EECS3101 Algorithms (Summer'26)

Digest of Lecture 1, May 5

Course Expectations, Assessments, and Foundations for Algorithm Analysis

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 10

Closing Reminder 11

Main Ideas

- The first lecture set up how EECS3101 will run: where to find materials, how assessments work, what classroom expectations apply, and how students should communicate with Jackie.
- The course is designed around steady participation-based practice, individual tests, and a large cumulative final exam, so weekly review is part of the course strategy rather than an end-of-term activity.
- Assignments are meant to be low-stress learning and self-assessment opportunities, while programming and written tests are the main places where individual mastery is assessed.
- Jackie framed the course as an extension of data structures into algorithm design, analysis, correctness, and implementation, especially for trees, graphs, heaps, and dynamic programming.
- The technical preview introduced the core analysis mindset: an algorithm should be evaluated for both efficiency, such as Big-O running time, and correctness, through tests and mathematical arguments.
- The lecture ended by recalling sorting algorithms and heap ideas, using them as early examples of how the course will connect data structures, algorithms, asymptotic analysis, and formal reasoning.

Roadmap

Course welcome, communication, and classroom conduct

Jackie introduced the course as a setup lecture rather than a heavy technical lecture. Students were asked to save general syllabus questions for the course-structure discussion, and Jackie emphasized that substantial questions are usually better handled in person, after class or during office hours, than through long email exchanges. He also reviewed classroom conduct expectations: avoid side conversations, minimize phone use, do not wear headphones during lecture, and use laptops or tablets for course-related work rather than unrelated browsing or assignments from other courses.

Where the course materials live

The lecture distinguished two regular sites students must check. eClass is used for announcements, assignment/test access, submissions, and the written test. Jackie's lecture site is used for lecture recordings, slides, iPad notes, clean notes templates, tutorial materials, and extra examples. Jackie demonstrated that the lecture site can be found through a search such as "EECS Jackie lectures" and showed the row-based layout for lecture materials. The clean notes templates are intended to support active note-taking before or during class, while annotated notes and recordings support later review.

Assignments, tests, exam, and deadlines

Jackie explained that the course has three assignments worth 5% in total, two programming tests, one written test, and a cumulative final exam. Assignments are participation-based: a sincere best attempt submitted by the deadline earns full assignment credit, but deadlines are strict. The tests are individual work and are scheduled during Monday tutorial time in a computer lab, not the usual tutorial room. The programming tests are done in Eclipse and graded by JUnit test cases, including hidden tests consistent with the given requirements. The final exam is 50%, cumulative, closed book, three hours, paper-and-pen, and mostly written.

Missed tests, accommodations, attendance, and iClicker

For any missed test, Jackie emphasized that it is the student's responsibility to contact him by the end of the test day. Possible accommodation depends on the case and is not automatic. Students with official accommodations were told to contact Jackie and follow the course policy. Attendance is tracked through iClicker, but it is not a direct component of the grading scheme. Instead, it may help as one positive discretionary factor in borderline cases. Jackie asked students to set up iClicker and noted that formal attendance tracking would begin after trial checks.

Open letter: learning responsibility and AI use

Jackie connected the open letter to common student struggles in foundational courses. The main message was that effort alone is not enough: students need active engagement, steady preparation, honest self-assessment, and timely help-seeking. Because assignments are graded for participation, they become especially important for diagnosing gaps before high-stakes tests. Jackie also discussed AI tools such as ChatGPT: they may help automate routine tasks or provide explanations, but they can be wrong or fabricated, so students need enough foundational knowledge to evaluate, correct, and refine AI output.

Professional competence and engineering context

For software engineering students in particular, Jackie briefly connected course knowledge to professional responsibility. The point was not that this lecture teaches licensing rules, but that algorithmic knowledge and implementation competence matter in real engineering work. The course expectation is therefore not just to recognize concepts, but to build enough competence to implement, test, analyze, and justify algorithms responsibly.

Course topic preview: data structures, algorithms, and analysis

The lecture then moved from administration into a technical preview. Data structures were recalled as systematic ways of organizing and accessing data, including arrays, linked lists, stacks, queues, maps, trees, and graphs. Algorithms were described as step-by-step procedures, expressed either informally or in code. Analysis was split into efficiency and correctness: efficiency includes asymptotic running time such as Big-O, worst-case and average-

case analysis, and tradeoffs in time and space; correctness includes JUnit tests for implementation behaviour and mathematical arguments such as induction and loop invariants.

Sorting as the first formal review example

Using sorting, Jackie previewed how formal specifications will appear in the course. If an output array is indexed from 0 through n, a sorting condition can be written as a universal statement over adjacent positions: for every valid i with $0 \leq i \leq n-1$, the condition $A[i] \leq A[i+1]$ should hold. The exact index bound depends on the indexing convention, but the reasoning habit is the same: specify the output precisely before analyzing or proving an algorithm.

Sorting algorithm recall and heap preview

The final technical portion recalled familiar sorting algorithms. Insertion sort and selection sort were identified as typical nested-loop algorithms with $O(n^2)$ running time. Merge sort was recalled as recursive and consistently $O(n \log n)$. Quicksort was used to highlight average-case versus worst-case thinking: a median-like pivot gives balanced partitions and $O(n \log n)$ behaviour, while poor pivot choices can lead to $O(n^2)$. Heap sort was introduced through the heap data structure: a heap is a complete binary tree with a min-key property, giving height $O(\log n)$ and motivating later priority queues for algorithms such as Dijkstra's shortest path and Kruskal's minimum spanning tree.

Key Terms, Notation, and Structures

- **eClass**: the official course platform for announcements, assessment access, submissions, and the written eClass test.
- **Jackie's lecture site**: the site for recordings, slides, iPad notes, notes templates, extra examples, and tutorial materials.
- **Participation-based assignments**: assignments graded for a best attempt submitted on time; their main study function is practice and self-assessment.
- **Programming tests**: individual Eclipse-based tests graded by JUnit test cases; visible starter tests guide requirements, but hidden tests check additional valid cases.
- **Written test**: an in-person invigilated eClass test with multiple-choice and written/typed components.
- **Final exam**: a cumulative, closed-book, three-hour written exam worth 50% of the course grade.
- **iClicker**: the attendance tool used for coarse attendance tracking; attendance is not a fixed grade component but may support discretion in borderline cases.

- **Data structure**: a systematic way of organizing and accessing data, such as an array, linked list, stack, queue, map, tree, graph, or heap.
- **Algorithm**: a step-by-step procedure that can be described in words, pseudocode, diagrams, or implemented in a programming language.
- **Algorithm analysis**: the study of an algorithm's efficiency and correctness, including running time, space tradeoffs, tests, and proofs.
- **Big-O notation**: asymptotic notation used to express growth of running time or space, especially for worst-case or average-case comparisons.
- **Correctness**: the requirement that an algorithm produces the required output for all valid inputs, not just for a few examples.
- **JUnit test cases**: executable tests comparing expected and actual behaviour; they are central to programming assignments and programming-test grading.
- **Hidden tests**: grading tests not shown in advance; they are intended to check boundary cases or larger inputs still within the stated specification.
- **Predicate logic**: notation using implication, negation, and quantifiers such as $\forall$; it will be needed for specifications, loop invariants, and proof arguments.
- **Sortedness specification**: for an array indexed 0 through n, one way to express nondecreasing order is $\forall i\,(0 \le i \le n-1 \Rightarrow A[i] \le A[i+1])$.
- **Insertion sort and selection sort**: familiar iterative sorting algorithms often analyzed through nested loops and $O(n^2)$ running time.
- **Merge sort**: a recursive divide-and-conquer sorting algorithm with $O(n \log n)$ running time.
- **Quicksort**: a recursive sorting algorithm whose running time depends strongly on pivot quality; balanced partitions support $O(n \log n)$, while poor pivots can lead to $O(n^2)$.
- **Heap**: a complete binary tree with a key-order property, such as the min-key property, giving height $O(\log n)$.
- **Priority queue**: an abstract data type that can be implemented efficiently with a heap and will reappear in graph algorithms.

Key Examples and Demonstrations

Course-materials walkthrough

Setup: Jackie showed eClass and the lecture site, including where announcements, lecture slides, recordings, iPad notes, clean notes templates, and tutorial files appear.

Main point: Successful studying in this course depends on checking both sites regularly and using recordings and notes as review tools, not as excuses to postpone learning.

What to practise: Find the Lecture 1 row, identify the clean notes template versus annotated notes, and mark where assignments, tutorials, recordings, and test-related documents will appear.

Assessment structure as a study plan

Setup: Jackie explained the 5% participation-based assignments, three 15% tests, and the 50% cumulative final exam.

Main point: The grading scheme is also a study roadmap: assignments are for practice and diagnosis, programming tests assess implementation through JUnit, the written test assesses conceptual and proof-oriented work, and the final exam rewards steady cumulative review.

What to practise: Put the test dates into your calendar, connect each assignment to the test it prepares for, and use the assignments to identify what you cannot yet explain or implement independently.

Formal specification of sorting

Setup: The notes show an unsorted array being transformed by sorting into an ordered array, with adjacent positions i and $i+1$ highlighted.

Main point: Sorting is not just an intuitive idea; it can be specified formally by saying that every adjacent pair in the valid index range is ordered, such as $A[i] \leq A[i+1]$.

What to practise: Rewrite the sorting condition using both words and predicate notation, and adjust the upper bound correctly for the indexing convention you are using.

Sorting-algorithm complexity recall

Setup: Jackie asked students to recall the running times and implementation styles of insertion sort, selection sort, merge sort, quicksort, and heap sort.

Main point: The same computational problem can have several algorithms with different running-time behaviours, and the reason for the bound matters as much as the bound itself.

What to practise: For each sorting algorithm, state whether it is typically iterative or recursive, give the relevant Big-O bound discussed in lecture, and explain one sentence of reasoning for that bound.

Quicksort pivot quality

Setup: Jackie used quicksort to ask when the running time should be $O(n \log n)$ and when it can degrade to $O(n^2)$.

Main point: Balanced partitioning, approximately around a median pivot, supports $O(n \log n)$ behaviour; badly unbalanced partitions can behave like quadratic sorting.

What to practise: Trace quicksort on a small array with a good pivot and with a poor pivot, and compare the shapes of the recursive subproblems.

Heap properties and priority queues

Setup: The notes introduce heap sort through a heap drawn as a binary tree with two highlighted properties: completeness and the min-key property.

Main point: A complete binary tree gives logarithmic height, and the min-key property makes the root the right place to find the smallest key; together these properties support efficient priority queues.

What to practise: Draw a small min-heap, identify its root and subtrees, and explain why completeness matters for the $O(\log n)$ height claim.

Common Pitfalls and Misconceptions

Treating assignments as "only participation marks"

Pitfall: Assuming the assignments are unimportant because they are graded for best attempt rather than detailed correctness.

Why it causes problems: The assignments are where you discover whether you can actually trace, prove, implement, and test the material before the individual tests and final exam.

How to avoid it: Attempt each assignment seriously, compare your work against the posted solution, and contact Jackie promptly when your attempt reveals a gap.

Leaving review until just before a test

Pitfall: Waiting until a few days before a test or the final exam to begin serious preparation.

Why it causes problems: The course is cumulative, the exam is 50%, and later topics depend on earlier reasoning habits such as logic, Big-O, induction, and data-structure properties.

How to avoid it: Do a weekly review using the lecture materials, assignments, and this digest as a checklist, then ask for help while the gap is still small.

Using recordings as a replacement for first-pass learning

Pitfall: Skipping class because the recording will be available later.

Why it causes problems: Jackie emphasized that even partial comprehension during class makes later review more effective; recordings are meant mainly for catch-up and targeted review.

How to avoid it: Attend when possible, use the recording to revisit difficult segments, and pair the recording with the slides and iPad notes.

Expecting partial-credit code review in programming tests

Pitfall: Assuming Jackie or the TA will manually inspect nearly-correct code and award partial marks for intended logic.

Why it causes problems: Programming tests are graded by JUnit test cases, including hidden tests, so code that fails required behaviour may lose credit even if the idea seems close.

How to avoid it: Make your code compile, avoid exceptions such as null-pointer and array-index errors, satisfy the full specification, and test boundary cases beyond the visible starter tests.

Trusting AI output without verification

Pitfall: Treating ChatGPT or another AI tool as an authority for code, proofs, or explanations.

Why it causes problems: AI-generated output can be hallucinated, incomplete, or mismatched to the exact specification; relying on it without understanding can fail tests and weaken exam preparation.

How to avoid it: Use AI output only as something to inspect critically, then verify it against the course specification, examples, tests, and your own reasoning.

Confusing average-case and worst-case behaviour

Pitfall: Saying quicksort is simply $O(n \log n)$ without qualification.

Why it causes problems: Quicksort's behaviour depends on pivot quality; poor pivots can produce highly unbalanced recursion and $O(n^2)$ time.

How to avoid it: Always ask what case is being analyzed, what assumption is being made, and what structural feature of the algorithm justifies the bound.

Forgetting the exact missed-test responsibility

Pitfall: Missing a test and waiting several days before contacting Jackie.

Why it causes problems: The course policy places responsibility on the student to contact Jackie by the end of the test day; delayed contact can make accommodation harder or impossible.

How to avoid it: If a test is missed, contact Jackie the same day and be prepared to discuss the situation according to the syllabus policy.

What You Should Be Able to Do

After this lecture, you should be able to:

- locate the course eClass page and Jackie's lecture site, and explain what each site is used for;
- find lecture recordings, slides, clean iPad notes templates, annotated iPad notes, and tutorial materials on the lecture site;
- state the grading scheme: assignments worth 5%, three tests worth 15% each, and a cumulative final exam worth 50%;

- explain the purpose of participation-based assignments and why they still require a serious best attempt;
- describe the difference between programming tests, the written test, and the final exam in format, grading, and preparation strategy;
- state what to do if you miss a test and why contacting Jackie by the end of that day matters;
- set up iClicker and explain how attendance may be used as a discretionary, not automatic, factor;
- explain Jackie's expectations for email, office hours, classroom conduct, and in-person communication;
- summarize the open-letter message about steady preparation, honest self-assessment, and seeking help early;
- explain why AI tools must be checked against the course specification and your own understanding;
- distinguish between data structures, algorithms, efficiency analysis, and correctness analysis;
- write a formal adjacent-pair sorting condition using universal quantification and adjust the index range correctly;
- compare insertion sort, selection sort, merge sort, quicksort, and heap sort at the level of implementation style and asymptotic behaviour discussed in lecture;
- explain why quicksort's pivot choice affects whether its running time behaves like $O(n \log n)$ or $O(n^2)$;
- describe the two heap properties introduced in lecture: completeness and the min-key property;
- explain why heaps are relevant to priority queues and why priority queues will matter later for graph algorithms.

Suggested Practice

- Read the syllabus and open letter fully, then make a personal checklist of the policies that require action from you: deadlines, missed-test contact, iClicker setup, office hours, and allowed collaboration.
- Add the three test dates to your calendar: Programming Test 1 on June 22, Written Test on July 13, and Programming Test 2 on July 20.

- Visit both eClass and Jackie's lecture site; locate the Lecture 1 materials, Tutorial 1 recording, weekly schedule, and assessment-date document.
- Download or bookmark the clean iPad notes template for Lecture 1 and compare it with the annotated version after lecture.
- Review prerequisite logic from EECS1090, especially implication, negation, universal quantification, and how to read statements with $\forall$.
- Write the sorting condition for arrays under two indexing conventions: indices $0, \ldots, n$ and indices $0, \ldots, n-1$.
- Make a small table comparing insertion sort, selection sort, merge sort, quicksort, and heap sort by recursion/iteration, key idea, and Big-O behaviour discussed in lecture.
- Trace quicksort on a short array using a median-like pivot and then using a poor pivot; sketch the recursive subproblem sizes.
- Draw a small min-heap and check both properties: complete binary-tree shape and the min-key property at each relevant node.
- For programming-test preparation, start thinking in terms of specifications and test cases: ask what inputs are valid, what outputs are expected, and what hidden boundary cases might be reasonable.

Where to Review

- **iPad/PDF notes, page 1:** lecture title and scope: syllabus, introduction, open letter, responsibilities, standards, and recall of sorting problems and algorithms.
- **iPad/PDF notes, page 2:** course description and topic list, including Big-O analysis, worst-case versus average-case running time, tradeoffs, balanced BSTs, heaps, graphs, greedy algorithms, correctness, induction, and dynamic programming.
- **iPad/PDF notes, page 3:** sorting review, including the adjacent-pair sorting specification, sorting-algorithm running times, quicksort pivot issue, and heap properties.
- **Lecture recording:** revisit the early portion for course logistics, assessment rules, classroom conduct, eClass, the lecture site, and office-hour expectations.
- **Lecture recording:** revisit the middle portion for the syllabus, assignment policy, test dates, missed-test policy, accommodations, attendance, iClicker, and the open letter.
- **Lecture recording:** revisit the final portion for the technical preview of data structures, algorithms, efficiency, correctness, sorting, quicksort, heaps, and priority queues.
- **Course syllabus and schedule documents:** use these as the authority for deadlines, venues, office hours, assessment formats, and any later policy updates Jackie announces.

- **Open letter:** review the sections on studying actively, self-assessment, programming-test standards, and why foundational knowledge matters even when AI tools are available.
- **Optional background resources mentioned by Jackie:** use recommended algorithm texts or the linked Princeton-style resources only if you want extra reading beyond the course slides and notes.

Closing Reminder

This digest is a roadmap for Lecture 1, not a substitute for the syllabus, open letter, slides, iPad notes, recording, assignments, or your own practice. The most important next step is to turn the administrative information into concrete actions and the technical preview into active review: set up the tools, mark the dates, read the policies, revisit the sorting and heap examples, and start practising the habit of explaining, tracing, testing, and justifying your work.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 2, May 7

Asymptotic Analysis and Formal Upper (Big-O) Bounds

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 10

Closing Reminder 11

Main Ideas

- The lecture moved from course setup into the first core technical theme: asymptotic analysis of algorithms.
- Jackie connected computational problems, algorithms, and data structures by separating external specifications from internal implementation choices.
- Correctness was presented as the first criterion for an algorithm, before efficiency, with testing, loop invariants, and mathematical induction as the main tools to revisit.
- Running time analysis was shifted from absolute timings on a particular machine to relative running time functions based on primitive operations and input size.
- The lecture introduced the habit of approximating a running-time expression by its dominant term, after dropping lower-order terms and multiplicative constants.
- Growth-rate classes such as constant, logarithmic, linear, $n \log n$, quadratic, and exponential were compared, with naive recursive Fibonacci used as a warning about overlapping subproblems.
- The formal Big-O definition was introduced using a running-time function $f(n)$, a reference function $g(n)$, a constant c, and a threshold n_0.
- Several logistics were also important for study planning: re-download the updated Lecture 2 notes template, study slides 4–17, review the discrete math slides, and note that next Monday and Tuesday are lectures while next Thursday is Tutorial 2.

Roadmap

Announcements, materials, and preparation

Jackie began by pointing students back to the lecture site. Lecture 1 materials and Tutorial 1 materials were already posted, and the Lecture 2 notes template had been updated with one additional example, so anyone who downloaded it earlier should re-download it. Jackie also reminded students that course recordings and notes will appear on the lecture site rather than through daily email announcements.

The discrete math review slides were identified as background material rather than in-class lecture content. They review propositional logic, predicate logic, implication, biconditional, quantifiers, proof and disproof strategies, De Morgan's laws, and related tools from EECS1090. Jackie emphasized that these slides are dense but sufficient for this course; students should use them as needed rather than returning to a full discrete math textbook.

Jackie also announced upcoming practice material on OOP, linked lists, recursion, and trees, intended to help students prepare for later programming-test topics such as balanced binary search trees and tree rotations. For the following week, Monday and Tuesday would be Lectures 3 and 4, while Thursday would be Tutorial 2; students should go to the usual venue for each scheduled meeting.

Why algorithms need specifications

The first technical part used the notes' diagram of an algorithm as a black box with inputs and outputs, while internally containing steps and data structures. A computational problem specifies what relation between input and output should be achieved; an algorithm gives a step-by-step procedure for achieving it; and data structures support the operations used inside the algorithm.

Two examples framed this distinction. Sorting takes an input sequence and produces a sorted output sequence, but the internal algorithm could be merge sort, quick sort, selection sort, insertion sort, heap sort, or another method. Shortest paths in a weighted graph take a graph and a source vertex as input, with Dijkstra's algorithm later serving as one algorithmic solution under appropriate preconditions such as non-negative edge weights.

The key study point is to separate the specification from the implementation. Preconditions describe valid inputs; postconditions describe correct outputs upon termination. Internal choices such as a heap-based priority queue for Dijkstra's algorithm are implementation decisions that help the algorithm run efficiently, but they should not be confused with the problem specification itself.

Correctness before efficiency

Jackie emphasized that correctness is the first criterion for judging an algorithm. An efficient algorithm is not useful if it produces the wrong result or fails to terminate. The lecture identified three major correctness tools: software testing, loop invariants, and mathematical induction.

Testing is the practical habit students have already seen through JUnit and will continue to use in programming work. Loop invariants will become important for iterative algorithms, including the future treatment of Dijkstra's algorithm. Mathematical induction will be needed for reasoning about recursive structures, trees, graphs, and algorithmic properties. This part of the lecture explains why the discrete math review is not optional background: the logic and proof patterns will support later correctness arguments.

From absolute running time to relative running time

The next part moved from empirical timing to mathematical analysis. Jackie explained that measuring milliseconds on a machine is not the right primary tool for comparing algorithms, because the same algorithm may run at different absolute speeds on different hardware.

The notes used two machines as an example: a newer Mac with per-primitive-operation time $t = 2$, and an older Mac with $t = 10$. Two relative running-time functions, such as $7n - 2$ and $10n+3$, were compared at an example input size. The hardware-dependent time t multiplies the running time, but it does not change the asymptotic class. This motivates focusing on a relative running-time function $f(n)$, where n is the input size and $f(n)$ approximates the number of primitive operations.

Jackie also clarified what counts as a primitive operation: examples include assignments, array indexing, arithmetic, comparisons, logical operations, field access, and returns. A method call itself is not automatically primitive; students must reason about the body of the

method. A method that only prints a fixed string may be constant time, while a method that performs merge sort may be $O(n \log n)$.

Dominant terms and growth-rate intuition

The notes then worked through the expression

$$f(n) = 7n + 2n \log n + 3n^2.$$

The goal was not to compute an exact count, but to identify the asymptotic growth. Jackie highlighted three habits: identify the highest-power or dominant term, drop lower-order terms, and drop multiplicative constants. Since n^2 dominates both $n \log n$ and n, this expression is approximated by n^2 for asymptotic upper-bound purposes.

The growth-rate diagram on the notes compared common functions: constant, logarithmic, linear, $n \log n$, quadratic, cubic, and exponential. Jackie used this diagram to stress how quickly exponential running time becomes impractical. As a preview, naive recursive Fibonacci was used to show overlapping subproblems: the same smaller Fibonacci value is recomputed multiple times, causing exponential blowup. Dynamic programming was previewed as a later technique that avoids this recomputation and can reduce the Fibonacci-style computation to linear time with a table.

Formal Big-O definition

The last major technical segment introduced the formal definition of Big-O. The running-time function is $f(n)$, the reference function is $g(n)$, and the goal is to prove that $f(n)$ belongs to the family $O(g(n))$. Formally, this means finding a real constant $c > 0$ and an integer threshold $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

The notes' diagram showed that the inequality does not need to hold for every small input. The threshold n_0 marks where the upper-bound effect begins. Before n_0, $f(n)$ may be larger than $c \cdot g(n)$; from n_0 onward, the scaled reference function must stay at or above $f(n)$.

The worked example used $f(n) = 8n + 5$, $g(n) = n$, and chose $c = 9$. Jackie checked that $n_0 = 1$ does not work because $f(1) = 13$ is not at most 9. Solving $8n_0 + 5 = 9n_0$ gives $n_0 = 5$, so with $c = 9$, the bound starts at $n_0 = 5$. Jackie noted that in assessments students should usually identify the smallest possible n_0 for the chosen c.

Key Terms, Notation, and Structures

- **Computational problem**: A specification of what output should be produced for valid inputs, independent of the internal algorithm used.
- **Algorithm**: A step-by-step procedure for solving a computational problem.

- **Data structure**: A way to organize and access data so that an algorithm's steps can be carried out effectively.
- **Precondition**: A condition describing which inputs are valid before the algorithm is applied.
- **Postcondition**: A condition describing what must be true of the output after the algorithm terminates.
- **Termination**: The requirement that an algorithm eventually finishes, rather than running forever.
- **Correctness**: The property that an algorithm terminates and produces outputs satisfying the postcondition for valid inputs.
- **Testing**: A practical way to check correctness on selected cases; useful but not the same as a proof for all inputs.
- **Loop invariant**: A property maintained through loop iterations, later used to prove correctness of iterative algorithms.
- **Mathematical induction**: A proof technique that will support correctness arguments for recursive and structured objects such as trees and graphs.
- **Primitive operation**: A constant-time basic step such as assignment, array indexing, arithmetic, comparison, logical operation, field access, or returning from a method.
- **Method call**: Not automatically primitive; its cost depends on the work done inside the method body.
- **Input size** n: The measure of problem size used in a running-time function.
- **Relative running time** $f(n)$: A function estimating the number of primitive operations as a function of input size.
- **Absolute running time**: Hardware-dependent time, modeled in the lecture as relative running time multiplied by a per-operation time t.
- **Multiplicative constant**: A constant factor such as 2, 10, or 3 that can be dropped when focusing on asymptotic growth.
- **Lower-order term**: A term that grows more slowly than the dominant term, such as n or $n \log n$ when compared with n^2.
- **Dominant term**: The term that determines asymptotic growth, such as n^2 in $7n+2n \log n+3n^2$.
- **Reference function** $g(n)$: The function used as the comparison target in a Big-O proof.
- **Big-O membership**: The statement $f(n) \in O(g(n))$, meaning $f(n)$ is eventually upper bounded by $c \cdot g(n)$ for some $c > 0$.

- **Threshold** n_0: The input size from which the Big-O inequality must hold for all larger inputs.
- **Overlapping subproblems**: Repeated computation of the same subproblem, illustrated by naive recursive Fibonacci.
- **Dynamic programming**: A later technique previewed as a way to avoid repeated subproblem computation by storing results.
- **Dijkstra's algorithm**: A future shortest-path algorithm for weighted graphs, previewed as using a priority queue and heap.

Key Examples and Demonstrations

Problem, algorithm, and data structure separation

Setup: The notes show an algorithm as a box taking input and producing output, with internal steps and data structures, and connected to a computational problem.

Main point: Students should distinguish the external specification from the internal implementation. Sorting specifies a sorted output sequence; the algorithm might be selection sort, insertion sort, merge sort, quick sort, or heap sort. Shortest paths in a weighted graph specify desired paths or distances; Dijkstra's algorithm and a heap-based priority queue are internal choices for solving that problem.

What to practise: For a familiar problem such as sorting, write down the precondition, postcondition, one possible algorithm, and one data structure used internally. Keep these four items separate.

Absolute versus relative running time

Setup: The notes compare two machines, with per-operation times such as $t = 2$ and $t = 10$, and two relative running-time functions such as $7n - 2$ and $10n + 3$ at $n = 100$.

Main point: Absolute running time depends on the execution environment, while relative running time captures the algorithmic growth independent of hardware. Since the per-operation time is a multiplicative constant, it does not determine the Big-O class.

What to practise: Recompute the table for another input size, then explain in words why changing the machine affects milliseconds but not the asymptotic comparison.

Approximating $7n + 2n \log n + 3n^2$

Setup: The lecture analyzed $f(n) = 7n + 2n \log n + 3n^2$, with $\log n$ taken as base 2 unless otherwise stated.

Main point: The dominant term is n^2. The terms $7n$ and $2n \log n$ are lower order, and the constant factor 3 is dropped for the asymptotic approximation.

What to practise: For several expressions, circle the multiplicative constants, underline the lower-order terms, and box the dominant term before stating the Big-O class.

Growth-rate diagram and Fibonacci preview

Setup: The notes' growth-rate diagram compares constant, logarithmic, linear, $n \log n$, quadratic, cubic, and exponential functions. Jackie also sketched recursive Fibonacci, where $F(4) = F(3) + F(2)$ and $F(3) = F(2) + F(1)$.

Main point: Exponential growth can become impractical very quickly. The Fibonacci sketch shows why overlapping subproblems matter: the same subproblem can be recomputed many times in naive recursion. Dynamic programming will later address this by storing subproblem results.

What to practise: Draw the recursion tree for a small Fibonacci input and mark repeated subproblems. Then describe how a table would avoid recomputing them.

Big-O proof for $f(n) = 8n + 5$

Setup: The example asks students to prove $f(n) = 8n+5 \in O(n)$, using $g(n) = n$ and choosing $c = 9$.

Main point: A Big-O proof must give both a constant c and a threshold n_0. With $c = 9$, $n_0 = 1$ fails because $13 \leq 9$ is false. Solving $8n_0 + 5 = 9n_0$ gives $n_0 = 5$, so the inequality holds from $n = 5$ onward.

What to practise: Repeat the proof with a different choice of c, such as a larger value, and observe how the possible threshold n_0 changes.

Common Pitfalls and Misconceptions

Treating specifications and implementations as the same thing

Pitfall: Describing a computational problem by naming an algorithm or data structure only.

Why it causes problems: It hides the actual correctness requirement. For example, "use a heap" is not the same as specifying the shortest-path problem.

How to avoid it: First write the input, output, precondition, and postcondition; then choose the algorithm and supporting data structures.

Analyzing efficiency before checking correctness

Pitfall: Asking whether an algorithm is $O(n)$ or $O(n^2)$ before asking whether it terminates and produces correct output.

Why it causes problems: Running-time analysis of an incorrect algorithm does not establish that the algorithm solves the problem.

How to avoid it: Use tests, loop invariants, induction, and clear postconditions to reason about correctness first; then analyze efficiency.

Counting method calls as primitive operations

Pitfall: Treating every method call as one constant-time step.

Why it causes problems: A method call may hide substantial work, such as sorting or traversing a data structure.

How to avoid it: Ask what the method body does. Replace a method call by its known or derived cost before adding it into a running-time expression.

Using absolute running time as the main comparison

Pitfall: Comparing algorithms mainly by milliseconds from one computer.

Why it causes problems: Hardware speed changes absolute time but does not change the asymptotic growth class.

How to avoid it: Analyze the relative running-time function $f(n)$ and use absolute timing only as supplementary evidence.

Keeping lower-order terms and constants as if they control Big-O

Pitfall: Reporting a cluttered expression such as $7n + 2n \log n + 3n^2$ as the asymptotic class, or letting the constant 3 distract from the dominant term.

Why it causes problems: It misses the purpose of asymptotic approximation: describing growth for large input sizes.

How to avoid it: Identify the dominant term, drop lower-order terms, and drop multiplicative constants.

Assuming the Big-O inequality must work from $n = 1$

Pitfall: Choosing $n_0 = 1$ without checking the inequality.

Why it causes problems: The definition only requires the inequality to hold from some threshold onward. In the example with $c = 9$, $n_0 = 1$ fails.

How to avoid it: Substitute the proposed n_0 into the inequality, solve for the threshold when possible, and choose the smallest valid threshold requested by Jackie.

Misreading notation such as $2n \log n$

Pitfall: Confusing $2n \log n$ with a square-root term or misidentifying powers.

Why it causes problems: Misreading the expression leads directly to the wrong dominant term and wrong Big-O classification.

How to avoid it: Rewrite each term clearly before comparing growth rates; unless stated otherwise in this course, treat $\log n$ as base 2.

What You Should Be Able to Do

After this lecture, you should be able to:

- locate the updated Lecture 2 notes template, the posted Tutorial 1 materials, and the discrete math review slides;

- explain the difference between a computational problem, an algorithm, and a data structure;
- state preconditions, postconditions, and termination requirements for a simple algorithmic problem;
- explain why correctness must be considered before running-time efficiency;
- name testing, loop invariants, and mathematical induction as correctness tools, and explain the role of each at a high level;
- distinguish absolute running time from relative running time;
- identify common primitive operations and explain why a method call is not automatically primitive;
- explain why multiplicative constants from hardware speed or expression coefficients are dropped in asymptotic analysis;
- approximate a multi-term running-time expression by finding the dominant term;
- compare the rough growth of constant, logarithmic, linear, $n \log n$, quadratic, and exponential functions;
- recognize overlapping subproblems in a small recursion tree such as Fibonacci;
- state the formal Big-O definition using $f(n)$, $g(n)$, c, and n_0;
- prove a simple statement such as $8n + 5 \in O(n)$ by giving a valid c and n_0;
- explain why the Big-O inequality may fail for inputs smaller than n_0 without invalidating the proof.

Suggested Practice

- Re-download the Lecture 2 notes template if you saved it before class, then compare it with your own notes after reviewing the recording.
- Study slides 4–17 as assigned reading, with special attention to the relationship among problems, algorithms, data structures, correctness, and efficiency.
- Review the discrete math slides on implication, quantifiers, proof and disproof strategies, and De Morgan's laws; ask Jackie about any part that is still unclear.
- Write preconditions and postconditions for sorting and for a simple graph problem, without mentioning the internal data structure first.
- Take three expressions such as $5n^2+12n+7$, $4n \log n+9n$, and 2^n+n^3; identify the dominant term and state the Big-O class.

- For a small value such as $n = 4$ or $n = 5$, draw the naive Fibonacci recursion tree and mark repeated subproblems.
- Practise Big-O proofs by selecting c, solving for a threshold n_0, and verifying the inequality for all $n \geq n_0$.
- Prepare for the next week by expecting Lectures 3 and 4 on Monday and Tuesday, followed by Tutorial 2 on Thursday, where average-case or amortized analysis will be developed.
- Watch for Jackie's posted practice exercises on OOP, recursion, linked lists, and trees, especially if you want early preparation for balanced BST and rotation material.

Where to Review

- **Lecture 2 iPad notes, page 1**: overall lecture title and focus: asymptotic analysis, problems/algorithms/data structures, absolute versus relative running time, and Big-O definitions.
- **Lecture 2 iPad notes, page 2**: announcements and reminders, including posted materials, discrete math review, upcoming OOP/recursion practice, next week's lecture/tutorial plan, and office hour.
- **Lecture 2 iPad notes, page 3**: assigned reading reminder for slides 4–17.
- **Lecture 2 iPad notes, page 4**: main diagram connecting computational problems, algorithms, data structures, input/output, preconditions/postconditions, and correctness tools.
- **Lecture 2 iPad notes, page 5**: table comparing absolute and relative running time across different machines and algorithms.
- **Lecture 2 iPad notes, page 6**: dominant-term approximation for $7n + 2n \log n + 3n^2$, including lower-order terms and multiplicative constants.
- **Lecture 2 iPad notes, page 7**: growth-rate diagram and Fibonacci recursion sketch for overlapping subproblems and dynamic programming preview.
- **Lecture 2 iPad notes, pages 8–9**: Big-O membership, the roles of $f(n)$, $g(n)$, c, and n_0, and the worked example $8n + 5 \in O(n)$.
- **Lecture recording**: revisit the explanations around primitive operations, why method calls are not automatically primitive, and why $n_0 = 1$ fails in the Big-O example.
- **Discrete math review slides**: use them for implication, quantifiers, proof strategies, and logic notation before loop invariants and induction appear later.

Closing Reminder

This digest is only a roadmap for Lecture 2. To learn the material, review the recording, rework the iPad-note examples, study slides 4–17, and practise writing small Big-O proofs yourself. The important skill is not just recognizing a final answer such as $O(n^2)$, but explaining why the dominant term, constants, threshold n_0, and correctness conditions justify the conclusion.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 3, May 11

Big-O Proofs and Triangular-Sum Loop Analysis

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 7

Suggested Practice 8

Where to Review 9

Closing Reminder 9

Main Ideas

- The lecture continued the asymptotic-analysis unit by making the formal Big-O definition operational: identify a running-time function $f(n)$, a reference function $g(n)$, and witnesses c and n_0.
- Jackie connected Big-O notation to predicate logic: the proof structure is existential for the witnesses and universal for all sufficiently large input sizes.
- A practical theorem was introduced for polynomial running-time functions: choose C as the sum of the absolute values of the coefficients, and use $n_0 = 1$.
- The lecture warned that this polynomial shortcut does not automatically apply to functions involving terms such as $\log n$; the threshold n_0 must be checked explicitly.
- The second half shifted from proving bounds for mathematical functions to deriving bounds from code, using integer intervals, arithmetic sums, and a dependent nested-loop example.
- Students were reminded to review the discrete-math material, especially logical implication, quantifiers, and proof strategies, because these are needed for written work in the course.

Roadmap

Announcements, discrete-math review, and attendance setup

Jackie began by clarifying that this Monday session was Lecture 3, not a tutorial. Lecture 4 would follow on Tuesday, and Tutorial 2 would be held on Thursday in the Chemistry Building. Students were reminded to review the discrete-math slides on the lecture site, especially propositional logic, predicate logic, quantifiers, and proof strategies. Jackie also noted that this lecture would include one more trial attendance check, with actual attendance counting beginning on Tuesday.

Revisiting the Big-O picture

The first technical part revisited the Big-O visualization from the previous lecture. The running-time function $f(n)$ was compared with a reference function $g(n)$. Jackie emphasized that $g(n)$ alone may not upper-bound $f(n)$, so Big-O allows a multiplicative constant c to scale $g(n)$. The threshold n_0 marks where the upper-bound effect begins: for $n < n_0$, the inequality may fail; for $n \geq n_0$, the proof must show $f(n) \leq c \cdot g(n)$.

Big-O as a predicate-logic statement

The lecture then reformulated Big-O using nested quantifiers. The key structure is:

$$f(n) \in O(g(n)) \iff \exists c, n_0 \bullet c > 0 \wedge n_0 \geq 1 \wedge \big(\forall n \bullet n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n)\big)$$

The existential part corresponds to choosing witnesses c and n_0; the universal part says that, after the threshold, the inequality must hold for every valid input size. Jackie highlighted the implication symbol in the inner statement and posed a study question: what would change if that implication were replaced by conjunction?

A theorem for polynomial Big-O proofs

Jackie next introduced a proof strategy for polynomial running-time functions. If $f(n)$ is a polynomial of degree d, then $f(n)$ is in $O(n^d)$. A valid choice is

$$\begin{aligned} C &= |a_0| + |a_1| + \cdots + |a_d| \\ n_0 &= 1 \end{aligned}$$

where the a_i are the polynomial coefficients. The point is not just to guess the highest power, but to provide concrete witnesses that satisfy the formal definition.

Polynomial example and the logarithm warning

For the polynomial example

$$f(n) = 5n^4 - 3n^3 + 2n^2 - 4n + 1,$$

Jackie chose $g(n) = n^4$, computed $C = 15$, and used $n_0 = 1$, which is guaranteed because the function is polynomial only. The lecture then contrasted this with

$$f(n) = 3\log n + 2.$$

Here the most accurate upper-bound class discussed was $O(\log n)$, but using the same coefficient-sum idea with $C = 5$ does not make $n_0 = 1$ work, because $\log 1 = 0$. This was the key warning: for non-polynomial terms, especially logarithms, verify the threshold instead of assuming it.

Counting intervals and arithmetic sums

Before deriving a running time from code, Jackie reviewed two counting tools. First, a closed integer interval $[a, b]$ contains $b - a + 1$ integers. Second, the sum of an arithmetic sequence is

$$\frac{(\text{first term} + \text{last term}) \cdot (\text{number of terms})}{2}.$$

These formulas are not isolated facts; they are the counting machinery needed to analyze loops whose iteration counts form a sequence.

Deriving $O(n^2)$ from a dependent nested loop

The final worked code example analyzed a method in which the outer loop runs with $i = 0, 1, \ldots, n-1$, while the inner loop starts at $j = i$ and runs to $n - 1$. Jackie represented the possible (i, j) combinations as a triangular pattern. The inner statement executes

$$n + (n-1) + (n-2) + \cdots + 1 = \frac{(n+1) \cdot n}{2}$$

times, so the dominant term is n^2, giving an asymptotic upper bound of $O(n^2)$. Jackie connected this technique to the next lecture, where similar reasoning would be used for sorting algorithms such as selection sort.

Key Terms, Notation, and Structures

- $f(n)$: The running-time function for an algorithm, expressed as a function of input size n.
- $g(n)$: The reference function used to state a candidate Big-O upper bound.
- c **or** C: A positive multiplicative constant used to scale $g(n)$ so that it can upper-bound $f(n)$ eventually.
- n_0: The threshold input size from which the Big-O inequality must hold for all larger input sizes.
- **Witnesses**: The chosen values of c and n_0 in a Big-O proof.
- **Upper-bound effect**: Jackie's phrase for the point at which $f(n) \leq c \cdot g(n)$ begins to hold and continues to hold.
- **Predicate-logic formulation**: A formal statement using existential quantifiers for c, n_0 and a universal quantifier for all $n \geq n_0$.
- **Implication**: In $n \geq n_0 \implies f(n) \leq c \cdot g(n)$, implication restricts the obligation to input sizes at or beyond the threshold.
- **Polynomial degree**: The highest exponent in a polynomial; if $f(n)$ has degree d, the natural Big-O candidate is $O(n^d)$.
- **Coefficient-sum theorem**: For a polynomial, one valid witness choice is C equal to the sum of the absolute values of the coefficients and $n_0 = 1$.
- **Non-polynomial term**: A term such as $\log n$ that prevents automatic use of the polynomial-only guarantee $n_0 = 1$.
- **Closed interval** $[a, b]$: An inclusive integer range containing $b - a + 1$ values.
- **Arithmetic sequence**: A sequence with a common difference; its sum can be computed using first term, last term, and number of terms.
- **Triangular sum**: A sum of the form $n + (n-1) + \cdots + 1$, which simplifies to $n(n+1)/2$ and has dominant term n^2.
- **Dependent nested loop**: A nested loop where the inner-loop range depends on the outer-loop counter, so the total work must be summed rather than guessed.

Key Examples and Demonstrations

Big-O visualization with $f(n) = 8n + 5$ and $g(n) = n$

Setup: The notes showed $f(n) = 8n + 5$ as the running-time function and $g(n) = n$ as the reference function, then scaled $g(n)$ to $9 \cdot g(n) = 9n$.

Main point: The unscaled reference function does not upper-bound $f(n)$, but the scaled function does starting at $n_0 = 5$. This illustrates why both c and n_0 appear in the formal Big-O definition.

What to practise: Redraw the three curves $f(n)$, $g(n)$, and $c \cdot g(n)$, then explain in words what changes before and after n_0.

Predicate formulation of Big-O

Setup: Jackie asked students to express "$f(n)$ is in $O(g(n))$" using logical operators and quantifiers.

Main point: The proof pattern is: choose witnesses c and n_0, then prove a statement for every input size n at or beyond the threshold. The implication $n \geq n_0 \implies f(n) \leq c \cdot g(n)$ is essential because Big-O does not require the inequality to hold before n_0.

What to practise: Write the predicate-logic statement from memory, then answer Jackie's study question by comparing implication and conjunction using a truth table.

Polynomial proof example

Setup: The worked polynomial was

$$f(n) = 5n^4 - 3n^3 + 2n^2 - 4n + 1.$$

The goal was to prove $f(n) \in O(n^4)$.

Main point: Since this is polynomial only, the coefficient-sum theorem applies directly: choose $C = |5| + |-3| + |2| + |-4| + |1| = 15$ and $n_0 = 1$.

What to practise: Apply the same method to another polynomial: identify the highest power, compute C from absolute coefficient values, and verify the starting point.

Logarithmic example and threshold verification

Setup: The non-polynomial example was

$$f(n) = 3 \log n + 2.$$

The candidate upper bound discussed was $O(\log n)$.

Main point: Using $C = 5$ does not make $n_0 = 1$ valid, because $\log 1 = 0$ and the required inequality would become $2 \leq 0$. For functions involving $\log n$, the threshold must be tested rather than assumed.

What to practise: With base-2 logarithms, test $n_0 = 1$ and then $n_0 = 2$ for this example; explain why the first fails and why the next candidate works.

Closed intervals and arithmetic sequences

Setup: Jackie reviewed how many integers are in $[a, b]$, and how to sum sequences such as $1 + 2 + 3 + \cdots + 100$.

Main point: These formulas support loop analysis. The interval size is $b - a + 1$, and an arithmetic-sequence sum is

$$\frac{(\text{first} + \text{last}) \cdot \text{number of terms}}{2}.$$

What to practise: Compute interval sizes such as $[34, 100]$, then use the arithmetic-sum formula on short sequences before applying it to loop counts.

Triangular-sum nested loop

Setup: The code example initialized a sum, then used an outer loop over $i = 0, \ldots, n - 1$ and an inner loop over $j = i, \ldots, n - 1$ before returning the sum.

Main point: The number of executions of the inner statement is the number of valid (i, j) combinations. These form the pattern $n, (n - 1), (n - 2), \ldots, 1$, so the total work is $n(n + 1)/2$, which is $O(n^2)$.

What to practise: Recreate the triangular table of (i, j) values, count each row, and justify why the final answer is quadratic even though the inner-loop start changes with i.

Common Pitfalls and Misconceptions

Forgetting that $g(n)$ may need to be scaled

Pitfall: Assuming that $g(n)$ alone must upper-bound $f(n)$.

Why it causes problems: Big-O allows a multiplicative constant. Without $c \cdot g(n)$, many valid upper-bound relationships would look false.

How to avoid it: In every formal proof, explicitly identify $f(n)$, $g(n)$, c, and n_0.

Treating Big-O as if it must hold for every n

Pitfall: Requiring $f(n) \leq c \cdot g(n)$ even for input sizes before the threshold.

Why it causes problems: Big-O is asymptotic. The inequality only needs to hold for all $n \geq n_0$.

How to avoid it: Always state the threshold condition, and remember that values with $n < n_0$ are not part of the proof obligation.

Replacing implication by conjunction in the formal definition

Pitfall: Writing $n \geq n_0 \wedge f(n) \leq c \cdot g(n)$ where the definition needs $n \geq n_0 \Longrightarrow f(n) \leq c \cdot g(n)$.

Why it causes problems: Conjunction changes the meaning of the statement and no longer captures the idea that the inequality is required only when n is at or beyond the threshold.

How to avoid it: Review truth tables for implication and conjunction, then test both forms on values below n_0.

Assuming $n_0 = 1$ outside the polynomial theorem

Pitfall: Applying the polynomial-only shortcut to functions involving $\log n$ or other non-polynomial terms.

Why it causes problems: The example $3 \log n + 2$ shows that $n_0 = 1$ can fail immediately because $\log 1 = 0$.

How to avoid it: Use the shortcut only for polynomial functions. For non-polynomials, test the proposed threshold explicitly and increase n_0 if needed.

Guessing nested-loop cost without counting combinations

Pitfall: Looking at two loops and either blindly multiplying $n \cdot n$ or assuming the dependence of j on i automatically changes the class.

Why it causes problems: Dependent nested loops require summing the actual number of inner-loop executions. In this lecture, the exact pattern was triangular.

How to avoid it: Make a table of valid (i, j) pairs, count each row, and then simplify the resulting sum.

Confusing n with n_0

Pitfall: Treating n_0 as another running input variable.

Why it causes problems: n_0 is a fixed witness chosen in the proof; n is the general input-size variable over which the final condition is universally quantified.

How to avoid it: Say the roles aloud: "choose n_0 once; then prove the inequality for every $n \geq n_0$."

What You Should Be Able to Do

After this lecture, you should be able to:

- explain the roles of $f(n)$, $g(n)$, c, and n_0 in the formal Big-O definition;
- state the Big-O definition using nested quantifiers and the implication $n \geq n_0 \implies f(n) \leq c \cdot g(n)$;
- distinguish the existential choice of witnesses from the universal claim over all sufficiently large n;
- explain why replacing implication by conjunction changes the meaning of the Big-O definition;
- apply the polynomial theorem to prove that a polynomial of degree d is in $O(n^d)$;

- compute C as the sum of absolute coefficient values for a polynomial running-time function;
- recognize when a function contains non-polynomial terms and verify whether a proposed n_0 actually works;
- calculate the size of a closed integer interval $[a, b]$ using $b - a + 1$;
- use the arithmetic-sequence sum formula to count loop executions;
- derive the $O(n^2)$ bound for the triangular-sum nested loop by counting (i, j) combinations;
- prepare to reuse this loop-counting method for selection sort and insertion sort examples.

Suggested Practice

- Review the posted discrete-math slides, focusing on implication, conjunction, universal quantification, existential quantification, and proof strategies.
- Rewrite the Big-O definition in predicate logic without looking, then compare your version against Jackie's notes.
- Answer Jackie's study question: replace the implication in the formal definition by conjunction and use truth tables to explain what breaks.
- Redo the polynomial example $5n^4 - 3n^3 + 2n^2 - 4n + 1$ and justify why $C = 15$ and $n_0 = 1$ are valid.
- Attempt the Big-O practice examples shown in the notes before checking the posted solution slide.
- For $f(n) = 3\log n + 2$, verify explicitly that $n_0 = 1$ fails and test the next threshold value.
- Practise counting intervals: compute the sizes of $[0, x]$, $[34, 100]$, and two examples of your own.
- Recreate the triangular-sum table for the nested-loop code and explain why the rows have sizes $n, n - 1, \ldots, 1$.
- Before the next lecture, make sure you can use this triangular-sum reasoning without relying on the completed notes, since Jackie said it will be used again for sorting algorithms.

Where to Review

- **iPad/PDF notes, page 1:** lecture overview: Big-O using predicates, theorem for proving asymptotic upper bounds, and examples for proving and deriving bounds.
- **iPad/PDF notes, page 2:** announcements and reminders: Tutorial 2 on Thursday and discrete-math review topics.
- **iPad/PDF notes, pages 3–5:** Big-O visualization, predicate-logic formulation, and the study question about replacing implication by conjunction.
- **iPad/PDF notes, pages 6–7:** polynomial proof theorem and the worked example with $5n^4 - 3n^3 + 2n^2 - 4n + 1$.
- **iPad/PDF notes, pages 8–12:** additional Big-O proof examples for independent practice, including logarithmic and exponential-looking functions.
- **iPad/PDF notes, pages 16–17:** interval-size and arithmetic-sequence formulas used for loop counting.
- **iPad/PDF notes, page 18:** the triangular-sum nested-loop analysis and the $O(n^2)$ conclusion.
- **Lecture recording:** review the first half for the formal Big-O and predicate-logic discussion; review the second half for interval counting, arithmetic sums, and deriving a bound from code.
- **Lecture site:** use the discrete-math review slides and the posted proof-strategy video under the exercises section when reviewing the theorem.

Closing Reminder

This lecture is a bridge between formal mathematical definitions and practical code analysis. Use the digest to decide what to revisit, but do the actual work by rewriting the definitions, checking the logic, solving the practice functions, and retracing the nested-loop example from the notes and recording. The key skill is not just naming $O(n^2)$ or $O(n^d)$, but justifying the claim clearly with witnesses, thresholds, and counted operations.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 4, May 12

Sorting Orders, Selection and Insertion Sort, and Big-O Properties

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 7

Suggested Practice 8

Where to Review 8

Closing Reminder 9

Main Ideas

- The lecture used selection sort and insertion sort as concrete quadratic algorithms for practising asymptotic upper-bound analysis.
- The sorting problem was made more precise by distinguishing strict orders, such as increasing and decreasing, from non-strict orders, such as non-descending and non-ascending.
- Selection sort was analyzed through its repeated pattern of selecting the minimum from a shrinking unsorted portion and swapping it into a growing sorted portion.
- Insertion sort was analyzed through its repeated pattern of getting the first unsorted element and inserting it into the sorted portion, with shifting as the worst-case cost.
- The main analysis pattern was the triangular sum: dependent nested work such as $n + (n-1) + \cdots + 1$ or $1 + 2 + \cdots + (n-1)$ gives a quadratic upper bound.
- The lecture ended by treating Big-O classes as families of functions and comparing them using subset and proper-subset relations.
- Jackie also previewed Tutorial 2: 2D-array exercises with JUnit and the start of amortized/average running-time analysis for dynamic arrays.

Roadmap

Announcements and preparation

Jackie began by pointing students back to the updated Lecture 4 notes template and the posted Lecture 3 recording. Tutorial 2 will take place on Thursday in the same main lecture hall, not the usual tutorial location. It will include 2D-array exercises with JUnit and the first part of the dynamic-array amortized/average running-time analysis. Students were also reminded to review the propositional-logic and predicate-logic slides, since Big-O properties will use set-theoretic and logical reasoning.

Sorting orders of arrays

The lecture first clarified the terminology around sorting orders. Increasing/ascending and decreasing/descending are strict orders: adjacent values must move strictly up or strictly down. The "non" versions are different because they allow equality. For example, non-descending means "not descending", so the adjacent condition becomes $A[i] \leq A[i+1]$. This distinction matters because a test or exam question may use any of the four terms.

Selection sort as a shrinking-unsorted-region algorithm

Jackie then reviewed selection sort conceptually before looking at code. The algorithm repeatedly selects the minimum from the current unsorted portion and appends it to the sorted portion. In an array implementation, the append is realized by swapping the selected minimum with the first element of the unsorted portion. The iPad trace showed the sorted portion growing from size 0 to size n, while the unsorted portion shrinks.

Selection-sort running time

The cost of selection sort was derived by separating the two main operations. Selecting the minimum costs $n, n-1, n-2, \ldots, 1$ across the passes, while each swap costs constant time. Thus the total work has the form

$$1 \cdot n + \big(n + (n-1) + (n-2) + \cdots + 1\big),$$

whose dominant term is quadratic. The important habit is not to memorize only "selection sort is $O(n^2)$", but to see how the shrinking search range creates an arithmetic sum.

Insertion sort and worst-case shifting

Insertion sort was contrasted with selection sort. Instead of selecting the minimum from the whole unsorted portion, insertion sort gets the first element of the unsorted portion and inserts it into the correct position in the sorted portion. The inexpensive part is the get operation; the expensive part is the insertion, because it may require shifting many elements. In the worst case, the new element is smaller than everything already sorted, so the whole sorted block must shift right.

Insertion-sort running time

Jackie summarized insertion sort with a table: the steps are indexed by the element being inserted, the sorted portion grows from size 1 to size $n-1$, and the worst-case insertion cost grows as $1, 2, 3, \ldots, n-1$. The resulting sum

$$1 + 2 + 3 + \cdots + (n-1)$$

again has a quadratic dominant term. A useful comparison is that selection sort is most expensive at the beginning, while insertion sort is most expensive near the end.

Deriving a bound from Java-like selection-sort code

After the conceptual analysis, Jackie returned to a Java implementation of selection sort. The outer loop runs over i from 0 to $n-2$, and the inner loop starts from $j = i$ and goes to $n-1$. Since the inner loop length depends on i, the work must be summed as

$$n + (n-1) + (n-2) + \cdots + 2,$$

not computed by blindly multiplying two loop counts. The code analysis confirms the same $O(n^2)$ bound as the conceptual selection-sort analysis.

Big-O families and proper subsets

The last part introduced a more formal view of Big-O classes. $O(g(n))$ is a family, or set, of functions whose growth can be upper-bounded by a constant multiple of $g(n)$ for sufficiently large n. Because every function in $O(n)$ is also in $O(n^2)$, but not every function in $O(n^2)$ is in $O(n)$, the more accurate relation is

$$O(n) \subset O(n^2),$$

where $\subset$ is the proper-subset relation. Jackie connected this to predicate-logic language: every member of the smaller family is a member of the larger family, and the larger family has at least one member outside the smaller family.

Key Terms, Notation, and Structures

- **Sorting problem**: Given a sequence, return a permutation of the same contents arranged according to a specified order.
- **Increasing/ascending**: A strict upward order, represented locally by $A[i] < A[i+1]$.
- **Decreasing/descending**: A strict downward order, represented locally by $A[i] > A[i+1]$.
- **Non-descending**: An order that is not descending; equality is allowed, so adjacent values satisfy $A[i] \leq A[i+1]$.
- **Non-ascending**: An order that is not ascending; equality is allowed, so adjacent values satisfy $A[i] \geq A[i+1]$.
- **Selection sort**: Repeatedly select the minimum from the unsorted portion and swap it into the next sorted position.
- **Insertion sort**: Repeatedly get the first unsorted element and insert it into the growing sorted portion.
- **Sorted and unsorted portions**: A useful array partition for tracing both algorithms; one region grows while the other shrinks.
- **Select**: In selection sort, finding the minimum of a range; the cost depends on the size of the range.
- **Swap**: Exchanging two array elements; treated as constant time in this analysis.
- **Get**: Accessing the next unsorted element in insertion sort; this is array indexing and is constant time.
- **Insert/shift**: Moving elements to make space for the item being inserted; in the worst case, shifting costs linear time in the size of the sorted portion.
- **Triangular sum**: A sum such as $n + (n-1) + \cdots + 1$ or $1 + 2 + \cdots + (n-1)$, which produces a quadratic dominant term.

- **Dependent nested loop**: A nested loop where the inner loop bound depends on the outer-loop counter, such as starting j at i.
- $O(g(n))$ **as a family**: The set of all functions that can be asymptotically upper-bounded by a constant multiple of $g(n)$.
- **Subset vs. proper subset**: $S \subseteq T$ means every member of S is also in T; $S \subset T$ means this is true and T also has at least one member not in S.

Key Examples and Demonstrations

Sorting-order definitions

Setup: The iPad notes showed four array-order diagrams: decreasing/descending, increasing/ascending, non-descending, and non-ascending.

Main point: The "non" in non-descending and non-ascending should be read as logical negation. For example, non-descending is not the same as strictly increasing; it allows equality.

What to practise: For each of the four order names, write the adjacent-element condition using $A[i]$ and $A[i+1]$, then create a short example array with and without duplicate values.

Selection sort trace

Setup: The trace began with the whole array treated as unsorted, then repeatedly selected the minimum from the unsorted portion and swapped it into the next sorted position.

Main point: Selection sort grows the sorted portion by one element per pass. The select operation becomes cheaper over time because the unsorted portion shrinks.

What to practise: Retrace selection sort on a four- or five-element array, explicitly marking the unsorted range, the selected minimum, and the indices being swapped at each pass.

Selection-sort running-time derivation

Setup: The conceptual trace was generalized from a length-4 array to an array of length n.

Main point: The selection costs form the arithmetic sequence $n+(n-1)+\cdots+1$, while the swaps contribute only a linear lower-order term. The asymptotic upper bound is therefore $O(n^2)$.

What to practise: Reproduce the summation and identify which term is dominant. Then explain why the constant-time swaps do not change the final Big-O class.

Insertion sort and worst-case shifting

Setup: The notes traced insertion sort on a small array and then abstracted the worst case: the selected unsorted element is smaller than all elements in the sorted portion.

Main point: The costliest operation is not getting the element; it is shifting the sorted block to make space. In the worst case, the insertion cost grows with the size of the sorted portion.

What to practise: Trace insertion sort on an array that is reverse-sorted for the desired order, and count how many shifts happen at each insertion step.

Selection-sort Java loop analysis

Setup: The Java-like selection-sort code used an outer loop over i and an inner loop beginning at $j = i$.

Main point: Because j starts at i, the number of inner-loop executions changes by row. The right model is a triangular sum, not a simple product of two loop counts.

What to practise: Draw a table of (i, j) combinations for a small value such as $n = 5$, count the entries row by row, and then generalize the pattern to $n + (n-1) + \cdots + 2$.

Big-O family diagram

Setup: The final pages treated $O(n)$ and $O(n^2)$ as nested families of functions.

Main point: Functions such as $\log n$, $\sqrt{n}$, and n belong to $O(n)$ and also to $O(n^2)$, while a function such as $n \log n$ belongs to $O(n^2)$ but not to $O(n)$. This justifies using the proper-subset relation $O(n) \subset O(n^2)$.

What to practise: Give one function that belongs to both families, and one function that belongs to $O(n^2)$ but not $O(n)$. Explain each choice in words.

Common Pitfalls and Misconceptions

Confusing non-descending with strictly increasing

Pitfall: Treating non-descending order as though every adjacent value must strictly increase.

Why it causes problems: This incorrectly rejects valid arrays with duplicates or flat segments, such as $1, 2, 2, 3, 4$.

How to avoid it: Translate the word "non" as logical negation: non-descending gives $\neg(A[i] > A[i+1])$, which is $A[i] \leq A[i+1]$.

Treating all operations in the sorting descriptions as equally costly

Pitfall: Thinking that selection, swapping, getting, inserting, and shifting all contribute the same kind of cost.

Why it causes problems: It hides the actual reason the algorithms are quadratic. In selection sort, selecting a minimum over a range is the expensive part; in insertion sort, worst-case shifting is the expensive part.

How to avoid it: Label each conceptual operation before summing costs: select is range-dependent, swap is constant, get is constant, and insert/shift can grow with the sorted portion.

Multiplying dependent nested-loop bounds directly

Pitfall: Seeing two loops and immediately multiplying two loop counts, even when the inner loop starts at $j = i$.

Why it causes problems: The inner loop does not run the same number of times for every i, so direct multiplication gives the wrong reasoning pattern.

How to avoid it: Make a row-by-row table of valid (i, j) combinations and sum the row lengths as an arithmetic sequence.

Remembering only the final $O(n^2)$ result

Pitfall: Memorizing that selection sort and insertion sort are both $O(n^2)$ without being able to derive the bound.

Why it causes problems: Later topics will require deriving running times from new algorithms, not only recognizing familiar names.

How to avoid it: Practise deriving the summation from either the conceptual algorithm description or the Java-like loop structure.

Thinking of Big-O classes as isolated labels

Pitfall: Treating $O(n)$ and $O(n^2)$ as if a function must belong to exactly one class.

Why it causes problems: Big-O is an upper-bound family. A function with a tighter bound also belongs to looser upper-bound families.

How to avoid it: Think in nested sets: $O(n)$ is properly contained in $O(n^2)$, so every $O(n)$ function is also an $O(n^2)$ function, but not conversely.

What You Should Be Able to Do

After this lecture, you should be able to:

- distinguish sorting orders — increasing, decreasing, non-descending, and non-ascending orders — using adjacent-element inequalities;
- explain selection sort using sorted and unsorted portions of an array;
- trace selection sort by identifying the selected minimum and the swap performed at each pass;
- derive the $O(n^2)$ upper bound for selection sort from $n + (n - 1) + \cdots + 1$ plus linear swap cost;
- explain insertion sort using the ideas of getting the next unsorted element, inserting it, and shifting elements when necessary;
- derive the worst-case $O(n^2)$ upper bound for insertion sort from $1 + 2 + \cdots + (n - 1)$;
- compare where the cost is concentrated in selection sort versus insertion sort;

- analyze a Java-like nested loop where the inner loop begins at $j = i$ by summing row lengths instead of multiplying blindly;
- explain why $O(g(n))$ is a family of functions rather than a single function;
- justify why $O(n) \subset O(n^2)$ is the more accurate relation between these two Big-O families.

Suggested Practice

- Redraw the four sorting-order diagrams and write the corresponding inequality for each one.
- Trace selection sort on at least two arrays: one with distinct values and one with duplicates.
- Implement selection sort in Java, paying special attention to how the index of the current minimum is tracked before the swap.
- Reproduce the selection-sort running-time derivation from the conceptual algorithm, then repeat it from the Java-like nested loop.
- Trace insertion sort on a reverse-sorted array and count the number of shifts at each insertion step.
- Complete the insertion-sort code-analysis exercise from the notes and confirm that the answer is quadratic.
- For $O(n)$ and $O(n^2)$, list several example functions and place them in the correct family diagram.
- Review the propositional-logic and predicate-logic slides, especially implication, universal quantification, existential quantification, subset definitions, and proof strategies.
- Before Tutorial 2, review the triangular-sum analysis and the 2D-array/JUnit material from Tutorial 1, since Thursday's tutorial will build on both programming and running-time analysis skills.

Where to Review

- **Lecture 4 iPad notes, page 1**: lecture title and main focus: selection sort, insertion sort, deriving asymptotic upper bounds, and Big-O properties.
- **Lecture 4 iPad notes, page 2**: reminders about Tutorial 2, 2D-array exercises with JUnit, dynamic-array amortized/average running-time analysis, and the discrete-math review slides.
- **Lecture 4 iPad notes, page 3**: four sorting-order diagrams and the logical derivation of non-descending order.

- **Lecture 4 iPad notes, page 4**: selection-sort trace, shrinking unsorted portion, swap-based append, and the $n + (n-1) + \cdots + 1$ summation.
- **Lecture 4 iPad notes, page 5**: insertion-sort trace, worst-case shifting, table of insertion costs, and the $1 + 2 + \cdots + (n-1)$ summation.
- **Lecture 4 iPad notes, page 6**: selection-sort Java code and the row-by-row (i, j) loop analysis.
- **Lecture 4 iPad notes, page 7**: insertion-sort Java code left as an exercise for deriving the asymptotic upper bound.
- **Lecture 4 iPad notes, pages 8–9**: Big-O families, examples of functions in $O(n)$ and $O(n^2)$, and the subset/proper-subset relation.
- **Lecture recording**: revisit the sorting-order definitions, the selection/insertion traces, the dependent-loop question, and the final Big-O family discussion.
- **Posted tracing solution and review slides**: use the tracing solution for sorting-code practice and the logic slides for the set-theoretic notation used in Big-O properties.

Closing Reminder

This digest should help you decide what to revisit, but it is not enough by itself. Rewatch the relevant parts of the lecture, compare them with Jackie's iPad notes, retrace the sorting examples, redo the summations, and practise deriving bounds from code. The goal is to be able to explain the reasoning without relying on the final answer being memorized.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 5, May 19

Tree Foundations: Depth, Height, and Binary-Tree Bounds

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 9

Closing Reminder 10

Main Ideas

- The lecture began the self-balancing binary-search-tree unit by rebuilding the tree terminology and counting facts needed before rotations.
- Jackie connected this unit to Assignment 1 and Programming Test 1: depth, height, rotations, and later tree implementation work are not just abstract definitions.
- A data structure was classified as linear or nonlinear using the uniqueness of predecessors and successors: arrays, singly linked lists, and doubly linked lists are linear, while trees and graphs are nonlinear.
- Tree terminology was anchored through a family-tree diagram: root, edge, path, ancestor path, node depth, levels, and tree height.
- A key warning was that height is measured in edges and depends on the chosen root or subtree; the same larger diagram can give different heights when rooted at different nodes.
- The lecture derived the geometric-series formula and then applied it to binary trees, where the maximum number of nodes at level ℓ is 2^ℓ.
- For a binary tree of height h, the lecture established the node-count bounds

$$h + 1 \leq n \leq 2^{h+1} - 1.$$

- These counting facts prepare for BST performance analysis, where the height of the tree determines whether operations behave like $O(\log n)$ or degrade toward $O(n)$.

Roadmap

Starting the self-balancing BST unit

Jackie opened by situating the new topic in the course plan. The next few lectures will build from tree foundations toward binary search trees, height-balance, and rotations. Assignment 1 will be a programming assignment tied to these ideas, and Programming Test 1 after the reading week will be related to the assignment material. Jackie also reminded students to finish reviewing the earlier asymptotic-analysis materials and the propositional/predicate-logic review, since some self-study content may matter later.

Linear vs. nonlinear data structures

The first technical page compared arrays, singly linked lists, doubly linked lists, trees, and graphs. The criterion was whether an element has a unique predecessor and a unique successor. Arrays, SLLs, and DLLs are linear because movement through the structure follows a single previous/next pattern, up to boundary cases. Trees are nonlinear because a node can have multiple children, and graphs are nonlinear because a node can have multiple incoming and outgoing edges.

General tree terminology

The lecture then shifted to a family-tree diagram rooted at David. Jackie used it to review edges, parent-child links, paths, and ancestor paths. For example, the ancestor path of Vanessa goes from Vanessa to Elsa, then Chris, then David. The sequence includes the starting node, while the path length is counted by the number of edges between consecutive nodes.

Depth, levels, and height

Depth was defined as the number of parent links from a node to the root. In the David-rooted tree, David has depth 0, children such as Ernesto and Chris have depth 1, nodes such as Elsa and Anna have depth 2, and the deepest displayed nodes have depth 3. Nodes with the same depth form a level, so levels are labelled $0, 1, 2, \ldots$. Height of a tree is the maximum depth among the descendants of the chosen root.

Why the chosen root matters

Jackie emphasized that height is not an absolute property of a node floating in isolation; it is computed after fixing a root and considering the corresponding tree or subtree. In the notes, the tree rooted at David has height 3, while the subtree rooted at Elsa has height 1. Once a subtree root is chosen, the rest of the larger tree outside that subtree is ignored.

Reviewing geometric sequences

Before applying counting formulas to binary trees, Jackie reviewed geometric sequences. A sequence such as $3+6+12+24$ can be rewritten as $3\cdot 2^0+3\cdot 2^1+3\cdot 2^2+3\cdot 2^3$, revealing an initial term, a common factor, and a number of terms. The general sum was derived by writing S_k, multiplying by the common factor r, subtracting the two aligned equations, and solving:

$$S_k = \frac{I\cdot(r^k-1)}{r-1}.$$

Jackie noted that students should either memorize this formula or be able to re-derive it quickly for closed-book assessments.

Applying the geometric sum to binary trees

For a binary tree, each level can have at most twice as many nodes as the previous level. Thus level 0 has at most 2^0 node, level 1 has at most 2^1 nodes, level 2 has at most 2^2 nodes, and level h has at most 2^h nodes. Because levels are labelled from 0 to h, there are $h+1$ terms in the sum. Substituting $I = 1$, $r = 2$, and $k = h+1$ gives the maximum total node count

$$2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1.$$

Bounding the number of nodes from the height

The final technical development stated the first binary-tree bound. Given height h, the smallest possible binary tree with that height is a chain with one node per level, so it has $h+1$ nodes. The largest possible binary tree with that height fills every available position at every level, so it has $2^{h+1}-1$ nodes. For the example $h=3$, the minimum is 4 nodes and the maximum is 15 nodes.

Why these facts matter for BSTs

Jackie previewed the next step: bounds involving n and h matter because the running time of searching, insertion, deletion, and rotation-related work in a BST is governed by height. A balanced tree keeps height logarithmic in the number of nodes, while a poorly shaped tree can have linear height. This is the motivation for studying height-balance and rotations next.

Key Terms, Notation, and Structures

- **Linear data structure**: A structure where each position has at most one predecessor and at most one successor, up to boundary cases.
- **Nonlinear data structure**: A structure where nodes may have multiple predecessors or multiple successors; trees and graphs are the key examples in this lecture.
- **Array**: A linear structure where an element at index i has predecessor $i-1$ and successor $i+1$ when those indices exist.
- **Singly linked list**: A linear linked structure with a unique next reference from each non-tail node.
- **Doubly linked list**: A linear linked structure with previous and next references, giving a unique predecessor and successor for interior nodes.
- **Tree**: A nonlinear hierarchical structure; a node has at most one parent but may have multiple children.
- **Graph**: A nonlinear structure that may have multiple incoming and outgoing edges at a node.
- **Root**: The node from which the tree or subtree is considered to begin.
- **Edge**: A parent-child link between two nodes in a tree.
- **Path**: A sequence of connected nodes; its length is counted by the number of edges, not the number of nodes.
- **Ancestor path**: The path from a node up through parent links to the root, including the starting node.

- **Depth of a node**: The number of edges from that node to the root.
- **Level**: The collection of nodes with the same depth; level labels begin at 0.
- **Height of a tree or subtree**: The maximum depth among descendants of the chosen root; height is measured in edges.
- **Subtree**: The tree obtained by choosing a node as root and considering its descendants while ignoring the rest of the larger tree.
- **Geometric sequence**: A sequence where consecutive terms differ by a common factor r.
- S_k: The sum of k geometric terms, written in this lecture as $S_k = \frac{I \cdot (r^k - 1)}{r - 1}$.
- **Binary tree level count**: Level ℓ of a binary tree has at most 2^ℓ nodes.
- **Maximum binary-tree node count**: A binary tree of height h has at most $2^{h+1} - 1$ nodes.
- **Node-count bounds for height** h: For a binary tree of height h, $h + 1 \leq n \leq 2^{h+1} - 1$.
- **Degenerate tree**: A chain-like tree with one node per level; it gives the minimum node count for a fixed height.
- **Full binary tree**: In this lecture's counting context, a tree where every possible node position up to height h is filled.
- **Tree traversals**: Preorder, inorder, and postorder traversals were recalled as upcoming recursive tree-processing patterns.

Key Examples and Demonstrations

Classifying structures as linear or nonlinear

Setup: The notes showed an array, a singly linked list, a doubly linked list, a tree, and a graph.

Main point: Arrays, SLLs, and DLLs satisfy the unique-predecessor/unique-successor idea, while trees and graphs fail it because branching creates multiple successors or multiple incoming/outgoing relationships.

What to practise: For each structure, choose one interior element or node and identify its predecessor(s) and successor(s). Then explain why boundary cases such as the first array element do not break the linear classification.

Ancestor path of Vanessa

Setup: In the family-tree diagram rooted at David, Vanessa appears below Elsa, who appears below Chris, who appears below David.

Main point: The ancestor path includes the starting node and then follows parent links to the root: Vanessa, Elsa, Chris, David. The path length is 3 because there are 3 edges.

What to practise: Pick two other nodes in the same diagram and write their ancestor paths. Count the number of edges and compare that count with the node's depth.

Depth, levels, and height in the David tree

Setup: The notes labelled the David-rooted tree by depths and levels: David at depth 0, his children at depth 1, deeper nodes at depths 2 and 3.

Main point: Depth is a node property, levels group equal depths, and height is the maximum depth in the chosen tree. The David-rooted tree has height 3 and therefore levels $0, 1, 2, 3$.

What to practise: Redraw the diagram without the notes, label every node by depth, then write the level sets and identify the height.

Different roots give different heights

Setup: The notes compared the tree rooted at David with the subtree rooted at Elsa.

Main point: The David-rooted tree has height 3, while the Elsa-rooted subtree has height 1. This illustrates why the root must be fixed before height is computed.

What to practise: Repeat the calculation using Chris as the root. Ignore nodes outside Chris's subtree and count the longest descendant path from Chris.

Deriving the geometric-series formula

Setup: Jackie wrote a geometric sequence using initial term I, common factor r, and k terms, then aligned S_k with $r \cdot S_k$.

Main point: Multiplying by r shifts the terms so that most terms cancel when subtracting. The resulting formula is

$$S_k = \frac{I \cdot (r^k - 1)}{r - 1}.$$

What to practise: Re-derive the formula from a blank page. Pay special attention to why the last exponent is $k - 1$ in S_k and why multiplying by r creates a final r^k term.

Maximum nodes in a binary tree

Setup: The binary-tree page built the tree level by level and recorded maximum nodes at each level.

Main point: The level counts form the geometric sequence $2^0, 2^1, 2^2, \ldots, 2^h$. Because the labels run from 0 to h, the number of terms is $h + 1$, and the total maximum is $2^{h+1} - 1$.

What to practise: For $h = 0, 1, 2, 3$, draw the fullest possible binary tree, count the nodes by hand, and verify the formula.

Bounds for a binary tree of height 3

Setup: The final page compared two binary trees with height 3: a chain and a fully filled tree.

Main point: The chain gives the minimum $h + 1 = 4$ nodes, while the full binary tree gives the maximum $2^4 - 1 = 15$ nodes.

What to practise: Generalize the two pictures: draw the chain case for height h, then draw the full case symbolically and explain where each side of $h + 1 \leq n \leq 2^{h+1} - 1$ comes from.

Common Pitfalls and Misconceptions

Treating a boundary case as a violation of linearity

Pitfall: Thinking that an array is not linear because the first element has no predecessor or the last element has no successor.

Why it causes problems: The criterion is uniqueness, not the existence of both neighbours at every boundary. A missing predecessor is not the same as multiple predecessors.

How to avoid it: Check whether an element can have more than one predecessor or successor. Boundary positions are allowed to have none on one side.

Computing height without fixing the root

Pitfall: Looking at a node inside a larger tree and trying to assign one absolute height without specifying the tree or subtree root.

Why it causes problems: Height depends on which root has been chosen. The David-rooted tree and the Elsa-rooted subtree in the notes have different heights.

How to avoid it: Begin every height calculation by saying "the tree rooted at . . . ". Then ignore anything outside that subtree.

Counting nodes instead of edges for depth and height

Pitfall: Reporting the number of nodes in a path instead of the number of edges.

Why it causes problems: Depth and height in this lecture are measured in edges. For example, a root has depth 0, not depth 1.

How to avoid it: Count links, not boxes. If a path lists m nodes, it has $m - 1$ edges.

Confusing the last level label with the number of levels

Pitfall: Seeing a tree of height h and assuming it has h levels.

Why it causes problems: Levels are labelled $0, 1, \ldots, h$, so there are $h + 1$ levels. This off-by-one error changes the geometric-series exponent.

How to avoid it: Write the interval of level labels explicitly as $[0, h]$, then count its size as $h + 1$.

Using the wrong parameters in the geometric sum

Pitfall: Plugging arbitrary values into $S_k = \frac{I \cdot (r^k - 1)}{r - 1}$ without identifying I, r, and k from the structure being counted.

Why it causes problems: For binary-tree maximum counts, the correct values are $I = 1$, $r = 2$, and $k = h + 1$. A wrong k gives a wrong total.

How to avoid it: Before applying the formula, write the actual sequence: $2^0 + 2^1 + \cdots + 2^h$.

Confusing per-level count with cumulative count

Pitfall: Treating 2^h as the total number of nodes up to height h.

Why it causes problems: 2^h is only the maximum number of nodes at the bottom level. The total is the sum of all levels.

How to avoid it: Distinguish "at level h" from "in the whole tree up to level h". Use $2^{h+1} - 1$ for the full cumulative maximum.

Confusing BST search with array binary search

Pitfall: Assuming that every search in a binary search tree is automatically $O(\log n)$ because binary search on a sorted array is $O(\log n)$.

Why it causes problems: BST search depends on the tree height. A balanced tree can be logarithmic, but a degenerate tree can behave linearly.

How to avoid it: Tie BST running time to height first. Then ask whether the height is logarithmic or linear in the number of nodes.

What You Should Be Able to Do

After this lecture, you should be able to:

- distinguish linear and nonlinear data structures using the unique-predecessor and unique-successor criteria;
- explain why arrays, singly linked lists, and doubly linked lists are linear, while trees and graphs are nonlinear;
- identify roots, edges, parent-child links, paths, and ancestor paths in a tree diagram;
- trace an ancestor path from a node to the root and compute its length in edges;
- compute the depth of nodes and group nodes into levels starting from level 0;
- compute the height of a tree or subtree after first specifying the chosen root;
- explain why a tree of height h has levels labelled 0 through h, but has $h + 1$ levels;
- derive or recall the geometric-series formula $S_k = \frac{I \cdot (r^k - 1)}{r - 1}$;
- apply the geometric-series formula to binary-tree level counts using $I = 1$, $r = 2$, and $k = h + 1$;
- justify why level ℓ of a binary tree has at most 2^ℓ nodes;
- derive the maximum node count $2^{h+1} - 1$ for a binary tree of height h;

- justify the bounds $h + 1 \leq n \leq 2^{h+1} - 1$ for a binary tree of height h;
- explain why tree height will control the running time of BST operations;
- prepare for the next topics: traversals, BST search properties, height balance, and rotations.

Suggested Practice

- Redraw the linear-vs.-nonlinear structures page and, for each structure, mark one predecessor and one successor relationship.
- Use the family-tree diagram to compute at least three ancestor paths, including their node sequences and edge counts.
- Label every node in the David-rooted tree with its depth, then write the level sets explicitly.
- Recompute the height of the tree when rooted at David, Elsa, and Chris; explain why the answers differ.
- Re-derive the geometric-series formula without looking at the notes, aiming to finish the derivation in one or two minutes.
- For $h = 0, 1, 2, 3, 4$, compute the minimum and maximum number of nodes in a binary tree of height h and verify the pattern.
- Draw a full binary tree of height 3 and check that the per-level counts are $1, 2, 4, 8$ and the total is 15.
- Prepare for the next lecture by reviewing preorder, inorder, and postorder traversals on a small binary tree.
- Keep reviewing the asymptotic-analysis slides and the logic review slides, especially the self-study parts Jackie identified as still relevant.
- Before Assignment 1 is released, practise small recursive Java methods for tree-like structures, especially methods that compute depth, height, or traversal order.

Where to Review

- **iPad/PDF notes, page 1**: Lecture topic list for the self-balanced BST unit, including linear vs. nonlinear structures, depth vs. height, binary-tree node bounds, and tree traversals.
- **iPad/PDF notes, page 2**: Announcements about Tutorial 2 materials, dynamic-array amortized analysis, and the logic review slides.

- **iPad/PDF notes, page 3**: The linear-vs.-nonlinear structures comparison for arrays, SLLs, DLLs, trees, and graphs.
- **iPad/PDF notes, pages 4–6**: The general-tree terminology diagrams, including ancestor paths, depth, levels, height, and the effect of choosing different roots.
- **iPad/PDF notes, page 7**: The geometric-sequence derivation using S_k and $r \cdot S_k$.
- **iPad/PDF notes, page 8**: Binary-tree levels and maximum node counts per level, leading to $2^{h+1} - 1$.
- **iPad/PDF notes, pages 9–10**: The two binary-tree bounding questions and the completed bound on the number of nodes given height h.
- **Lecture recording**: Rewatch the parts on height, the h vs. $h + 1$ clarification, and the geometric-sum derivation if the formulas feel like memorized facts rather than reasoning you can reproduce.
- **Course site**: Review the earlier asymptotic-analysis slides and the propositional/predicate-logic review slides Jackie flagged for self-study.
- **Upcoming tutorial preparation**: Revisit Tutorial 2's dynamic-array analysis before the doubling-strategy follow-up, since Jackie said the proof pattern will be similar.

Closing Reminder

This digest is a roadmap for the lecture, not a substitute for the recording, slides, iPad notes, or your own practice. The most important active work after this lecture is to redraw the diagrams, recompute depths and heights from chosen roots, re-derive the geometric sum, and use the formulas on small binary trees until the off-by-one details are automatic. These foundations will be used again when Jackie moves into traversals, BST operations, height balance, rotations, Assignment 1, and Programming Test 1 preparation.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 6, May 21

BST Height Bounds, Traversal Properties, Search, and Insertion

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 10

Suggested Practice 11

Where to Review 11

Closing Reminder 12

Main Ideas

- The lecture continued the self-balancing BST unit by completing the height-vs.-node-count view needed to reason about the running time of tree operations.
- For a binary tree with n nodes, Jackie emphasized the height bounds

$$\log_2(n+1) - 1 \leq h \leq n - 1,$$

  and connected the two extremes to balanced and slanted tree shapes.
- Tree traversal was reviewed as a systematic way to linearize a tree: preorder visits a parent before its children, postorder visits a parent after its children, and inorder is specific to binary trees.
- The BST search property was stated recursively: every key in the left subtree is strictly smaller than the node's key, and every key in the right subtree is strictly larger.
- The BST sorting property follows from inorder traversal: inorder traversal of a BST produces a key sequence sorted in ascending order.
- Predecessor and successor were introduced through the sorted inorder sequence and then related back to their positions in the tree.
- Searching a BST was demonstrated in both successful and unsuccessful cases; an unsuccessful search ends at an external node.
- BST insertion was framed as "search first": if the key already exists, update the value; if the search reaches an external node, convert that external node into a new internal node.
- Assignment 1 and Programming Test 1 were repeatedly connected to these ideas: students will need the tree concepts, Java recursion/background study, and BST implementation skills.

Roadmap

Tutorial, assignment, and preparation plan

Jackie began with logistics for the next tutorial and Assignment 1. Tutorial 3 will resume at the usual Monday time in CLHE and will cover the doubling strategy for dynamic arrays plus hands-on generic BST exercises using JUnit. Jackie also said Assignment 1 may be released earlier than originally planned, while keeping the same deadline, so students can use the extra time for background study on Java, recursion, linked lists, and trees before Programming Test 1 after the reading week.

Bounding height from the number of nodes

The lecture first completed the second binary-tree counting property. Lecture 5 bounded n given height h; this lecture reversed the viewpoint and bounded h given n. The key result was

$$\log_2(n+1) - 1 \leq h \leq n - 1.$$

The lower bound corresponds to the most balanced, filled-out shape; the upper bound corresponds to a slanted, chain-like tree.

Why height controls running time

Jackie tied the height bounds directly to operations such as search, insertion, deletion, and later rotations. If a tree is balanced, height is about $\log n$, so height-dependent operations can be logarithmic. If the tree degenerates into a chain, height is about n, so the same operations can become linear. This is the performance reason for studying self-balancing BSTs.

The $n = 7$ example and the height formula

For $n = 7$, the notes showed a minimum-height tree of height 2 and a maximum-height slanted tree of height 6. Jackie derived the lower-bound formula by starting from the full-binary-tree node count

$$n = 2^{h+1} - 1$$

then rearranging it to solve for h:

$$h = \log_2(n+1) - 1.$$

The example also included the practical note that when the logarithm does not produce an integer, one must use the appropriate rounded height; asymptotically, the important growth is $O(\log n)$.

Preorder and postorder traversals on a general tree

The next review topic was tree traversal. Jackie asked students to state the difference between preorder and postorder. In preorder, the parent is visited before recursively visiting the children from left to right. In postorder, the children are recursively visited first, and the parent is visited after them. The David-rooted family tree was used again to produce and analyze the two sequences.

Binary-tree traversal exercise

The notes then showed a binary expression tree and listed preorder, inorder, and postorder as the three traversal targets. Jackie left this as an exercise and noted that inorder traversal only makes sense for binary trees, not for an arbitrary general tree. The point was to practise recursive traversal patterns before using them in BSTs and later graph traversal ideas.

BST search property

The lecture then moved from binary trees to binary search trees. Jackie defined each BST node as storing an entry, consisting of a key and a value. The search property concerns the key: for every node and every subtree, keys in the left subtree must be strictly smaller than the node's key, and keys in the right subtree must be strictly larger. Jackie corrected the possible "less than or equal" wording and emphasized that duplicate keys are not allowed.

BST sorting property through inorder traversal

Using the sample BST in the notes, Jackie showed that an inorder traversal linearizes the nonlinear BST into the sorted sequence

$$8, 17, 21, 28, 29, 32, 44, 54, 65, 76, 80, 82, 88, 93, 97.$$

The root key behaves like a pivot: the inorder traversal of the left subtree appears before the root key, and the inorder traversal of the right subtree appears after it. Jackie compared this structure to the idea behind quicksort, where values are split around a pivot.

Predecessor, successor, and valid replacement ranges

The sorted inorder sequence was then used to identify predecessor and successor. For example, the predecessor of 76 is 65 and its successor is 80, so a replacement key for 76 must be strictly between them. A second exercise used key 65, whose predecessor is 54 and successor is 76. These examples prepare for BST deletion, where replacement often uses either the predecessor or successor.

A visual summary of inorder traversal in a BST

Jackie summarized the relative order around a parent node p: the rightmost node of the left subtree gives the largest key smaller than p, and the leftmost node of the right subtree gives the smallest key larger than p. In other words, the predecessor of p is the maximum key in the left subtree, and the successor of p is the minimum key in the right subtree.

BST search and external nodes

The search operation was demonstrated with two cases. Searching for 65 succeeds and returns the internal node with matching key. Searching for 68 follows the BST comparisons until it reaches an external node, so the search is unsuccessful and returns that external node. Jackie explained that external nodes are special sentinel nodes: their stored key/value fields are not meaningful for the search or sorting properties.

BST insertion as search plus update or conversion

Finally, Jackie visualized insertion. To insert entry $(28,$"suyeon"$)$, the search finds an existing internal node with key 28, so insertion replaces the stored value. To insert entry

$(68,$ "yuna"), the search reaches the external node where 68 belongs; insertion then converts that external node into an internal node storing the new entry and attaches new external children. Jackie warned that parent and child links must be maintained correctly in the Java implementation.

Key Terms, Notation, and Structures

- **Binary tree (BT)**: A tree in which each node has at most two children; the lecture continued using BTs before adding search-specific constraints.
- **Number of nodes** n: The total number of nodes considered in the tree.
- **Height** h: The maximum number of edges from the chosen root to a deepest node.
- **Height bounds from** n: For a binary tree with n nodes, $\log_2(n+1) - 1 \leq h \leq n - 1$.
- **Balanced tree shape**: The shape that keeps height small by filling levels as much as possible; it gives logarithmic height.
- **Slanted or degenerate tree**: A chain-like shape whose height is $n - 1$, producing linear height.
- **Traversal**: A systematic method for visiting every node in a tree.
- **Preorder traversal**: Visit the parent first, then recursively visit children from left to right.
- **Postorder traversal**: Recursively visit children first, then visit the parent.
- **Inorder traversal**: For a binary tree, recursively visit the left subtree, then the parent, then the right subtree.
- **Binary search tree (BST)**: A binary tree satisfying a recursive key-ordering property at every subtree.
- **Entry**: The key-value pair stored at a BST node; searching and ordering are based on the key, not the value.
- **Key**: The part of an entry used for comparisons, searching, and maintaining the BST property.
- **Value**: The associated data stored with a key; if an insertion uses an existing key, the value is replaced.
- **Search property**: For every node, keys in the left subtree are strictly smaller than the node's key, and keys in the right subtree are strictly larger.
- **Sorting property**: Inorder traversal of a BST gives the keys in sorted ascending order.

- **Left subtree (LST) and right subtree (RST)**: The subtrees rooted at the left and right children of a node; in a BST they appear before and after the parent key in the inorder sequence.
- **Predecessor**: The key immediately before a given key in the sorted inorder sequence.
- **Successor**: The key immediately after a given key in the sorted inorder sequence.
- **Minimum key**: The leftmost meaningful internal node in a BST.
- **Maximum key**: The rightmost meaningful internal node in a BST.
- **Internal node**: A meaningful BST node storing an entry.
- **External node**: A sentinel leaf node shown as a square in the notes; its key/value fields are not meaningful for the BST's sorted key sequence.
- **Successful search**: A search that returns an internal node whose key matches the target.
- **Unsuccessful search**: A search that ends at and returns an external node where the missing key would belong.
- **Insertion**: A BST operation that first searches; it either updates an existing value or converts the returned external node into a new internal node.

Key Examples and Demonstrations

Bounding the height when $n = 7$

Setup: The notes compared two binary trees with 7 nodes: a filled, balanced tree and a slanted chain.

Main point: The balanced shape gives minimum height $\log_2(7 + 1) - 1 = 2$, while the chain gives maximum height $7 - 1 = 6$. These are the two extremes behind $O(\log n)$ and $O(n)$ height-dependent operations.

What to practise: Draw the two shapes for $n = 7$ without looking at the notes. Then repeat for a different small n and identify which shape is trying to minimize or maximize height.

Deriving the logarithmic lower bound on height

Setup: Jackie reused the maximum-node-count formula for a full binary tree, $n = 2^{h+1} - 1$.

Main point: Rearranging gives $h = \log_2(n + 1) - 1$. This explains why a well-filled binary tree has logarithmic height in the number of nodes.

What to practise: Starting from $n = 2^{h+1} - 1$, solve for h on paper. Then explain why this derivation corresponds to the minimum possible height for a fixed n.

Preorder and postorder on the David tree

Setup: The general tree rooted at David was used to compare preorder and postorder traversals.

Main point: Preorder places the root of a subtree before its descendants, while postorder places it after its descendants. In the David example, David appears first in preorder and last in postorder.

What to practise: Trace both traversals on the same tree and mark which recursive subtree each block of the sequence belongs to. Do not just memorize the final list.

Binary expression tree traversal exercise

Setup: The notes showed a binary expression tree and left space for preorder, inorder, and postorder traversal results.

Main point: Binary trees support all three traversal orders. The exercise is useful for practising recursive structure before coding tree methods.

What to practise: Complete all three traversal sequences yourself. Then explain how each result changes depending on when the parent operator is visited relative to its operands.

Inorder traversal linearizes a BST

Setup: The sample BST rooted at 44 was linearized by inorder traversal.

Main point: The nonlinear tree becomes the sorted key sequence

$$8, 17, 21, 28, 29, 32, 44, 54, 65, 76, 80, 82, 88, 93, 97.$$

The keys from the left subtree of 44 appear before 44, and the keys from the right subtree appear after 44.

What to practise: Redraw the BST and divide the inorder sequence into the left-subtree part, root key, and right-subtree part. Then repeat the same subdivision for the subtree rooted at 88.

Replacement range for key 76

Setup: In the sorted inorder sequence, 65 comes immediately before 76 and 80 comes immediately after it.

Main point: To replace 76 without breaking the BST property, the replacement key must be strictly between its predecessor and successor. In the integer examples, this gives possible keys 66 through 79.

What to practise: Locate 76 in both the tree and the inorder sequence. Explain why a visually nearby tree node is not enough; the predecessor and successor are defined by the inorder order.

Replacement range for key 65

Setup: The second exercise used key 65 in the same BST.

Main point: The predecessor is 54 and the successor is 76, so valid integer replacements are 55 through 75. This example is useful because one of the neighbors is easier to see in the tree than the other.

What to practise: Pick two other internal nodes from the BST and find their predecessor and successor from the sorted sequence. Then check whether your answer matches the tree structure.

Predecessor and successor around a parent node

Setup: Jackie summarized a generic BST node p with a left subtree and a right subtree.

Main point: The predecessor of p is the rightmost internal node in the left subtree, and the successor of p is the leftmost internal node in the right subtree. Equivalently, they are the largest key less than p and the smallest key greater than p.

What to practise: For several nodes in the sample BST, find the predecessor by moving to the left subtree and then as far right as possible; find the successor by moving to the right subtree and then as far left as possible.

Searching for 65 and 68

Setup: The notes compared a successful search for 65 with an unsuccessful search for 68.

Main point: Searching for 65 returns the matching internal node. Searching for 68 follows comparisons until it reaches the external node under the path where 68 would belong.

What to practise: Trace both searches step by step, writing each comparison and whether it sends the search left or right. Stop only when you reach either a matching internal node or an external node.

Insertion cases: update value or convert external node

Setup: The notes used insertion of $(28,$ "suyeon") and $(68,$ "yuna").

Main point: If the key already exists, insertion replaces the value associated with that key. If the key is absent, the unsuccessful search returns an external node; insertion converts that external node into an internal node storing the new entry and gives it new external children.

What to practise: Simulate both insertion cases by hand. For the absent-key case, explicitly show the parent link that must be preserved and the two new external children that must be attached.

Common Pitfalls and Misconceptions

Treating a BST as automatically balanced

Pitfall: Assuming that every BST has height $O(\log n)$.

Why it causes problems: The search property alone does not prevent a tree from becoming slanted. A slanted BST can have height $n-1$, making search, insertion, and deletion linear.

How to avoid it: Separate the BST search property from the height-balance property. When analyzing running time, first ask what the height can be.

Forgetting strict inequalities in the search property

Pitfall: Writing "less than or equal to" or allowing duplicate keys in the left or right subtree.

Why it causes problems: The lecture's BST convention uses unique keys. Allowing equality breaks the intended search-property definition and makes update-vs.-insert behavior ambiguous.

How to avoid it: State the property with strict comparisons: left-subtree keys are $<$ the parent key and right-subtree keys are $>$ the parent key.

Checking only immediate children

Pitfall: Verifying only that a left child is smaller and a right child is larger, while ignoring deeper nodes.

Why it causes problems: The search property is recursive and must hold for every subtree, not just for one parent-child pair.

How to avoid it: For any node, reason about the entire left subtree and the entire right subtree. Then repeat the check recursively for subtrees.

Confusing the tree picture with the inorder sequence

Pitfall: Trying to find predecessor or successor only by visual closeness in the drawing.

Why it causes problems: The predecessor and successor are defined by the sorted inorder sequence. A node that looks nearby in the picture may not be the immediate sorted neighbor.

How to avoid it: Use inorder traversal or the left-subtree/right-subtree rule: predecessor is the maximum of the left subtree; successor is the minimum of the right subtree.

Including external nodes as meaningful keys

Pitfall: Treating square external nodes as part of the sorted sequence or as meaningful key-value entries.

Why it causes problems: External nodes are sentinel leaves used by the implementation convention. Their fields may contain placeholder values, but those values do not participate in the BST search or sorting property.

How to avoid it: When tracing searches or traversals for BST keys, include only internal nodes as meaningful entries.

Stopping an unsuccessful search too early

Pitfall: Concluding that a key is absent before following comparisons all the way to the external node where it would belong.

Why it causes problems: That final external node is exactly what insertion needs. If you stop early, you lose the location where the new entry should be placed.

How to avoid it: Continue left/right comparisons until either the key matches an internal node or the search reaches an external node.

Breaking links during insertion

Pitfall: When converting an external node into an internal node, losing the existing parent link or failing to attach new external children.

Why it causes problems: BST implementation depends on correct parent and child references. Broken links will cause later searches, traversals, insertions, or rotations to fail.

How to avoid it: In the absent-key insertion case, think of the operation as converting the returned external node into an internal node while preserving its parent relationship and creating its two external children.

What You Should Be Able to Do

After this lecture, you should be able to:

- State and explain the height bounds $\log_2(n+1) - 1 \leq h \leq n - 1$ for a binary tree with n nodes.
- Draw minimum-height and maximum-height binary trees for a small fixed number of nodes.
- Explain why balanced tree shapes lead to $O(\log n)$ height while slanted tree shapes lead to $O(n)$ height.
- Re-derive $h = \log_2(n+1) - 1$ from $n = 2^{h+1} - 1$.
- Distinguish preorder, postorder, and inorder traversals by when the parent is visited.
- Trace preorder and postorder traversals on a general tree.
- Trace preorder, inorder, and postorder traversals on a binary tree.
- Define the BST search property using strict inequalities and the full left and right subtrees.
- Explain why duplicate keys are not allowed under the BST convention used in lecture.
- Use inorder traversal to produce the sorted sequence of keys from a BST.
- Identify predecessor and successor from both an inorder sequence and the tree structure.
- Determine a valid key range for replacing a node's key without breaking the BST property.
- Trace a successful BST search that returns an internal node.
- Trace an unsuccessful BST search that returns the external node where the missing key belongs.

- Explain the two insertion cases: updating the value of an existing key, or converting an external node into a new internal node.
- Describe the implementation role of external nodes and why their key/value fields are not meaningful for the BST's logical contents.
- Prepare for Tutorial 3 by reviewing dynamic-array doubling and the Java/JUnit background needed for generic BST exercises.

Suggested Practice

- Redraw the $n = 7$ minimum-height and maximum-height binary trees and annotate their heights.
- Starting from $n = 2^{h+1} - 1$, solve for h without notes; then explain what tree shape makes this bound relevant.
- Complete the binary expression tree traversal exercise from the notes: preorder, inorder, and postorder.
- On the David tree, trace preorder and postorder again and mark where each recursive subtree contributes to the final sequence.
- For the sample BST, redo the inorder traversal and verify that the sequence is sorted.
- For at least five keys in the sample BST, identify predecessor and successor using both the inorder sequence and the tree rule.
- Create replacement-range questions like the ones for 76 and 65, then check whether the proposed replacement would preserve the BST property.
- Trace searches for keys that exist and keys that do not exist; for absent keys, identify the exact external node returned.
- Simulate insertion for one existing key and one absent key, showing whether the value is replaced or an external node is converted.
- Before Tutorial 3, review the Tutorial 2 dynamic-array analysis and come ready to compare fixed increments with doubling.
- Use Jackie's planned Assignment 1 background materials, when posted, to refresh Java recursion, linked structures, and tree implementation patterns.

Where to Review

- **iPad/PDF page 1**: Lecture topic overview for Self-Balanced BSTs, with emphasis on BST search/sorting properties, searching, and insertion.

- **iPad/PDF page 2**: Tutorial 3 reminders: dynamic arrays Part 2 and generic BST exercises with JUnit.
- **iPad/PDF page 3**: Height bounds for a binary tree with n nodes, including the $n = 7$ minimum-height and maximum-height examples.
- **iPad/PDF page 4**: Preorder and postorder traversal on the David-rooted general tree.
- **iPad/PDF page 5**: Binary expression tree traversal exercise for preorder, inorder, and postorder.
- **iPad/PDF pages 6–7**: BST search and sorting properties, including the sample BST and its inorder sorted sequence.
- **iPad/PDF pages 8–9**: Replacement-range exercises for keys 76 and 65, using predecessor and successor.
- **iPad/PDF page 10**: Visual summary of inorder traversal around a parent node, including min, max, predecessor, and successor positions.
- **iPad/PDF page 11**: Successful and unsuccessful BST search examples using keys 65 and 68.
- **iPad/PDF page 12**: BST insertion examples for an existing key and an absent key.
- **Lecture recording**: Rewatch the segments on height bounds, traversal definitions, the search/sorting property, predecessor/successor, and external-node insertion if any step feels mechanical rather than understood.
- **Upcoming materials**: Watch for Jackie's Assignment 1 background-study section and use it to review Java recursion, linked structures, and tree coding before the programming test preparation becomes urgent.

Closing Reminder

This digest is a roadmap for review, not a substitute for the lecture, the iPad/PDF notes, the recording, or hands-on practice. The most important work after this lecture is active: redraw the trees, trace the traversals, follow BST searches by comparisons, identify predecessor and successor from the inorder sequence, and simulate insertion carefully enough that the external-node convention and parent/child links make sense before you implement them in Java.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 7, May 26

RT of BST Search, Height Balance Property, and Left Rotation after Insertion

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 11

Closing Reminder 11

Main Ideas

- The lecture connected BST search to tree shape: search is $O(h)$, so a balanced tree can give $O(\log n)$ behaviour while an ill-balanced tree can degrade to $O(n)$.
- Jackie reviewed successful and unsuccessful BST search, emphasizing that an unsuccessful search returns the specific external node where the missing key belongs.
- The contrast between a balanced BST and a slanted BST showed why binary search on a sorted array is not automatically the same as search in an arbitrary BST.
- A sorted or otherwise unlucky insertion order can create a BST whose height is linear in the number of internal nodes.
- The Height-Balance Property (HBP) was introduced as the extra condition that turns a BST into a balanced BST suitable for logarithmic search, insertion, and deletion.
- Students practised annotating node heights from external nodes upward and checking the height difference between each internal node's two children.
- After an insertion, only the new node's ancestor path needs to be checked for imbalance; the first/lowest unbalanced node is named A.
- The rebalancing rule identifies A, its child B, and grandchild C along the ancestor path; same-direction slants require a single rotation, while different-direction slants require a double rotation.
- A right-right imbalance was fixed by a single left rotation, with the inorder traversal order preserved as the key correctness invariant.
- Jackie pointed students back to Tutorial 3 resources, May 21 deletion exercises, optional pre-study materials, and upcoming Assignment 1 preparation.

Roadmap

Tutorial 3 resources, deletion practice, and Assignment 1 preparation

Jackie began by pointing students to the Tutorial 3 recording and resources on the lecture site, and to optional pre-study materials on eClass for recursion, singly linked lists, and trees. These materials are optional, but Jackie recommended them for students who want to regain fluency with recursive thinking and linked structures before Programming Assignment 1 and Programming Test 1. Jackie also strongly emphasized the assigned BST deletion exercises from the May 21 notes; Assignment 1 will not require deletion, but implementing deletion is useful preparation for Programming Test 1.

Searching through the small BST from Tutorial 3

The first technical review continued the BST construction from Tutorial 3: internal nodes such as $(28,$"alan"$)$, $(21,$"mark"$)$, and $(35,$"tom"$)$ were connected with four external nodes. Searching for key 21 from the root succeeds and returns the internal node storing 21. Searching for key 33 is unsuccessful, but it does not simply return a meaningless failure value; it returns the external node between predecessor 28 and successor 35, which is exactly where 33 would be inserted.

Why an unsuccessful search must return the correct external node

A student asked why the search for 33 could not return a different external node, such as the one to the right of 35. Jackie explained that if insertion used that wrong external node, placing 33 there would violate the BST search property because 33 is not larger than 35. The returned external node is therefore not arbitrary: it is the only insertion location consistent with the sorted inorder order.

Running time of BST search: $O(h)$ first

Jackie then used the recursive search routine to explain running time. Each recursive step goes left or right by one level, so the most accurate bound is $O(h)$, where h is the height of the BST. If the tree is balanced, then h is about $\log n$; if the tree is ill-balanced, then h can be linear in n. The same operation can therefore be described as $O(h)$, $O(\log n)$, or $O(n)$ depending on whether the tree height is known and whether the tree is balanced.

BST search versus binary search on a sorted array

The notes compared two BSTs with the same sorted inorder sequence. In the balanced tree, searching behaves similarly to binary search on the corresponding sorted array because each decision roughly cuts down the remaining possibilities. In the ill-balanced tree, the same kind of search can become like a linear scan through the sorted sequence. The lesson was that the BST property alone does not guarantee logarithmic search; the shape of the tree matters.

How bad insertion orders create linear height

To motivate balancing, Jackie considered insertion sequences such as decreasing keys

$$100, 75, 68, 60, 50, 1$$

Inserting these keys into an initially empty BST produces a chain-like tree with height 5 for 6 internal nodes, so $h = n - 1$. Jackie also listed increasing and in-between insertion sequences as practice cases. These examples explain why a BST must do more than just maintain the search property.

Defining node height and the Height-Balance Property

The next step was to define the height of a node recursively. An external node has height 0; an internal node has height

$$1 + \textbf{max}\{\text{height(left child)}, \text{height(right child)}\}.$$

The Height-Balance Property says that for every internal node, the absolute difference between the heights of its left and right children is at most 1. A BST that also satisfies this property is treated as a balanced BST, so its height-dependent operations can be kept logarithmic.

Checking HBP by annotating heights bottom-up

Jackie walked through a sample tree by first marking all external nodes with height 0, then computing internal-node heights upward. After that, each internal node was checked by comparing its two child heights. The first tree satisfied HBP. After inserting key 55, the heights changed along the insertion path, and node 48 became the first unbalanced node because its two child heights differed by 2.

Why only the ancestor path matters after insertion

After the break, Jackie summarized the rebalancing logic. When a new node n is inserted, only nodes on n's ancestor path can have their heights changed. If all nodes on that path remain balanced, no rotation is needed. If at least one ancestor becomes unbalanced, the algorithm uses the first/lowest unbalanced node on that path. Jackie emphasized that "lowest" means closest to the newly inserted node, not closest to the root.

Identifying A, B, and C

The rebalancing algorithm names the lowest unbalanced node A. Then B is A's child on the ancestor path from the new node upward, and C is B's child on that same path. The alignment of A, B, and C determines the repair: left-left or right-right alignments use a single rotation on the middle node B; left-right or right-left alignments use a double rotation on the lower node C.

Single left rotation and the inorder invariant

The lecture then focused on the right-right case, where A's right child is B and B's right child is C. Jackie first wrote the inorder traversal order of the abstract structure as

$$T_1, A, T_2, B, T_3, C, T_4.$$

A left rotation promotes B to the root of the local subtree, places A as B's left child and C as B's right child, and reattaches T_1, T_2, T_3, T_4 so that the same inorder order is preserved. This preservation of inorder order is the reason the rotation keeps the BST property valid.

Concrete left-rotation example after inserting 100

The final worked example began with an empty BST and inserted $44, 17, 78, 32, 50, 88, 95$, producing a balanced tree. After inserting 100, the lowest unbalanced node was 88, so $A = 88$, $B = 95$, and $C = 100$. Because the path was right-right, Jackie performed a left rotation on the middle node $B = 95$, making 95 the new root of that local subtree, with 88 on the left and 100 on the right. Heights were then updated to confirm the tree was balanced again.

What comes next

Jackie indicated that Monday's tutorial will include more sophisticated rotation examples and a deletion example. After that, the course will move on to graphs from scratch, with dynamic programming as the final major topic later in the semester.

Key Terms, Notation, and Structures

- **Internal node**: A BST node that stores a meaningful key/value entry and participates in the search and sorting properties.
- **External node**: A placeholder or sentinel node representing an empty child position; its key/value contents are not meaningful for inorder order or search comparisons.
- **Successful search**: A BST search that returns the internal node whose key matches the search key.
- **Unsuccessful search**: A BST search that returns the external node where the missing key should be inserted.
- **Predecessor and successor**: The keys immediately before and after a position in the sorted inorder sequence.
- **Search property**: For each internal node, all keys in the left subtree are smaller than the node's key, and all keys in the right subtree are larger.
- **Sorting property**: Inorder traversal of a BST gives the keys in ascending order.
- **BST search running time**: The accurate bound is $O(h)$, where h is the tree height; this can become $O(\log n)$ for balanced trees or $O(n)$ for ill-balanced trees.
- **Balanced BST**: In this lecture, a BST that also satisfies the Height-Balance Property, so height-dependent operations remain logarithmic.
- **Ill-balanced BST**: A BST whose shape is too chain-like, so its height can be linear in the number of nodes.
- **Height of an external node**: 0.

- **Height of an internal node**: $1 + \max\{h_L, h_R\}$, where h_L and h_R are the heights of the left and right children.
- **Height-Balance Property (HBP)**: For every internal node v, $|h_{\text{LST of } v} - h_{\text{RST of } v}| \leq 1$.
- **Ancestor path**: The path from a newly inserted node upward through parents toward the root; after insertion, this is the path whose balance must be checked.
- **Lowest unbalanced node**: The first unbalanced node encountered when moving upward from the new node; this node is named A in the rotation algorithm.
- **A, B, and C**: In trinode restructuring, A is the lowest unbalanced node, B is A's child on the ancestor path, and C is B's child on the same path.
- **Same-direction slant**: A left-left or right-right A-B-C pattern, repaired using a single rotation on B.
- **Different-direction slant**: A left-right or right-left A-B-C pattern, repaired using a double rotation on C.
- **Single left rotation**: The single rotation used for the right-right case; it promotes B and preserves inorder traversal.
- T_1, T_2, T_3, T_4: Abstract subtrees used to reason about rotation while preserving the inorder sequence, e.g., $T_1, A, T_2, B, T_3, C, T_4$.

Key Examples and Demonstrations

Tracing successful and unsuccessful search

Setup: The Tutorial 3 BST contained internal nodes 28, 21, and 35, plus four external nodes.

Main point: Searching for 21 returns the internal node 21, while searching for 33 returns the external node between 28 and 35, because that is the only insertion position that preserves the BST property.

What to practise: Redraw the small tree, label the four external nodes, and trace searches for keys that fall into each possible gap.

Comparing balanced and ill-balanced BST search

Setup: The notes compared two BSTs with the same inorder sequence of 15 keys: one balanced and one slanted.

Main point: The balanced tree search resembles binary search on a sorted array, while the slanted tree search resembles a linear scan. The BST property gives sorted order; balance gives efficient height.

What to practise: On both trees, trace a search path and count how many edges are followed before the search terminates.

Constructing a worst-case BST from insertion order

Setup: Jackie inserted decreasing keys $100, 75, 68, 60, 50, 1$ into an empty BST.

Main point: A valid BST can still have height $n-1$ if the insertion order makes it a chain, so search can become $O(n)$.

What to practise: Build the decreasing-key example, then try the increasing-key and in-between-key sequences listed in the notes.

Annotating heights and checking HBP

Setup: The notes used a sample BST and then asked whether it remained balanced after inserting 55 and after inserting 63.

Main point: Heights are computed bottom-up from external nodes. A node is balanced if its two child heights differ by at most 1. After inserting 55, node 48 becomes the first unbalanced node on the new node's ancestor path.

What to practise: Recompute every height on the page, then mark each internal node as balanced or unbalanced by comparing its two child heights.

Using A, B, and C to choose a rotation

Setup: After an insertion, the notes describe the rebalancing rule using the lowest unbalanced node A, then child B, then grandchild C along the ancestor path.

Main point: The alignment of A, B, C decides the rotation type: same slant gives a single rotation on B; different slant gives a double rotation on C.

What to practise: Draw all four cases: left-left, right-right, left-right, and right-left. For each, identify which node is rotated and whether the repair is single or double.

Abstract single left rotation

Setup: The abstract right-right shape was written using nodes A, B, C and subtrees T_1, T_2, T_3, T_4.

Main point: A left rotation promotes B and reattaches the subtrees so that the inorder traversal remains $T_1, A, T_2, B, T_3, C, T_4$.

What to practise: Before memorizing the picture, explain why each subtree must go where it goes by using the inorder sequence.

Concrete single left rotation after inserting 100

Setup: Starting from a balanced BST built from $44, 17, 78, 32, 50, 88, 95$, insert 100.

Main point: The insertion creates a right-right imbalance with $A = 88$, $B = 95$, and $C = 100$. A single left rotation on B restores balance while preserving the BST order.

What to practise: Redraw the before and after trees, annotate heights, and verify both HBP and inorder order after the rotation.

Common Pitfalls and Misconceptions

Treating an unsuccessful search as just failure

Pitfall: Thinking that an unsuccessful BST search should simply return `null` or any external node.

Why it causes problems: Insertion depends on the exact external node returned by search. Returning the wrong external node can place the new key where it violates the BST property.

How to avoid it: Trace the comparison path from the root until it terminates. The final external node is the only correct insertion position for that key.

Assuming every BST has $O(\log n)$ search

Pitfall: Saying that BST search is automatically logarithmic because it feels like binary search.

Why it causes problems: A BST can be a chain. In that case $h = n - 1$, and search is $O(n)$.

How to avoid it: State the precise bound first: search is $O(h)$. Only then specialize to $O(\log n)$ when the tree is balanced.

Forgetting that external nodes are placeholders

Pitfall: Treating external nodes as if their stored key/value fields are meaningful entries in the BST.

Why it causes problems: This corrupts reasoning about inorder order, search, insertion, and height annotations.

How to avoid it: Use external nodes as structural placeholders and insertion targets, not as keys in the sorted sequence.

Misidentifying the unbalanced node

Pitfall: Looking at two child heights that differ by more than 1 and labelling one of the children as unbalanced.

Why it causes problems: The node whose HBP is being checked is the parent. If its child heights differ too much, the parent is the unbalanced node.

How to avoid it: At each internal node, compare the heights of its left and right children; then decide whether that internal node is balanced.

Checking the wrong part of the tree after insertion

Pitfall: Re-checking arbitrary nodes,
or choosing an unbalanced node that is not the first/lowest one on the ancestor path.

Why it causes problems: The rotation algorithm depends on the unique lowest unbalanced node A. Choosing a higher one can lead to the wrong local repair.

How to avoid it: Start from the new node and move upward. The first ancestor that violates HBP is A.

Doing rotations by picture memorization only

Pitfall: Memorizing a rotation diagram without understanding why subtrees such as T_2 move to a particular position.

Why it causes problems: It becomes easy to reattach a subtree on the wrong side and break the BST sorting property.

How to avoid it: Use inorder preservation as the check: before and after rotation, the sequence must remain $T_1, A, T_2, B, T_3, C, T_4$ for the right-right left-rotation case.

Starting a rotation from an incorrectly drawn tree

Pitfall: Rushing into a rotation after drawing the insertion result incorrectly.

Why it causes problems: Even a correct rotation algorithm will give the wrong answer if the initial BST or height labels are wrong.

How to avoid it: Before rotating, double-check the insertion path, the external nodes, the heights, and the lowest unbalanced node.

Using generated practice without verification

Pitfall: Asking an AI tool for rotation exercises and trusting every generated solution automatically.

Why it causes problems: Generated examples can contain subtle BST, height, or rotation errors.

How to avoid it: Use generated exercises only as practice prompts, then verify the tree, inorder order, heights, and HBP yourself.

What You Should Be Able to Do

After this lecture, you should be able to:

- distinguish internal nodes from external nodes in the BST implementation convention used in lecture;
- trace a successful BST search and identify the returned internal node;
- trace an unsuccessful BST search and justify why the returned external node is the correct insertion location;
- explain why BST search is $O(h)$ and why this can become either $O(\log n)$ or $O(n)$ depending on tree shape;
- compare BST search in a balanced tree with binary search on a sorted array;

- construct a chain-like BST from a decreasing or increasing insertion sequence and compute its height;
- compute node heights using the recursive definition with external nodes of height 0;
- check the Height-Balance Property at each internal node by comparing child heights;
- explain why insertion can only affect balance along the new node's ancestor path;
- identify the lowest unbalanced node A after an insertion, then determine B and C;
- decide whether an A-B-C pattern requires a single or double rotation;
- perform a single left rotation for a right-right imbalance using the T_1, T_2, T_3, T_4 template;
- verify after a rotation that the inorder traversal order is preserved and the local height balance is restored;
- locate the posted deletion exercises and use them as practice for Programming Test 1 preparation.

Suggested Practice

- Recreate the small BST from Tutorial 3 and trace searches for 21, 33, and several other missing keys that should end at different external nodes.
- Revisit the May 21 BST deletion exercises, especially the four deletion cases and the most sophisticated case Jackie highlighted.
- Implement or pseudocode the BST search routine so that successful search returns an internal node and unsuccessful search returns the correct external node.
- Draw the BST produced by the decreasing insertion sequence $100, 75, 68, 60, 50, 1$, then repeat with the increasing and in-between sequences listed in the notes.
- On the HBP example page, recompute every height and every child-height difference without looking at Jackie's annotations.
- Insert 55 into the HBP example and explain why 48 is the first unbalanced node; then attempt the 63 exercise left in the notes.
- For each of the four A-B-C shapes, write whether the case is same-slant or different-slant and whether the repair is single or double rotation.
- Redraw the single left rotation template and justify each subtree location using inorder traversal rather than memory.
- Redo the concrete insertion of 100 into the tree built from $44, 17, 78, 32, 50, 88, 95$, then annotate the heights before and after rotation.
- Use the optional pre-study materials if recursion, linked structures, or tree-node manipulation still feel rusty before Assignment 1.

Where to Review

- **Lecture 7 iPad/PDF notes, page 2:** reminders about Tutorial 3, the May 21 deletion exercises, and pre-study materials.
- **Page 3:** tracing BST search with internal and external nodes, including the unsuccessful search for 33.
- **Page 4:** the recursive BST search routine and the three running-time views: $O(h)$, $O(\log n)$, and $O(n)$.
- **Page 5:** the balanced-vs.-ill-balanced BST comparison and the link to binary search on a sorted array.
- **Page 6:** worst-case insertion sequences that produce linear-height BSTs.
- **Page 7:** height calculation, HBP checking, and the questions about inserting 55 and 63.
- **Page 8:** the compact summary "BST + HBP = balanced BST" and the logarithmic-operation motivation.
- **Page 9:** the after-insertion rebalancing algorithm and the A, B, C same-slant vs. different-slant decision.
- **Page 10:** the abstract single left rotation and the inorder sequence $T_1, A, T_2, B, T_3, C, T_4$.
- **Page 11:** the concrete left-rotation example after inserting 100.
- **Tutorial 3 recording and starter resources:** review the BST node setup, external nodes, and Java exercises if the implementation details are unclear.
- **May 21 notes and deletion walkthrough:** use the BST deletion exercises as extra practice for Programming Test 1 preparation.

Closing Reminder

This digest is only a roadmap. To learn this lecture properly, review the recording and Jackie's iPad notes, redraw the BSTs, retrace the searches, recompute heights, and practise rotations by hand. The central skill is not recognizing a diagram after seeing it once; it is being able to build the tree, find the ancestor path, identify A, B, C, preserve inorder order, and verify balance yourself.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 8, May 28

Graph Definitions, Directions, Degrees, and Edge Bounds

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 11

Closing Reminder 12

Main Ideas

- The lecture opened the graph unit by defining a graph formally as $G = (V, E)$, where V is the set of vertices and E is the set of edges.
- Jackie emphasized precise graph vocabulary: use *vertex/vertices* and *edge*, not informal language such as circles and lines.
- Graphs were presented as a general structure for modeling relationships among entities, with trees as a special case inside the larger class of graphs.
- Directed, undirected, and mixed graphs were distinguished by whether edges encode direction, dependency, flow, order, or bidirectional connection.
- Self-loops and parallel edges were introduced so that students can recognize when a graph is *simple* and when special cases are present.
- Vertex degree was defined through incident edges, with in-degree and out-degree needed for directed graphs.
- Three foundational graph properties were introduced: the sum of degrees in an undirected graph, the equality of total in-degree and total out-degree in a directed graph, and the maximum number of edges in a simple undirected graph.
- The bound $m \leq \frac{n \cdot (n-1)}{2}$ was connected to complete graphs and to the asymptotic fact that $m = O(n^2)$ for simple undirected graphs.
- Jackie previewed the graph unit ahead: DFS, BFS, topological sort, minimum spanning trees, Dijkstra's shortest path algorithm, greedy algorithms, proof techniques, and graph implementation.
- Tutorial 4 will focus on rotations, Assignment 1 will be tree-based, and deletion practice remains important for Programming Test 1 preparation.

Roadmap

Logistics before starting graphs

Jackie began by pointing students toward the coming Monday tutorial in CLH, where the focus will be tree rotations, including double rotations. Assignment 1 is scheduled for release on Wednesday, June 3, possibly one or two days earlier, and it will be about trees rather than graphs. Jackie also reminded students to practise BST deletion using the previously assigned exercises from the May 21 notes, because deletion is not the main Assignment 1 task but may matter for Programming Test 1.

Course roadmap for the rest of the term

The lecture then situated the graph unit within the course plan. Graphs will occupy a substantial block of lectures, roughly through Lecture 20, while dynamic programming is planned for Tutorial 9 and the final four lectures if time permits. Jackie previewed DFS, BFS, topological sorting, minimum spanning trees, Dijkstra's algorithm, greedy algorithms, correctness reasoning with preconditions/postconditions and loop invariants, mathematical induction, and graph implementation.

Graphs as models of relationships

The technical content started from the motivation that graphs model relationships among entities. A graph is more general than a tree: trees have a strong hierarchical structure, but graphs allow richer relationships, including multiple predecessors and successors. Jackie connected this to applications such as course prerequisites, road maps, control flow diagrams, social networks, co-authorship networks, web pages, and biological interaction networks.

The formal definition $G = (V, E)$

Jackie defined a graph as a pair $G = (V, E)$, where V is the set of vertices and E is the set of edges. In the page-3 diagram, the vertices were A, B, C, D, E, F, so $|V| = 6$, and the edges could be listed as AC, CD, DE, AB, BF, AE, so $|E| = 6$. The lecture emphasized that students must be able to move in both directions: extract V and E from a picture, and draw a graph from given sets.

Smallest valid graphs and an invalid edge-only picture

Because V and E are sets, the empty graph with $V = \varnothing$ and $E = \varnothing$ is valid. A one-vertex graph with no edges is also valid. By contrast, a graph with one edge but no vertices is not valid, because each edge must have endpoints. A student's suggestion that a minimum graph might have $|E| = |V| - 1$ was connected to a later special case about spanning structures, not to the absolute minimum graph.

Graphs, trees, and logic notation

Jackie placed trees inside graphs conceptually: every tree is a graph, but many graphs are not trees. The lecture also connected graph notation to predicate logic. When we write quantified statements over vertices or edges, V and E become universes of discourse. For example, $\forall x \in V : P(x)$ means the property is being asserted for every vertex, while a statement about $(a, b) \in E$ refers to vertices connected directly by an edge.

Directed, undirected, and mixed edges

For a directed edge from u to v, $(u, v) \in E$ but generally $(v, u) \notin E$; u is the origin and v is the destination. For an undirected edge, uv and vu are conceptually the same edge and should

be counted once. Jackie connected this to future implementation work: an undirected edge's equality test must treat the two endpoint orders as the same. The road-map example with intersections and one-way/two-way streets illustrated a mixed graph containing both directed and undirected edges.

Applications that depend on direction

Direction matters when a relationship represents dependency, flow, or order. The course prerequisite example used arrows such as EECS2030 → EECS2101 → EECS3101 and MATH1090 → EECS3101. A control flow diagram used directed edges to model possible execution paths through statements and conditionals. By contrast, a friendship graph or two-way road segment is naturally undirected unless the application says otherwise.

Self-loops, parallel edges, and simple graphs

A self-loop is an edge from a vertex to itself. Parallel edges are distinct edges with the same endpoints, possibly distinguished by labels, meanings, directions, or weights. A simple graph has neither self-loops nor parallel edges. Many graph properties in the lecture explicitly assume simplicity, so the definitions are not just vocabulary: they control when formulas may be used safely.

Degree, in-degree, and out-degree

The degree of a vertex counts edges incident on it. In a directed graph, this splits into in-degree and out-degree. In the directed example, vertex A had outgoing edges m and o, incoming edge q, and therefore $\deg(A) = 3$ with out-degree 2 and in-degree 1. This example is the model for how to read degrees from diagrams.

Three basic graph properties

For a simple undirected graph with m edges, Jackie introduced

$$\sum_{v \in V} \deg(v) = 2 \cdot m.$$

The example graph had degrees $3, 2, 2, 2, 2, 3$, which sum to $14 = 2 \cdot 7$. For a directed graph, Jackie introduced

$$\sum_{v \in V} \text{in-degree}(v) = \sum_{v \in V} \text{out-degree}(v),$$

with both totals equal to 7 in the example. Finally, for a simple undirected graph with n vertices and m edges,

$$m \leq \frac{n \cdot (n-1)}{2}.$$

Equality gives a complete graph, and the bound implies $m = O(n^2)$. Jackie indicated that a proof of the degree-sum property by mathematical induction will come next time.

Key Terms, Notation, and Structures

- **Graph**: A structure $G = (V, E)$ consisting of a set of vertices V and a set of edges E.
- **Vertex / vertices**: The entities in a graph; *vertex* is singular and *vertices* is plural.
- **Edge**: A relationship or connection between vertices; Jackie also mentioned the term arc, but used edge throughout the lecture.
- **Cardinality**: The size of a set; $|V|$ is the number of vertices and $|E|$ is the number of edges.
- n **and** m: Standard graph-size variables, where $n = |V|$ and $m = |E|$.
- **Empty graph**: A valid graph with $V = \varnothing$ and $E = \varnothing$.
- **One-vertex graph**: A valid graph with one vertex and no edges.
- **Directed edge**: An ordered edge (u, v) from origin u to destination v; direction must be respected.
- **Undirected edge**: An edge with endpoints u and v but no direction; uv and vu represent the same edge.
- **Mixed graph**: A graph containing both directed and undirected edges.
- **Origin and destination**: For a directed edge (u, v), u is the origin and v is the destination.
- **Endpoint**: A vertex at one end of an edge; undirected edges are best thought of by endpoints rather than origin/destination.
- **Self-loop / self-edge**: An edge from a vertex to itself, such as (u, u).
- **Parallel edges**: Multiple distinct edges connecting the same endpoints, possibly with different labels, meanings, or weights.
- **Simple graph**: A graph with no self-loops and no parallel edges.
- **Not-simple graph**: A graph containing at least one self-loop or at least one pair of parallel edges.
- **Incident on**: An edge is incident on a vertex if the edge is connected to that vertex; incident edges count toward degree.
- **Degree**: For a vertex v, $\deg(v)$ is the number of incident edges.
- **In-degree and out-degree**: In a directed graph, in-degree counts incoming edges and out-degree counts outgoing edges.
- **Property**: For a simple undirected graph, $\sum_{v \in V} \deg(v) = 2 \cdot m$.

- **Directed degree-sum property**: For a directed graph, total in-degree equals total out-degree.
- **Complete graph**: A simple undirected graph in which every pair of distinct vertices is connected by an edge; it has $m = \frac{n \cdot (n-1)}{2}$ edges.
- $m = O(n^2)$: In a simple undirected graph, the number of edges is at most quadratic in the number of vertices.
- **Universe of discourse**: The set over which a quantified variable ranges, such as V for vertices or E for edges.
- **DFS and BFS**: The two main graph traversal algorithms previewed in this lecture.
- **Topological sort**: A graph algorithm previewed for dependency/order problems, such as prerequisite structures.
- **Dijkstra's shortest path algorithm**: A later graph algorithm used as an example of greedy reasoning and correctness via invariants.
- `equals` **for undirected edges**: A future implementation concern: an edge from u to v should compare equal to an edge from v to u when the graph is undirected.

Key Examples and Demonstrations

Extracting V and E from a graph picture

Setup: The notes used a graph with vertices A, B, C, D, E, F
and edges such as AC, CD, DE, AB, BF, AE.

Main point: A graph picture is not just a drawing; it is a representation of two sets. Students should be able to write V, write E, and compute $|V|$ and $|E|$ accurately.

What to practise: Redraw the page-3 graph, list its vertices and edges, and then check the edge count again without relying on the labels.

Empty, one-vertex, and invalid edge-only graphs

Setup: Jackie asked what the smallest possible graph could be.

Main point: The empty graph is valid, and a one-vertex graph with no edges is valid. A line with no endpoint vertices is not a valid graph because an edge must have endpoints.

What to practise: Write down several very small $G = (V, E)$ examples and decide which ones are valid graphs.

Prerequisites as a directed graph

Setup: Jackie used prerequisite relationships such as EECS2030 → EECS2101 → EECS3101 and MATH1090 → EECS3101.

Main point: The arrows encode dependency and order. This is the kind of structure where topological sort will later be useful.

What to practise: Draw a small prerequisite graph from courses you know and explain what each arrow means.

Road maps and control flow as directed or mixed graphs

Setup: The lecture used a road-map sketch with intersections and one-way/two-way roads, plus a control-flow sketch with initialization, assignment, and a conditional branch.

Main point: The same graph vocabulary applies across different applications, but the direction of edges depends on the meaning: roads may be one-way or two-way, while control flow follows program execution.

What to practise: Draw one mixed road-map graph and one directed control-flow graph, then label which edges are directed and which are undirected.

Self-loops and parallel edges

Setup: The notes showed an edge from u back to u and multiple edges between u and v.

Main point: These structures are allowed in some graphs, but they make the graph not simple. Parallel edges can represent distinct meanings, such as different routes, airlines, labels, or weights between the same endpoints.

What to practise: Given a graph drawing, mark every self-loop and every pair of parallel edges, then decide whether the graph is simple.

Computing degree for vertex A

Setup: In the directed degree example, vertex A had outgoing edges m and o, and incoming edge q.

Main point: For directed graphs, compute in-degree and out-degree separately before adding them for total degree.

What to practise: Repeat the degree computation for vertices B, C, D, E, F in the same diagram.

Checking the undirected degree-sum property

Setup: The notes used a simple undirected graph with six vertices and seven edges.

Main point: The degrees $3, 2, 2, 2, 2, 3$ sum to 14, and $2 \cdot m = 2 \cdot 7 = 14$. Each undirected edge contributes once to each of its two endpoints.

What to practise: For a new simple undirected graph, make a degree table and verify $\sum_{v \in V} \deg(v) = 2 \cdot m$.

Checking total in-degree equals total out-degree

Setup: The notes used a directed version of the example graph and recorded in-degree/out-degree values in a table.

Main point: Every directed edge leaves exactly one vertex and enters exactly one vertex, so total out-degree and total in-degree match.

What to practise: Make your own directed graph, compute all in-degrees and out-degrees, and confirm that both totals match.

Bounding the number of edges and recognizing complete graphs

Setup: The notes used a complete graph on five vertices.

Main point: In a simple undirected graph, each vertex can connect to at most $n-1$ other vertices, and the count $n \cdot (n-1)$ double-counts every undirected edge. Dividing by 2 gives $m \leq \frac{n \cdot (n-1)}{2}$. For $n = 5$, equality gives 10 edges.

What to practise: Draw complete graphs for $n = 3, 4, 5$, count their edges, and compare your counts to $\frac{n \cdot (n-1)}{2}$.

Common Pitfalls and Misconceptions

Using informal words instead of graph terminology

Pitfall: Calling vertices "circles" or edges "lines" in technical explanations.

Why it causes problems: Informal language hides the formal structure $G = (V, E)$ and makes later definitions, proofs, and implementations harder to state precisely.

How to avoid it: Say vertex, vertices, and edge consistently, even when referring to a picture.

Double-counting undirected edges

Pitfall: Treating uv and vu as two different edges in an undirected graph.

Why it causes problems: This gives wrong values for $|E|$, breaks degree-counting examples, and leads to a wrong implementation of edge equality.

How to avoid it: For undirected graphs, treat an edge as an unordered connection between endpoints and count it once.

Confusing the minimum graph with a spanning case

Pitfall: Answering that a minimum graph must satisfy $|E| = |V| - 1$.

Why it causes problems: That relationship belongs to special connected/spanning structures later in the course, not to the definition of a graph in general.

How to avoid it: Start from the set definition: the empty graph $V = \varnothing, E = \varnothing$ is the smallest valid graph.

Ignoring edge direction

Pitfall: Assuming that if $(u, v) \in E$, then $(v, u) \in E$ automatically.

Why it causes problems: This is false for directed graphs and will lead to wrong traversals, wrong dependencies, and wrong algorithm behaviour.

How to avoid it: Check whether the graph is directed, undirected, or mixed before tracing or counting anything.

Using simple-graph formulas without checking simplicity

Pitfall: Applying formulas such as $m \leq \frac{n \cdot (n-1)}{2}$ without checking for self-loops or parallel edges.

Why it causes problems: The bound was stated for simple undirected graphs. Self-loops and parallel edges change what can be counted.

How to avoid it: Before using a graph property, read its assumptions: simple, directed/undirected, and the meaning of n and m.

Mixing up degree, in-degree, and out-degree

Pitfall: Counting only outgoing edges, or only incoming edges, when asked for total degree in a directed graph.

Why it causes problems: Degree-related tables and formulas will be wrong, especially when verifying the directed degree-sum property.

How to avoid it: For directed graphs, first compute in-degree and out-degree separately, then add them only when total degree is requested.

Misreading the edge bound as the wrong Big-O relationship

Pitfall: Confusing which quantity is bounded in $m \leq \frac{n \cdot (n-1)}{2}$.

Why it causes problems: The conclusion is about the number of edges: in a simple undirected graph, $m = O(n^2)$.

How to avoid it: Always identify $n = |V|$ and $m = |E|$ before interpreting a formula asymptotically.

Dropping tree practice because the graph unit has started

Pitfall: Assuming the start of graphs means tree rotations and deletions no longer matter.

Why it causes problems: Tutorial 4, Assignment 1, and Programming Test 1 preparation still depend on the tree material.

How to avoid it: Keep practising rotations and deletion cases while starting the graph vocabulary and counting work.

What You Should Be Able to Do

After this lecture, you should be able to:

- state the formal definition of a graph as $G = (V, E)$;
- use the words vertex, vertices, and edge correctly;

- extract V, E, $|V|$, and $|E|$ from a graph diagram;
- draw a graph when given V and E;
- distinguish valid small graphs from invalid edge-only pictures;
- explain why trees are special cases of graphs;
- interpret V and E as universes of discourse in quantified statements;
- distinguish directed, undirected, and mixed graphs;
- identify the origin and destination of a directed edge;
- explain why uv and vu represent the same edge in an undirected graph;
- model simple situations such as prerequisites, control flow, road maps, and social connections using appropriate graph directionality;
- identify self-loops and parallel edges;
- determine whether a graph is simple or not simple;
- compute degree, in-degree, and out-degree from a diagram;
- verify $\sum_{v \in V} \deg(v) = 2 \cdot m$ on a simple undirected graph;
- verify that total in-degree equals total out-degree on a directed graph;
- derive or explain the intuition behind $m \leq \frac{n \cdot (n-1)}{2}$ for simple undirected graphs;
- recognize a complete graph and compute its number of edges;
- explain why $m = O(n^2)$ for simple undirected graphs;
- describe at a high level what DFS, BFS, topological sort, MST, Dijkstra's algorithm, and greedy correctness proofs will contribute later in the graph unit;
- identify the near-term study tasks connected to Tutorial 4, Assignment 1, and Programming Test 1.

Suggested Practice

- Redraw the page-3 graph and write both V and E without looking at the notes; then compare your edge count with the notes.
- Create three tiny graphs: an empty graph, a one-vertex graph, and a graph with two vertices and one edge; write each as $G = (V, E)$.
- Practise converting a list of edges into a diagram, especially when some edges are directed and others are undirected.

- Draw a small prerequisite graph from your own courses and explain which edges encode dependency.
- Draw a small control-flow graph for an `if/else` statement or loop and label the directed edges.
- For at least two graphs, mark all self-loops and parallel edges, then decide whether each graph is simple.
- Make a degree table for an undirected graph and check the handshaking lemma.
- Make separate in-degree and out-degree tables for a directed graph and check that the two totals match.
- Draw complete graphs with 3, 4, and 5 vertices and verify the formula $\frac{n\cdot(n-1)}{2}$.
- Review the May 21 deletion exercises and the posted deletion walkthrough before Programming Test 1 preparation becomes urgent.
- Prepare for Tutorial 4 by reviewing the single-rotation example from Lecture 7 and then trying a double-rotation example by hand.
- Use the optional pre-study materials from 2101 selectively if you need more practice with recursion, trees, or singly linked lists.

Where to Review

- **Lecture 8 notes, page 1**: topic overview for the start of the graph unit.
- **Lecture 8 notes, page 2**: reminders about Tutorial 4, deletion exercises, and pre-study materials.
- **Lecture 8 notes, page 3**: formal graph definition $G = (V, E)$, the example graph with A through F, valid small graphs, and the tree-vs-graph picture.
- **Lecture 8 notes, page 4**: universe of discourse examples for quantifying over vertices and edges.
- **Lecture 8 notes, page 5**: directed vs. undirected edges, origin/destination, endpoint equality, applications, control flow, and mixed road-map graphs.
- **Lecture 8 notes, page 6**: self-loops, parallel edges, simple graphs, and not-simple graphs.
- **Lecture 8 notes, page 7**: incident edges, degree, in-degree, out-degree, and the worked degree example for vertex A.
- **Lecture 8 notes, pages 8 and 9**: sum of degrees for simple undirected graphs and the example degree table.

- **Lecture 8 notes, pages 10 and 11**: equality of total in-degree and total out-degree for directed graphs.
- **Lecture 8 notes, pages 12 and 13**: the edge bound $m \leq \frac{n \cdot (n-1)}{2}$, complete graphs, and the $m = O(n^2)$ interpretation.
- **Lecture recording**: revisit the opening roadmap, the graph definition examples, the directed/undirected discussion, and the three graph properties.
- **May 21 notes and deletion resources**: use these for BST deletion practice, especially for Programming Test 1 preparation.
- **Tutorial 4 materials**: use them to consolidate tree rotations before Assignment 1.
- **Graph ADT slides**: use them as the stable slide reference for the graph vocabulary and properties introduced here.

Closing Reminder

This digest gives you the roadmap for the first graph lecture, but it is not enough by itself. You should still review Jackie's iPad/PDF notes, revisit the recording where the diagrams and examples were explained, and practise translating between pictures, sets, and formulas. The graph unit will build quickly from these definitions into traversal, correctness proofs, greedy algorithms, and implementation, so make the basic vocabulary and counting properties automatic now. At the same time, keep up with tree rotations and deletion practice for the near-term assignment and programming-test preparation.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 9, June 2

Proving vs. Disproving Properties, Induction, Paths, Cycles, and Subgraphs

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 3

Key Examples and Demonstrations 4

Common Pitfalls and Misconceptions 5

What You Should Be Able to Do 6

Suggested Practice 7

Where to Review 8

Closing Reminder 8

Main Ideas

- The lecture continued the graph unit by moving from applying graph properties on examples to proving or disproving such properties formally.
- The central proof example was the sum-of-degrees property for simple, undirected graphs: $\sum_{v \in V} \deg(v) = 2 \cdot |E|$.
- Jackie used this proof to review the mathematical-induction pattern: base cases, inductive hypothesis, and an inductive step that strictly extends the input and uses the hypothesis.
- The lecture introduced graph paths, cycles, simple versus non-simple variants, and the reachability problem.
- The lecture introduced subgraphs as valid graphs formed from subsets of vertices and edges, preparing for spanning, connected, cyclic, acyclic, forest, tree, and spanning-tree terminology.
- Assignment 1 had been released, Tutorial 4's tree-rotation material was available, and Jackie reminded students to practise deletion and to review linked lists before later graph assignments.

Roadmap

Assignment and tutorial reminders

Jackie began by pointing students back to Assignment 1 and the recording of Tutorial 4 on tree rotations. The immediate technical preparation is still BST work: complete Assignment 1 before the deadline, use office hours for clarifications when stuck, and practise deletion from BSTs even though Assignment 1 does not require it. Jackie also emphasized pre-study for recursion, trees, and singly linked lists: Assignment 1 does not use linked lists, but later graph assignments will.

From checking examples to proving or disproving properties

The lecture then returned to the graph properties from the previous class. A property is a claim that can be evaluated as true or false under specified assumptions, such as "simple" and "undirected". Applying the property to small example graphs is useful for sketching and intuition, but it is not the final justification. The final outcome should be either a proof that every graph satisfying the assumptions also satisfies the property, or a disproof by giving one valid witness/counterexample.

The mathematical induction template

Jackie reviewed induction as a proof structure analogous to recursion. A proof by induction needs base cases for small legitimate inputs, an inductive hypothesis that assumes the property for input size k, and an inductive step that proves the property for a strictly larger input.

The step must use the inductive hypothesis and must cover all minimal ways to extend the object. Jackie stressed that the extension is usually $k+1$, but can be different depending on the structure being proved.

Proof of the degree-sum property

The main proof used induction on $|V|$, the number of vertices, for a simple, undirected graph $G = (V, E)$. Jackie chose vertices rather than edges because a graph with a fixed number of edges can have many incompatible vertex structures. The base cases were the empty graph and the one-vertex graph, both with no edges. The one-vertex-one-edge case was rejected because it would be a self-loop and therefore not a simple graph.

For the inductive step, Jackie visualized an arbitrary graph with $|V| = k$ and $|E| = m$ as a cloud satisfying the inductive hypothesis. A new vertex y is added, and d existing vertices are connected to y, where $0 \leq d \leq k$. The extended graph has $k+1$ vertices and $m+d$ edges. The degree sum increases by $2 \cdot d$, because each new undirected edge contributes one degree to y and one degree to an old endpoint. Using the inductive hypothesis gives

$$\sum_{v \in V \cup \{y\}} \deg(v) = 2 \cdot m + 2 \cdot d = 2 \cdot (m + d),$$

which is exactly twice the number of edges in the extended graph.

Paths, cycles, and reachability

After the proof, the lecture moved to path and cycle terminology. A path can be represented either by alternating vertices and edges or, when there are no parallel edges to disambiguate, by a sequence of vertices. Jackie distinguished simple paths, simple cycles, non-simple paths, and non-simple cycles by checking whether vertices or internal cycles are repeated. Jackie also introduced the reachability problem: given a graph $G = (V, E)$ and vertices $x, y \in V$, determine whether there is a path from x to y, and if so produce one.

Subgraphs and validity

The final technical segment introduced subgraphs. For $G = (V, E)$, a graph $G' = (V', E')$ is a subgraph when $V' \subseteq V$, $E' \subseteq E$, and G' is itself a valid graph. Jackie emphasized the last condition: an edge cannot be included without its endpoint vertices. The empty graph is a valid minimum subgraph, and the original graph is the maximum subgraph of itself. The lecture ended by previewing subgraph properties such as cyclic versus acyclic, spanning, and connected.

Key Terms, Notation, and Structures

- **Property**: A graph-related claim that can be evaluated as true or false under stated assumptions.

- **Assumption**: A condition such as "simple" or "undirected" that limits which graphs the property is about.
- **Proof**: A universal justification that every graph satisfying the assumptions also satisfies the claimed property.
- **Disproof, witness, counterexample**: A single valid graph satisfying the assumptions but violating the property.
- **Mathematical induction**: A proof technique with base cases, an inductive hypothesis, and an inductive step.
- **Inductive hypothesis (IH)**: The assumed truth of the property for an arbitrary input of size k, used as a helper step in the proof.
- **Minimal extension**: The smallest structural growth needed in the inductive step, such as adding one vertex to move from k to $k+1$ vertices.
- **Degree-sum property**: For a simple, undirected graph, $\sum_{v \in V} \deg(v) = 2 \cdot |E|$.
- **Simple path**: A path containing no cycle; in the lecture examples, this meant no repeated internal vertex.
- **Simple cycle**: A closed path that starts and ends at the same vertex, with the middle portion forming a simple path.
- **Non-simple path or cycle**: A path or cycle containing repeated vertices or an internal cycle.
- **Reachability problem**: Given $G = (V, E)$ and $x, y \in V$, decide whether a path exists between x and y.
- **Subgraph**: A graph $G' = (V', E')$ with $V' \subseteq V$, $E' \subseteq E$, and with every edge in E' having its endpoint vertices in V'.
- **Cyclic vs. acyclic**: A cyclic graph contains a cycle; an acyclic graph contains no cycle.
- **Spanning and connected**: Previewed subgraph properties to be developed next; spanning concerns retaining all vertices, while connected concerns path-based reachability among vertices.

Key Examples and Demonstrations

Proof-or-disproof workflow for graph properties

Setup: Start with a graph property under stated assumptions, such as a property about simple, undirected graphs.

Main point: Trying a few small graphs is only an intuition step. A final answer needs either a proof for all graphs satisfying the assumptions or one valid counterexample.

What to practise: For each new graph property, first test a few small graphs, then write down whether you are aiming for a proof or a disproof before starting the formal argument.

Induction proof of the degree-sum property

Setup: Prove $\sum_{v \in V} \deg(v) = 2 \cdot |E|$ for every simple, undirected graph $G = (V, E)$, using induction on $|V|$.

Main point: The base cases show the property for graphs with zero or one vertex. The inductive step adds a new vertex y and connects it to d old vertices. The old graph contributes $2 \cdot m$ by the IH, and the new edges contribute $2 \cdot d$, giving $2(m + d)$.

What to practise: Rewrite the proof without looking at the notes, making sure you state the assumptions, choose $|V|$ as the induction variable, reject invalid base cases, and explicitly use the IH in the algebra.

Path and cycle examples

Setup: The notes used a graph with vertices such as A, B, C, D, E, F and labelled edges to show sequences like F, D, E, A, E, D, C, A, E, $B, F, E, D, F, D, C, A, B$, and F, D, E, F, B.

Main point: The examples separate four cases: simple path, simple cycle, non-simple cycle, and non-simple path. The key check is not whether the drawing "looks circular", but whether vertices are repeated and whether the middle portion contains a cycle.

What to practise: For any proposed path or cycle, mark repeated vertices and decide whether the word "simple" is justified. Then rewrite the same path using both edge-labelled and vertex-only notation when possible.

Subgraph examples and invalid subgraphs

Setup: The lecture considered the empty graph, the whole graph, a subgraph such as $G_3 = (\{A, B, F\}, \{m\})$, and an invalid construction such as $(\varnothing, \{m\})$.

Main point: Subgraphs are not arbitrary pairs of vertex and edge subsets. The edge set must be compatible with the vertex set, because an edge cannot exist without its endpoints.

What to practise: Choose several vertex subsets and edge subsets from the lecture graph. For each one, decide whether it is a valid subgraph, then redraw the valid ones separately from the original graph.

Common Pitfalls and Misconceptions

Treating examples as a proof

Pitfall: Showing that a property works on one or two small graphs and calling that a proof.

Why it causes problems: Examples can support intuition, but they do not justify a universal claim over all graphs satisfying the assumptions.

How to avoid it: Use examples only as a sketching phase. Then either write a universal proof or give one valid counterexample.

Writing an induction step that does not use the IH

Pitfall: Proving the $k+1$ case from scratch, or doing algebra without connecting back to the inductive hypothesis.

Why it causes problems: The proof no longer has the structure of mathematical induction; it does not show how truth at size k supports truth at the larger size.

How to avoid it: In the inductive step, explicitly identify where the IH is used. In the degree-sum proof, the old graph's degree sum is replaced by $2 \cdot m$ by the IH.

Forgetting the assumptions of the property

Pitfall: Including a self-loop base case when proving a property for simple graphs.

Why it causes problems: A graph with one vertex and one self-loop is not simple, so it is outside the property being proved.

How to avoid it: Before testing a case or using a counterexample, check that it satisfies all assumptions in the property statement.

Choosing an awkward induction variable

Pitfall: Inducting on the number of edges when the structure of the graph is more naturally controlled by the number of vertices.

Why it causes problems: A fixed number of edges does not determine enough about the graph's vertex structure, making base cases and extensions harder to organize.

How to avoid it: For graph proofs in this part of the course, first consider induction on $|V|$, unless the property clearly suggests another measure.

Assuming every path or cycle is simple

Pitfall: Reading "path" or "cycle" as if "simple" were automatically included.

Why it causes problems: Non-simple paths and cycles may repeat vertices or contain internal cycles, so arguments based on no repetition become invalid.

How to avoid it: When the word "simple" is absent, do not assume it. Check the sequence explicitly for repeated vertices and internal cycles.

Constructing an invalid subgraph

Pitfall: Choosing an edge for E' while omitting one or both of its endpoint vertices from V'.

Why it causes problems: The resulting pair (V', E') is not even a valid graph, so it cannot be a subgraph.

How to avoid it: For every edge included in E', verify that both endpoint vertices are included in V'.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain why applying a property to examples is useful but not a formal proof;
- distinguish between proving a universal graph property and disproving it by witness/counterexample;
- state the base cases, inductive hypothesis, and inductive step in a graph induction proof;
- explain why Jackie chose induction on $|V|$ rather than $|E|$ for the degree-sum proof;
- reproduce the induction proof that $\sum_{v \in V} \deg(v) = 2 \cdot |E|$ for simple, undirected graphs;
- explain why a one-vertex graph with one self-loop is not a valid base case for a property about simple graphs;
- represent a path using either alternating vertices and edges or a vertex sequence when no ambiguity is introduced;
- classify a given sequence as a simple path, simple cycle, non-simple path, or non-simple cycle;
- state the reachability problem for two vertices in a graph;
- define a subgraph $G' = (V', E')$ of $G = (V, E)$ and check whether a proposed G' is a valid graph;
- prepare for the next lecture by distinguishing cyclic/acyclic, spanning, and connected as different subgraph properties.

Suggested Practice

- Review the Tutorial 4 tree-rotation recording and use it while working through Assignment 1, especially if rotations or height updates are still unclear.
- Practise BST deletion from the slides and notes, since Jackie again connected deletion practice to later preparation.
- Rewrite the degree-sum induction proof from memory, then compare your version against the notes and check whether your inductive step explicitly uses the IH.
- Create two small graph properties of your own: try to prove one and disprove the other by a counterexample.
- Draw a small graph and list one simple path, one simple cycle, one non-simple path, and one non-simple cycle.
- From the lecture's graph, construct several candidate subgraphs and reject any candidate that includes an edge without its endpoint vertices.
- Practise singly linked list operations such as insert, delete, and traversal, because Jackie said linked lists will be useful for upcoming graph implementations.

Where to Review

- **Lecture 9 iPad/PDF notes, page 2:** Assignment 1 release, Tutorial 4 tree rotations, deletion exercises, and programming-assignment pre-study reminders.
- **Lecture 9 iPad/PDF notes, pages 3–4:**
  The proof-vs-disproof strategy and the mathematical-induction template.
- **Lecture 9 iPad/PDF notes, page 5:** The full degree-sum induction proof, including base cases, the cloud diagram, the new vertex y, and the d-edge extension.
- **Lecture 9 iPad/PDF notes, pages 6–7:** Path and cycle examples, including the distinction between simple and non-simple cycles and the self-loop edge case.
- **Lecture 9 iPad/PDF notes, pages 8–10:** Subgraph definitions, examples, and the preview of cyclic/acyclic, spanning, and connected properties.
- **Lecture recording:** Rewatch the first half for the induction proof and the second half for path, cycle, reachability, and subgraph terminology.
- **Course materials:** Revisit the math review lecture for quantifiers, witnesses, counterexamples, and induction; revisit Tutorial 4 and the self-balancing BST slides while completing Assignment 1.

Closing Reminder

This digest is only a roadmap. To learn this lecture properly, review Jackie's notes and the recording, then actively redo the proof, classify paths and cycles yourself, and construct valid and invalid subgraphs by hand. For the programming side of the course, keep working on Assignment 1 and use the linked-list pre-study as preparation for the graph implementations that come later.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 10, June 4

Spanning Subgraphs, Connected Components, Forests, and Spanning Trees

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas

- The lecture continued the graph vocabulary sequence by moving from general subgraphs to spanning subgraphs, connected graphs, connected components, forests, trees, and spanning trees.
- A spanning subgraph must keep all vertices of the original graph, but it may keep only some, or none, of the original edges.
- Connectedness is a separate path-based property: a graph is connected when every pair of distinct vertices is reachable through some path.
- Connected components are maximal connected subgraphs; they explain how a disconnected graph breaks into pieces and prepare for later DFS/BFS traversal work.
- Forests, trees, and spanning trees were organized by the properties undirected, acyclic, connected, and spanning.
- The lecture emphasized that a spanning tree has exactly $|E| = |V| - 1$ edges, but simply choosing $|V| - 1$ edges is not enough unless the graph is also acyclic and connected.
- Jackie reminded students that these definitions matter for the closed-book written test and exam, and assigned the taxonomy table as self-study.

Roadmap

Assignment and tutorial reminders

Jackie opened by reminding students to get started with Assignment 1 and to ask for guidance early. Tutorial 5 on the following Monday was announced as a tutorial focused mainly on mathematical induction, including properties from the slides and an induction example on trees. Jackie also planned, if time allowed, to review how to construct trees in Java for JUnit-style examples and to revisit BST deletion.

Recap: paths, cycles, and legitimate subgraphs

The lecture briefly reviewed path and cycle terminology from the previous lecture. A cycle means that a traversal returns to the same vertex, and a path can contain a cycle even if its first and last vertices are different. Jackie also reviewed the subgraph definition: a subgraph is made from subsets of vertices and edges, but it must still be a legitimate graph. Vertices may appear without incident edges, but an edge cannot be included unless both endpoint vertices are included.

Spanning subgraphs

The first new definition was a spanning subgraph. For a graph $G = (V, E)$, a subgraph $G' = (V', E')$ is spanning when $V' = V$ and $E' \subseteq E$. Jackie described this as a subgraph that "spans through" or covers all original vertices. The edge set can be empty or partial, as long as the result remains a valid graph.

Jackie used several copies of the same graph with vertices A, B, C, D, E, F. The first copy kept all vertices and no edges; it was spanning but disconnected. Another copy kept all vertices and a few edges; it was still spanning, but not necessarily connected. Later copies showed that a spanning subgraph can be connected, cyclic, acyclic, or a spanning tree depending on which edges are kept. The major lesson is that spanning is about vertex coverage, not reachability or absence of cycles.

Subgraphs versus spanning subgraphs

The lecture then connected the graph definitions to set language. Every spanning subgraph is a subgraph, but not every subgraph is spanning. A subgraph that is not spanning has $V' \subsetneq V$, e.g., $|V'| < |V|$, so at least one original vertex has not been included. This distinction matters because many later definitions add one property at a time, and students must not collapse them into a single idea.

Connected graphs

Jackie next defined connectivity. A graph G is connected when, for every pair of distinct vertices $x, y \in V$, there is a path between x and y. The negation is also important: a graph is not connected when there exist distinct vertices x, y that are not reachable from each other. Jackie stressed that connectivity is based on paths, not only on direct edges.

A key contrast was that a subgraph can be connected but not spanning, and a spanning subgraph can be disconnected. The empty-edge spanning subgraph $(V, \varnothing)$ is the simplest counterexample to the mistaken idea that spanning implies connected.

Connected components

A connected component, abbreviated CC, was defined as a maximal connected subgraph of G. Maximal means that no further connection can extend it into a larger connected subgraph with more vertices. Jackie used a drawing with three separated pieces and identified three connected components.

The important property is that vertices from two distinct connected components cannot be reachable from each other. If such a path existed, the two components would actually be one larger connected component. Jackie also noted that a graph is connected if and only if it has exactly one connected component; if it is disconnected, it has more than one. In the three-component example, two new edges are enough to make the graph connected by linking the components together.

Forests

The next definition was forest. A forest is an undirected, acyclic subgraph. It does not have to be connected. Jackie linked the absence of cycles to the idea that any two vertices are connected by at most one path. Zero paths between a pair of vertices is allowed in a forest, because a forest may have multiple connected components.

Jackie also explained the opposite direction: if two or more different paths exist between two vertices x and y, then the graph contains a cycle. One can travel from x to y along one path and return from y to x along another, forming a cycle. Therefore such a subgraph is not a forest.

Trees and spanning trees

A tree was defined as a connected forest. Equivalently, it is an undirected, connected, acyclic subgraph. This also explains the relation between forests and trees: every tree is a forest, but a forest with more than one connected component is not a tree. Jackie pointed out that individual acyclic connected components of a forest can themselves be trees.

Finally, Jackie defined a spanning tree in two equivalent ways: a connected spanning subgraph that is acyclic, or a spanning subgraph that is also a tree. Jackie then introduced the important property that if G is a spanning tree, then $|E| = |V| - 1$. For the six-vertex example, a spanning tree should have five edges. The notes showed several valid examples and one invalid attempt: choosing five edges can still fail if those edges create a cycle and leave the graph not properly connected as a spanning tree.

Taxonomy table as self-study

The lecture ended with a table exercise. Students were asked to fill in whether each graph type – subgraph, spanning subgraph, forest, tree, and spanning tree – must satisfy each property: undirected, acyclic, connected, and spanning. Jackie framed this as a useful way to consolidate the definitions after a vocabulary-heavy lecture.

Key Terms, Notation, and Structures

- **Subgraph**: For $G = (V, E)$, a graph $G' = (V', E')$ with $V' \subseteq V$, $E' \subseteq E$, and with every edge in E' having both endpoints in V'.
- **Spanning subgraph**: A subgraph $G' = (V', E')$ of $G = (V, E)$ such that $V' = V$; it covers all original vertices.
- **Subgraph but not spanning**: A subgraph with $V' \subsetneq V$, so at least one vertex of the original graph not included.
- **Connected graph**: A graph in which every pair of distinct vertices is connected by at least one path.
- **Disconnected graph**: A graph with at least one pair of distinct vertices that are not reachable from each other.

- **Connected component (CC)**: A maximal connected subgraph; it cannot be extended into a larger connected subgraph by adding more reachable vertices.
- **Maximal**: Cannot be made larger while preserving the required property; for CCs, this means no larger connected subgraph contains it.
- **Cyclic**: Contains at least one cycle.
- **Acyclic**: Contains no cycle.
- **Forest**: An undirected, acyclic subgraph; it may be disconnected.
- **Tree**: A connected forest, equivalently an undirected, connected, acyclic subgraph.
- **Spanning tree**: A spanning subgraph that is also a tree; equivalently, a connected, acyclic spanning subgraph.
- **At most one path**: In a forest, any two vertices have either zero or one path between them; two or more distinct paths imply a cycle.
- **Spanning-tree edge count**: If G is a spanning tree, then $|E| = |V| - 1$. This is a necessary property that must be used together with connectivity and acyclicity.
- **Taxonomy properties**: The four properties Jackie asked students to organize are undirected, acyclic, connected, and spanning.

Key Examples and Demonstrations

Spanning subgraphs on the six-vertex graph

Setup: The notes used a graph with vertices A, B, C, D, E, F and labelled edges, then created several spanning subgraphs by keeping all vertices while choosing different edge sets.

Main point: A spanning subgraph can be disconnected, connected, cyclic, or acyclic. The condition $V' = V$ says only that all vertices are included; it says nothing by itself about paths or cycles.

What to practise: Redraw the examples G_1, G_2, G_3, G_4 from the notes. For each one, mark whether it is spanning, connected, and cyclic or acyclic, and justify each label by pointing to vertices, paths, or cycles.

Subgraph but not spanning

Setup: The notes asked for a condition on $G' = (V', E')$ that makes it a subgraph but not a spanning subgraph of $G = (V, E)$.

Main point: The condition is $V' \subsetneq V$ while still satisfying the ordinary subgraph requirements. This connects graph terminology to subset and proper-subset reasoning.

What to practise: Construct two examples from the same original graph: one spanning subgraph and one non-spanning subgraph. Then explain the difference using $V' = V$ versus $V' \subsetneq V$, not just by looking at the drawing.

Connectedness and the spanning counterexample

Setup: Jackie considered whether a spanning subgraph must also be connected and used $(V, \varnothing)$ as a counterexample.

Main point: Spanning and connectivity are independent. A graph with all vertices but no edges is spanning, but it has no paths between distinct vertices.

What to practise: Give your own example of a connected subgraph that is not spanning, and a spanning subgraph that is not connected. For each one, identify the vertex pair or omitted vertex that proves your claim.

Connected components in a disconnected graph

Setup: The notes showed a graph separated into three connected components.

Main point: Each component is connected internally and maximal: if two components had any path between them, they would not be distinct components. With three components, two added edges can connect them into one connected graph.

What to practise: On the page-7 diagram, choose one vertex from each pair of different components and explain why no path can exist between them. Then sketch two possible new edges that would make the whole graph connected.

Forests and path multiplicity

Setup: The forest example showed an undirected acyclic subgraph with multiple connected pieces, then contrasted it with a situation where two paths exist between the same vertices x and y.

Main point: A forest does not require connectivity, so zero paths between some vertices is allowed. But two or more distinct paths between a pair of vertices create a cycle, which violates the forest definition.

What to practise: Pick two vertices in the forest example that are not reachable and explain why that does not break the definition. Then draw two distinct paths between vertices x and y and trace the resulting cycle.

Tree and spanning-tree examples

Setup: Jackie defined a tree as a connected forest and a spanning tree as a spanning subgraph that is also a tree. The notes then showed several spanning-tree candidates on the same six-vertex graph.

Main point: A spanning tree on six vertices has five edges and must be connected and acyclic. The invalid candidate in the notes illustrates that an edge count alone does not guarantee a spanning tree if a cycle is introduced.

What to practise: Draw at least two valid spanning trees for the lecture graph. Then draw one invalid five-edge choice and explain exactly which required property fails.

Common Pitfalls and Misconceptions

Assuming spanning implies connected

Pitfall: Thinking that a spanning subgraph must be connected because it includes all vertices.

Why it causes problems: Spanning says nothing about edges or paths. The subgraph $(V, \varnothing)$ includes all vertices but is disconnected unless there is only one vertex.

How to avoid it: Check spanning by asking whether $V' = V$. Check connectivity separately by asking whether every pair of distinct vertices has a path.

Assuming connected implies spanning

Pitfall: Thinking that a connected subgraph must cover all vertices of the original graph.

Why it causes problems: A subgraph can be connected among the vertices it includes while still omitting other original vertices.

How to avoid it: After checking paths inside G', separately compare V' with the original vertex set V.

Including edges without endpoints

Pitfall: Choosing an edge for E' without also including both of its endpoint vertices in V'.

Why it causes problems: The result is not a legitimate graph, so it cannot be a valid subgraph.

How to avoid it: For every edge you include, circle both endpoints and confirm that both are in V'.

Treating $|E| = |V| - 1$ as sufficient by itself

Pitfall: Deciding that a graph is a spanning tree only because it has one fewer edge than vertices.

Why it causes problems: A five-edge choice on six vertices can still create a cycle or leave the graph disconnected, so it may fail to be a spanning tree.

How to avoid it: Use the edge count as a necessary check, then also verify spanning, connectivity, and acyclicity.

Thinking every forest is a tree

Pitfall: Treating a forest as if it must be connected.

Why it causes problems: A forest can have multiple connected components; it only requires undirectedness and no cycles.

How to avoid it: Remember: tree = connected forest. If a forest has more than one connected component, it is not a tree.

Confusing multiple paths with shortest paths

Pitfall: Bringing shortest paths into the definition of a cycle or forest.

Why it causes problems: The lecture's forest test is about the existence of more than one path between two vertices, not about which path is shortest.

How to avoid it: For cycle and forest questions, ask only whether zero, one, or two-or-more paths exist between a pair of vertices.

What You Should Be Able to Do

After this lecture, you should be able to:

- define subgraph, spanning subgraph, connected graph, connected component, forest, tree, and spanning tree;
- decide whether a proposed $G' = (V', E')$ is a legitimate subgraph of $G = (V, E)$;
- distinguish $V' \subseteq V$, $V' \subsetneq V$, and $V' = V$ in the context of subgraphs and spanning subgraphs;
- construct examples showing that spanning does not imply connected, and connected does not imply spanning;
- use reachability between every pair of distinct vertices to test connectivity;
- identify connected components and explain why different components cannot have paths between them;
- explain why a graph is connected exactly when it has one connected component;
- explain why a forest may be disconnected but must be acyclic;
- use the "at most one path" idea to recognize why two distinct paths between vertices imply a cycle;
- distinguish a forest from a tree and a tree from a spanning tree;
- apply the spanning-tree edge-count property $|E| = |V| - 1$ without treating it as the only requirement;
- complete the taxonomy table by deciding which graph types must be undirected, acyclic, connected, or spanning.

Suggested Practice

- Fill in the taxonomy table from the final page of the notes, using a check mark when a property must hold and a question mark when it may or may not hold.

- Redraw the lecture graph with vertices A, B, C, D, E, F, then construct at least four subgraphs: one empty-edge spanning subgraph, one disconnected spanning subgraph, one connected cyclic spanning subgraph, and one spanning tree.
- For each constructed subgraph, write three short justifications: whether it is spanning, whether it is connected, and whether it is acyclic.
- Take a disconnected graph and label all its connected components; then add the minimum number of edges needed to make it connected.
- Practise the forest test by choosing pairs of vertices and deciding whether they have zero, one, or more than one path between them.
- Draw several five-edge candidates on a six-vertex graph and decide which are spanning trees; be sure to catch candidates that have the right edge count but contain a cycle.
- Continue Assignment 1 work and bring questions early; for graph preparation, keep reviewing the linked-list background that Jackie mentioned for later assignments.
- Before the closed-book written test or exam, rehearse the definitions aloud without looking at the table, then check them against the official notes.

Where to Review

- **iPad/PDF notes, page 2**: review the reminders about Assignment 1, Tutorial 5 on mathematical induction, BST deletion practice, and pre-study materials.
- **iPad/PDF notes, page 3**: review the definition of spanning subgraph and the examples G_1 through G_4, especially the distinction between spanning, connected, cyclic, and acyclic.
- **iPad/PDF notes, page 4**: review the set-theoretic relation between subgraphs and spanning subgraphs, including the proper-subset condition for a subgraph that is not spanning.
- **iPad/PDF notes, page 5**: review the formal connectivity definition and the counterexample showing that a spanning subgraph need not be connected.
- **iPad/PDF notes, pages 6–7**: review connected components, maximality, the example with three CCs, and the connection between connected graphs and the number of CCs.
- **iPad/PDF notes, page 8**: review forests, acyclicity, the "at most one path" condition, and the two-path argument for finding a cycle.
- **iPad/PDF notes, page 9**: review trees as connected forests and the relation between forests, trees, and connected components.
- **iPad/PDF notes, page 10**: review the two definitions of spanning tree, the property $|E| = |V| - 1$, and the valid versus invalid spanning-tree candidates.

- **iPad/PDF notes, page 11**: complete the taxonomy table as self-study and use it as a definition checklist.
- **Lecture recording**: revisit the spoken explanations around spanning versus connected, connected components, and the forest/tree/spanning-tree distinctions if the definitions feel similar.

Closing Reminder

This lecture was definition-heavy, but the definitions are not just vocabulary. They determine which graph properties, proofs, algorithms, and later implementations are valid. Use this digest only as a roadmap: return to Jackie's notes and the recording, redraw the examples, complete the taxonomy table, and practise deciding properties from actual subgraphs. For the closed-book written test and exam, you should be able to state the definitions and apply them without relying on this digest.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 11, June 9

Spanning Trees, Graph Properties, and Tracing DFS on Undirected Graphs

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 9

Closing Reminder 10

Main Ideas

- The lecture finished the first pass through core graph-structure vocabulary by connecting spanning trees, forests, trees, connectedness, acyclicity, and edge-count properties.
- Jackie reviewed how to use examples and witnesses to test graph-property statements before deciding whether to prove or disprove them.
- A spanning tree on a graph with n vertices must use $n - 1$ edges, but choosing $n - 1$ edges from the original graph is only a candidate-selection process, not a guarantee of a spanning tree.
- The choose operator $\binom{m}{n-1}$ gives an upper bound on possible spanning trees when the original graph has m edges, because some selected edge sets may create cycles.
- The lecture introduced graph traversal, especially depth-first search (DFS) on undirected graphs, using a LIFO stack, discovery edges, back edges, and explicit push/pop tracing.
- The DFS trace connected earlier tree vocabulary to algorithms: on a tree, DFS behaves like pre-order traversal; on a connected graph, the discovery edges produced by DFS form a spanning tree.
- Assignment 1 is active, the first test is scheduled for June 22, the programming-test guide will be released soon, and the Monday tutorial before reading week will focus on DFS tracing.

Roadmap

Administrative setup and upcoming support

Jackie began by directing students to focus seriously on Assignment 1 and to ask for help early if any part becomes difficult. Reading week begins on Tuesday, but the Monday tutorial before reading week will still run and will be led by Jackie; its main purpose will be additional DFS tracing practice. Jackie also reminded students that the first test is on June 22, the Monday after reading week, and that a programming-test guide will be released soon. Extra office-hour support, including a possible Zoom hour, will be announced later.

Spanning trees and the choose-operator upper bound

The first technical return point was the spanning-tree material from the previous lecture. A spanning tree was reviewed as a connected, acyclic spanning subgraph, and the already-proved property $|E| = |V| - 1$ was used to reason about candidate spanning trees. In the notes example, the original graph has six vertices and eight edges, so any spanning tree must choose five edges. The number $\binom{8}{5} = 56$ counts all possible ways to choose five edges from eight, but Jackie emphasized that this is only an upper bound on the number of spanning trees: some five-edge choices create a cycle and therefore are not trees.

Taxonomy of graph structures

Jackie then filled in the graph taxonomy table comparing subgraph, spanning subgraph, forest, tree, and spanning tree against the properties undirected, acyclic, connected, and spanning. The important habit is not to memorize the table mechanically, but to derive each entry from definitions. For example, a forest must be acyclic but need not be connected or spanning; a tree must be connected and acyclic but need not be spanning; a spanning tree must satisfy all the relevant properties.

Connected components inside a forest

A student asked for a reminder about connected components, and Jackie returned to the forest example from the previous lecture. The key point was that a forest may have more than one connected component, but each connected component must still be acyclic. If a forest happens to be connected, then it is a tree; if it is not connected, it remains a forest but is not a tree.

Quick spanning-tree identification exercise

The lecture included a multiple-choice-style exercise asking which displayed subgraph is a spanning tree. The correct option was the one that was simultaneously spanning, acyclic, and connected. Jackie also used the wrong options diagnostically: one option was spanning and connected but cyclic, while another was connected and acyclic but not spanning. This reinforced that no single property is enough by itself.

Proving versus disproving graph-property statements

Jackie revisited the list of graph properties relating structure to $|V|$ and $|E|$. The first property, "if G is a spanning tree, then $m = n - 1$," had been proved in Tutorial 5. Jackie then showed how to disprove the general statement "if G contains a cycle, then $m \geq n$" by giving a witness: a triangle together with an isolated vertex has a cycle, but has $m = 3$ and $n = 4$, so $m \geq n$ is false. The lecture also clarified that adding connectedness as an assumption changes the situation: for connected cyclic graphs, $m \geq n$ does hold.

Roadmap for graph traversal algorithms

The lecture then moved from graph vocabulary to traversal. Jackie previewed three related topics: DFS on undirected graphs, DFS on directed graphs, and BFS on undirected graphs. For undirected DFS, the edge types are discovery edges and back edges. Directed DFS will later require vertex colors – white, gray, and black – and four edge types: discovery, back, forward, and cross. BFS on undirected graphs will use discovery and cross edges. The rest of this lecture focused on the basic undirected DFS case.

DFS on a tree

Jackie introduced DFS as a process that starts from a chosen vertex, follows incident edges as far as possible, and backtracks only when the current vertex cannot extend to an unvisited vertex. In the first example the graph was itself a tree. With adjacent vertices visited in alphabetic order, the trace used a LIFO stack to record the active path. Since the graph had no cycle, every edge encountered by DFS was a discovery edge. Jackie connected this case to pre-order traversal: on a tree, DFS corresponds to visiting the parent before continuing through children.

DFS on an undirected graph with cycles

The main trace used a graph with vertices A, B, C, D, E, start vertex A, and alphabetic visiting of adjacent vertices. Jackie tracked four kinds of information that could appear in tests: discovery edges, back edges, push/discovery order of vertices, and pop/backtracking order of vertices. In the trace, an edge to an unvisited vertex became a discovery edge; an edge to a vertex already visited and still on the current DFS path became a back edge. The example showed discovery edges forming a spanning tree of the connected graph and back edges revealing cycles.

Implementation and assessment connections

Jackie noted that Assignment 2 will involve implementing graphs using an edge-list strategy and implementing DFS. Students may implement DFS recursively, but Jackie recommended the iterative, stack-based style because it matches the paper trace and is easier to reason about. Jackie also clarified a student question: visited status belongs to vertices, while discovery/back classification belongs to edges.

Key Terms, Notation, and Structures

- **Spanning tree**: A connected, acyclic spanning subgraph; equivalently, a spanning subgraph that is also a tree.
- **Tree edge / discovery edge**: In DFS, an edge used to reach a vertex that has not yet been visited.
- **Back edge**: In undirected DFS, an edge that leads to an already visited vertex on the current DFS path; its presence indicates a cycle.
- **Forest**: An undirected acyclic graph or subgraph; it may have multiple connected components.
- **Tree**: A connected forest; it is acyclic and connected, but not necessarily spanning relative to a larger graph.
- **Connected component**: A maximal connected subgraph; in a forest, each connected component is acyclic and is itself a tree.

- **Choose operator**: $\binom{n}{i} = \frac{n!}{i!(n-i)!}$, read as "$n$ choose i", counts the number of ways to choose i items from n items.
- **Candidate spanning-tree edge sets**: In a graph with m edges and n vertices, $\binom{m}{n-1}$ counts all ways to select $n-1$ edges, but this can include cyclic or disconnected choices.
- **DFS stack**: A LIFO stack used in the iterative DFS trace to remember the current path and support backtracking.
- **Visited vertex**: A vertex marked when it is first encountered; Jackie emphasized that visited status is stored at the vertex level.
- **Marked edge**: An edge classified during DFS as discovery or back in the undirected case; edge classification is separate from vertex visited status.
- **Backtracking / popping**: The moment when the current vertex is removed from the stack because all of its incident edges have already been explored or marked.
- **Alphabetic visiting order**: The lecture's DFS traces assume adjacent vertices are considered in alphabetic order unless an exercise explicitly states an exception.
- **Directed DFS roadmap**: Later DFS on directed graphs will use vertex colors white → gray → black and discovery, back, forward, and cross edges.
- **BFS roadmap**: Later BFS on undirected graphs will use discovery and cross edges, not the same edge classification pattern as undirected DFS.

Key Examples and Demonstrations

Using $\binom{8}{5}$ as an upper bound

Setup: The notes reviewed a graph with six vertices and eight edges. Since a spanning tree on six vertices must have five edges, Jackie considered $\binom{8}{5}$.

Main point: The value 56 counts five-edge selections, not guaranteed spanning trees. Some five-edge selections create cycles, so the actual number of spanning trees is strictly smaller in the example.

What to practise: Pick several five-edge subsets from the graph and classify each one as a spanning tree or not by checking spanning, connectedness, and acyclicity.

The taxonomy table

Setup: The table compared subgraph, spanning subgraph, forest, tree, and spanning tree against four properties: undirected, acyclic, connected, and spanning.

Main point: The table is a summary of definitions, not a memorization shortcut. For instance, a forest is acyclic but may be disconnected, while a tree is a connected forest.

What to practise: Recreate the table from definitions without looking, then justify every check mark or question mark in one sentence.

Disproving a false graph-property implication

Setup: Jackie considered the statement: if a graph contains a cycle, then $m \geq n$.

Main point: The statement is false in general. A triangle plus one isolated vertex is cyclic, has $m = 3$, has $n = 4$, and therefore violates $m \geq n$. This is a witness for disproof.

What to practise: For each property statement on the graph-properties slide, first decide whether a proof or a counterexample is needed; then either sketch a proof idea or construct a valid witness.

DFS on a tree

Setup: The first DFS trace used a tree rooted visually at A, with adjacent vertices visited in alphabetic order.

Main point: DFS uses a stack and behaves like pre-order traversal on a tree. Because there are no cycles, every edge encountered is a discovery edge.

What to practise: Redraw the tree, trace the stack push/pop sequence, and explain why no back edge can appear.

DFS on a cyclic undirected graph

Setup: The main DFS example started at A in an undirected graph with vertices A, B, C, D, E, using alphabetic order for adjacent vertices.

Main point: The trace tracked the order of edge identification, the discovery edges, the back edges, the vertex push order, and the vertex pop order. Discovery edges led to new vertices; back edges led to visited vertices on the current DFS path.

What to practise: Reproduce the trace without looking, then redo the variants where the first edge from A is forced to be AC, AD, or AE. Compare how discovery and back edges change.

Discovery edges as a spanning tree

Setup: After completing DFS on the connected example graph, Jackie counted the discovery edges and compared them with the number of vertices.

Main point: In a connected graph, the DFS discovery edges form a spanning tree. If the original graph is disconnected, a DFS started in one component only gives a spanning tree of that connected component.

What to practise: For a disconnected graph, start DFS in one component and identify exactly which vertices and edges belong to the DFS tree for that component.

Common Pitfalls and Misconceptions

Confusing an upper bound with an exact count

Pitfall: Treating $\binom{m}{n-1}$ as the exact number of spanning trees.

Why it causes problems: The formula counts all ways to choose $n-1$ edges, including choices that create cycles or fail to connect all vertices.

How to avoid it: Call $\binom{m}{n-1}$ an upper bound unless you have separately verified that every selected edge set is a spanning tree.

Forgetting that "tree" does not mean "spanning tree"

Pitfall: Assuming every tree automatically spans the original graph.

Why it causes problems: A tree can be a connected acyclic subgraph on only some of the vertices. Spanning requires covering all vertices of the original graph.

How to avoid it: Check the vertex set separately: a spanning tree must include all original vertices, be connected, and be acyclic.

Using the wrong direction of implication

Pitfall: Reversing statements such as "if G is a spanning tree, then $m = n - 1$".

Why it causes problems: The converse is not automatically true. A graph can have $n-1$ edges and still fail to be a spanning tree because of cycles or disconnectedness.

How to avoid it: Translate the English statement into a formal implication before proving or disproving it, and use witnesses when the statement is false.

Assuming cyclic implies $m \geq n$ without connectedness

Pitfall: Believing that any cyclic graph must have at least as many edges as vertices.

Why it causes problems: A cyclic graph can have isolated vertices or multiple components. The triangle-plus-isolated-vertex example is cyclic but has $m = 3 < n = 4$.

How to avoid it: Check all assumptions. The connectedness assumption changes the edge-count conclusion.

Popping from the DFS stack too early

Pitfall: Backtracking from a vertex before all of its incident edges have been considered.

Why it causes problems: The trace may miss edges, misclassify edge types, or produce the wrong push/pop order.

How to avoid it: At the current vertex, inspect incident edges in the required order and pop only after every incident edge has been identified as discovery or back.

Mixing vertex state with edge classification

Pitfall: Treating "visited" as an edge property or treating discovery/back as a vertex property.

Why it causes problems: DFS implementation and tracing become unclear, especially when an already visited vertex is reached through a new edge.

How to avoid it: Store visited status on vertices; classify each explored edge separately as discovery or back in the undirected case.

Ignoring the specified neighbor order

Pitfall: Tracing DFS using an arbitrary order when the question specifies alphabetic order or an exception.

Why it causes problems: DFS can still be valid algorithmically, but your trace will not match the expected answer for the test or exercise.

How to avoid it: Before tracing, write the start vertex and the neighbor-order rule beside the graph.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain why a spanning tree must have $n-1$ edges when it spans n vertices;
- use $\binom{m}{n-1}$ to compute an upper bound on the number of possible spanning trees;
- distinguish subgraphs, spanning subgraphs, forests, trees, and spanning trees using acyclicity, connectedness, and spanning;
- explain why a forest can be disconnected and why each connected component of a forest is acyclic;
- identify whether a displayed subgraph is a spanning tree by checking all required properties;
- disprove a false graph-property statement by constructing a valid witness;
- explain why adding connectedness can change whether an edge-count property is true;
- describe the high-level differences among undirected DFS, directed DFS, and undirected BFS as previewed in the lecture;
- trace iterative DFS on an undirected graph using a stack and a specified neighbor order;
- list discovery edges, back edges, push order, pop order, and edge-identification order from a DFS trace;
- explain why DFS on a tree has only discovery edges and corresponds to pre-order traversal;
- explain why DFS discovery edges form a spanning tree when the graph is connected;
- prepare questions for Assignment 1 and for the June 22 test using the Monday tutorial and reading-week support.

Suggested Practice

- Watch the posted short video on the choose operator, then compute a few values such as $\binom{8}{5}$ and explain what the result counts in graph terms.
- Recreate the taxonomy table from the notes and justify each entry from definitions rather than memory.
- Draw at least three five-edge selections from the six-vertex, eight-edge graph and decide which are spanning trees.
- For the properties on the structure-vs.-$|V|$-and-$|E|$ slide, decide which ones should be proved and which ones should be disproved.
- Construct your own witness for a false implication, then write explicitly which part of the implication is true and which part is false.
- Retrace the DFS tree example and verify that the push order matches a pre-order traversal.
- Retrace DFS Example 1(a) from start vertex A, recording discovery edges, back edges, push order, pop order, and edge-identification order.
- Attempt DFS Examples 1(b), 1(c), and 1(d) before the next class, paying attention to the stated exception about which edge from A is visited first.
- For Assignment 1, keep working early and bring concrete questions to Jackie if a tree, deletion, or testing issue is blocking you.
- Before the Monday tutorial, prepare at least one DFS trace question and one Assignment 1 or test-preparation question.

Where to Review

- **iPad/PDF notes, pages 1–3:** lecture title, spanning-tree review, the $|E| = |V| - 1$ property, and the choose-operator setup for bounding spanning trees.
- **iPad/PDF notes, page 4:** the taxonomy table for subgraph, spanning subgraph, forest, tree, and spanning tree.
- **iPad/PDF notes, page 5:** the quick spanning-tree identification exercise and the reasons the incorrect options fail.
- **iPad/PDF notes, page 6:** graph-property examples involving spanning trees, forests, connected acyclic graphs, connected cyclic graphs, and a counterexample witness.
- **iPad/PDF notes, page 7:** the roadmap for DFS and BFS edge classifications.

- **iPad/PDF notes, pages 8–10:** DFS definition, stack-based tracing, tree example, undirected cyclic-graph example, and discovery/back edge marking.
- **iPad/PDF notes, pages 11–13:** extra DFS trace variants where the first edge from A is forced to be AC, AD, or AE.
- **Lecture recording:** revisit the opening logistics for Assignment 1, the June 22 test reminder, and reading-week support; then focus on the choose-operator discussion, the counterexample for the cyclic-property claim, and the DFS traces.
- **Tutorial 5 materials:** review the mathematical induction proofs and the posted material on one-line binary-tree construction if you are still working through Assignment 1 starter tests.
- **Upcoming Monday tutorial:** use it for DFS tracing practice and for questions about Assignment 1 or test preparation.

Closing Reminder

This digest is only a roadmap. To learn this lecture well, you still need to review Jackie's iPad notes, watch the relevant parts of the recording, and actively retrace the graph examples. In particular, DFS tracing is not something to learn passively: draw the graph, choose the start vertex and neighbor order, maintain the stack, mark each edge, and check your push/pop sequence. Bring concrete questions to the Monday tutorial and use reading week to connect the graph vocabulary, proof/disproof habits, and DFS tracing before the June 22 test.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 12, June 11

DFS on Directed Graphs: Vertex Colors & Timestamps, Edge Classification

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 7

Suggested Practice 8

Where to Review 8

Closing Reminder 9

Main Ideas

- The lecture moved from DFS on undirected graphs to DFS on directed graphs, where tracing requires more information than the stack alone.
- A single DFS run begins from a designated start vertex and produces a DFS tree; traversing a whole graph may require multiple DFS runs, producing a DFS forest.
- Directed DFS uses vertex colors: white for unvisited, gray for discovered and still on the stack, and black for finished and popped.
- Each vertex has two timestamps, a discovery time $\mathrm{d}(x)$ and a finish time $\mathrm{f}(x)$, recorded by a global timer that advances on pushes and pops.
- Directed DFS has four edge categories: discovery/tree, back, forward, and cross; classification depends on the runtime color and the DFS-tree relationship between vertices.
- The main example traced a directed graph on vertices A, B, C, D, E, recording the stack, colors, timestamps, and edge types in one complete run starting from A.
- Jackie emphasized that students should practise this trace over the reading week because the same example will later be used to state more general properties of d and f.

Roadmap

Terminology for directed DFS

The lecture began with a compact overview of the terms students need before tracing DFS on directed graphs: DFS run, start vertex, DFS tree, DFS forest, ancestor, proper ancestor, descendant, proper descendant, vertex colors, timestamps, and four directed edge categories. Jackie framed these terms as essential preparation for the properties of DFS that will be generalized after the reading week.

DFS run, DFS tree, and DFS forest

A DFS run is one execution of DFS starting from a designated unvisited start vertex. In one run, the discovery edges form a DFS tree whose root is the start vertex. If the whole graph cannot be covered by one run, then DFS must be restarted from another unvisited vertex; the collection of resulting DFS trees is a DFS forest. The lecture only fully traced one DFS run, but Jackie noted that multiple-run cases will be discussed later.

Ancestor and descendant language

Jackie used the DFS tree to define ancestor and descendant relationships. In the example tree, the ancestors of D include D, C, B, A, while the proper ancestors of D exclude D itself. Similarly, descendants include the vertex itself, while proper descendants exclude

it. This language matters because forward, back, and cross edges are defined relative to ancestor/descendant relationships in the DFS tree.

Vertex colors and the stack

The lecture then introduced the standard color lifecycle white → gray → black. A white vertex has not been visited and is not on the stack. A gray vertex has been discovered, is still on the stack, and is not finished. A black vertex is finished, backtracked, and popped. Jackie emphasized that the gray vertices match the current stack contents during the trace.

Discovery and finish timestamps

Each vertex x has a discovery time $\mathrm{d}(x)$, recorded when x is pushed and turns gray, and a finish time $\mathrm{f}(x)$, recorded when x is popped and turns black. The global timer starts at 0 and increments whenever a vertex is pushed or popped. For every vertex, $\mathrm{f}(x) > \mathrm{d}(x)$, because a vertex must be discovered before it can finish.

Four directed edge types

For directed DFS, non-discovery edges are not all back edges. A discovery edge goes to a white vertex and becomes part of the DFS tree. A back edge goes from a vertex to a gray ancestor. A forward edge goes from a vertex to a black descendant but is not itself a tree edge. A cross edge goes to a black vertex that is neither an ancestor nor a descendant in the relevant DFS tree relationship. Jackie first gave a visual intuition on the A, B, C, D, E graph, then made the classification precise during the trace.

Setting up the full trace

Before tracing, Jackie set up several pieces of bookkeeping: a push/pop stack table, a timestamp table with d and f for each vertex, and a color table listing the white, gray, and black vertices at each timer value. Because the example has five vertices, there are ten push/pop events, so the timer table runs from the initial state $t = 0$ through $t = 10$.

The directed DFS trace from A

The run starts with all vertices white and begins at A. Following the directed edges and the chosen order, the trace discovers A, B, C, D, producing discovery edges $A \to B$, $B \to C$, and $C \to D$. When $D \to A$ is handled, A is still gray and is an ancestor of D, so this edge is classified as a back edge. After D, C, B finish, the edge $A \to C$ is handled when C is already black and is a descendant of A, so it is a forward edge. Then $A \to E$ discovers E, and $E \to C$ is a cross edge because C is already finished and is not in an ancestor/descendant relationship with E. Finally E and A finish.

Why this trace matters

The trace is not only an example; it is the model for how students should handle directed DFS in assignments, tests, and later analysis. Jackie emphasized that stack-only tracing is not enough for directed graphs. Students must update the stack, colors, d-times, f-times, and edge classifications consistently.

Key Terms, Notation, and Structures

- **DFS run**: One execution or application of DFS starting from a designated unvisited start vertex.
- **Start vertex**: The first vertex of a DFS run; in the DFS tree for that run, it becomes the root.
- **DFS tree**: The tree formed by the discovery edges of one DFS run.
- **DFS forest**: A collection of DFS trees obtained when more than one DFS run is needed to traverse the graph.
- **Ancestor / proper ancestor**: A vertex is its own ancestor, but not its own proper ancestor; proper ancestor means strictly above the vertex in the DFS tree.
- **Descendant / proper descendant**: A vertex is its own descendant, but not its own proper descendant; proper descendant means strictly below the vertex in the DFS tree.
- **White vertex**: An unvisited vertex that has not yet been pushed onto the stack.
- **Gray vertex**: A discovered but unfinished vertex that is currently on the DFS stack.
- **Black vertex**: A finished vertex whose outgoing edges have been handled and that has been popped from the stack.
- **Global timer** t: A counter used to record when push and pop events occur during DFS.
- **Discovery time** $\mathrm{d}(x)$: The timer value recorded when vertex x is first discovered, pushed, and colored gray.
- **Finish time** $\mathrm{f}(x)$: The timer value recorded when vertex x is finished, popped, and colored black.
- **Healthiness property**: For every vertex x, $\mathrm{f}(x) > \mathrm{d}(x)$.
- **Discovery edge / tree edge**: A directed edge used to discover a white vertex; these edges form the DFS tree for the run.
- **Back edge**: An edge from the current vertex to a gray ancestor; in the example, $D \to A$ is a back edge.

- **Forward edge**: A non-tree edge from a vertex to a black descendant; in the example, $A \to C$ is a forward edge.
- **Cross edge**: An edge to a black vertex that is not an ancestor or descendant of the source in the DFS tree relation; in the example, $E \to C$ is a cross edge.
- **Outgoing edge**: In a directed graph, DFS explores edges leaving the current vertex; incoming edges are handled when their own source vertices are explored.
- **LIFO stack**: The stack discipline that causes descendants to finish before their ancestors.

Key Examples and Demonstrations

Visual classification before the trace

Setup: The notes show a directed graph on A, B, C, D, E, with a DFS tree rooted at A using discovery edges $A \to B$, $B \to C$, $C \to D$, and $A \to E$.

Main point: The same graph contains all four directed DFS edge types: discovery edges in the tree, $D \to A$ as a back edge, $A \to C$ as a forward edge, and $E \to C$ as a cross edge. Jackie corrected the visual discussion to make clear that $A \to C$ is forward, not back.

What to practise: Redraw the graph and DFS tree, then label each non-tree edge by asking whether it goes to an ancestor, descendant, or a vertex in another subtree.

Color lifecycle and timestamp setup

Setup: Before the full trace, Jackie set up a stack table, a d/f table, and a color table with white, gray, and black columns.

Main point: Directed DFS needs disciplined bookkeeping. A push event changes a vertex from white to gray and records d; a pop event changes a vertex from gray to black and records f. The gray column and the stack should agree at every step.

What to practise: Make a blank table for the five-vertex example before looking at the completed notes. Fill in the timer values, colors, stack contents, and d/f entries yourself.

Complete DFS trace from A

Setup: The main trace starts at A, with all vertices initially white and the timer initially 0.

Main point: The trace discovers A, B, C, D, handles $D \to A$ as a back edge, finishes D, C, B, handles $A \to C$ as a forward edge, discovers E, handles $E \to C$ as a cross edge, and finally finishes E and A. The resulting times are $\mathrm{d}(A) = 1$, $\mathrm{d}(B) = 2$, $\mathrm{d}(C) = 3$, $\mathrm{d}(D) = 4$, $\mathrm{d}(E) = 8$, and $\mathrm{f}(D) = 5$, $\mathrm{f}(C) = 6$, $\mathrm{f}(B) = 7$, $\mathrm{f}(E) = 9$, $\mathrm{f}(A) = 10$.

What to practise: Reproduce the trace from a blank copy of the graph. For each timer value, say what changed: which vertex was pushed or popped, which color changed, and whether an edge was classified.

Justifying each non-tree edge

Setup: Three non-tree edges were emphasized in the example: $D \to A$, $A \to C$, and $E \to C$.

Main point: $D \to A$ is back because it goes to a gray ancestor. $A \to C$ is forward because C is a finished descendant of A. $E \to C$ is cross because C is already finished and is neither an ancestor nor a descendant of E.

What to practise: For each edge, write a one-sentence classification justification using color and the DFS-tree relationship at the moment the edge is handled.

Multiple DFS runs as a future extension

Setup: Jackie briefly explained that a graph may require more than one DFS run if the first start vertex cannot reach every vertex.

Main point: One DFS run gives one DFS tree; multiple runs give a DFS forest. The detailed cases were postponed until after the reading week.

What to practise: Sketch a small directed graph where starting at one vertex cannot reach all vertices, then identify where a second DFS run would need to begin.

Common Pitfalls and Misconceptions

Using undirected DFS edge logic on directed graphs

Pitfall: Assuming that every non-discovery edge in directed DFS is a back edge.

Why it causes problems: Directed DFS has four edge categories. A non-tree edge could be back, forward, or cross, depending on the destination color and DFS-tree relationship.

How to avoid it: For each directed edge, ask whether the destination is white, gray, or black at the moment the edge is handled, and then check the ancestor/descendant relationship.

Tracing only the stack

Pitfall: Recording only the push/pop stack sequence and ignoring colors and timestamps.

Why it causes problems: Stack information alone is not enough to justify forward and cross edges or later mathematical properties involving d and f.

How to avoid it: Always maintain four views together: stack, white/gray/black sets, d/f table, and edge labels.

Classifying edges from the final picture only

Pitfall: Looking at the final DFS tree and trying to classify edges without considering when the edge was explored.

Why it causes problems: DFS edge classification is algorithmic. It depends on the runtime state of the destination vertex when the edge is handled.

How to avoid it: During a trace, classify each edge immediately when it is encountered, using the current color and available timestamp information.

Considering incoming edges from the current vertex

Pitfall: At a vertex x, treating incoming edges as choices for where DFS can go next.

Why it causes problems: In a directed graph, DFS from x follows outgoing edges from x, not edges that point into x.

How to avoid it: Before choosing the next edge, list only the outgoing edges of the current vertex.

Forgetting a timer or color update

Pitfall: Pushing or popping a vertex without incrementing the timer or updating the color table.

Why it causes problems: One missed update can make later edge classifications inconsistent, especially for forward and cross edges.

How to avoid it: Use a strict rule: every push records d and turns the vertex gray; every pop records f and turns the vertex black.

Ignoring the word "proper"

Pitfall: Listing a vertex itself as a proper ancestor or proper descendant.

Why it causes problems: Proper ancestor and proper descendant exclude the vertex itself, just as a proper subset excludes equality.

How to avoid it: First list the inclusive ancestors or descendants, then remove the vertex itself when the word "proper" appears.

What You Should Be Able to Do

After this lecture, you should be able to:

- define a DFS run, DFS tree, DFS forest, start vertex, and root of a DFS tree;
- identify ancestors, proper ancestors, descendants, and proper descendants from a DFS tree;
- explain the meaning of white, gray, and black vertices during directed DFS;
- explain why the gray vertices correspond to the current DFS stack;
- use a global timer to record $\mathrm{d}(x)$ and $\mathrm{f}(x)$ for every vertex;
- verify the basic property $\mathrm{f}(x) > \mathrm{d}(x)$ for each vertex after a trace;
- trace a directed DFS run from a specified start vertex using stack, color, and timestamp tables;
- classify directed DFS edges as discovery, back, forward, or cross;

- justify the classifications $D \to A$ as back, $A \to C$ as forward, and $E \to C$ as cross in the lecture example;
- explain why directed DFS must consider outgoing edges from the current vertex;
- explain why multiple DFS runs may be needed to traverse a whole graph, and why the result is called a DFS forest;
- prepare for later post-reading-week properties by using d, f, colors, and tree relationships consistently.

Suggested Practice

- Re-trace the full directed DFS example from A without looking at the completed notes, then compare your stack, color table, d/f table, and edge labels against the lecture notes.
- For every edge in the A, B, C, D, E example, write the exact reason for its classification: destination color, ancestor/descendant relation, and whether it became part of the DFS tree.
- Practise the update sequence aloud or in writing: push means discover, turn gray, increment timer, record d; pop means finish, turn black, increment timer, record f.
- Redraw the DFS tree from the discovery edges and list ancestors/proper ancestors and descendants/proper descendants for several vertices.
- Create a small directed graph where one DFS run cannot reach all vertices, and sketch how multiple DFS runs would produce a DFS forest.
- Review the previous undirected DFS trace and compare why undirected DFS needed only discovery/back edges while directed DFS needs four edge types.
- During the reading week, practise enough that the trace mechanics are automatic before Jackie generalizes the timestamp properties after the break.

Where to Review

- **iPad/PDF notes, page 1**: the lecture title and scope: DFS on directed graphs, notations, and tracing.
- **iPad/PDF notes, page 2**: the directed graph notation page, including DFS run, DFS tree, DFS forest, ancestor/descendant terminology, vertex colors, and timestamps.
- **iPad/PDF notes, page 3**: the full directed DFS trace, including the push/pop list, d/f table, white/gray/black table, and edge classifications.
- **Lecture recording**: review the explanation of white, gray, and black first, then the timestamp setup, then the full trace from A; do not skip directly to the final table.

- **Slides 31–32**: review the compact terminology and color/timestamp narrative that Jackie referenced at the start of the lecture.
- **Previous DFS lecture**: review undirected DFS and discovery/back edges so that the differences in the directed case are clear.
- **Upcoming reading-week practice**: use the directed DFS example as preparation for the later general properties of d and f and for the more complicated tracing work Jackie said will be assigned.

Closing Reminder

Directed DFS is a bookkeeping-heavy topic. This digest can remind you what to track, but it cannot replace doing the trace yourself. Revisit Jackie's iPad notes and the recording, redraw the graph, rebuild the tables, and practise classifying each edge from the runtime state. The goal is not to memorize the final table; the goal is to be able to produce and justify it independently.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 13, June 23

Edge List Strategy, Back-References, and DFS Properties

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 7

Suggested Practice 8

Where to Review 9

Closing Reminder 9

Main Ideas

- Assignment 2 was previewed as a two-part assignment: written tracing and induction work, plus Java implementation of graphs using an edge list strategy.
- An edge list graph stores its vertices and edges in separate doubly linked lists; the lecture focused on visualizing the run-time object and pointer structure rather than treating the representation as an abstract black box.
- Header and trailer sentinels simplify doubly linked list operations, while direct node references allow a non-sentinel node to be removed in $O(1)$ time.
- The key implementation idea is a back-reference: each vertex or edge stores the linked-list node that contains it, so its own list node can be located without a linear search.
- Removing a graph vertex is a cascading operation: remove the vertex itself and also every incident edge, which exposes an important limitation of the edge list representation.
- The lecture also began formalizing directed DFS properties by relating white, gray, and black colors to whether $\mathrm{d}(x)$ and $\mathrm{f}(x)$ are defined, including the property $\mathrm{d}(x) < \mathrm{f}(x)$.
- Jackie connected this work to the remaining graph assignments and to Programming Test 2, which will cover the programming components of Assignments 2 and 3.

Roadmap

Assignment 2 and the remaining graph work

Jackie said that Assignment 2 would be released on June 24. Its written part will involve tracing and one or more induction questions, while its programming part introduces graph implementation. Practise induction rather than only reading solutions, and expect to build the required graph methods yourself. Programming Test 2 is scheduled for July 20 and will be based on the programming parts of Assignments 2 and 3; it is also scheduled on the same Monday as the written test. Jackie confirmed that there is no Programming Test 3.

Why start with an edge list?

The lecture introduced three graph representation strategies: edge list, adjacency list, and adjacency matrix. The current focus is the edge list because it gives a concrete foundation for understanding the others, even though it is not always the most efficient representation. The key task is to be able to draw the run-time structure and then reason about what an operation must search or update.

Doubly linked nodes, lists, and sentinels

The provided support structure is a doubly linked list whose nodes store an element plus `prev` and `next` references. Every list has a *header* sentinel and a *trailer* sentinel. In an empty list, the header's next reference reaches the trailer, the trailer's previous reference reaches the header, and the sentinels do not count toward `size`. This is familiar linked-list machinery, but it is essential background for the graph implementation.

Removing a known linked-list node in constant time

Given the actual node x to remove, the list can reconnect x's predecessor directly to its successor, clear x's own two links, and decrement `size`. No traversal is needed, so this list-level removal is $O(1)$. Jackie emphasized that assigning a local variable such as `node` to `null` does not change the list structure; the neighboring objects must be updated.

Edge list graph objects and generic roles

An edge list graph maintains two doubly linked lists: one whose elements are vertices and one whose elements are edges. A vertex has one generic element type. An edge has an element type of its own, such as a weight, and references to its origin and destination vertices. The assignment uses a simplified class organization, but the intended design distinction remains: a graph has vertex and edge collections, and an edge connects two vertex objects.

Back-references: the central implementation mechanism

Each vertex stores `vertexListPosition`, a reference to the doubly linked node in the vertices list whose element is that vertex. Each edge similarly stores `edgeListPosition`. Thus, if `v1` is a vertex, `v1.vertexListPosition.getElement()` is the same vertex object, and the list can remove that node directly by passing `v1.vertexListPosition` to `vertices.remove`. Here, "position" means a node reference, not a numeric index.

Incident edges and cascading removal

The back-reference lets the graph remove the vertex's own node efficiently, but this alone does not complete `removeVertex`. Every edge incident to that vertex must also be removed from the edge list. In an edge list representation, finding those incident edges requires scanning the entire edge list, which motivates later comparison with adjacency lists. For the undirected graphs used in the assignments, edge equality must treat an edge between A and B as equivalent to one between B and A.

Directed DFS: colors and timestamps as formal state

The final portion returned to the directed DFS trace from the pre-reading-week tutorial. White means that neither discovery nor finish time has been assigned; gray means discovered and currently on the stack, so $\mathrm{d}(x)$ is defined but $\mathrm{f}(x)$ is not; black means finished, so both are defined. The lecture used vertex E as an example: it is gray at $t = 7$ with

$\mathrm{d}(E) = 7$ and undefined $\mathrm{f}(E)$, then black at $t = 10$ with $\mathrm{f}(E) = 10$. For every completed vertex, $\mathrm{d}(x) < \mathrm{f}(x)$.

Using the lecture digests appropriately

Jackie explained the purpose of the posted digests: they identify main ideas, key terms, examples, common mistakes, learning outcomes, and practice targets. Use them as a selective roadmap for cumulative review, especially for the final exam, but return to the recording, slides, iPad notes, code, and active practice for full details.

Key Terms, Notation, and Structures

- **Edge list:** A graph representation that stores a list of vertices and a separate list of edges. It is simple to visualize, but operations such as finding all incident edges can require a full scan of the edge list.
- **Adjacency list / adjacency matrix:** Two alternative graph representations to be compared later with the edge list strategy.
- `DLNode<E>`**:** A doubly linked node that stores an element of type `E` and references to its previous and next nodes.
- `DoublyLinkedList<E>`**:** A list with a `size` field and permanent header and trailer sentinel nodes.
- **Header and trailer sentinels:** Guard nodes that simplify list operations; their elements are not meaningful data and they are excluded from the list size.
- **Predecessor and successor:** The nodes immediately before and after a node in a doubly linked list. Updating their references is the core of node removal.
- $O(1)$ **node removal:** If the node reference is already known, remove it by reconnecting its predecessor and successor, clearing the removed node's links, and decrementing `size`; no list traversal is needed.
- **Vertex element type** V**:** The type of data stored in a vertex object.
- **Edge element type** E**:** The type of data stored on an edge, such as a weight; an edge also stores origin and destination vertex references.
- `vertexListPosition`**:** A back-reference from a vertex to the `DLNode` that contains it in the vertices list.
- `edgeListPosition`**:** The analogous back-reference from an edge to its node in the edges list.
- **Position:** In this context, a reference to a linked-list node, not a numeric list index.

- **Incident edge:** An edge that touches a given vertex. Removing a vertex requires removing all of its incident edges as well.
- **Undirected edge equality:** In an undirected graph, an edge joining A and B should be treated as equal to an edge joining B and A.
- **DFS colors and timestamps:** White means $\mathrm{d}(x)$ and $\mathrm{f}(x)$ are undefined; gray means only $\mathrm{d}(x)$ is defined; black means both are defined. The symbol $\perp$ denotes an undefined value.
- $\forall x \bullet x \in V \Rightarrow \mathrm{d}(x) < \mathrm{f}(x)$**:** A basic DFS property: a vertex must be discovered before it can finish.

Key Examples and Demonstrations

Sentinel-based empty doubly linked list

Setup: An empty `DoublyLinkedList` has `size = 0`, a header sentinel, and a trailer sentinel. The header points forward to the trailer, while the trailer points backward to the header.

Main point: Sentinels are permanent structural nodes, not vertex or edge data. They reduce special-case logic at the ends of the list.

What to practise: Draw the empty list and label every `prev/next` reference, including the two references that are `null`. Then explain why the sentinels are excluded from `size`.

Removing a known node from a doubly linked list

Setup: A node x is known to be an interior node in a doubly linked list. Let p be its predecessor and s its successor.

Main point: Reconnect p to s, reconnect s to p, clear x's two links, and decrement the list size. Because the node reference is given, no traversal is necessary.

What to practise: Redraw the three-node removal diagram from the notes and write the pointer changes in order. Then explain in words why assigning the local variable `node` to `null` would not remove x from the list.

Back-reference from a vertex to its containing list node

Setup: The vertices list contains nodes whose elements are vertex objects such as `v1` and `v2`. Each vertex object points back to the node that contains it through `vertexListPosition`.

Main point: A vertex can locate its own list node directly. This supports removal by passing `v1.vertexListPosition` to `vertices.remove` without searching from the header. The corresponding idea applies to edges through `edgeListPosition`.

What to practise: Draw two vertex objects and the two list nodes that contain them. Add the back-references, then justify why `v1 == v1.vertexListPosition.getElement()` evaluates to true.

Run-time structure of an undirected edge list graph

Setup: The notes show an undirected graph with vertices A, B, C, D and edges o, p, q, r, represented by a doubly linked list of vertices and a doubly linked list of edges.

Main point: Each edge object stores references to its endpoint vertex objects, while each vertex and edge stores a back-reference to its own node in the appropriate list. This diagram is the basis for reasoning about running time and graph mutations.

What to practise: Redraw the full structure from the notes: both lists, the edge origin/destination references, and the two kinds of back-reference. Then identify every edge incident to A and explain why an edge list must inspect all edges to find them.

DFS color state and timestamp definedness

Setup: The notes revisit the directed DFS trace on A, B, C, D, E, F, focusing on the status of E at different times.

Main point: At $t = 0$, all vertices are white and both timestamps are undefined. When E is discovered at $t = 7$, it becomes gray with $\mathrm{d}(E) = 7$ and undefined $\mathrm{f}(E)$. When it finishes at $t = 10$, it becomes black with both values defined.

What to practise: Use the completed trace to choose one other vertex and identify a white, gray, and black moment if possible. State whether d and f are defined in each state, then verify $\mathrm{d}(x) < \mathrm{f}(x)$.

Common Pitfalls and Misconceptions

Assigning a local node variable to `null`

Pitfall: Thinking that `node = null` removes the corresponding node from a doubly linked list.

Why it causes problems: This only changes the local reference variable. The predecessor and successor still point to the existing node, so the list structure has not changed.

How to avoid it: Update the predecessor and successor links first, clear the removed node's own links, and update `size`.

Treating sentinels as ordinary list elements

Pitfall: Counting the header and trailer as data nodes or trying to store graph vertices in them.

Why it causes problems: Sentinels are structural guards with no meaningful element. Including them in normal iteration or size reasoning creates incorrect boundary behavior.

How to avoid it: Keep the distinction explicit in every diagram: header and trailer are present, but `size` counts only real element nodes.

Confusing a position with an index

Pitfall: Reading `vertexListPosition` as an integer location in the list.

Why it causes problems: The constant-time benefit comes from holding the actual `DLNode` reference. An index would not directly identify a linked-list node without traversal.

How to avoid it: Whenever you see "position," draw the arrow from the vertex or edge object back to its containing list node.

Assuming vertex removal is only one list removal

Pitfall: Removing a vertex from the vertices list but leaving its incident edges in the edges list.

Why it causes problems: The graph then contains edges that refer to a removed vertex, so the representation is inconsistent.

How to avoid it: Treat `removeVertex` as a cascading operation: identify every incident edge, remove those edges, and then remove the vertex. Distinguish the $O(1)$ removal of a known list node from the full cost of finding incident edges in an edge list.

Using directed equality for an undirected edge

Pitfall: Treating (A, B) and (B, A) as different edges in the undirected assignment setting.

Why it causes problems: It gives the wrong graph semantics and can break edge lookup or duplicate prevention.

How to avoid it: Check both endpoint orders when implementing equality for an undirected edge.

Treating a digest as a full substitute

Pitfall: Reading only the digest and skipping the diagrams, recordings, code, and active exercises.

Why it causes problems: The edge list strategy depends on seeing and reconstructing pointer relationships, and the DFS properties depend on the full trace.

How to avoid it: Use the digest to choose where to return, then redraw, trace, implement, and test the primary course materials.

What You Should Be Able to Do

After this lecture, you should be able to:

- describe the two parts of Assignment 2 and identify the written tracing/induction work and the graph-programming work that require preparation;
- explain why an edge list graph maintains separate collections of vertices and edges;
- draw the empty sentinel-based doubly linked list and explain why sentinels are excluded from `size`;
- trace a correct $O(1)$ removal of a known non-sentinel doubly linked node;

- distinguish changing a local reference from changing the underlying object structure;
- identify the generic roles of vertex payloads, edge payloads, origins, and destinations;
- draw an edge list run-time structure containing vertices, edges, list nodes, endpoint references, and back-references;
- explain how `vertexListPosition` and `edgeListPosition` allow direct access to a containing list node;
- justify why a back-reference makes a single list-node removal $O(1)$, while finding all incident edges in an edge list still requires a scan;
- explain why `removeVertex` must also remove all incident edges;
- implement undirected edge equality that treats the two endpoint orders consistently;
- state the white, gray, and black DFS conditions for discovery and finish timestamps; and
- verify from a DFS trace that $\mathrm{d}(x) < \mathrm{f}(x)$ for every completed vertex.

Suggested Practice

- Before starting Assignment 2, review the relevant mathematical induction examples and practise at least one proof without looking at a solution.
- Draw an empty `DoublyLinkedList` with header and trailer sentinels. Then insert a few ordinary nodes and manually trace removing a middle node.
- Recreate the notes' edge list diagram for A, B, C, D and o, p, q, r, including both doubly linked lists, every endpoint reference, and every back-reference.
- Given a vertex such as A, scan the edge list by hand and collect its incident edges. Explain why this is not a direct lookup in the edge list strategy.
- When Assignment 2 is released, read the starter code carefully and map every supplied `DLNode`, list, vertex, and edge field to the diagrams from this lecture before implementing methods.
- Write or extend tests for vertex and edge removal. In particular, check that removing a vertex also removes all incident edges and leaves no dangling endpoint references.
- Review the directed DFS trace from the pre-reading-week tutorial. For white, gray, and black vertices, record whether d and f are defined, then verify $\mathrm{d}(x) < \mathrm{f}(x)$.
- Plan ahead for July 20, when Programming Test 2 and the written test are both scheduled. Use Assignment 2 and the early Assignment 3 programming materials, when posted, as practice rather than waiting until the test week.

Where to Review

- **iPad/PDF notes, page 3:** The `DLNode<E>` and `DoublyLinkedList<E>` structures, the empty-list sentinels, and the pointer updates for removing a known node.
- **iPad/PDF notes, page 4:** The edge list class structure, generic roles of vertices and edges, and the declarations of `vertexListPosition` and `edgeListPosition`.
- **iPad/PDF notes, page 5:** The detailed back-reference diagram and the conceptual removal call that passes `v1.vertexListPosition` to `vertices.remove`.
- **iPad/PDF notes, page 6:** The complete run-time edge list visualization for the undirected graph with vertices A, B, C, D and edges o, p, q, r; use this page to reason about incident-edge lookup and removal.
- **iPad/PDF notes, page 7 and directed DFS slide 33:** Color states, timestamp definedness, and the property $\mathrm{d}(x) < \mathrm{f}(x)$. Revisit the full directed DFS trace in Tutorial 6 when the table is not immediately clear.
- **Lecture recording:** Review the edge list and back-reference explanation when implementing Assignment 2, especially the discussion of why `node = null` does not remove a node and why vertex removal cascades to incident edges.
- **Assignment 2 materials:** Once posted, use the starter code and written questions as the primary implementation and practice target. The graph slides around 36–43 provide the interface context and edge list diagrams behind the assignment.

Closing Reminder

The most important follow-up from this lecture is active reconstruction. Do not merely recognize the edge list diagrams: redraw the lists, vertices, edges, endpoint links, and back-references until you can explain every arrow and predict what changes during a removal. Then connect the diagrams to the Assignment 2 starter code and test your own implementation. For DFS, return to the complete trace whenever the color/timestamp relationship is not automatic. This digest is a roadmap; the recording, Jackie's iPad notes, slides, Assignment 2, and your own tracing and coding practice remain essential.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 14, June 25

DFS Vertex & Edge Properties, BFS Foundations

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 3

Key Examples and Demonstrations 4

Common Pitfalls and Misconceptions 5

What You Should Be Able to Do 6

Suggested Practice 7

Where to Review 7

Closing Reminder 8

Main Ideas

- Assignment 2 is now available: its written induction and undirected-DFS tracing work is useful Written Test practice, while its programming work is the main preparation for Programming Test 2.
- DFS edge labels on directed graphs can be justified precisely from colour, stack status, discovery times, finish times, and ancestor/descendant relationships rather than from a drawing alone.
- Discovery/finish times behave as nested or non-overlapping intervals: interval containment identifies ancestry, while non-overlap identifies vertices that are neither ancestors nor descendants.
- Breadth-first search (BFS) explores an undirected graph level by level using a FIFO queue; its discovery edges form a spanning tree, while same-level non-discovery edges are cross edges.

Roadmap

Assignment 2 and assessment preparation

Assignment 2 contains two induction problems, an undirected-DFS trace, and a graph-programming part. The programming part is the most direct preparation for Programming Test 2. Jackie advised beginning preparation during the two test-free weeks: the Written Test is on Monday, July 13 and covers material through Lecture 17 on Tuesday, July 7; Programming Test 2 follows on Monday, July 20. For Written Test preparation, prioritize the iPad notes, then the slides and assigned exercises; use the digests selectively to identify important ideas, examples, and misconceptions.

DFS edge types: use state and timestamps, not only the picture

The lecture returned to the directed-DFS trace to formalize the four edge types. A discovery (tree) edge is processed from a gray origin to a white destination. A back edge goes from a descendant to a proper ancestor that is still active (gray). A forward edge goes from an ancestor to a proper descendant that has already finished (black). A cross edge connects vertices in different DFS subtrees, so neither endpoint is an ancestor or descendant of the other. The global time counter changes only when a vertex is discovered or finished, not when an already-known edge is classified.

Back edges, forward edges, and DFS forests

For a back edge (u, v), v is a proper ancestor of u: it was discovered earlier and is not yet finished. This is the directed-DFS edge type that certifies a cycle. For a forward edge, the destination is a finished proper descendant. The trace also illustrated why a graph may need more than one DFS run: a start vertex reaches only vertices reachable from it, and a full

DFS uses a driver loop to begin a new run from any remaining white vertex. Such separate runs form a DFS forest, and edges between its subtrees are cross edges.

Ancestor, descendant, and neither from d/f intervals

Jackie represented each vertex x by the interval $[d(x), f(x)]$. An ancestor's interval strictly contains every descendant's interval. By contrast, vertices in completed sibling branches have non-overlapping intervals. The notes used C and E as a left-to-right example, then introduced a hypothetical later branch H to show the opposite ordering. Adding a later sibling branch extends the finish time of an ancestor such as the root, but it does not change timestamps in already completed sibling subtrees.

BFS introduction: queue, levels, and edge types

The lecture then introduced BFS on undirected graphs. BFS uses a FIFO queue rather than the DFS stack: visit every adjacent vertex at the current level before progressing to more distant levels. The start vertex is level 0; a vertex's level is the number of edges in a shortest path from the start in the trace. In the tree-like example, the discovery edges formed a spanning tree. A second, partially traced graph showed the additional BFS edge type: an edge such as (B, C) is a cross edge when it connects two vertices at the same level. The full second trace was left for continuation.

Key Terms, Notation, and Structures

- **Discovery time $d(x)$ and finish time $f(x)$:** timestamps assigned when DFS discovers and finishes vertex x, respectively; the global counter advances on these push/pop events only.
- **Vertex colours in DFS:** white means undiscovered (d, f undefined), gray means discovered and active on the DFS stack (d defined, f undefined), and black means finished (d, f defined).
- **Discovery/tree edge:** an edge from a gray origin to a white destination; it discovers a vertex and extends the DFS or BFS spanning tree.
- **Back edge in directed DFS:** an edge from a descendant to a gray proper ancestor. In the lecture's timestamp language, the ancestor was discovered earlier and has undefined finish time. A back edge implies a directed cycle.
- **Forward edge in directed DFS:** an edge from an ancestor to a proper descendant that has already finished. At the time the edge is processed, the destination's finish time is defined and earlier than the current time.
- **Cross edge:** in directed DFS, an edge between vertices in different DFS subtrees; in the BFS discussion, an edge joining vertices at the same level.

- **Proper ancestor / proper descendant:** excludes the vertex itself. This distinction matters because a vertex is an ancestor and descendant of itself in the non-proper sense.
- **DFS interval:** the interval $[d(x), f(x)]$. After DFS completes,

$$u \text{ is an ancestor of } v \iff d(u) < d(v) \ \wedge \ f(v) < f(u).$$

- **Neither ancestor nor descendant:** completed intervals do not overlap:

$$f(u) < d(v) \ \vee \ f(v) < d(u).$$

- **DFS forest and driver loop:** a full DFS must start a new run from every still-white vertex after an earlier run finishes; multiple runs produce a forest rather than one tree.
- **BFS FIFO queue:** vertices are *enqueued* at the back and *dequeued* from the front, producing level-by-level exploration rather than DFS-style backtracking.
- **BFS level:** the number of edges from the start vertex in the BFS traversal; discovery edges in a connected graph form a spanning tree with $|V| - 1$ edges.

Key Examples and Demonstrations

Directed DFS: classifying discovery, back, forward, and cross edges

Setup: Jackie revisited a directed graph whose DFS tree contains a branch $A \to B \to C \to D$, followed by a separate branch through E.

Main point: Classify an edge using the destination's state and the DFS-tree relationship. In the trace, tree edges discover white vertices; $D \to A$ is back because A is a still-gray proper ancestor; after backtracking, $A \to C$ is forward because C is a finished proper descendant; and an edge from the new E branch to C is cross because the vertices lie in different subtrees. Only the back-edge case establishes a directed cycle.

What to practise: Redraw the DFS tree separately from the original graph. For each non-tree edge, state the origin/destination colours, whether the destination is an ancestor, descendant, or neither, and whether the current time changes.

Timestamp intervals for ancestry and non-overlap

Setup: Use the completed trace values $A : [1, 12]$, $C : [3, 4]$, and $E : [7, 10]$.

Main point: A is an ancestor of both C and E because each smaller interval is nested inside A's interval. C and E are neither ancestor nor descendant because $f(C) < d(E)$. If a later branch through H begins after E finishes, then $f(E) < d(H)$; the root's finish time moves later, but the completed C and E timestamps remain unchanged.

What to practise: Draw a time line for a small completed DFS trace. Test one ancestor pair and one unrelated pair using both required inequalities, rather than relying only on the diagram.

BFS levels and the FIFO queue

Setup: Start BFS at A in a tree-like undirected graph. Process adjacent vertices alphabetically.

Main point: Enqueue A at level 0, then discover B, C, D at level 1, E, F at level 2, and G, H at level 3. The discovery edges form a spanning tree. The queue controls which vertex is processed next; BFS does not backtrack like DFS.

What to practise: Recreate the queue after every enqueue/dequeue and label each vertex by level. Verify that every discovered vertex first appears one level farther from the start.

BFS cross edge at one level

Setup: In the second undirected BFS example, A discovers B, C, D at level 1.

Main point: When processing B, the edge (B, C) is not a discovery edge because C is already visited at the same level. It is a cross edge. The edge from B to a newly reached vertex such as E is instead a discovery edge to level 2.

What to practise: Continue the trace from the lecture only after first marking all vertices already discovered at each level. Explain why a same-level edge cannot create a new BFS-tree child.

Common Pitfalls and Misconceptions

Incrementing DFS time for every edge

Pitfall: Increasing the global DFS time whenever an edge is classified.

Why it causes problems: Discovery and finish timestamps no longer match pushes and pops, so the trace and all timestamp-based edge tests become incorrect.

How to avoid it: Increment time only when a vertex is discovered/pushed or finished/popped. Keep the same current time while classifying a back, forward, or cross edge.

Misstating the colours for a discovery edge

Pitfall: Treating a discovery edge as gray-to-black or otherwise forgetting the destination state before discovery.

Why it causes problems: A black vertex is already finished and cannot be newly discovered.

How to avoid it: Before processing a discovery edge, say explicitly: gray origin, white destination; after the push, the destination becomes gray.

Calling every non-tree edge evidence of a cycle

Pitfall: Inferring a directed cycle from a forward or cross edge.

Why it causes problems: Those edges do not point from a descendant to an active ancestor.

How to avoid it: In directed DFS, reserve the cycle conclusion for back edges only. Check that the destination is a gray proper ancestor.

Using only discovery times to claim ancestry

Pitfall: Concluding that u is an ancestor of v just because $d(u) < d(v)$.

Why it causes problems: u and v may be in different sibling subtrees, discovered at different times.

How to avoid it: Check interval containment: require both $d(u) < d(v)$ and $f(v) < f(u)$.

Treating BFS like DFS

Pitfall: Using a stack/backtracking mental model for BFS or treating an edge to an already visited same-level vertex as a new discovery.

Why it causes problems: BFS order, levels, and the spanning tree become incorrect.

How to avoid it: Maintain a FIFO queue and process the queue front. Record a discovery edge only when it reaches a previously unvisited vertex; same-level edges are cross edges in this undirected-BFS discussion.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain how Assignment 2 supports preparation for the Written Test and Programming Test 2, and plan review using the iPad notes, slides, exercises, and digests;
- classify directed-DFS edges as discovery, back, forward, or cross using colours, stack state, timestamps, and DFS-tree relationships;
- justify why a directed back edge implies a cycle, while a forward or cross edge does not establish one;
- trace which DFS actions change the global time and assign correct discovery and finish times;
- use $[d(x), f(x)]$ intervals to determine whether one vertex is an ancestor, descendant, or neither relative to another;
- explain why a full DFS may require multiple runs and produce a DFS forest;
- trace BFS on a small undirected graph using a FIFO queue, levels, and alphabetic adjacency order when specified;
- identify BFS discovery edges and same-level cross edges, and construct the BFS spanning tree from the discovery edges.

Suggested Practice

- Start Assignment 2 early. Complete the induction and undirected-DFS written work as active Written Test practice, and use the programming work to build the graph foundation needed for Programming Test 2.
- Revisit the directed DFS trace from the tutorial before Reading Week. For every edge, write its type, the relevant vertex colours, the time at which it is identified, and whether time changes.
- Create two completed DFS timestamp tables: one for an ancestor/descendant pair and one for unrelated siblings. Verify both interval tests.
- Add a later sibling branch to a DFS tree and predict which finish times change before checking your trace.
- Redraw the first BFS example and maintain an explicit FIFO queue. Label every vertex with its BFS level and count the $|V| - 1$ discovery edges in the spanning tree.
- Continue the partial second BFS example from the notes. Mark same-level edges as cross edges and distinguish them from edges that first discover a level-2 vertex.
- Before the Written Test, work through the end-of-lecture exercises or at least compare your reasoning against their posted solutions. Use the digest sections selectively to locate gaps, then return to the official notes and recording for details.

Where to Review

- **iPad/PDF notes, Page 3:** DFS discovery, back, forward, and cross edges, including the colour/timestamp conditions and the cycle observation.
- **iPad/PDF notes, Page 4:** interval diagrams for d/f timestamps, the ancestor test, and the two non-overlap cases for vertices that are neither ancestors nor descendants.
- **iPad/PDF notes, Page 5:** BFS setup, FIFO queue, levels, and the discovery-edge spanning tree.
- **iPad/PDF notes, Page 6:** the partial second BFS trace, including the same-level cross edge (B, C).
- **DFS slides, slide 33:** the formal timestamp properties discussed in the lecture; use the interval examples to interpret each condition.
- **BFS slides, beginning at slide 44:** the BFS definitions and subsequent examples.
- **Tutorial 6 and its recording:** the earlier directed-DFS trace used to ground the edge classifications and timestamp intervals.
- **Assignment 2 and the lecture-site exercises:** practise the induction, DFS tracing, and graph-programming skills that connect directly to the remaining assessments.

Closing Reminder

This digest is a roadmap, not a replacement for the lecture recording, Jackie's iPad notes, slides, Assignment 2, or independent practice. For DFS, actively redraw the tree and interval diagrams before applying a formula. For BFS, keep a real queue and level labels rather than trying to reason from the graph picture alone. Use the official materials to fill in the full traces and details, and ask Jackie when a definition or classification remains unclear.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 15, June 29

Testing Exceptions, BFS Tracing, DFS vs. BFS: Connectivity

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 7

Suggested Practice 8

Where to Review 8

Closing Reminder 9

Main Ideas

- JUnit starter tests that use `try/catch` and `fail()` tell you whether a graph operation is supposed to accept an input or throw an exception.
- Breadth-first search (BFS) on an undirected graph uses a FIFO queue to discover vertices level by level, with the level equal to the number of edges on a shortest path from the start vertex.
- In the BFS tracing model used here, each explored edge is either a discovery edge to a newly visited vertex or a cross edge to an already visited vertex; there are no DFS-style back or forward edges.
- DFS and BFS each have running time $O(n + m)$ under the stated incident-edge-access assumption, but representation details matter and dense graphs give the quadratic case $O(n^2)$.
- One DFS/BFS run on an undirected graph covers exactly the connected component containing the start vertex; directed graphs require the separate ideas of weak and strong connectivity.
- Assignment 2's starter tests and graph implementation work are direct preparation for the remaining graph assessments, so use the tests as specifications rather than as opaque code.

Roadmap

JUnit tests that expect or forbid exceptions

Jackie began with a programming-reading skill needed for Assignment 2. A method declared with "`throws SomeException`" must be handled by the caller, here through a `try/catch` block. The placement of `fail()` reveals the contract: for a legal input, an exception is not expected, so `fail()` belongs in the `catch`; for an illegal input, an exception is expected, so `fail()` belongs immediately after the method call in the `try`. Read this control flow before implementing the corresponding graph method.

Completing the BFS trace

The notes then completed BFS on the undirected six-vertex example. Start vertex A is level 0; its newly discovered neighbours B, C, and D form level 1; and E and F become level 2. The FIFO queue determines the processing order, so after discovering B, C, and D from A, process B before C and D.

Marking BFS vertices and edges

Before BFS, all vertices are unvisited and all edges are unmarked. When a vertex is first encountered, mark it visited and enqueue it immediately. When an already identified edge is encountered from the other endpoint, do not classify it again. In this undirected-BFS discussion, a new vertex gives a discovery edge; a previously visited vertex gives a cross edge. Cross edges may connect vertices at the same level or adjacent levels.

Tracing assumptions and alternative starts

Alphabetical order among adjacent vertices makes a trace deterministic, but a stated exception to that order must be followed exactly. The lecture also emphasized that changing the start vertex changes the level partition, even though a BFS of a connected undirected graph still reaches every vertex.

DFS/BFS running time and graph density

For DFS, each vertex is pushed and popped once; for BFS, each vertex is enqueued and dequeued once. Each edge receives its classification once. This motivates $O(n + m)$, where $n = |V|$ and $m = |E|$, provided the implementation can obtain the relevant incident-edge information efficiently. The edge-list strategy does not meet the stated constant-time incident-edge-access assumption. For a complete undirected graph, $m = \frac{n \cdot (n-1)}{2}$, so $O(n+m)$ simplifies to $O(n^2)$.

Connected components in undirected graphs

A single DFS or BFS visits all vertices and edges in the connected component containing its start vertex. Therefore, one traversal run covers an entire undirected graph exactly when the graph is connected. If the graph has several components, begin another run from an unvisited vertex for each remaining component.

Directed reachability: weak versus strong connectivity

Direction changes the role of the start vertex. A run beginning at one directed-graph vertex may reach every vertex, whereas a different start can reach only a proper subset. A directed graph is weakly connected if it becomes connected when edge directions are ignored. It is strongly connected only when directed paths exist between every pair of vertices. The lecture used the missing path from G to A as a witness that its directed example was not strongly connected.

Counting connected components

Assuming DFS or BFS is already available, repeat a traversal from an unvisited vertex while a counter of discovered vertices is less than the graph's number of vertices. Increment the counter when a vertex is discovered. The number of traversal runs is the number of connected components; terminate once the counter equals $|V|$.

Key Terms, Notation, and Structures

- `throws SomeException`**:** A declaration that the method may throw a checked exception, so callers must catch it or declare that they throw it.
- **Legal versus illegal input:** Legal input should complete without the specified exception; illegal input is a case where the required exception should occur.
- `fail()`**:** A JUnit assertion that immediately fails the test. Its location indicates which execution path is unacceptable.
- **BFS:** A graph traversal that uses a first-in, first-out (FIFO) queue and explores outward by levels from a chosen start vertex.
- **BFS level:** The number of edges on a shortest path from the start vertex to the vertex. The start vertex is level 0.
- **Discovery edge (BFS):** An edge that first reaches an unvisited vertex; it becomes part of the BFS spanning tree.
- **Cross edge (BFS):** An explored non-tree edge whose other endpoint is already visited. In the lecture examples, it can connect the same level or the next level.
- **Queue front:** The next vertex to dequeue and process in BFS. Processing must follow FIFO order, not visual left-to-right layout.
- $n = |V|$**,** $m = |E|$**:** The numbers of vertices and edges. The traversal bound is expressed as $O(n + m)$ under the stated access assumption.
- **Complete graph:** An undirected graph with the maximum number of edges, $m = \frac{n \cdot (n-1)}{2}$; its traversal bound is $O(n^2)$.
- **Connected component:** A maximal connected part of an undirected graph. One traversal run covers the component containing its start vertex.
- **Weakly connected directed graph:** A directed graph that is connected after all edge directions are ignored.
- **Strongly connected directed graph:** A directed graph in which a directed path exists from every vertex to every other vertex.
- `count`**:** The counter in the component-counting sketch. It is incremented whenever a traversal discovers a vertex and is compared with $|V|$ to decide whether more runs are needed.

Key Examples and Demonstrations

Reading exception tests as specifications

Setup: A method call appears inside `try/catch`, and the method may throw `SomeException`.

Main point: For legal input, the test should fail only if the exception occurs: `fail()` is placed in `catch`. For illegal input, the test should fail if control reaches the line after the call: `fail()` is placed immediately after the call in `try`, while the expected `catch` can be empty.

What to practise: For each Assignment 2 starter test involving an exception, trace both control-flow possibilities and state which one passes.

BFS trace from A

Setup: In the notes' undirected graph, process adjacent vertices alphabetically. Start at A, enqueue it, and assign level 0.

Main point: The trace discovers B, C, and D from A (level 1), then discovers E from B and F from C (level 2). The queue order is A, B, C, D, E, F for both enqueueing and dequeueing in this example. Discovery edges are AB, AC, AD, BE, CF; BC and CD are same-level cross edges, while CE and DF are cross edges to the next level.

What to practise: Redraw the graph, maintain the queue explicitly after every discovery/dequeue, and number the edges in the lecture order instead of relying on the visual layout.

From $O(n+m)$ to the dense-graph case

Setup: Compare a traversal's vertex actions and edge classifications with a complete undirected graph.

Main point: The general traversal accounting is one vertex action pair per vertex and one edge identification per edge, producing $O(n+m)$ under the stated assumption. Substituting $m = \frac{n\cdot(n-1)}{2}$ gives the dense-graph simplification $O(n^2)$.

What to practise: Explain in words why the edge-list representation needs separate analysis before claiming the $O(n+m)$ bound.

One traversal per connected component

Setup: An undirected input graph may have more than one disconnected part, while DFS/BFS is already implemented.

Main point: One traversal from an unvisited start covers exactly one component. Repeating from another unvisited vertex covers another component. A counter updated on every discovery reaches $|V|$ only after all components have been visited; the number of runs is the number of components.

What to practise: Simulate the loop on a graph with two components and write down the counter after each run.

Weak versus strong directed connectivity

Setup: The lecture's directed graph becomes connected if directions are removed, but no directed path leads from G to A.

Main point: Ignoring directions establishes weak connectivity only. The missing directed path is a concrete witness that strong connectivity fails. The lecture then showed two possible sets of added directed edges for turning the example into a strongly connected graph.

What to practise: Use the diagram to test paths in both directions for a few vertex pairs, then verify one of Jackie's proposed augmentations rather than memorizing it.

Common Pitfalls and Misconceptions

Putting `fail()` on the wrong branch

Pitfall: Reading `try/catch` without asking whether the input is legal or illegal.

Why it causes problems: You may implement an operation that throws on valid input, or one that silently accepts an invalid request when the specification requires an exception.

How to avoid it: First state the expectation. If no exception is expected, `fail()` belongs in `catch`. If an exception is expected, `fail()` belongs after the method call in `try`.

Forgetting that discovery requires immediate enqueueing

Pitfall: Marking a newly found vertex as visited but forgetting to enqueue it, as nearly happened for F in the trace.

Why it causes problems: The vertex will not be processed at the correct BFS level, so the queue order and later edge classifications become wrong.

How to avoid it: Use the same mini-checklist every time: discovery edge → mark visited → assign level → enqueue.

Mixing BFS markings with DFS colours

Pitfall: Treating the lecture's green BFS marking as the DFS white/gray/black state system.

Why it causes problems: It encourages DFS-style reasoning about backtracking and edge types during BFS.

How to avoid it: In these BFS traces, green means only "visited." Keep the FIFO queue and level labels visible; do not use the DFS stack-state interpretation.

Using DFS edge labels in BFS

Pitfall: Calling a BFS non-tree edge a back or forward edge.

Why it causes problems: The lecture's BFS model classifies edges only as discovery or cross edges.

How to avoid it: When an endpoint is already visited, record a cross edge and then note whether it connects the same level or the next level when asked.

Stating $O(n+m)$ without its representation assumption

Pitfall: Treating the usual DFS/BFS bound as automatic for every graph representation.

Why it causes problems: The traversal repeatedly needs incident-edge information; an edge list does not provide the stated constant-time incident-edge access.

How to avoid it: State the assumption before using the bound and revisit the actual cost of finding incident edges for the chosen representation.

Confusing connected, weakly connected, and strongly connected

Pitfall: Assuming one directed traversal proves strong connectivity, or ignoring direction when strong connectivity is required.

Why it causes problems: Starting from one convenient vertex can hide missing directed paths in the opposite direction.

How to avoid it: For a directed graph, test reachability in the required direction. To refute strong connectivity, give one ordered pair such as G to A with no directed path.

What You Should Be Able to Do

After this lecture, you should be able to:

- interpret a JUnit `try/catch` test to determine whether a legal or illegal input should throw an exception;
- place and explain `fail()` correctly for both "exception not expected" and "exception expected" cases;
- trace BFS on an undirected graph using a FIFO queue, a stated neighbour order, and explicit enqueue/dequeue records;
- assign BFS levels and explain their shortest-edge-distance meaning from the chosen start vertex;
- identify BFS discovery and cross edges, including cross edges that connect the same level or the next level;
- distinguish a DFS-style traversal picture from a BFS-style traversal picture when the ordering assumptions are given;
- justify the $O(n+m)$ DFS/BFS bound under its incident-edge-access assumption and simplify it to $O(n^2)$ for a complete graph;
- explain why one undirected traversal covers one connected component and design a repeated-traversal approach to count components;
- distinguish weakly connected from strongly connected directed graphs and use reachability to justify the distinction.

Suggested Practice

- Read the Assignment 2 starter tests that use exceptions. For each one, write a one-line statement of the intended contract before you implement the method.
- Retrace the A-start BFS example from the notes. Keep four separate records: the queue, each vertex's level, the discovery order, and the edge classification/order.
- Repeat the BFS trace from a different start vertex such as D, C, or E. Compare the level partition with the original trace and follow alphabetical ordering unless an explicit exception is given.
- Attempt the posted "tracing breadth first search using a FIFO queue" exercise before viewing its solution video. Check not only the final tree but also enqueue/dequeue order and cross-edge classifications.
- On a small graph, mark every BFS edge as discovery or cross and label cross edges as same-level or next-level. Do not use back/forward labels.
- Derive $O(n + m)$ from the "once per vertex / once per edge" accounting, then substitute $m = \frac{n \cdot (n-1)}{2}$ to obtain the complete-graph bound.
- Draw an undirected graph with at least two components. Simulate repeated DFS or BFS runs, update a discovery counter, and verify that the number of runs equals the number of components.
- Take a weakly connected directed graph, identify a missing directed path, and try adding a few edges to make paths possible in both directions for representative pairs.

Where to Review

- **iPad/PDF notes, page 3:** Review the two JUnit `try/catch` patterns and the different positions of `fail()`.
- **Page 4:** Revisit the BFS marking rule before tracing: unvisited vertices/edges at the start, then discovery versus cross edges.
- **Page 5:** Redo BFS Example 1(a), including the queue, levels L_0, L_1, L_2, and the same-level versus next-level cross edges.
- **Pages 6–8:** Use the alternative ordering prompts and the larger BFS exercise to practise respecting stated traversal assumptions.
- **Page 9:** Review the DFS/BFS time-complexity diagram, the $O(n + m)$ accounting, the incident-edge-access assumption, and the complete-graph simplification.
- **Pages 10–12:** Review connected components, directed versus undirected start-vertex behaviour, weak/strong connectivity, and the counter-based component-counting loop.

- **Lecture recording:** Revisit the segments on exception-test control flow, the completed BFS trace, and the discussion of why one traversal does not necessarily cover a disconnected graph.
- **Assignment 2 and the lecture site:** Read the starter tests alongside the programming specification, and use the posted FIFO-queue BFS exercise/video for additional tracing practice.

Closing Reminder

Use this digest to decide what to revisit, not as a substitute for working through the code and graph traces yourself. For this lecture, active practice means reading the starter tests line by line, maintaining a real FIFO queue while tracing BFS, and drawing components or directed paths rather than relying on visual intuition alone. Return to Jackie's iPad notes, slides, recording, Assignment 2, and posted exercises whenever a condition, edge classification, or implementation detail is not yet clear.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 16, June 30

Weighted Graphs and Tracing Dijkstra's Algorithm

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 3

Key Examples and Demonstrations 4

Common Pitfalls and Misconceptions 5

What You Should Be Able to Do 6

Suggested Practice 6

Where to Review 7

Closing Reminder 7

Main Ideas

- A *weighted graph* attaches a numerical cost to each edge; shortest-path questions minimize the total cost of a path rather than the number of edges.
- Dijkstra's algorithm computes shortest paths from one chosen *source* vertex to every other reachable vertex, under the essential assumption that all edge weights are non-negative.
- The algorithm maintains a current best distance $D[v]$, an ancestor $a[v]$, and a minimum-priority queue of vertices that have not yet been finalized.
- The central update is *relaxation*: improve a neighbor's D-value only when the path through the current vertex is strictly cheaper, and update its ancestor at the same time.
- A vertex's D-value becomes final only when that vertex is removed from the priority queue; the final ancestor links form a shortest-path tree that can reconstruct the actual paths.
- Assignment 2 is due Wednesday, July 8. The Written Test is on Monday, July 13 during tutorial time and covers material through Lecture 17; the programming parts of Assignments 2 and 3 prepare you for Programming Test 2.

Roadmap

Assessment planning and this week's direction

Jackie reminded you to keep moving on Assignment 2, which is due Wednesday, July 8. The Written Test is scheduled for Monday, July 13 during tutorial time and includes material through Lecture 17. Tutorial 7 will focus on loop invariants, which will support later reasoning about iterative algorithms such as Dijkstra's algorithm. Assignment 3 will include further Dijkstra practice and a mathematical-induction proof connection.

Weighted graphs, path costs, and shortest distances

A weighted graph adds a cost to each edge. The lecture used an airport-route diagram to distinguish the weight $w(u, v)$ of one edge from the weight $w(p)$ of an entire path, found by summing its edge weights. The theoretical shortest-path distance $d(u, v)$ is the minimum path weight among all paths from u to v; if no path exists, $d(u, v) = \infty$. For this course's Dijkstra tracing, weights are assumed non-negative.

What Dijkstra's algorithm maintains

With source vertex s, initialize $D[s] = 0$ and $a[s] =$ `nil`. For every other vertex v, initialize $D[v] = \infty$ and $a[v] =$ `nil`. Here $D[v]$ is the best distance found *so far*, whereas $d(s, v)$ is the true shortest distance that the algorithm aims to establish. A minimum-priority queue Q, keyed by D-values, selects the next vertex to process.

Extract-minimum, relaxation, and finalization

At each iteration, extract from Q the vertex u with minimum $D[u]$. For an incident edge from u to a neighbor v that is still in Q, compare $D[u]+w(u,v)$ with $D[v]$. If the new route is cheaper, replace $D[v]$ with that smaller value and set $a[v] = u$. After all applicable incident edges are considered, remove u from Q: its $D[u]$ is then finalized and equals $d(s,u)$.

The lecture trace from source W

The notes trace a four-vertex weighted graph with source W. Processing W gives tentative values $D[X] = 40$, $D[Y] = 30$, and $D[Z] = 20$. Next, processing Z improves Y to 25; processing Y improves X to 30. This illustrates why a tentative D-value may change before its vertex is extracted from Q.

Recovering actual shortest paths

The final ancestor values do not store full paths repeatedly. Instead, follow ancestors backward from a destination to the source, stop at `nil`, and reverse the resulting sequence. In the trace, $a[X] = Y$, $a[Y] = Z$, and $a[Z] = W$, so the shortest path from W to X is $W \to Z \to Y \to X$.

Key Terms, Notation, and Structures

- **Weighted graph**: A graph whose edges carry numerical costs; the lecture treats lower cost as better.
- $w(u,v)$: The weight of the edge between vertices u and v.
- **Path weight** $w(p)$: For $p = (v_0, v_1, \ldots, v_k)$,

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

- **Shortest-path distance** $d(u,v)$: The minimum path weight among all paths from u to v; use ∞ when no such path exists.
- **Non-negative-weight assumption**: Dijkstra's algorithm requires $w(u,v) \geq 0$ for every edge considered in the graph.
- **Source vertex** s: The fixed starting vertex from which Dijkstra's algorithm computes paths to all other vertices.
- **Tentative distance** $D[v]$: The algorithm's current best known distance from the source to v. It is not automatically final merely because it was updated.
- **Ancestor** $a[v]$: The preceding vertex on the current best path to v. The source begins with $a[s] =$ `nil`.

- **Minimum-priority queue** Q: Stores unfinalized vertices and supports selecting a vertex with minimum D-value. A heap is the efficient implementation to review later.
- **Relaxation**: For current vertex u and eligible neighbor v, test whether $D[u] + w(u, v) < D[v]$. If true, update both $D[v]$ and $a[v]$.
- **Finalized vertex**: A vertex removed from Q. At that point, its estimate is correct: $D[v] = d(s, v)$.
- **Shortest-path tree**: The tree encoded by final ancestor links, rooted at the source; it supports reconstructing one shortest path to each reachable destination.

Key Examples and Demonstrations

Airport routes: edge weights versus path weight

Setup: The weighted airport graph attaches a distance-like cost to each route, for example $w(\text{BOS}, \text{MIA}) = 1258$. A multi-edge route such as LAX → DFW → ORD → JFK has a weight obtained by adding the three edge weights.

Main point: An individual edge weight is not the same thing as the weight of a path, and neither is automatically the shortest distance between two vertices.

What to practise: Choose two visibly different routes between the same cities, calculate both path weights, and explain which one is shorter.

Dijkstra trace on W, X, Y, Z

Setup: Start at W. The example has edges $WX = 40$, $WY = 30$, $WZ = 20$, $ZY = 5$, and $YX = 5$. Initialize $D[W] = 0$, all other D-values to ∞, and all ancestors to `nil`.

Main point: The important sequence is not just the final answer: extract W, then Z, then Y, then X, applying relaxation whenever a cheaper route is found. The final distances are $D[W] = 0$, $D[Z] = 20$, $D[Y] = 25$, and $D[X] = 30$.

What to practise: Recreate the priority queue after every extraction, record every relaxation comparison, and state exactly when each vertex becomes finalized.

Ancestor links and path reconstruction

Setup: At termination, the trace gives $a[Z] = W$, $a[Y] = Z$, and $a[X] = Y$.

Main point: To find the path to X, follow the ancestors backward: $X, Y, Z, W,$ `nil`, then reverse the non-`nil` sequence.

What to practise: Reconstruct the paths from W to Z, Y, and X, and verify that their total weights equal the finalized D-values.

Common Pitfalls and Misconceptions

Confusing $D[v]$ with $d(s, v)$

Pitfall: Treating an updated $D[v]$ as the true shortest distance immediately.

Why it causes problems: $D[v]$ is only the shortest path found so far; the lecture trace improves $D[Y]$ from 30 to 25, and $D[X]$ from 40 to 30.

How to avoid it: State that $D[v]$ is finalized only when v is extracted from the priority queue.

Updating the distance but not the ancestor

Pitfall: Changing $D[v]$ after finding a cheaper route but leaving $a[v]$ unchanged.

Why it causes problems: The final distance may look correct, but the ancestor structure will no longer reconstruct the path that produced it.

How to avoid it: Treat relaxation as a paired update: whenever the strict inequality succeeds, set both $D[v]$ and $a[v] = u$.

Using Dijkstra's algorithm with negative edge weights

Pitfall: Ignoring the non-negative-weight assumption.

Why it causes problems: The finalization reasoning used by Dijkstra's algorithm is not valid when a later route can reduce a finalized distance through a negative edge.

How to avoid it: Check the assumption $w(u, v) \geq 0$ before applying this algorithm.

Treating the priority queue like a FIFO queue

Pitfall: Processing vertices in insertion order rather than by current minimum D-value.

Why it causes problems: Dijkstra's greedy selection rule is what makes extracted vertices safe to finalize.

How to avoid it: At the beginning of each iteration, explicitly compare the remaining D-values and choose the minimum key.

Relaxing edges to finalized vertices

Pitfall: Continuing to consider an incident edge whose destination has already been removed from Q.

Why it causes problems: The lecture's procedure treats removed vertices as finalized, so those destinations are no longer candidates for relaxation.

How to avoid it: For a current vertex u, consider only incident edges leading to vertices still in Q.

Reading ancestor links in the wrong direction

Pitfall: Reporting $X \to Y \to Z \to W$ as the route from W to X.

Why it causes problems: Ancestor pointers are followed from destination back toward source.

How to avoid it: Follow the links backward to `nil`, then reverse the sequence before reporting the source-to-destination path.

What You Should Be Able to Do

After this lecture, you should be able to:

- distinguish an edge weight $w(u, v)$, a path weight $w(p)$, a true shortest distance $d(u, v)$, and a tentative algorithm value $D[v]$;
- calculate the weight of a given path and explain why a shortest path minimizes total weight rather than number of edges;
- state the non-negative-weight assumption needed for Dijkstra's algorithm;
- initialize Dijkstra's algorithm correctly for a chosen source vertex, including D, a, and the priority queue;
- trace one Dijkstra iteration: extract the minimum-D vertex, identify eligible incident edges, perform relaxation, and finalize the extracted vertex;
- explain why a vertex still in the priority queue is not yet finalized even when its current D-value happens to equal the final answer;
- record final ancestor links and reconstruct a shortest source-to-destination path by reversing an ancestor chain;
- prepare for the Written Test and Programming Test 2 by connecting Assignment 2 and the later Assignment 3 practice to the graph algorithms studied in class.

Suggested Practice

- Re-trace the W, X, Y, Z example from scratch without looking at the completed table. For every iteration, write the contents of Q, the current D-values, each relaxation comparison, and the vertex that becomes finalized.
- Repeat the same graph with Z as the source vertex, as Jackie suggested. Compare the new ancestor tree and distances with the W-source trace.
- Practise using $w(u, v)$, $w(p)$, $d(u, v)$, $D[v]$, and $a[v]$ in short explanations. Be able to say what each notation means without mixing their roles.
- For each successful relaxation, verify both parts of the update: the new distance and the new ancestor. For each unsuccessful comparison, explain why no update occurs.

- Use the final ancestor map to reconstruct paths, then add their edge weights to check the corresponding finalized D-values.
- Review priority queues and heaps from the prerequisite data-structures material so that later discussion of efficient `extract-min` operations is easier to follow.
- Continue Assignment 2 before its July 8 deadline, and be ready to apply Dijkstra's algorithm and induction reasoning when Assignment 3 is released.

Where to Review

- **iPad/PDF notes, page 3:** weighted-graph terminology, $w(u, v)$, path weights, $d(u, v)$, and the non-negative assumption.
- **iPad/PDF notes, pages 4–5:** the W, X, Y, Z graph and the compact Dijkstra trace, including priority-queue selection and finalization.
- **iPad/PDF notes, page 7:** the detailed relaxation idea for one candidate edge; revisit this when the update rule feels mechanical rather than meaningful.
- **iPad/PDF notes, page 8:** final ancestor links and the reversal needed to reconstruct shortest paths from the source.
- **Lecture recording:** review the weighted-graph notation segment, then the full Dijkstra trace. Replay the portions where Y improves through Z and X improves through Y.
- **Course slides 54–55:** review the weighted-graph definitions and notation that Jackie identified as the starting reference for this topic.
- **Assignment 2 and upcoming Assignment 3:** use these for graph implementation practice and the later Dijkstra/induction connection; use the official assignment instructions for exact requirements.

Closing Reminder

Dijkstra's algorithm becomes much clearer when you actively trace it: maintain the priority queue, compare alternative path costs, update both D and a when relaxation succeeds, and reconstruct paths from the final ancestor links. Revisit the recording, Jackie's iPad notes, slides, and assignment materials, then practise on a changed source vertex rather than relying on recognition of the lecture example alone. This digest is a roadmap, not a replacement for working through the trace yourself.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 17, July 6

Dijkstra's Algorithm: Contracts, Loop Invariant, and Negative Weights

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 3

Key Examples and Demonstrations 4

Common Pitfalls and Misconceptions 4

What You Should Be Able to Do 5

Suggested Practice 6

Where to Review 6

Closing Reminder 7

Main Ideas

- The correctness of an iterative algorithm is organized around a precondition Q, an initialization S_{init}, a loop invariant I, a stay condition B, and a postcondition R; a correct termination state satisfies $I \wedge \neg B \Rightarrow R$.
- Dijkstra's shortest-path algorithm requires every existing edge weight to be non-negative, whether the weighted graph is directed or undirected.
- Distinguish $D(v)$, the algorithm's current computed distance for one vertex, from $d(s, v)$, the true shortest-path weight from source s to vertex v.
- Dijkstra's loop invariant says that every finalized vertex already has its correct shortest-path distance; the set of finalized vertices grows one extraction at a time.
- A negative edge can make Dijkstra finalize a vertex too early, breaking the loop invariant; an algorithm such as Bellman-Ford is needed when negative weights must be supported.

Roadmap

From contracts to a safe terminal state

Jackie first revisited the proof pattern for iterative algorithms. Starting from a state satisfying the **precondition** Q, the initialization S_{init} must establish the **loop invariant** I. Each iteration runs only while the **stay condition** B is true and must preserve I. When the loop exits, the intended safe state has I, $\neg B$, and the **postcondition** R all true. The key proof obligation is therefore

$$(I \wedge \neg B) \Rightarrow R.$$

Revisit Tutorial 7 if this workflow or the difference between a stay condition and a loop invariant is not yet clear.

Dijkstra's contract and precise notation

For Dijkstra's algorithm, the key precondition is that every existing edge (u, v) has $w(u, v) \geq 0$. A zero-weight edge is still an edge; a missing edge is simply not considered. The graph may be directed or undirected, but directions must be respected in a directed graph.

Upon termination, two outcomes are required: for every vertex v, $D(v) = d(s, v)$, and the reverse of the ancestor chain reconstructs a true shortest path from source s to v. Keep the notation exact: $D(v)$ is a mutable program value for one vertex, whereas $d(s, v)$ is the theoretical true distance between two vertices.

Dijkstra's loop invariant: finalized vertices only

During execution, not every $D(v)$ is final. The loop invariant restricts the postcondition to the vertices already removed from the priority queue Q (the finalized vertices):

$$\forall u \in V \bullet u \notin Q \Rightarrow D(u) = d(s, u)$$

Equivalently, let S be the conceptual set of finalized vertices:

$$\forall u \in V \bullet u \in S \Rightarrow D(u) = d(s, u)$$

The implication matters: only vertices satisfying the antecedent need to meet the equality. In the earlier W, Z, Y, X example, S grows from $\varnothing$ to $\{W\}$, then $\{W, Z\}$, and so on. At termination, $S = V$, so this restricted statement becomes the full distance postcondition.

Why negative weights break Dijkstra's algorithm

Jackie then traced a directed counterexample with source A, edges $A \to B$ of weight 2, $A \to C$ of weight 4, and $C \to B$ of weight -3. After processing A, Dijkstra records $D(B) = 2$ and $D(C) = 4$, so it extracts and finalizes B next. But the true shortest distance is $d(A, B) = 1$ through $A \to C \to B$, so the loop invariant fails as soon as B is finalized.

Do not "fix" this by later relaxing $C \to B$: after B has been removed from Q, that edge is not eligible for Dijkstra's relaxation rule. This is why the non-negative-weight precondition is essential rather than optional. Bellman-Ford was mentioned as an alternative when negative weights are required.

Assessment and preparation reminders

Assignment 2 is due Wednesday, July 8; submit before consulting the solution, which Jackie planned to release Thursday morning or earlier. Assignment 3 is a continuation of Assignment 2's programming work and includes loop-invariant reasoning. The written test is Monday, July 13 and covers material through Lecture 18 and Tutorial 8; Programming Test 2 is Monday, July 20 and is based on Assignments 2 and 3.

Key Terms, Notation, and Structures

- **Precondition** Q: A required property of the input state before the algorithm begins.
- **Postcondition** R: The required relationship between the input and final output after successful termination.
- **Loop invariant** I: A Boolean property established by initialization and preserved by every completed loop iteration.
- **Stay condition** B: The Boolean condition that determines whether another iteration executes; loop exit means $\neg B$.
- **Safe state**: A terminating state satisfying $I \wedge \neg B \wedge R$; correctness requires $(I \wedge \neg B) \Rightarrow R$.
- $w(u, v)$: The weight of an existing edge (u, v). Dijkstra's algorithm requires $w(u, v) \geq 0$ for every edge considered.
- $D(v)$: The algorithm's best distance found *so far* for vertex v; it is a program variable and is finalized only when v is extracted from the priority queue.

- $d(s, v)$: The true minimum weight among all paths from source s to destination v; it is the theoretical target, not a value that Dijkstra's algorithm directly updates.
- **Priority queue** Q: The collection of unfinalized vertices, keyed by their current D values; Dijkstra extracts a minimum-D vertex each iteration.
- **Finalized set** S: The conceptual set of vertices removed from Q. For a vertex u, $u \notin Q$ is equivalent to $u \in S$.
- **Ancestor pointer**: The stored predecessor for a vertex on its current best path; reversing the final ancestor chain gives a shortest path from the source.
- **Relaxation**: Checking whether a path through the current vertex improves an unfinalized neighbor's D value, then updating the distance and ancestor when it does.

Key Examples and Demonstrations

Growing the invariant in the W, Z, Y, X trace

Setup: In the earlier Dijkstra trace, vertices were removed from the priority queue in the order W, Z, Y, X. The finalized set evolves as $\varnothing$, $\{W\}$, $\{W, Z\}$, $\{W, Z, Y\}$, then V.

Main point: The invariant is cumulative. After each extraction, recheck the correctness equality for every vertex already in S, not only for the newly removed vertex.

What to practise: Revisit the original trace and write the invariant after each extraction. Explain why the final invariant has the same scope as the postcondition.

The negative-edge counterexample

Setup: Use source A and directed edges $A \to B$ with weight 2, $A \to C$ with weight 4, and $C \to B$ with weight -3.

Main point: Dijkstra extracts B before C because $2 < 4$, but B's true shortest distance is 1 through C. Thus, the algorithm finalizes an incorrect value and violates its loop invariant.

What to practise: Trace the first two iterations and explicitly compare $D(B) = 2$ with $d(A, B) = 1$. Then identify why a later attempt to update B from C is not a valid Dijkstra step.

Common Pitfalls and Misconceptions

Confusing $D(v)$ with $d(s, v)$

Pitfall: Treating $D(v)$ as though it were automatically the true shortest distance, or writing $D(s, v)$ and $d(v)$ as if the two notations had the same arguments.

Why it causes problems: $D(v)$ is only the best value found so far and can change before v is finalized; $d(s, v)$ is the true mathematical distance from a specified source.

How to avoid it: Say aloud what each expression means while tracing: "capital D of one vertex, current value" versus "little d from source to destination, true value."

Using Dijkstra's algorithm with a negative edge

Pitfall: Assuming Dijkstra's greedy extraction remains safe when some edge has negative weight.

Why it causes problems: A path discovered later can reduce the distance of a vertex that Dijkstra has already finalized, which destroys the invariant and can produce an incorrect answer.

How to avoid it: Check the non-negative-weight precondition before choosing Dijkstra. A weight of zero is allowed; the issue is a weight below zero.

Revisiting a finalized vertex

Pitfall: Relaxing an edge to a destination that has already been removed from the priority queue.

Why it causes problems: That is no longer Dijkstra's algorithm: relaxation applies only to destinations still in Q.

How to avoid it: In every relaxation step, first ask whether the destination vertex remains in the priority queue.

Checking the invariant over the wrong scope

Pitfall: Requiring $D(v) = d(s, v)$ for all vertices too early, or forgetting that previously finalized vertices remain part of the invariant.

Why it causes problems: Before termination, unfinalized vertices may still have provisional D values; meanwhile, every member of S must continue to satisfy the equality.

How to avoid it: Write the implication form of the invariant and track the growing set S after each extraction.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain how Q, S_{init}, I, B, and R fit together in a correctness argument for an iterative algorithm;
- state Dijkstra's non-negative-edge-weight precondition and explain why a zero-weight edge is still valid;
- distinguish $D(v)$ from $d(s, v)$ precisely, including their different roles and arguments;
- state Dijkstra's two postcondition goals: correct shortest-path weights and recoverable shortest paths through ancestors;
- state Dijkstra's loop invariant using either $u \notin Q$ or $u \in S$;
- trace how S grows as vertices are extracted and identify exactly which distance equalities the invariant requires after each iteration;

- use the negative-edge example to show a loop-invariant violation and explain why Dijkstra cannot repair a vertex after it is finalized;
- identify Bellman-Ford as an alternative when the application requires negative edge weights, without assuming it has the same running-time cost as Dijkstra's algorithm.

Suggested Practice

- Revisit Tutorial 7 and redraw the safe-state argument: I is maintained, $\neg B$ holds at exit, and together they must imply R.
- Redo the Lecture 16 Dijkstra trace from the source and, after every extraction, write the current finalized set S and the exact invariant statement it requires.
- Trace the $A \to B$, $A \to C$, $C \to B$ negative-edge graph. Stop at the point where B is finalized and explain in one or two sentences why the invariant fails.
- Practise writing the two distance notations correctly: $D(v)$ for the current program value and $d(s, v)$ for the true source-to-destination distance.
- Before the next lecture, review the procedural specification of Dijkstra's algorithm on slide 69 and compare each step against the earlier tracing example, as Jackie requested.
- Complete Assignment 2 before its deadline; after submitting, study both its written and programming solutions, then start Assignment 3 promptly because it extends the same programming work and uses loop-invariant reasoning.

Where to Review

- **Lecture 17 iPad/PDF notes, page 3 (slide 67):** visualization of initialization, repeated iterations, the final safe state, and $(I \wedge \neg B) \Rightarrow R$.
- **Lecture 17 iPad/PDF notes, page 4:** Dijkstra's precondition and postconditions, including the distinction between $D(v)$ and $d(s, v)$.
- **Lecture 17 iPad/PDF notes, page 5 (slide 68):** the two equivalent forms of Dijkstra's loop invariant and the W, Z, Y, X finalized-set trace.
- **Lecture 17 iPad/PDF notes, page 6:** the negative-edge counterexample and the exact point where the loop invariant fails.
- **Lecture 16:** revisit the original Dijkstra trace, priority queue operations, and ancestor-path reconstruction before formal reasoning about the invariant.
- **Tutorial 7:** revisit preconditions, postconditions, loop invariants, and the use of a restricted postcondition to describe intermediate progress.
- **Slide 69:** review Dijkstra's procedural specification before Lecture 18, then compare it line by line with the tracing steps from Lecture 16.

Closing Reminder

This digest is a roadmap, not a replacement for the Lecture 16–17 recordings, Jackie's iPad/PDF notes, Tutorial 7, assignments, or your own tracing and proof practice. For Dijkstra's algorithm, make your review active: trace the priority queue, track D, ancestors, and S, and justify why each finalized vertex is correct rather than merely memorizing the final formulas.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 18, July 7

Priority Queues, Heaps, and Time Complexity of Dijkstra's Algorithm: S_{init}

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 8

Suggested Practice 8

Where to Review 9

Closing Reminder 9

Main Ideas

- This lecture reviewed priority queues as an abstract data type and explained why Dijkstra's algorithm needs an efficient priority-queue implementation.
- Jackie contrasted priority queues with FIFO queues and dictionaries: priority-queue keys determine removal priority, not searchable identity.
- The lecture rebuilt the heap data structure from complete binary trees, then added the Heap Order Property to explain why the root stores the minimum key.
- Complete binary tree bounds and the geometric sequence sum were used to justify why heaps have height $O(\log n)$.
- Heap insertion and deletion were connected to up-heap and down-heap bubbling, each taking $O(\log n)$ time.
- The Dijkstra pseudocode from the graph slides was revisited enough to identify the initialization cost: inserting all n vertices into a heap-backed priority queue costs $O(n \log n)$.
- Assignment 2 was due the next day, Assignment 3 was to be released, Written Test 2 and Programming Test 2 remained on schedule, and the final exam time was tentatively announced.

Roadmap

Administrative reminders and study timeline

Jackie announced a tentative final exam time of Sunday, August 9, from 10 AM to 1 PM, with the location still to be finalized by the registrar office. Assignment 2 was due Wednesday, July 8, Assignment 3 was to be released on Wednesday, Written Test 2 was scheduled for Monday, July 13, and Programming Test 2 for Monday, July 20. Jackie also pointed students to the heap insertion-versus-deletion exercises and video, which should be completed before the written test.

Priority queues as an abstract data type

The lecture began by defining a priority queue as a collection of entries, where each entry has a key and an element. In a minimum-key priority queue, the entry with the smallest key has the highest priority, so `removeMin()` or `extractMin()` removes the minimum-key entry. The example entries $(9, A), (3, B), (6, C), (3, D), (1, E), (2, F)$ showed that $(1, E)$ is removed first and that equal-priority entries, such as the two key-3 entries, may be chosen in either order.

Priority queues versus FIFO queues and dictionaries

Jackie then separated three ideas that are easy to confuse. A FIFO queue removes entries according to chronological insertion order. A dictionary, often implemented using a BST or balanced BST, uses keys to identify and search for entries. A priority queue uses keys only to rank priority: in Dijkstra's algorithm, the key is the current D value, so the vertex with the smallest current distance estimate is extracted next. Heap keys may be duplicated and should not be treated like dictionary keys.

Why heaps are needed

A student correctly anticipated that repeatedly scanning an array or linked list to find the minimum would be inefficient. Jackie used this to motivate heaps as the specialized data structure for priority queues. Arrays and linked lists are useful review cases, but heaps are the structure that supports the priority-queue operations needed by Dijkstra efficiently.

Complete binary trees and geometric sums

Before defining heaps, Jackie reviewed the geometric sequence sum and the node counts of binary trees. A complete binary tree of height h is analyzed by separating levels 0 through $h-2$, level $h-1$, and the bottom level h. The minimum number of nodes is 2^h because a tree of height h must have at least one node at level h; the maximum number is $2^{h+1}-1$, the full binary tree case.

Three criteria for completeness

The notes formalized completeness using three criteria. C1: every node at levels 0 through $h-2$ has exactly two children. C2: nodes at level $h-1$ may have 0, 1, or 2 children. C3: children at level h must be filled from left to right. Jackie emphasized that a node at level $h-1$ with a right child but no left child violates C3, even if the other levels look reasonable.

Heap structural and relational properties

A heap first satisfies the structural property: it is a complete binary tree. This gives height $O(\log n)$, which is why ancestor-path repairs can be logarithmic. A heap also satisfies the relational Heap Order Property: for a min-heap, every non-root node has key greater than or equal to its parent's key. This makes the root the minimum key of the whole heap, but it does not sort the left and right subtrees the way a BST does.

Insertion, deletion, and bubbling

The lecture summarized the key operations that students must practise with the posted exercises. Insertion places the new entry at the next valid bottom-level position so the tree remains complete, then uses up-heap bubbling to restore the Heap Order Property. Deletion or `extractMin()` removes the root, replaces it with the rightmost bottom-level node to

preserve completeness, and then uses down-heap bubbling. Each bubbling path has length $O(\log n)$.

Heapsort as a useful comparison

Jackie briefly connected heaps to heapsort. Conceptually, heapsort builds a heap by inserting all items, then repeatedly removes the minimum. This resembles selection sort in spirit, but each removal is logarithmic rather than linear, giving the familiar $O(n \log n)$ behaviour.

Dijkstra initialization and the first complexity bound

At the end, Jackie returned to Dijkstra's algorithm on slide 69 of the graph deck. The input is a directed or undirected weighted graph and a source vertex, with non-negative edge weights as the key precondition. The initialization sets the source distance, gives other vertices $+\infty$, sets ancestors to `nil`, and inserts all vertices into the priority queue. The first important bound is therefore $O(n \log n)$ for initialization, because there are n heap insertions and each takes $O(\log n)$. The rest of the Dijkstra complexity analysis was left for Tutorial 8.

Key Terms, Notation, and Structures

- **Priority queue (PQ)**: an abstract data type storing entries as key-element pairs and removing entries according to priority.
- **Minimum-key priority queue**: a priority queue where the smaller key has the higher priority; `extractMin()` removes a minimum-key entry.
- **FIFO queue**: a queue that removes entries in chronological insertion order rather than priority order.
- **Dictionary**: an ADT where keys are used to identify and search for entries, often implemented using a BST or balanced BST.
- **Heap**: a binary tree used to implement a priority queue efficiently; it must satisfy both a structural property and a relational property.
- **Complete binary tree**: a binary tree satisfying C1, C2, and C3: full enough above the bottom, flexible at level $h - 1$, and filled left-to-right at level h.
- **Full binary tree**: the maximum case for a complete binary tree of height h, with all levels filled and $2^{h+1} - 1$ nodes.
- **Height** h: the number of edges on a longest path from the root to a deepest node; for heaps, $h = O(\log n)$.
- **Heap Order Property (HOP)**: for a min-heap, every non-root node n satisfies $\text{key}(n) \geq \text{key}(\text{parent}(n))$.

- **Up-heap bubbling**: the repair process after insertion, moving a newly inserted entry upward until HOP is restored.
- **Down-heap bubbling**: the repair process after deleting the root, moving the replacement entry downward until HOP is restored.
- n **and** m: in the Dijkstra discussion, $n = |V|$ is the number of vertices and $m = |E|$ is the number of edges.
- $D[v]$ **or** $D(v)$: the current distance estimate used as the priority key for vertex v in Dijkstra's priority queue.
- $+\infty$: the initialization value for non-source vertices before any shorter path has been discovered.
- **Ancestor pointer**: the predecessor information used to reconstruct shortest paths after Dijkstra terminates.

Key Examples and Demonstrations

Priority-queue entry example

Setup: The notes used entries such as $(9, A), (3, B), (6, C), (3, D), (1, E), (2, F)$ to illustrate priority.

Main point: In a minimum-key priority queue, $(1, E)$ has the highest priority because 1 is the smallest key. Entries with the same key, such as the two key-3 entries, have equal priority and may be removed in either order.

What to practise: Given a small set of key-element pairs, identify the removal order under a minimum-key priority queue and explain how it differs from FIFO order.

Complete binary tree criteria C1–C3

Setup: Jackie divided a tree of height h into levels 0 through $h - 2$, level $h - 1$, and level h.

Main point: Completeness is not just a vague idea of being "almost full". You must check C1, C2, and C3 precisely. In particular, bottom-level nodes must be filled left-to-right; a right-only child at level h violates completeness.

What to practise: Draw several small binary trees and label their levels. For each one, state whether C1, C2, and C3 pass or fail.

Minimum and maximum node counts

Setup: For a complete binary tree of height h, the lecture summed the full levels above the bottom and then considered the minimum and maximum number of nodes at level h.

Main point: The shared base levels contribute $2^h - 1$ nodes. The minimum complete tree of height h adds one bottom-level node, giving 2^h total nodes. The maximum case fills level h, giving $2^{h+1} - 1$ nodes.

What to practise: Re-derive these formulas using the geometric sequence sum and explain why zero nodes at level h would contradict the assumption that the height is h.

Heap structural and relational checks

Setup: Jackie used several small example trees and asked which ones were heaps.

Main point: A tree must pass both checks: it must be complete, and it must satisfy HOP. A tree may be complete but violate HOP, or satisfy parent-child key order locally but fail completeness.

What to practise: For each example heap/non-heap in the notes, identify exactly which property passes and which property fails.

Insertion versus deletion in a heap

Setup: The assigned exercise pages show inserting a new entry $(2, T)$ and deleting the root/minimum from a heap.

Main point: Insertion repairs upward; deletion repairs downward. Both repairs are bounded by the heap height, so both are $O(\log n)$.

What to practise: Complete the posted heap insertion and deletion exercises. Do not only watch the video; redraw the heap after each swap and verify both completeness and HOP after each operation.

Dijkstra initialization complexity

Setup: The Dijkstra pseudocode initializes distance and ancestor values, then inserts all vertices into the priority queue.

Main point: The loop that inserts all vertices dominates initialization: n insertions times $O(\log n)$ per heap insertion gives $O(n \log n)$.

What to practise: Annotate slide 69 line by line. For each initialization step, write its cost and justify why the total simplifies to $O(n \log n)$.

Common Pitfalls and Misconceptions

Treating heap keys like dictionary keys

Pitfall: Assuming the key in a heap is meant for efficient searching, like the key in a dictionary or BST.

Why it causes problems: Heap keys only impose parent-child priority order. They do not tell you whether to search left or right, and arbitrary search in a heap can degrade to $O(n)$.

How to avoid it: Use heaps for priority operations such as insertion and extracting the minimum. Use a dictionary or balanced BST when searchable unique keys are the main requirement.

Using BST logic inside a heap

Pitfall: Comparing the root with a target key and then deciding to search only one subtree, as if the heap were a binary search tree.

Why it causes problems: The left and right subtrees of a heap are not ordered relative to each other. The only guaranteed relationship is between a node and its parent.

How to avoid it: Remember that HOP is vertical, not horizontal: child key ≥ parent key in a min-heap, but siblings and subtrees have no BST-style ordering.

Forgetting the left-to-right requirement for completeness

Pitfall: Accepting a tree as complete when a node at the second-last level has a right child but no left child.

Why it causes problems: This violates C3, so the tree is not complete and therefore cannot be a heap even if the key order looks fine.

How to avoid it: Check bottom-level filling from left to right. During deletion, replace the root using the rightmost bottom-level node, not an arbitrary leaf.

Thinking a complete tree of height h can have no nodes at level h

Pitfall: Counting the minimum nodes of a complete binary tree of height h without including any node at the deepest level.

Why it causes problems: If level h is empty, the tree's height is not h; it is at most $h-1$.

How to avoid it: In the minimum case, add exactly one node at level h after summing the full levels above it.

Calling heap insertion or deletion constant time

Pitfall: Not separating the easy root access from the repair work needed to restore heap validity.

Why it causes problems: Accessing the root is $O(1)$, but insertion may require up-heap bubbling and deletion may require down-heap bubbling, each over a path of height $O(\log n)$.

How to avoid it: In running-time analysis, include the structural and relational repair steps, not just the first access or assignment.

Underestimating Dijkstra's initialization cost

Pitfall: Saying Dijkstra initialization is just $O(n)$ because it scans vertices once.

Why it causes problems: The initialization also inserts all n vertices into a heap-backed priority queue, and each insertion is $O(\log n)$.

How to avoid it: Break initialization into lines: constant setup, $O(n)$ assignments, and n heap insertions, giving $O(n \log n)$ overall.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain what a minimum-key priority queue is and what `extractMin()` removes;
- distinguish priority queues from FIFO queues and dictionaries;
- explain why heap keys are not used for efficient arbitrary searching;
- state and apply C1, C2, and C3 for complete binary trees;
- compute the minimum and maximum number of nodes in a complete binary tree of height h;
- define a heap using both the structural property and the Heap Order Property;
- identify whether a small binary tree is a heap and justify the answer by naming the property that passes or fails;
- describe insertion with up-heap bubbling and deletion with down-heap bubbling;
- justify why heap insertion and deletion are $O(\log n)$;
- explain the conceptual $O(n \log n)$ behaviour of heapsort;
- annotate the initialization part of Dijkstra's pseudocode and justify its $O(n \log n)$ cost;
- connect the heap review to the upcoming Dijkstra time-complexity analysis in Tutorial 8.

Suggested Practice

- Complete the posted heap insertion-versus-deletion exercises and watch the accompanying video before Written Test 2.
- Redraw the insertion of $(2, T)$ into the example heap and mark each up-heap swap.
- Redraw the deletion of the root/minimum from the example heap and mark why the rightmost bottom-level node must be used as the replacement.
- For several small binary trees, check C1, C2, and C3 explicitly, then check HOP separately.
- Re-derive the complete-tree node bounds 2^h and $2^{h+1} - 1$ from the geometric sequence sum.
- Compare a heap and a balanced BST on the same set of key values; explain which operations each structure supports efficiently.
- Review slide 69 of the graph notes and write the running time beside each initialization line of Dijkstra's algorithm.
- Submit Assignment 2 by its deadline, then study the posted solution once it is released; start Assignment 3 when it appears.

Where to Review

- **Lecture 18 iPad/PDF notes, pages 1–2**: lecture title and administrative reminders, including the tentative final exam time and upcoming assessment dates.
- **Page 3**: priority queue definition, min-key priority, duplicate keys, and comparison with FIFO queues.
- **Page 4**: exercise prompt for implementing priority-queue operations using a linear data structure such as an array or linked list.
- **Pages 5–7**: geometric sequence sum, role of keys, and complete-versus-full binary tree criteria and node bounds.
- **Pages 9–11**: heap structural property, Heap Order Property, and example heap/non-heap checks.
- **Page 12**: Dijkstra pseudocode and the initialization-time discussion leading to $O(n \log n)$.
- **Pages 13–15**: assigned heap insertion and deletion exercises; use these together with the posted exercise video.
- **Lecture recording**: revisit the explanations distinguishing heaps from BSTs, the complete-tree criteria, the root-replacement rule for deletion, and the Dijkstra initialization analysis.
- **Graph slides, slide 69**: review the procedural specification of Dijkstra's algorithm before Tutorial 8.

Closing Reminder

This digest is only a roadmap. To learn the material, you still need to review Jackie's iPad notes and recording, complete the heap insertion/deletion exercises, trace the examples by hand, and connect the heap operation costs back to Dijkstra's pseudocode. Reading the summary is not enough: the testable skill is being able to draw, check, repair, and analyze the structures yourself.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 19, July 14

DAGs, Topological Sort: DFS, Time Complexity, Tracing

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 9

Closing Reminder 10

Main Ideas

- This lecture introduced directed acyclic graphs (DAGs) as directed graphs whose edges model dependency constraints and whose lack of cycles rules out circular dependency.
- A topological order was defined as a linearized DAG: every vertex appears exactly once, and every directed edge is respected by the order.
- Jackie emphasized that a topological order is usually not unique; vertices with no edge constraint between them may be moved around without making the order illegal.
- The lecture formalized the legality test for a topological order using indices: for an order $\langle v_1, v_2, \ldots, v_n \rangle$, every edge (v_i, v_j) must satisfy $i < j$.
- Topological sort was presented as a dependency-based "sort", not a comparison-based sort like merge sort or quicksort.
- A DFS-based topological sort was developed: run DFS from unvisited vertices, add a vertex to the front of the order when it backtracks, and accumulate one shared order list across all DFS runs.
- The algorithm runs in $O(|V| + |E|)$ time because each vertex is pushed and popped once and each edge is examined once.
- The lecture ended by introducing a practical application: sequentializing parallel transactional updates by building a dependency DAG and applying topological sort.
- For course logistics, Programming Test 2 is on Monday, July 20; Assignment 2's solution and Assignment 3 are released, and the final exam is finalized for Sunday, August 9, 10 AM–1 PM, with room still TBD.

Roadmap

Announcements and graph-representation review

Jackie began with assessment and preparation reminders. The written test requires manual grading, Assignment 2's solution is available, Assignment 3 has been released, Programming Test 2 is on Monday, July 20, and the final exam is finalized for Sunday, August 9, 10 AM–1 PM, with the room still to be announced. Jackie also reminded students to review all three graph representations: edge list, adjacency list, and adjacency matrix. Edge lists are especially relevant to Programming Test 2, while adjacency lists and matrices should be studied more carefully for later review and the exam.

Directed acyclic graphs as dependency models

The technical material began with the definition of a DAG. A graph has vertices and edges; "directed" means each edge has an origin and destination; and "acyclic" means there is no

directed cycle. The key interpretation is dependency: an edge $v \to u$ means v must happen before u. A cycle would create a circular dependency, which is why topological sorting requires a DAG as input.

Course prerequisites as a DAG

Jackie's first major example used courses as vertices and prerequisites as directed edges. A chain such as $1011 \to 1021 \to 2030 \to 2101 \to 3101$ expresses required ordering, while other edges such as $1019 \to 2101$, $1090 \to 3101$, and $1801 \to 1802$ add independent constraints. The example also showed that a DAG does not have to be connected: separate components create independent ordering choices.

Topological order: linearizing a DAG

A topological order is a linear sequence of the vertices of a DAG. Jackie gave two criteria: every vertex from the DAG must appear in the order, and for every edge (x, y), x must appear before y in the sequence. Importantly, "before" does not mean "immediately before". Other vertices may appear between x and y as long as the relative order of the edge endpoints is preserved.

Using indices to test legality

The lecture then translated the informal rule into an index-based predicate. Given an order $\langle v_1, v_2, \ldots, v_n \rangle$, pick any two distinct indices i and j. If (v_i, v_j) is an edge in the DAG, then the order is legal only if $i < j$. If the two vertices do not form an edge, their relative order is unconstrained. This is the precise reasoning pattern to use when checking whether a proposed topological order is legal.

Why topological "sort" is not comparison sorting

Jackie stressed that topological sort is not sorting by numeric or lexicographic values. The word "sort" here means arranging vertices according to dependency edges in a DAG. The output is a single legal topological order, not necessarily the only one. This distinction matters because the algorithm is based on graph traversal, especially DFS, rather than on comparing data values.

DFS-based topological sort

The core algorithm was presented as a slight extension of DFS. The top-level method initializes a shared `order` list, then scans all vertices and calls `DFS_topo` on any unvisited vertex. This outer loop is necessary because the DAG may have multiple components. Inside `DFS_topo`, vertices are pushed onto a stack, outgoing edges are examined to discover unvisited destinations, and a vertex is added to the front of `order` only when the algorithm backtracks from it.

Running time and the role of `addFirst`

The DFS-based algorithm remains linear: each vertex is pushed and popped once, and each edge is considered once, so the running time is $O(|V| + |E|)$. The main extension is the shared order list. Jackie emphasized `addFirst`: a vertex is prepended when it backtracks, so the final order is the reverse of the backtracking order. Appending instead would reverse the intended order.

Tracing topological sort on a sample DAG

Jackie traced the algorithm on a DAG with vertices A, B, C, D, E, F. Under the lecture's chosen traversal order, the first DFS run from E produced the partial order $\langle E, F, D\rangle$ after backtracking through D, F, and E. A second run from B prepended B, C, giving $\langle B, C, E, F, D\rangle$. A final run from A prepended A, yielding $\langle A, B, C, E, F, D\rangle$. The exact order depends on traversal choices, but the method and legality test are the same.

Sequentializing parallel transactional updates

The final topic introduced a new application: parallel constraints involving pre-state variables such as a, b, c, d and post-state variables such as a', b', c', d'. These constraints are not ordinary sequential assignment statements; they specify values that should hold after a conceptual simultaneous update. Jackie framed the upcoming task as converting such parallel constraints into a correct sequence of assignments by identifying dependencies, building a DAG, and then topologically sorting it.

Key Terms, Notation, and Structures

- **DAG**: A directed acyclic graph; its directed edges encode dependency constraints, and its lack of cycles rules out circular dependencies.
- **Dependency edge** $v \to u$: A directed edge meaning that v must occur before u.
- **Topological order**: A linear sequence containing every vertex exactly once and respecting every dependency edge.
- **Topological sort**: An algorithm that takes a DAG as input and returns one legal topological order.
- **Legal order criteria**: Every vertex appears in the order, and every edge $(x, y) \in E$ has x appearing before y.
- **Index formulation**: For $\langle v_1, \ldots, v_n\rangle$, a legal order satisfies $\forall i, j$ with $1 \le i, j \le n$ and $i \ne j$, if $(v_i, v_j) \in E$, then $i < j$.
- **Unconstrained pair**: Two vertices with no edge between them; their relative order may change without affecting legality, unless another dependency indirectly constrains them.

- **Predecessor/successor**: In Jackie's discussion of linear orders, these refer to immediate sequence neighbors, not merely any earlier or later vertex.
- `DFS_topo`: A DFS-style helper used by topological sort to visit outgoing edges and prepend a vertex to the order when it backtracks.
- `order`: The shared list that accumulates the topological order across all DFS runs; it should not be reinitialized for each component.
- `addFirst`: The list operation used when a vertex backtracks; it makes the final order the reverse of the DFS finishing order.
- **Discovery edge**: In the DFS trace, an outgoing edge whose destination has not yet been visited.
- **Back edge**: An edge to an ancestor in DFS; finding one would indicate a cycle, so it should not occur in a DAG.
- $O(|V| + |E|)$: The running time of DFS-based topological sort, because vertices and edges are each processed a constant number of times.
- **Pre-state and post-state**: In the update example, unprimed variables such as a refer to values before the update, while primed variables such as a' refer to values after the update.

Key Examples and Demonstrations

Course prerequisite DAG

Setup: Courses were drawn as vertices, with directed edges representing prerequisite dependencies such as $1011 \to 1021 \to 2030 \to 2101 \to 3101$, plus additional dependencies such as $1019 \to 2101$, $1090 \to 3101$, and $1801 \to 1802$.

Main point: The diagram showed that a DAG can model real dependency constraints and can be disconnected; separate components give flexibility in the final linear order.

What to practise: Redraw the course DAG and write at least two different legal topological orders. For each proposed order, verify every edge one by one.

Checking a proposed topological order

Setup: Jackie checked a proposed sequence such as

$$\langle 1801, 1011, 1021, 1019, 2030, 1090, 2101, 3101, 1802 \rangle.$$

Main point: The order is legal because every edge's origin appears somewhere before its destination. For example, 1021 appears before 2030, and 1801 appears before 1802, even though many other vertices may be interleaved.

What to practise: Take a legal order and swap two vertices that have no edge constraint between them, such as 1021 and 1019 in the lecture example. Explain why the new order remains legal or identify which edge it violates.

Formal legality using indices

Setup: The lecture indexed a linear order as $\langle v_1, v_2, \ldots, v_n \rangle$.

Main point: The predicate $(v_i, v_j) \in E \Rightarrow i < j$ captures the exact meaning of "the origin occurs before the destination". It applies only to vertex pairs that are actually edges in the DAG.

What to practise: Given a small DAG and a candidate order, label the indices and check the predicate for every edge. Do not check non-edge pairs as if they were required dependencies.

DFS-based topological sort trace

Setup: The sample DAG used vertices A, B, C, D, E, F, and the trace selected unvisited start vertices in a chosen order for illustration.

Main point: The trace showed how `DFS_topo` accumulates the order by prepending vertices on backtracking. One run from E yielded $\langle E, F, D \rangle$, the next run from B yielded $\langle B, C, E, F, D \rangle$, and the final run from A yielded $\langle A, B, C, E, F, D \rangle$.

What to practise: Recreate the stack trace, explicitly marking push, pop, visited status, and the changing `order` list. Then verify the final order against the graph's edges.

Parallel transactional update constraints

Setup: Jackie introduced constraints such as

$$c' = b' - a' + d \quad \wedge \quad b' = a' * d' \quad \wedge \quad d' = a + c + 5 \quad \wedge \quad a' = a + d'.$$

The task is to sequentialize the specified transactional updates.

Main point: These are not ordinary assignments to execute top to bottom. They describe required relationships between pre-state and post-state values. The dependency structure can be modeled as a DAG, and topological sort can help choose a valid assignment order.

What to practise: Before reviewing the next lecture, identify which post-state variables depend on which other post-state variables. Do not try to finish the full solution until you can justify the dependency graph.

Common Pitfalls and Misconceptions

Treating every earlier-later pair as a dependency

Pitfall: Assuming that if one vertex appears before another in a legal order, there must be an edge between them.

Why it causes problems: A topological order may include many unconstrained pairs. Over-reading the sequence can make you reject valid orders or invent dependencies that are not in the DAG.

How to avoid it: Always check the DAG's edges. Only an edge (x, y) forces x to appear before y.

Confusing "before" with "immediately before"

Pitfall: Thinking that if (x, y) is an edge, then x must be the immediate predecessor of y in the topological order.

Why it causes problems: The lecture examples allow other vertices to appear between an edge's origin and destination. Immediate adjacency is not required.

How to avoid it: Use the index rule: for edge (v_i, v_j), require $i < j$, not $j = i + 1$.

Assuming a topological order is unique

Pitfall: Looking for the one correct order.

Why it causes problems: DAGs often have many valid topological orders, especially when vertices are in different components or not connected by dependency edges.

How to avoid it: Verify an order by the two criteria instead of comparing it against a single expected sequence.

Using comparison-sort thinking for topological sort

Pitfall: Treating topological sort like sorting numbers, strings, or keys.

Why it causes problems: Topological sort is not based on value comparison. It is based on dependency edges, so ordinary sorting intuition does not explain the algorithm.

How to avoid it: Before sorting, identify what each edge means and which order constraints it imposes.

Appending on backtracking instead of prepending

Pitfall: Using ordinary `add` at the end of the list when a vertex backtracks.

Why it causes problems: Jackie emphasized that `addFirst` is the crucial extension: the final order is the reverse of backtracking order. Appending would produce the wrong direction.

How to avoid it: When tracing, write down the pop/backtracking order first, then reverse it or use `addFirst` at every backtrack.

Reinitializing the order list for each DFS run

Pitfall: Treating each component's DFS as a separate topological sort with its own result list.

Why it causes problems: A disconnected DAG still needs one topological order containing all vertices exactly once.

How to avoid it: Keep one shared `order` object and pass it into each `DFS_topo` call.

Reading parallel constraints as sequential assignments

Pitfall: Executing the displayed equations from top to bottom as if they were ordinary program statements.

Why it causes problems: The constraints describe simultaneous post-state relationships, not an already chosen execution sequence. A top-to-bottom assignment order may use a value before it has been correctly computed.

How to avoid it: First separate pre-state from post-state variables, then build the dependency graph before choosing an assignment order.

What You Should Be Able to Do

After this lecture, you should be able to:

- define a directed acyclic graph and explain why cycles correspond to circular dependencies;
- interpret a directed edge $v \to u$ as a dependency requiring v to occur before u;
- construct or read a small dependency DAG, such as a course-prerequisite graph;
- state the two criteria for a legal topological order;
- verify a proposed topological order by checking every directed edge;
- use the index rule $(v_i, v_j) \in E \Rightarrow i < j$ to formalize legality;
- explain why vertices with no edge constraint may appear in different relative orders;
- distinguish topological sort from comparison-based sorting;
- outline the DFS-based topological sort algorithm, including the outer loop over unvisited vertices;
- trace `DFS_topo` using a stack, visited markers, and a shared `order` list;
- explain why `addFirst` is used when a vertex backtracks;
- justify the $O(|V| + |E|)$ running time of DFS-based topological sort;
- explain why a discovered back edge would violate the DAG precondition;
- describe the pre-state/post-state idea in the parallel update example;
- prepare for Programming Test 2 by reviewing Assignment 2's edge list implementation and Assignment 3 materials.

Suggested Practice

- Redraw the course-prerequisite DAG and produce at least three legal topological orders; for each order, mark which edge constraints are being checked.
- Take one legal order and deliberately swap two vertices. Decide whether the swap preserves legality by checking the affected edges, not by intuition alone.
- Rewrite the legality condition using indices for a new small DAG, then test a proposed order against the formula.
- Retrace the A, B, C, D, E, F DFS example, recording the stack contents, the visited set, each backtracking event, and the evolving `order` list.
- Modify the DFS trace by choosing a different unvisited start vertex, then check whether the resulting order is still legal.
- Explain in your own words why `addFirst` gives the reverse of the backtracking order and why that is what topological sort needs.
- Review the graph representation material from the previous lectures, especially edge lists for Programming Test 2 and adjacency list/matrix ideas for later exam preparation.
- Review Assignment 2's solution and the Assignment 3 guide/solution materials as Jackie releases them; Programming Test 2 is based on Assignments 2 and 3.
- For the parallel update problem, list the post-state variables and sketch which ones appear on the right-hand side of other equations. Save the full sequentialization for the follow-up solution discussion.

Where to Review

- **iPad/PDF notes, page 1**: lecture topic overview: DAGs, topological sort time complexity, tracing, and linearizing updates.
- **iPad/PDF notes, page 2**: announcements about the finalized exam time, released assignments, Programming Test 2 date, and guide.
- **iPad/PDF notes, page 3**: DAG definition, dependency interpretation of direction, the course-prerequisite DAG, and the first topological order criteria.
- **iPad/PDF notes, page 4**: index-based formulation of legal topological orders.
- **iPad/PDF notes, page 5**: examples of legal orders and the idea that moving unconstrained vertices does not necessarily make an order illegal.
- **iPad/PDF notes, pages 6–7**: topological sort as a DFS extension, the shared `order` list, `DFS_topo`, and $O(|V| + |E|)$ running time.

- **iPad/PDF notes, page 8**: the main DFS trace producing $\langle A, B, C, E, F, D \rangle$.
- **iPad/PDF notes, page 9**: additional activity-DAG practice for topological ordering.
- **iPad/PDF notes, page 10**: the parallel transactional update example and the sequentialization task.
- **Lecture recording**: revisit the spoken explanation of the course-prerequisite example, the index predicate, the DFS trace, and the warnings about `addFirst` and non-unique orders.
- **Assignments and test preparation**: review Assignment 2's edge list implementation, Assignment 3 materials, and the Programming Test 2 guide.

Closing Reminder

This digest is a roadmap for reviewing the lecture, not a replacement for the lecture recording, slides, Jackie's iPad notes, assignments, or your own tracing practice. To learn this material, you should actively redraw DAGs, check topological orders against edge constraints, trace the DFS-based algorithm with a stack, and explain why `addFirst` produces the correct order. For Programming Test 2, also return to the Assignment 2 and Assignment 3 materials rather than relying on this digest alone.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 20, July 16

Topological Sort for Linearizing Updates, Tracing Kruskal's MST Algorithm

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 10

Closing Reminder 10

Main Ideas

- This lecture finished the topological-sort application introduced previously: sequentializing parallel transactional updates by turning post-state dependencies into a DAG.
- Jackie emphasized the difference between logical equalities, which specify simultaneous constraints, and programming assignments, whose order matters and may overwrite values.
- A naive implementation that follows the written order of the constraints can mix pre-state and post-state values and therefore compute the wrong result.
- The correct workflow is to identify dependencies among post-state variables, build the DAG, perform topological sort, and then translate the resulting order into assignments.
- Pre-state values must be preserved separately before the assignments begin, so later computations can still refer to the original values when the specification requires them.
- The lecture then began the final graph topic: minimum spanning trees (MSTs) for simple, undirected, weighted graphs.
- An MST was presented as a minimum-cost spanning tree: it covers all vertices, has no cycles, and contains exactly $|V| - 1$ edges.
- Kruskal's algorithm was introduced as a greedy algorithm that repeatedly considers the lowest-weight remaining edge and adds it only when it connects two different clusters.
- For Programming Test 2, Jackie reminded students to redo the programming parts of Assignments 2 and 3, review the released guide and solutions, and practise adapting DFS and BFS to solve graph problems.

Roadmap

Programming Test 2 preparation and logistics

Jackie began by reminding students that Programming Test 2 is on Monday, July 20 and covers the programming parts of Assignments 2 and 3. Assignment 3's programming solution has been released, and the guide for Programming Test 2 is also available. Jackie recommended a preparation pattern similar to Programming Test 1: redo the assignments first, then practise adapting graph traversal methods to answer related questions. In particular, students should revisit the graph slide about extending DFS to answer questions such as path existence and number of connected components; the same kind of adaptation also applies to BFS.

Jackie also reminded students that Assignment 3's written part is still due next week and should be attempted before relying on the solution. The final exam date is fixed, and more details about format and location will be shared later.

Parallel constraints are not ordinary assignments

The lecture then returned to the topological-sort example from the previous class. The input was a conjunction of logical constraints such as

$$c' = b' - a' + d, \quad b' = a' \cdot d', \quad d' = a + c + 5, \quad a' = a + d'.$$

The prime notation denotes post-state values, while unprimed variables denote pre-state values. Jackie stressed that these are logical equalities describing a simultaneous update, not sequential programming assignments. The order of conjuncts in the specification is not automatically a safe execution order.

Why the naive implementation fails

To show the danger, Jackie used pre-state values $a = 1$, $b = 2$, $c = 3$, and $d = 4$. A correct evaluation gives $d' = 9$, $a' = 10$, $b' = 90$, and $c' = 84$. However, if students simply translate the constraints in their written order into assignments such as $c := b - a + d$, then $b := a \cdot d$, then $d := a + c + 5$, the value of c has already been overwritten before d is computed. The later calculation then uses a post-state c where the specification required the pre-state c, producing an incorrect result.

Building the dependency DAG

The next step was to ask what the vertices and arrows in the DAG should mean. The vertices are the post-state variables a', b', c', and d'. An arrow $x' \to y'$ means that x' must be evaluated before y' because y' depends on x' on the right-hand side of its defining constraint. For example, $b' = a' \cdot d'$ gives two edges, $a' \to b'$ and $d' \to b'$; $c' = b' - a' + d$ gives $b' \to c'$ and $a' \to c'$; and $a' = a + d'$ gives $d' \to a'$. The constraint $d' = a + c + 5$ uses only pre-state variables on the right-hand side, so it contributes no post-state dependency edges.

Jackie warned against doing extra simplification while drawing the graph. The DAG should faithfully record each direct dependency from the input. Simplification, optimization, or reasoning about which edges feel redundant is a separate concern; the graph-construction phase should represent the specification directly.

Topological sort gives the assignment order

Starting a DFS-based topological sort at d', and following the demonstrated visitation order, the trace pushed d', then a', then b', then c'. The backtracking sequence was c', b', a', d', and because the DFS-based topological sort adds finished vertices to the front of the order, the usable order is

$$\langle d', a', b', c' \rangle.$$

This is the order Jackie then used for the implementation: compute d first, then a, then b, then c.

Preserving pre-state values in the implementation

The topological order alone is not enough if assignments overwrite variables that later expressions still need in their pre-state form. Jackie therefore introduced the implementation strategy of first saving the pre-state values, for example using variables such as `preA`, `preB`, `preC`, and `preD`. After that, assignments can safely follow the topological order:

$$d := \texttt{preA} + \texttt{preC} + 5,$$
$$a := \texttt{preA} + d,$$
$$b := a \cdot d,$$
$$c := b - a + \texttt{preD}.$$

This final translation connects the abstract DAG and topological sort back to executable sequential code.

Minimum spanning trees

After the break, Jackie introduced the minimum spanning tree problem. The input is assumed to be a simple, undirected, weighted graph. A tree is an undirected, acyclic, connected subgraph; a spanning tree is a tree that covers all vertices; and a minimum spanning tree is a spanning tree whose total edge cost is as small as possible. Jackie emphasized that a spanning tree on $|V|$ vertices contains $|V|-1$ edges, and that the output can be represented as a set of edge objects, such as a `HashSet` of edges.

Kruskal's algorithm: first trace

The lecture ended by beginning a trace of Kruskal's MST algorithm. Kruskal's uses a priority queue whose entries are edges keyed by edge weight, unlike Dijkstra's algorithm, whose priority queue stores vertices keyed by distance estimates. The algorithm starts with each vertex in its own cluster and $T = \varnothing$. At each iteration it considers the minimum-weight remaining edge. If the edge connects two different clusters, the edge is accepted into T and the two clusters merge. If the endpoints are already in the same cluster, the edge is an internal edge and is rejected because adding it would create a cycle. The trace accepted edges such as (A, B), (B, C), (A, D), (E, F), and (E, D), and rejected (D, C) because D and C were already in the same cluster.

Key Terms, Notation, and Structures

- **Logical equality** =: A constraint in the specification; its conjuncts describe required relationships and are not executed one by one.
- **Programming assignment** :=: A sequential update operation; changing the order of assignments can change the result.
- **Pre-state variable**: An unprimed variable such as a, b, c, or d, referring to the value before the transactional update.

- **Post-state variable**: A primed variable such as a', b', c', or d', referring to the value after the transactional update.
- **Dependency DAG**: A directed acyclic graph whose vertices are post-state variables and whose edges record which post-state values must be computed first.
- **Topological order**: A legal linear order of DAG vertices such that every edge points from an earlier vertex to a later vertex.
- **DFS-based topological sort**: A modified DFS in which a vertex is added to the front of the output order when it backtracks.
- **Minimum spanning tree (MST)**: A spanning tree of a weighted graph with minimum total edge weight among all spanning trees.
- **Spanning tree size**: If a graph has $|V|$ vertices, then any spanning tree has exactly $|V|-1$ edges.
- **Tree as a set of edges**: In this lecture, T denotes the set of selected edges forming the spanning tree being built.
- **Tree weight**: For a tree T, $w(T) = \sum_{(u,v)\in T} w(u,v)$.
- **Greedy method**: An algorithm design pattern that extends a partial solution by repeatedly making the best available local choice.
- **Kruskal's algorithm**: A greedy MST algorithm that considers edges in increasing weight order and accepts an edge only if it joins two different clusters.
- **Cluster**: In the Kruskal trace, a set of vertices currently connected by the edges selected so far.
- **Internal edge**: An edge whose endpoints are already in the same cluster; Kruskal's rejects it because it would create a cycle.
- **Priority queue in Kruskal**: Stores edges with edge weight as the key.
- **Priority queue in Dijkstra**: Stores vertices with the current distance estimate as the key.

Key Examples and Demonstrations

Naive sequentialization of parallel constraints

Setup: The constraints used $c' = b' - a' + d$, $b' = a' \cdot d'$, $d' = a + c + 5$, and $a' = a + d'$, with sample pre-state values $a = 1$, $b = 2$, $c = 3$, and $d = 4$.

Main point: A naive sequence of assignments can overwrite a value too early. In Jackie's trace, updating c first caused a later computation of d to use the new c when the specification required the old c.

What to practise: Recreate the trace twice: once using the naive order and once using the dependency-respecting order. Explain exactly where the wrong value first enters the computation.

Constructing the dependency DAG

Setup: The DAG has vertices a', b', c', and d'. Edges come only from post-state variables appearing on the right-hand side of another post-state definition.

Main point: Each direct dependency becomes its own edge. For example, $b' = a' \cdot d'$ contributes $a' \to b'$ and $d' \to b'$, while $d' = a + c + 5$ contributes no edge because it uses only pre-state values.

What to practise: Given a new system of primed/unprimed equations, underline every post-state variable on the right-hand side and draw one arrow from that variable to the post-state variable being defined.

DFS-based topological sort on the dependency DAG

Setup: Starting from d', the trace pushed d', a', b', and c' and then backtracked in the reverse direction.

Main point: The assignment order is obtained by adding finished vertices to the front of the order. The resulting usable order was $\langle d', a', b', c' \rangle$.

What to practise: Trace the same DAG using the alternative incident-edge visitation choice Jackie mentioned. Check that the resulting order still respects the dependencies.

Correct implementation with saved pre-state values

Setup: Before overwriting a, b, c, or d, Jackie saved the old values in separate pre-state storage.

Main point: The topological order tells you when post-state variables can be assigned, but the implementation also needs preserved pre-state values whenever an expression refers to an unprimed variable.

What to practise: Write the final sequential code from the topological order and annotate each occurrence of a variable as either pre-state storage or current post-state variable.

MST definition from the airport graph

Setup: The iPad notes showed a weighted, undirected graph and a highlighted spanning-tree output.

Main point: An MST is a set of edges that spans all vertices, remains acyclic and connected, and minimizes the sum of weights. With 9 vertices, the spanning tree shown has 8 edges.

What to practise: For a smaller weighted graph, list two different spanning trees, compute their total weights, and explain which one is cheaper.

Kruskal trace with clusters

Setup: The trace began with vertices A through H, each in its own cluster, and considered edges in increasing weight order.

Main point: Accepted edges merge clusters and enter T; rejected internal edges do not change either the clusters or T. The edge (D, C) was rejected because D and C were already in the same cluster.

What to practise: Continue the trace from where lecture stopped, always writing the minimum edge considered, the cluster change, and the current set T.

Common Pitfalls and Misconceptions

Treating parallel constraints as ordinary assignments

Pitfall: Reading a conjunction of logical constraints from top to bottom and translating it directly into assignments.

Why it causes problems: Assignments overwrite variables. A later expression may accidentally use a post-state value when the specification requires a pre-state value.

How to avoid it: First identify dependencies, build the DAG, topologically sort it, and save pre-state values before beginning the assignments.

Putting pre-state variables into the dependency DAG

Pitfall: Drawing vertices for both a, b, c, d and a', b', c', d'.

Why it causes problems: The topological-order problem in this lecture is about dependencies among post-state computations. Pre-state values are inputs to those computations, not computations that need to be scheduled.

How to avoid it: Use only post-state variables as DAG vertices, and handle pre-state values separately through saved variables such as `preA` and `preC`.

Reversing dependency arrows

Pitfall: Drawing $b' \to a'$ from $b' = a' \cdot d'$.

Why it causes problems: That arrow says b' must be computed before a', which is the opposite of the dependency.

How to avoid it: Read every arrow as "must be evaluated before". If y' uses x' on its right-hand side, draw $x' \to y'$.

Simplifying the DAG too early

Pitfall: Replacing multiple direct dependencies with a guessed chain because the chain seems to imply the same order in a small example.

Why it causes problems: A guessed simplification may omit a real dependency, and it mixes graph construction with later analysis.

How to avoid it: Separate concerns: first draw every direct dependency from the specification, then let topological sort produce an order.

Forgetting that a spanning tree has exactly $|V| - 1$ edges

Pitfall: Expecting Kruskal's algorithm to accept as many edges as it considers.

Why it causes problems: Kruskal may consider and reject edges that would create cycles, so the number of iterations can be larger than $|V| - 1$.

How to avoid it: Count accepted edges in T, not considered edges. Stop only when T has $|V| - 1$ accepted edges.

Adding internal edges in Kruskal's algorithm

Pitfall: Adding a low-weight edge even though its endpoints are already in the same cluster.

Why it causes problems: The selected edge set would become cyclic, so it would no longer be a tree.

How to avoid it: Before accepting an edge, check the clusters of its two endpoints. Accept only if the clusters are different.

Confusing Kruskal's priority queue with Dijkstra's priority queue

Pitfall: Assuming the two algorithms use the same kind of priority-queue entries and keys.

Why it causes problems: Dijkstra prioritizes vertices by distance estimates, while Kruskal prioritizes edges by weights.

How to avoid it: For each algorithm, state the priority-queue entry type and key before tracing.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain why logical constraints with primed variables should not be implemented by blindly following their written order;
- distinguish pre-state variables, post-state variables, logical equality, and programming assignment;
- construct the dependency DAG for a system of post-state constraints;
- draw the correct direction for dependency edges and justify each edge using the right-hand side of a constraint;
- trace a DFS-based topological sort on the dependency DAG and obtain a valid computation order;
- translate a topological order into sequential assignments while preserving required pre-state values;

- define tree, forest, spanning tree, minimum spanning tree, and tree weight in the MST setting;
- justify why an MST on $|V|$ vertices has $|V| - 1$ edges;
- trace Kruskal's algorithm by maintaining a priority queue of edges, clusters of vertices, and the growing edge set T;
- identify when Kruskal rejects an internal edge and explain why adding it would create a cycle;
- compare the priority-queue roles in Dijkstra's algorithm and Kruskal's algorithm;
- prepare for Programming Test 2 by redoing Assignments 2 and 3 and practising DFS/BFS adaptations.

Suggested Practice

- Redo the programming parts of Assignments 2 and 3 before attempting extra practice for Programming Test 2.
- Revisit the graph slide Jackie mentioned about adapting DFS to answer path and connected-component questions; write small methods for those tasks, then try analogous BFS versions.
- Recreate the parallel-constraint example with $a = 1$, $b = 2$, $c = 3$, and $d = 4$; compute both the naive result and the correct result.
- Draw the dependency DAG for the four primed variables without looking at the notes, then compare every edge with the original constraints.
- Trace the DFS-based topological sort twice: once using Jackie's order and once using the alternative choice discussed in class.
- Write the correct sequential implementation using saved pre-state values, and label each variable occurrence as pre-state or post-state.
- For MSTs, take a small weighted undirected graph, propose a spanning tree, verify it has $|V| - 1$ edges, and compute its total weight.
- Continue the Kruskal trace from the lecture: for each iteration, record the minimum edge, whether it is accepted or rejected, the updated clusters, and the current T.
- Explain in words why rejected internal edges are safe to skip and why accepted edges merge clusters.

Where to Review

- Review the opening of the lecture recording for Programming Test 2 preparation advice, especially Jackie's recommendation to redo Assignments 2 and 3 before extra DFS/BFS practice.
- Review the iPad/PDF notes page 2 for the final exam, Assignment 2, Assignment 3, and Programming Test 2 reminders.
- Review the iPad/PDF notes page 4 for the wrong implementation of the parallel constraints and the sample pre-state/post-state calculation.
- Review the iPad/PDF notes page 3 for the dependency DAG, DFS-topological-sort trace, and correct implementation order $\langle d', a', b', c' \rangle$.
- Review the previous lecture's DAG and topological-sort material if the meaning of a topological order, DFS backtracking, or add-first construction is unclear.
- Review the iPad/PDF notes page 5 for the MST problem definition, including tree, spanning tree, minimum cost, $|T| = |V| - 1$, and representing the output as a set of edges.
- Review the iPad/PDF notes page 6 for the Kruskal trace table, especially the cluster column and the rejection of the internal edge (D, C).
- Use the released Programming Test 2 guide and the Assignment 3 programming solution as practice references, not just reading material.

Closing Reminder

This digest is only a roadmap for reviewing Lecture 20. To learn the material, revisit the recording, Jackie's iPad notes, graph slides, assignment solutions, and Programming Test 2 guide. The most important active work is to trace the topological-sort example yourself, write the sequential implementation with saved pre-state values, and continue the Kruskal trace while checking clusters and accepted edges. Reading the digest alone is not enough preparation for the programming test, later graph material, or the final exam.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 21, July 21

Topological Sort: DFS vs. BFS, Greedy Method, Partition vs. Cut

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 10

Closing Reminder 10

Main Ideas

- Jackie confirmed that the final exam is 10:00 AM-1:00 PM on Sunday, August 9, with the room shown in the notes as LAS B, and reminded students that the exam is worth 50% of the final grade.
- The lecture first closed the topological-sort discussion by explaining why, in DFS-based topological sort, the final order is controlled by backtracking order, not by the order in which adjacent vertices are first pushed.
- A small DAG with edges (S, A), (S, B), and (B, A) was used to show why BFS is not a suitable basis for topological sort.
- The Kruskal trace from the previous lecture was completed: internal edges were discarded, useful edges merged clusters, and the final MST contained exactly $|V| - 1$ edges.
- Jackie emphasized that Kruskal's algorithm may take more than $|V| - 1$ iterations because some extracted edges can be rejected as internal edges.
- The lecture introduced the greedy-method pattern: build a solution step by step, choose among feasible extensions, make the locally cheapest choice according to a cost function, and never undo a chosen step.
- Dijkstra's algorithm was revisited as a greedy algorithm by matching its finalized set, priority queue, D-value cost function, and irreversible removal step to the greedy pattern.
- The final part introduced partition, cluster, and $C(v)$ notation as preparation for the cut and cut-property discussion used to reason about Kruskal's correctness.

Roadmap

Announcements and exam preparation context

Jackie opened by shifting attention from tests to the final exam. The final exam was confirmed for 10:00 AM-1:00 PM on Sunday, August 9, with the location shown in the notes as LAS B. Jackie also reminded students that Assignment 3 was due Wednesday, July 22, and that the written-part solution would be released after the deadline. The written test grading and Programming Test 2 grading were still in progress, with results intended to be released as soon as possible, ideally before the withdrawal deadline.

The study implication was direct: students should begin preparing for the 50% final exam, watch for Jackie's later exam-format guidance, and review Assignment 3 carefully once the solution is posted, especially the inductive proof for Dijkstra's correctness.

Revisiting topological sort: edge visiting order versus backtracking order

The lecture first returned to the parallel-update example from the previous class. The DAG had post-state vertices such as A', B', C', and D', where an edge means one post-state value must be evaluated before another. Jackie revisited the question of what happens if, starting from D', the DFS visits B' before A' rather than following the earlier alphabetical route.

The important point is that the push order can change, but the correctness of DFS-based topological sort comes from the backtracking order. A vertex is added to the topological order only when all vertices depending on it have already been handled. Therefore, the final ordering is tied to when vertices are popped/backtracked, not merely when they are discovered.

Why BFS is not the right traversal for topological sort

Jackie then used the small DAG

$$g = (\{S, A, B\}, \{(S, A), (S, B), (B, A)\})$$

to contrast DFS and BFS. In the DFS/topological-sort version, the stack discipline allows a vertex to backtrack only after all relevant outgoing-dependency work has been completed. For example, A can backtrack when no unvisited vertex depends on it, B can backtrack after A is already handled, and then S can backtrack after both have been handled.

BFS behaves differently. It enqueues adjacent vertices and can dequeue S after placing A and B in the queue, but that dequeue action is not based on dependency completion. The FIFO queue order does not give the same "finish after dependents" guarantee that DFS backtracking gives. This is the main reason Jackie emphasized that BFS should not be used to implement topological sort.

Completing Kruskal's MST trace

The lecture then resumed the Kruskal trace from the previous lecture. The input graph had eight vertices, so any spanning tree must contain $|V| - 1 = 7$ edges. However, this does not mean Kruskal must finish in exactly seven iterations: internal edges may be extracted from the priority queue and rejected.

Starting from the partial state with five accepted edges, Jackie traced the remaining steps. The edge (C, F) of weight 6 was an internal edge because its endpoints were already in the same cluster, so it was rejected. The edge (F, G) of weight 7 connected different clusters and became e_6. The edge (E, H) of weight 8 connected the last remaining cluster, became e_7, and completed the MST. The trace also highlighted that choosing (G, H) of weight 9 would still connect the graph but would not give a minimum spanning tree because (E, H) was cheaper.

Tie-breaking and non-unique MSTs

Jackie pointed out that in an earlier step there was a tie between two weight-3 edges, such as (A, D) and (D, C). Either choice can be acceptable if the question gives no tie-breaking

rule. If a test or exam question specifies a tie rule, such as alphabetical order, students must follow that rule. This also illustrates that an MST is not necessarily unique.

The greedy-method pattern

After the trace, Jackie abstracted from Kruskal and Dijkstra to the greedy method. A greedy algorithm is iterative, but not every iterative algorithm is greedy. The pattern has four ingredients: construct a complete solution step by step; maintain multiple feasible choices for extending the partial solution; choose the locally best or cheapest option according to a cost or score function; and do not undo earlier choices.

The visualization on the notes showed a partial solution growing toward an imagined complete solution. The greedy step chooses an extension that stays within the intended optimal solution, so the algorithm can keep expanding without backtracking or replacing earlier choices. Jackie also emphasized that greedy algorithms are often easy to trace, but proving their correctness is usually the harder part.

Dijkstra as an example of a greedy algorithm

Jackie then revisited Dijkstra's shortest-path pseudocode and matched it to the greedy pattern. The while loop builds the solution step by step. The set of feasible choices is stored in the priority queue Q, containing vertices not yet finalized. The greedy step is to extract the vertex with minimum current D-value. The no-regret step is that once a vertex is finalized and removed from Q, it is never put back for further processing.

The lecture also noted that after each greedy step, the solution is still only partial: the finalized set S still needs later iterations to extend it until all required vertices are finalized.

Partition, cluster, and $C(v)$ notation

The final technical segment prepared students for the later cut-property proof of Kruskal's correctness. Jackie introduced a *partition* as a set whose members are sets of vertices, and a *cluster* as one member of such a partition. In Kruskal's trace, the initial partition has every vertex in its own singleton cluster; the final partition has all vertices in one cluster.

The notation $C(A)$ means the cluster containing vertex A at the current moment in the algorithm. This cluster can grow as tree edges merge clusters. If $C(A) \neq C(B)$, then edge (A, B) connects two different clusters and can be a tree/discovery edge. If $C(A) = C(B)$, then edge (A, B) is internal and should not be added because it would create a cycle.

Key Terms, Notation, and Structures

- **Topological sort**: A DFS-based method for producing a legal linear order of a DAG, where vertices are added according to finishing/backtracking order.
- **Backtracking order**: The order in which vertices finish and are popped from the DFS stack; this is the order that matters for DFS-based topological sort.

- **Push order**: The order in which vertices are first discovered and placed on the DFS stack; Jackie emphasized that this is not the same thing as the final topological order.
- **BFS queue discipline**: BFS uses a FIFO queue and dequeues a vertex after enqueuing adjacent vertices, not after dependency completion, which is why it is not suitable for topological sort.
- **MST**: A minimum spanning tree is a spanning tree of a weighted, undirected graph whose total edge weight is minimum among all spanning trees.
- **Spanning-tree size**: If a spanning tree covers $|V|$ vertices, then its edge set T satisfies $|T| = |V| - 1$.
- **Kruskal's algorithm**: A greedy MST algorithm that repeatedly extracts the lowest-weight remaining edge and adds it only if the edge connects two different clusters.
- **Priority queue in Kruskal**: The priority queue stores edges, keyed by edge weight $w(e)$, so extracted edge weights appear in non-descending order.
- **Tree edge or discovery edge**: In Kruskal's trace, an edge whose endpoints are in different clusters; adding it merges those clusters and extends T.
- **Internal edge**: An edge whose endpoints are already in the same cluster; adding it would create a cycle, so Kruskal discards it.
- **Greedy method**: An algorithm design pattern that constructs a solution iteratively by making a locally best feasible choice and never undoing it.
- **Cost or score function**: A function used to rank feasible choices, such as $D(v)$ in Dijkstra's algorithm or $w(e)$ in Kruskal's algorithm.
- **No regret**: The greedy-algorithm principle that once a choice is made, the algorithm does not later replace or undo it.
- **Dijkstra's finalized set** S: The partial solution in Dijkstra's algorithm: vertices whose shortest-path information has been finalized.
- **Dijkstra's priority queue** Q: The collection of remaining feasible choices, keyed by the current D-value.
- **Partition**: A set of non-empty, disjoint vertex sets whose union is the full vertex set.
- **Cluster**: One set of vertices inside a partition; in Kruskal's trace, clusters correspond to connected components formed by the edges accepted so far.
- $C(v)$: The current cluster containing vertex v; this notation is dynamic because clusters grow through merging.
- **Cut**: A concept announced for the next lecture; it will be used with partitions and clusters to state the cut property for Kruskal's correctness.

Key Examples and Demonstrations

Topological sort with a changed incident-edge order

Setup: Jackie reused the dependency DAG for the parallel-update example with vertices A', B', C', and D'. The trace started from D', but this time considered what happens if B' is visited before A'.

Main point: Changing the order of visiting outgoing edges can change the push order, but the DFS-based topological sort is justified by backtracking order. The example reinforced that a vertex should be finalized in the topological order only after the relevant vertices reachable through outgoing dependencies have already been handled.

What to practise: Redo the trace twice: once with A' visited before B', and once with B' visited before A'. For each trace, write the push order, pop/backtracking order, and resulting topological order.

DFS versus BFS on the S, A, B DAG

Setup: The notes gave the graph $g = (\{S, A, B\}, \{(S, A), (S, B), (B, A)\})$. The dependency $B \to A$ means B must be respected before A in any legal topological order.

Main point: DFS works because stack backtracking naturally waits until dependent work has been handled. BFS does not work for topological sort because dequeue order is driven by queue mechanics, not by the completion of dependencies.

What to practise: Draw the graph from the set notation, then run both DFS/topological sort and BFS. Explain in words why the BFS dequeue sequence is not a valid basis for topological sorting.

Completing the Kruskal trace

Setup: The Kruskal trace had an eight-vertex graph and a partially constructed tree T. Since $|V| = 8$, the completed MST needed 7 accepted edges.

Main point: Jackie completed the trace by rejecting (C, F) as an internal edge, accepting (F, G), and then accepting (E, H). The final accepted edge set had seven edges, while the full algorithm trace took nine iterations because two internal edges were considered and rejected.

What to practise: Reproduce the entire table from the notes: iteration, minimum edge, processing decision, resulting partition, and current T. Then try the tie variant where (D, C) is chosen before (A, D), and check that you still get a valid MST if all rules are followed.

Greedy method visualized

Setup: The notes illustrated a partial solution inside an imagined complete optimal solution, with several possible choices for extending it.

Main point: The greedy step chooses the cheapest feasible extension right now, according to a cost function, and the algorithm keeps that choice permanently. This visualization prepares students to think beyond tracing and toward correctness arguments.

What to practise: For both Dijkstra and Kruskal, identify the partial solution, the feasible choices, the cost function, the greedy choice, and the point at which the choice becomes irreversible.

Dijkstra's algorithm as a greedy algorithm

Setup: Jackie revisited the Dijkstra pseudocode with the priority queue Q, finalized set S, current distance estimate $D(v)$, and removal from Q.

Main point: Dijkstra fits the greedy template: the solution grows one finalized vertex at a time, choices come from Q, the cost function is $D(v)$, the greedy step extracts the minimum, and the selected vertex is never returned to the queue.

What to practise: Take a Dijkstra trace you have already studied and annotate each iteration with the four greedy ingredients.

Partition and cluster notation in Kruskal's table

Setup: The last page of the notes used the Kruskal table to define partitions, clusters, and $C(A)$, the cluster containing A.

Main point: Clusters evolve by merging. The condition $C(u) \neq C(v)$ means edge (u, v) connects different clusters and can extend the tree; $C(u) = C(v)$ means the edge is internal and should be rejected.

What to practise: At each row of the Kruskal table, write $C(A)$, $C(B)$, or another requested $C(v)$, and decide whether the current edge is a tree edge or an internal edge.

Common Pitfalls and Misconceptions

Mistaking push order for topological order

Pitfall: Thinking that the order in which DFS first discovers or pushes vertices directly determines the final topological order.

Why it causes problems: Topological sort depends on when vertices finish, not merely when they are first seen. Focusing only on push order can lead to an invalid explanation of the algorithm.

How to avoid it: Track both push and pop/backtracking order in every trace. The topological order is built from finishing/backtracking information.

Trying to use BFS for topological sort

Pitfall: Assuming that because BFS and DFS are both graph traversals, either one can be modified in the same way for topological sort.

Why it causes problems: BFS dequeue order is not based on dependency completion. In the S, A, B example, the queue mechanics can process vertices in an order that does not respect the dependency $B \to A$.

How to avoid it: Use DFS-based topological sort for this course's approach, and be able to explain why stack backtracking gives the needed dependency behavior.

Confusing accepted edges with considered edges in Kruskal

Pitfall: Counting every extracted edge as part of the MST.

Why it causes problems: Kruskal considers edges in non-descending weight order, but internal edges are rejected because they would create cycles.

How to avoid it: Maintain a separate set T of accepted edges. Only add an edge to T when its endpoints are in different clusters.

Assuming Kruskal takes exactly $|V| - 1$ iterations

Pitfall: Treating $|V| - 1$ as the number of loop iterations rather than the final number of MST edges.

Why it causes problems: Internal edges can be extracted and rejected, so the loop may run more times than the number of accepted edges.

How to avoid it: In a trace, count accepted tree edges separately from total iterations. The final MST has $|V| - 1$ edges, but the trace may include rejected edges.

Assuming the MST is always unique

Pitfall: Believing there is only one possible correct MST for a graph.

Why it causes problems: Equal-weight choices can lead to different valid MSTs with the same total cost. This matters in traces and exam answers.

How to avoid it: If the question gives a tie-breaking rule, follow it. If no rule is given, choose any minimum edge consistently and justify your trace.

Treating $C(v)$ as fixed

Pitfall: Thinking that $C(A)$ always denotes the same singleton set $\{A\}$.

Why it causes problems: In Kruskal's algorithm, clusters grow by merging. After edge (A, B) is accepted, $C(A)$ becomes $\{A, B\}$, and it can grow further.

How to avoid it: Always interpret $C(v)$ relative to the current row of the trace table.

Thinking greedy correctness is automatic

Pitfall: Assuming that a locally cheapest choice must always lead to a globally optimal solution without proof.

Why it causes problems: Greedy algorithms can be easy to trace but hard to justify. The later cut-property discussion is needed because correctness is not automatic from the word "greedy".

How to avoid it: Separate three tasks: tracing the algorithm, identifying the greedy choice, and proving that the choice is safe.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain why DFS-based topological sort depends on backtracking order rather than push order;
- trace the parallel-update DAG again under a different incident-edge visiting order and still determine a legal topological order;
- draw the graph $g = (\{S, A, B\}, \{(S, A), (S, B), (B, A)\})$ and use it to explain why BFS is not suitable for topological sort;
- complete a partially filled Kruskal trace by identifying the next minimum edge, checking whether it is internal, updating clusters, and updating T;
- distinguish between an edge that is considered, an internal edge that is rejected, and a tree edge that is accepted;
- explain why an MST on $|V|$ vertices has $|V| - 1$ edges but Kruskal may require more than $|V| - 1$ iterations;
- handle equal-weight choices in Kruskal according to either a stated tie-breaking rule or a consistent valid choice when no rule is given;
- state the four ingredients of the greedy method and recognize them in an algorithm trace;
- map Dijkstra's algorithm to the greedy-method ingredients using S, Q, $D(v)$, `extract_min`, and removal from Q;
- define partition and cluster in the context of Kruskal's algorithm;
- interpret $C(v)$ as the current cluster containing vertex v, not as a fixed set;
- use $C(u) = C(v)$ and $C(u) \neq C(v)$ to decide whether an edge is internal or a tree edge;
- identify which pages of the notes to revisit before the next lecture on cut and cut property.

Suggested Practice

- Redo the topological-sort trace on the A', B', C', D' dependency graph with both incident-edge visiting orders discussed in class.
- For the S, A, B example, write one DFS/topological-sort trace and one BFS trace, then explain why the BFS trace does not provide the right dependency guarantee.
- Reproduce the full Kruskal table from the notes, including the rejected internal edges (D, C) and (C, F), the accepted edges e_1 through e_7, and the resulting partitions.
- Try the alternate tie choice in Kruskal: choose (D, C) before (A, D) when both have weight 3, then trace carefully and compare the resulting MST.
- Take Dijkstra's pseudocode and label the four greedy ingredients directly on the code: partial solution, feasible choices, cost function/greedy step, and no-regret step.

- For several rows of the Kruskal table, compute examples such as $C(A)$, $C(D)$, $C(F)$, and $C(H)$, then decide whether the current edge should merge clusters.
- After Assignment 3 is submitted, review Jackie's written solution, especially the inductive proof for Dijkstra's correctness.
- Before Thursday's class, review the Kruskal table and make sure the meanings of partition, cluster, and $C(v)$ are comfortable; they are needed for the cut-property discussion.

Where to Review

- **iPad/PDF notes, page 1:** lecture scope: topological sort with BFS versus DFS, greedy method, Dijkstra, Kruskal, and partition/cluster/cut.
- **iPad/PDF notes, page 2:** announcements: final exam time and location, posted digests, and Assignment 3 due date.
- **iPad/PDF notes, page 3:** the parallel-update DAG and the repeated topological-sort trace when B' is visited before A'.
- **iPad/PDF notes, page 4:** the S, A, B graph showing why DFS supports topological sort but BFS does not.
- **iPad/PDF notes, page 5:** the completed Kruskal trace table, including internal edges, accepted edges, clusters, and final T.
- **iPad/PDF notes, pages 6-7:** the four greedy-method ingredients and the visual model of extending a partial solution.
- **iPad/PDF notes, page 8:** Dijkstra's algorithm annotated as a greedy method.
- **iPad/PDF notes, page 9:** partition, cluster, and $C(v)$ notation in the Kruskal trace table.
- **Lecture recording:** revisit the opening logistics, the DFS versus BFS comparison, the Kruskal trace continuation, the greedy-method explanation, and the final discussion of partition and cluster notation.
- **Course materials:** review Assignment 3 and its written solution after the deadline, especially the Dijkstra correctness proof Jackie highlighted.

Closing Reminder

This digest is only a roadmap. To prepare well, revisit the recording and Jackie's iPad notes, redraw the DAG and MST diagrams, redo the DFS/topological-sort and Kruskal traces by hand, and practise explaining why each accepted or rejected edge is justified. The key work is active: trace, compare, justify, and connect the examples to the greedy-method pattern rather than only rereading the summary.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 22, July 23

Kruskal's Algorithm: Greed Method, Cut Property, and Correctness

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 10

Closing Reminder 11

Main Ideas

- This was the final graph lecture. The main technical goal was to finish the Kruskal/MST discussion by connecting the trace, the pseudocode, the greedy-method pattern, and the cut property.
- Kruskal's algorithm was reviewed as a greedy algorithm: it builds a set T of accepted edges step by step, chooses from a priority queue of edges keyed by weight, adds only edges that connect different clusters, and never revisits a removed edge.
- The lecture clarified the cluster operations behind Kruskal: $C(u)$ means the current cluster containing u; finding clusters lets the algorithm detect internal edges; merging clusters corresponds to accepting a tree edge.
- The key correctness idea was the cut property. For this course, Jackie treated it as an axiom that explains why Kruskal's locally cheapest safe edge can be trusted as part of some minimum spanning tree.
- Students were also pointed to assigned graph exercises on DFS versus BFS, back edges, cross edges, and cycles, but the main in-class technical development was Kruskal's greedy correctness argument.
- Logistically, the course has no remaining tests except the final exam. The final exam is cumulative, worth 50%, and scheduled for 10 AM–1 PM on Sunday, August 9 in LAS B.

Roadmap

Course status and exam preparation

Jackie began by reminding the class that the remaining major assessment is the final exam. The exam will be cumulative and will cover lectures, tutorials, assignments, study materials, and lecture digests. Jackie planned to discuss format, expectations, and preparation tips in the following Tuesday lecture, but also emphasized that students should not wait until then to begin studying.

Assignment 3 had already been due. Jackie said the written-part solution would be released soon and specifically told students to pay attention to the Dijkstra correctness proof, because it illustrates both mathematical-induction technique and important insight into Dijkstra's algorithm. Programming Test 2 grading was in progress.

Dynamic programming preview and remaining course plan

Jackie previewed the transition away from graphs. Lecture 23 would run during the Monday tutorial time in CRH, Lecture 24 would follow on Tuesday, and Tutorial 9 would be the final Thursday session. The next topic would be dynamic programming. Jackie described it as challenging, useful, and relevant for interviews, and strongly indicated that students should expect dynamic programming to appear on the final exam.

Reviewing Kruskal's pseudocode

The technical lecture resumed with Kruskal's algorithm for finding a minimum spanning tree of a simple, undirected, weighted, connected graph $G = (V, E)$. The output is a minimum spanning tree T, represented as a set of edges. The algorithm initializes each vertex v in its own cluster $C(v) = \{v\}$, initializes a priority queue Q containing the edges E keyed by their weights, and starts with $T = \varnothing$.

The main loop continues while $|T| \neq n - 1$, where $n = |V|$. In each iteration, the algorithm removes the minimum-weight edge (u, v) from Q. If $C(u) \neq C(v)$, the edge is added to T, and the two clusters are merged. If $C(u) = C(v)$, the edge is internal and is not added, because it would create a cycle.

Find, union, and the scope of implementation detail

Jackie named the two cluster operations behind Kruskal. The *find* operation locates the cluster containing a vertex, such as $C(u)$. The *union* or *merge* operation combines two different clusters after a tree edge is accepted.

A student asked whether one could implement clusters using a set of sets, such as a hash set of hash sets. Jackie acknowledged the question but made the boundary clear: Union-Find implementation details and Kruskal's time complexity are not required for this course or the exam. The important point for this lecture is the conceptual role of finding and merging clusters.

Kruskal as a greedy algorithm

Jackie then matched Kruskal to the four greedy ingredients introduced earlier. First, the while loop constructs the solution step by step by growing T. Second, the feasible choices are the edges still remaining in Q. Third, the greedy step is Q.`removeMin`(), which chooses the edge with minimum weight according to the cost function $w(u, v)$. Fourth, there is no regret: once an edge is removed from Q, it is never put back or processed again.

The important subtlety is that being removed from Q is not the same as being accepted into T. If the edge is internal, it is discarded. If it connects two different clusters, it is accepted as a tree edge and the corresponding clusters are merged.

Partitions, clusters, and cuts

Jackie revisited the previous lecture's definitions of *partition* and *cluster*. A partition is a set whose members are sets of vertices; a cluster is one of those member sets. Initially, each vertex is its own cluster. As Kruskal accepts tree edges, clusters grow by merging. The notation $C(A)$ means the cluster currently containing A, not a fixed singleton set.

The lecture then defined a *cut* as a way to view the vertices as two sets, typically S and $V - S$. Jackie emphasized that a cut does not necessarily match the existing clusters one-for-one. Sometimes the cut corresponds to two clusters; at other times, one side of the cut is a cluster of interest and the other side is the union of all remaining vertices.

Two cut examples from the Kruskal table

The first example used the state before iteration 9 of the Kruskal trace, where the partition had two clusters: $\{A, B, C, D, E, F, G\}$ and $\{H\}$. In this case, the cut can be viewed directly as those two clusters, and edge (E, H) crosses the cut.

The second example used the state before iteration 6, where there were more than two clusters. Jackie focused on the cluster $\{A, B, C, D\}$ as one side S, and treated all remaining vertices $\{E, F, G, H\}$ as $V - S$. Edge (D, E) crosses that cut because one endpoint is in the focused cluster and the other endpoint is outside it.

The cut property and why Kruskal is safe

The final technical idea was the cut property. Jackie described it as an axiom for this course, not something students need to prove. It applies at the end of an iteration where a tree edge is added to T. If Kruskal chooses an edge $e = (x, y)$ with minimum weight among edges crossing the cut $\{C(x), V - C(x)\}$, and $y \notin C(x)$, then adding e is a safe greedy step.

The intuition is that after adding such an edge, the growing set T is still contained in some MST. This is the loop-invariant style guarantee: after each successful tree-edge iteration, the edges chosen so far are part of some minimum spanning tree. When $|T| = n-1$, the algorithm stops, and T is the MST returned by Kruskal.

Assigned graph exercise: DFS, BFS, and cycle-related edges

The iPad/PDF notes ended with assigned exercises on DFS versus BFS, back edges, cross edges, and cycles. Jackie did not work through those exercise details in the main lecture, so students should treat the slides as practice prompts rather than as completed solutions. The key study action is to revisit the exercise materials and check which statements are true for DFS back edges and BFS cross edges.

Key Terms, Notation, and Structures

- **Kruskal's algorithm**: A greedy algorithm for finding an MST by repeatedly considering the remaining minimum-weight edge and accepting it only if it connects two different clusters.
- **Minimum spanning tree** T: The edge set being constructed; it should eventually be a spanning tree with minimum total weight.
- $n = |V|$: The number of vertices in the input graph. Kruskal exits when $|T| = n - 1$.
- **Priority queue** Q **in Kruskal**: A priority queue containing edges, keyed by edge weight $w(u, v)$.
- **Priority queue contrast with Dijkstra**: In Dijkstra, Q stores vertices keyed by current D-value; in Kruskal, Q stores edges keyed by edge weight.

- Q.`removeMin`(): The operation that retrieves and removes the minimum-weight remaining edge; the edge is not processed again later.
- $C(v)$: The current cluster containing vertex v.
- **Find operation**: Given a vertex v, locate the cluster $C(v)$ containing it.
- **Union or merge operation**: Combine two different clusters into one cluster after accepting a tree edge.
- **Union-Find**: The usual data-structure family for supporting find and union; Jackie named it but kept its implementation details outside the course/exam scope.
- **Tree edge**: In Kruskal, an edge (u, v) with $C(u) \neq C(v)$; it can be added to T and causes a merge.
- **Internal edge**: An edge (u, v) with $C(u) = C(v)$; it is rejected because adding it would create a cycle.
- **Partition**: A set of clusters, where each cluster is a set of vertices.
- **Cluster**: One member of a partition; in Kruskal, clusters correspond to the connected components formed by accepted edges so far.
- **Cut**: A two-set view of the vertices, usually written as $\{S, V - S\}$. A cut does not have to be the same thing as the current partition into clusters.
- **Crossing a cut**: An edge crosses a cut if its endpoints lie on opposite sides of the cut.
- **Cut property**: The axiom used in this lecture to justify Kruskal's greedy choice: choosing a minimum-weight edge crossing the relevant cut is safe for some MST.
- **Loop-invariant view**: After each successful tree-edge addition, the current set T is contained in some MST; this remains true until termination.
- **Back edge and cross edge**: Terms appearing in the assigned DFS/BFS exercise on cycles; the lecture notes posed these as practice questions rather than working through a full solution in class.

Key Examples and Demonstrations

Reading Kruskal's pseudocode

Setup: The notes showed the pseudocode for FIND-MST-KRUSKAL: initialize singleton clusters $C(v) = \{v\}$, put all edges into a priority queue Q keyed by weight, set $T = \varnothing$, and loop until $|T| = n - 1$.

Main point: The algorithm's structure explains both how Kruskal traces are performed and why the cluster test matters. Removing the minimum edge gives the greedy candidate;

checking $C(u) \neq C(v)$ decides whether it is safe to accept as a tree edge or must be rejected as internal.

What to practise: Rewrite the pseudocode in words. For each line, identify whether it initializes clusters, initializes choices, makes the greedy choice, checks for a cycle, extends T, or merges clusters.

Find and union in the cluster test

Setup: Jackie focused on the lines that refer to $C(u)$ and $C(v)$. The algorithm must determine the cluster containing each endpoint of the current edge.

Main point: The test $C(u) = C(v)$ means the edge stays inside one already-connected component and would form a cycle. The test $C(u) \neq C(v)$ means the edge connects two components and can be accepted. The subsequent merge is a union of the two clusters.

What to practise: In the Kruskal trace table, pick several rows and explicitly write $C(u)$ and $C(v)$ before the row is processed. Then decide whether the row is an internal edge or a tree edge.

Kruskal mapped to the greedy ingredients

Setup: Jackie revisited the four greedy ingredients and applied them directly to Kruskal's pseudocode.

Main point: The partial solution is T. The feasible choices are edges remaining in Q. The cost function is $w(u, v)$. The greedy step is Q.`removeMin`(). The no-regret feature is that each removed edge is processed only once, and accepted edges are not later removed from T.

What to practise: Make a four-row table for Kruskal with one row per greedy ingredient. Then make the same table for Dijkstra and compare the two algorithms, especially what is stored in the priority queue and what the cost function is.

Cuts at iterations 9 and 6 of the Kruskal trace

Setup: The notes used the earlier Kruskal trace table. Before iteration 9, the partition can be viewed as two clusters, $\{A, B, C, D, E, F, G\}$ and $\{H\}$. Before iteration 6, one possible cut focuses on $\{A, B, C, D\}$ and groups all other vertices as $\{E, F, G, H\}$.

Main point: These examples show that a cut is a two-set view of the vertices, not necessarily the whole cluster partition. The edge (E, H) crosses the iteration-9 cut, and (D, E) crosses the iteration-6 cut.

What to practise: For each tree-edge row of the Kruskal trace, choose one endpoint x, write the cut $\{C(x), V - C(x)\}$, and identify the edge crossing the cut that Kruskal is adding.

Visualizing the cut property

Setup: Jackie drew an imagined MST as a larger set of edges and a current cluster $C(x)$ as one side of a cut. The chosen edge $e = (x, y)$ crosses from $C(x)$ to $V - C(x)$.

Main point: The cut property says that if e is a minimum-weight edge crossing that cut, then adding e keeps the partial solution on track toward some MST. This is why the greedy step does not need to be undone later.

What to practise: Explain the statement "there exists an MST containing the current T plus e" in your own words. Then connect that statement to the loop-invariant idea that every accepted edge remains safe.

Assigned DFS/BFS edge exercise

Setup: The last pages of the notes listed assigned exercises under "DFS vs. BFS: Back Edges, Cross Edges, Cycles" and asked whether a DFS back edge or a BFS cross edge always implies a cycle.

Main point: The lecture did not complete this exercise in class. Its purpose is to make students compare DFS and BFS traversal structure carefully instead of transferring facts from one traversal to the other without checking.

What to practise: Work through the diagrams in the notes. For each marked edge, decide what kind of edge it is under the stated traversal and whether the graph must contain a cycle.

Common Pitfalls and Misconceptions

Treating every extracted edge as an MST edge

Pitfall: Assuming that once an edge is removed from Q, it automatically belongs to T.

Why it causes problems: Internal edges are removed from Q but rejected. Adding them would create a cycle, so they cannot be part of the tree being constructed.

How to avoid it: Separate the action "remove from Q" from the action "add to T". Add the edge only after checking $C(u) \neq C(v)$.

Confusing Kruskal's priority queue with Dijkstra's priority queue

Pitfall: Using the wrong kind of object or key for the priority queue.

Why it causes problems: Dijkstra's priority queue stores vertices keyed by $D(v)$, while Kruskal's priority queue stores edges keyed by $w(u, v)$. Mixing them changes the algorithm.

How to avoid it: Before tracing or writing pseudocode, state explicitly: "What is in Q? What is the key?"

Thinking that iterative automatically means greedy

Pitfall: Calling any algorithm with a loop greedy.

Why it causes problems: Greedy algorithms require more than iteration: they must choose a locally best feasible option according to a cost or score function, and that choice must not be undone.

How to avoid it: Check all four ingredients: step-by-step construction, feasible choices, a greedy choice rule, and no regret.

Assuming Union-Find implementation details are exam material

Pitfall: Spending disproportionate exam-study time on implementing the data structure for clusters.

Why it causes problems: Jackie explicitly kept Union-Find implementation details and Kruskal's time complexity outside the required scope for this course.

How to avoid it: Study the conceptual operations: find $C(v)$, test whether two clusters are equal, and merge different clusters. Do not make Union-Find implementation your main review target unless you are exploring beyond the course.

Treating a cut as the same thing as the current partition

Pitfall: Believing that a cut must list all current clusters separately.

Why it causes problems: A cut is a two-set division of V, while a partition in the Kruskal table may contain many clusters. For example, before iteration 6, the cut can be $\{\{A, B, C, D\}, \{E, F, G, H\}\}$, even though the partition has more than two clusters.

How to avoid it: When forming a cut for a chosen edge (x, y), focus on $C(x)$ as one side and $V - C(x)$ as the other side.

Applying the cut property to the wrong kind of iteration

Pitfall: Trying to use the cut property as if every iteration of Kruskal adds a useful edge.

Why it causes problems: The lecture framed the cut property for iterations where a tree edge is added to T. Internal-edge iterations do not merge clusters and do not extend the MST under construction.

How to avoid it: First classify the edge. If $C(u) = C(v)$, it is internal and rejected. If $C(u) \neq C(v)$, then analyze the relevant cut.

Thinking greedy correctness is just obvious

Pitfall: Saying "Kruskal is correct because it always picks the cheapest edge" without explaining why this local choice is safe.

Why it causes problems: Many greedy-looking strategies are not correct. Kruskal needs the cut property to justify that the chosen edge can belong to some MST.

How to avoid it: State the safety argument: after each accepted tree edge, T is still a subset of some MST.

Overstating the DFS/BFS edge exercise

Pitfall: Memorizing answers about back edges, cross edges, and cycles without working through the traversal diagrams.

Why it causes problems: The assigned exercise is meant to test careful traversal reasoning. DFS and BFS classify edges differently, so a fact from one traversal should not be copied blindly to the other.

How to avoid it: Redraw the exercise graphs, mark the traversal tree, identify the non-tree edges, and then reason about cycles from the actual directed or undirected structure shown.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain the input, output, initialization, main loop, and exit condition of Kruskal's algorithm;
- state why Kruskal's priority queue stores edges keyed by edge weight, not vertices keyed by D-values;
- distinguish between removing an edge from Q, accepting an edge into T, and rejecting an internal edge;
- use $C(u)$ and $C(v)$ to decide whether an edge is a tree edge or an internal edge;
- explain the roles of the find and union/merge operations at the conceptual level;
- map Kruskal's algorithm to the four greedy ingredients;
- define partition, cluster, and cut, and explain how they differ;
- construct a cut of the form $\{C(x), V - C(x)\}$ for a chosen tree edge $e = (x, y)$;
- identify whether a given edge crosses a proposed cut;
- state the cut property as the safety rule behind Kruskal's greedy choice;
- explain the loop-invariant idea that after each accepted tree edge, the current T is contained in some MST;
- describe why, once $|T| = n - 1$, the final T is the MST returned by the algorithm;
- identify what is and is not required for the course: conceptual Kruskal, clusters, cuts, and cut property are required; Kruskal time complexity and Union-Find implementation details are not;
- begin preparing for the cumulative final exam and recognize that dynamic programming will be an important upcoming topic.

Suggested Practice

- Reproduce Kruskal's pseudocode from the notes without looking, then annotate each line with its purpose.
- Using the Kruskal trace table, choose several rows and compute $C(u)$ and $C(v)$ before the edge is processed. Decide whether the edge is accepted or rejected.
- For every accepted tree edge in the trace, form the cut $\{C(x), V - C(x)\}$ for one endpoint x, and identify why the chosen edge crosses the cut.
- Write the four greedy ingredients for Kruskal and for Dijkstra side by side. Compare the partial solution, feasible choices, cost function, greedy step, and no-regret action.
- Explain the cut property in your own words without using the word "magic". Your explanation should include the idea that the current T remains part of some MST.
- Watch or review the posted exercise walkthrough on analyzing the cut property in Kruskal's MST algorithm, especially the connection between the bullet points on the slide and the trace table.
- Work through the assigned DFS/BFS exercise on back edges, cross edges, and cycles. Do not assume the answer; reason from the traversal and the graph.
- Review the Assignment 3 written solution when posted, especially the Dijkstra induction proof, and compare its proof style to the loop-invariant reasoning for Kruskal.
- Start final-exam preparation now: gather lecture notes, tutorial materials, assignments, solutions, digest files, and your own traces before the exam-format discussion.
- Before the Monday dynamic-programming lecture, download or print the new dynamic-programming notes when Jackie posts them.

Where to Review

- **iPad/PDF notes, page 1**: lecture topic overview: partition, cluster, cut, cut property, and assigned DFS/BFS cycle exercises.
- **iPad/PDF notes, page 2**: announcements and reminders, including the confirmed final exam time and location.
- **iPad/PDF notes, page 3**: Kruskal's pseudocode, including the priority queue, $C(u)$, $C(v)$, the merge operation, and the exit condition $|T| = n - 1$.
- **iPad/PDF notes, page 4**: Kruskal as a greedy method, including the four greedy ingredients as they appear in the pseudocode.
- **iPad/PDF notes, page 5**: cut examples from the Kruskal trace, including iterations 6 and 9.

- **iPad/PDF notes, page 6**: cut-property visualization and the loop-invariant intuition for why accepted edges remain safe.
- **iPad/PDF notes, page 7**: the Kruskal pseudocode together with the trace table; use this to connect the algorithm to the trace.
- **iPad/PDF notes, pages 8–10**: assigned exercises on DFS versus BFS, back edges, cross edges, and cycles.
- **Lecture recording**: review the opening logistics, the pseudocode discussion, the cut examples, and the cut-property explanation if any part of the notation $C(v)$, $V - S$, or $T \subseteq$ some MST feels unclear.
- **Assignment 3 solution when posted**: revisit the Dijkstra induction proof as a model for algorithm-correctness reasoning.
- **Posted graph exercise video**: use the walkthrough on analyzing the cut property in Kruskal's MST algorithm as active exam preparation.

Closing Reminder

This digest is only a roadmap. To study this lecture properly, return to Jackie's iPad/PDF notes and the recording, redraw the Kruskal trace, practise forming cuts from clusters, and explain the cut property in your own words. The main goal is not just to memorize the pseudocode, but to understand why each accepted Kruskal edge is a safe greedy choice.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 23, July 27

Recursive Fibonacci: Complexities, Dynamic Programming: Principles

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 10

Closing Reminder 10

Main Ideas

- This lecture introduced dynamic programming as a way to make certain recursive solutions much more efficient by storing and reusing subproblem results.
- Jackie used recursive problem solving as the starting point: identify base cases, divide a large problem into strictly smaller subproblems, assume their solutions, and combine those solutions.
- The two key conditions for applying dynamic programming were overlapping subproblems and optimal substructure.
- Fibonacci was the main running example: the naive recursive solution is easy to derive from the recurrence, but it recomputes the same subproblems many times.
- The lecture analyzed naive recursive Fibonacci as using linear stack space but exponential running time, then motivated memoization and tabulation as two dynamic-programming approaches.
- Logistically, Jackie reminded students that the final exam is scheduled for 10 AM–1 PM on Sunday, August 9 in LAS B, that the Assignment 3 written solution would be released, and that this week's office hour is Wednesday, July 29, 3:30 PM–5 PM.

Roadmap

Final-week logistics and the transition to dynamic programming

Jackie began by reminding students that Tuesday and Thursday office hours for the week were cancelled, but that a 90-minute Wednesday office hour would run from 3:30 PM to 5 PM. Students with outstanding academic-consideration issues, such as a missed test, were especially encouraged to contact Jackie. Programming Test 2 grading was in progress, with results and feedback targeted for early the following week.

The lecture then started the final course topic: dynamic programming. Jackie framed it as a broad and important topic, but narrowed the course treatment to the examples and exercises needed for the final exam. Students were told that dynamic programming will be relevant for the exam and that they should study the posted dynamic-programming slides, especially from slide 3 onward.

Recursive problem solving as the starting point

The first technical page reviewed the general shape of a recursive solution. A small problem is solved directly by a base case. A large problem is handled by a recursive case: divide the original problem into strictly smaller subproblems, assume solutions to those subproblems through recursive calls, and then conquer by combining the subsolutions into a solution for the original problem.

Jackie used a visual convention in the notes: blue boxes represent problems, while green boxes represent solutions. This matters because dynamic programming is not introduced as a separate magic trick; it is introduced as a way to repair an inefficient recursive structure when the same subproblem solutions keep being recomputed.

The two dynamic-programming conditions

Jackie identified two conditions that make dynamic programming applicable. The first is *overlapping subproblems*: during the recursive computation, the same smaller problem may be solved more than once. This is mainly an efficiency concern, because repeated recomputation can make the runtime much larger than necessary.

The second is *optimal substructure*. Jackie explained this as the solution being compositional: the solution to the original problem can be built from the solutions to smaller subproblems. The exact combining operation depends on the problem. It might be addition, concatenation, or another operation, but there must be a meaningful way to assemble the larger solution from subsolutions.

Fibonacci as the main example

The lecture then moved to the Fibonacci sequence. The notes defined the sequence by indices, using a state/index such as i, and the recurrence

$$F_i = \begin{cases} 0, & i = 0, \\ 1, & i = 1, \\ F_{i-1} + F_{i-2}, & i > 1. \end{cases}$$

Jackie emphasized that identifying the state, base cases, and recursive case is a useful step before moving to code, especially as problems become more complex.

The recursive Java implementation follows the recurrence directly: handle negative input as invalid, return 0 for $i = 0$, return 1 for $i = 1$, and otherwise return `fib(i - 1) + fib(i - 2)`. The code is correct, but correctness alone does not imply efficiency.

Tracing `fib(5)` and seeing overlap

The recursion tree for `fib(5)` was used to show repeated work. Computing `fib(5)` requires `fib(4)` and `fib(3)`; computing `fib(4)` again requires `fib(3)` and `fib(2)`. As the tree expands, subproblems such as `fib(3)` and `fib(2)` appear in multiple places.

This trace establishes the two dynamic-programming conditions for Fibonacci. The repeated `fib(3)` and `fib(2)` calls show overlapping subproblems. The recurrence $F_i = F_{i-1} + F_{i-2}$ shows optimal substructure, because a larger Fibonacci value is composed from smaller Fibonacci values.

Space and time complexity of naive recursive Fibonacci

Jackie then separated space complexity from time complexity. The space cost comes from the recursion/call stack. Along the deepest path, the calls go from `fib(n)` to `fib(n-1)` to

`fib(n-2)` and so on until a base case, so the maximum stack depth is linear in the input size: $O(n)$.

The time cost is much worse. Jackie gave an intuitive exponential-growth sketch: computing `fib(n+2)` contains two separate computations of `fib(n)` as subtrees, so the work more than doubles when the input size increases by 2. Repeating this pattern gives exponential growth, summarized in the lecture as $O(2^n)$. The important lesson is not a formal proof detail, but the reason for the explosion: many overlapping subproblems are recomputed.

What dynamic programming changes

The final page introduced the two words in *dynamic programming*. "Programming" means a step-by-step computation of subproblems while storing their results. "Dynamic" means those intermediate results are computed and stored on the fly as the algorithm proceeds, either across loop iterations or across recursive calls.

The central improvement is caching. If the solution to a subproblem has already been computed, later work can look it up in constant time rather than recomputing it. Jackie connected this idea to memory/cache intuition: store the result once, then reuse it when the same subproblem appears again.

Top-down memoization versus bottom-up tabulation

Jackie introduced two dynamic-programming styles. The *top-down* approach keeps the recursive structure but adds *memoization*: store previously computed results in an array or table, then check the table before making more recursive calls. Jackie explicitly noted the spelling: *memoization*, not memorization.

The *bottom-up* approach uses iteration and *tabulation*. Instead of starting from the target call and recursing downward, it fills a table from smaller cases upward. Jackie previewed a trade-off: top-down memoization is often easier to derive from the recurrence but harder to trace; bottom-up tabulation may be harder to design from scratch, but once written it is usually easier to trace because it is mainly loop execution.

Preparation for the next class

Jackie ended by asking students to compare the posted `FibonacciTopDown` helper method with the ordinary recursive Fibonacci method. The specific goal is to identify what is different in the memoized version and why that difference makes it more efficient. The next lecture would trace the top-down and bottom-up versions in more detail and continue with more advanced dynamic-programming examples.

Key Terms, Notation, and Structures

- **Dynamic programming**: A method for improving certain recursive solutions by computing subproblem results step by step, storing them, and reusing them instead of recomputing them.

- **Overlapping subproblems**: The same subproblem occurs in multiple branches of a recursive computation, causing repeated work in the naive solution.
- **Optimal substructure**: The solution to the original problem can be composed from solutions to strictly smaller subproblems.
- **Base case**: A small enough input whose solution is returned directly, without further recursive calls.
- **Recursive case**: The case where a larger input is reduced to one or more strictly smaller subproblems.
- **Divide and conquer**: The recursive pattern of dividing a problem into subproblems, solving or assuming subsolutions, and combining them.
- **State**: A parameter that identifies a subproblem. In Fibonacci, the index i is the state.
- **Recurrence relation**: A mathematical specification of how a problem's value is defined from base cases and smaller states, such as $F_i = F_{i-1} + F_{i-2}$ for $i > 1$.
- **Fibonacci base cases**: $F_0 = 0$ and $F_1 = 1$.
- **Naive recursive Fibonacci**: The direct implementation of the recurrence using recursive calls; correct but inefficient because it recomputes overlapping subproblems.
- **Recursion tree**: A diagram of recursive calls, useful for seeing repeated subproblems and reasoning about runtime.
- **Call stack or recursion stack**: The runtime structure storing active recursive calls; its maximum depth determines the stack-space cost.
- **Memoization**: A top-down dynamic-programming technique that stores results of previously solved subproblems and looks them up before recomputing.
- **Tabulation**: A bottom-up dynamic-programming technique that fills a table iteratively from smaller states to larger states.
- **Cache/table**: A 1D, 2D, or higher-dimensional storage structure used to remember subproblem solutions; in this course, Jackie indicated that 1D and 2D cases are the relevant focus.
- **Constant-time lookup**: After a subproblem result is stored in an array/table, retrieving it can be treated as $O(1)$ in the examples discussed.
- **Complexities for naive recursive Fibonacci**: Space $O(n)$ from stack depth; time exponential, summarized as $O(2^n)$, due to repeated recursive calls.

Key Examples and Demonstrations

Recursive problem-solving diagram

Setup: The notes show a small problem solved directly and a large problem divided into several smaller subproblems, with blue boxes for problems and green boxes for solutions.

Main point: Recursion is organized around base cases and recursive cases. For dynamic programming, the important extra question is whether the subproblems overlap and whether their solutions can be composed into the larger solution.

What to practise: Pick a familiar recursive problem and label its state, base case, recursive case, subproblems, and combine/conquer step. Then ask whether any subproblem is likely to be solved more than once.

Fibonacci recurrence and Java implementation

Setup: The lecture defines Fibonacci using $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i > 1$, then translates this into a recursive Java method `fib(i)`.

Main point: The recurrence gives a clean recursive program, but a clean recursive program may still be inefficient. The recurrence is also the starting point for deriving dynamic-programming versions.

What to practise: Write the recurrence first, then write the Java method from the recurrence. Check the negative-input case, both base cases, and the recursive return expression.

Recursion tree for `fib(5)`

Setup: The notes expand `fib(5)` into `fib(4)` and `fib(3)`, then continue expanding until base cases are reached.

Main point: The tree makes the inefficiency visible: `fib(3)` and `fib(2)` are recomputed in different branches. This is the overlapping-subproblems condition.

What to practise: Redraw the `fib(5)` recursion tree and circle every repeated subproblem. Then explain how storing `fib(3)` after the first computation would avoid later work.

Stack-space intuition

Setup: Jackie compared the recursion tree with the call stack. The deepest chain follows calls like `fib(5)`, `fib(4)`, `fib(3)`, `fib(2)`, `fib(1)`.

Main point: The maximum number of active calls is proportional to the input size, so the stack-space complexity is $O(n)$, even though the total number of calls is much larger.

What to practise: Trace `fib(4)` using a stack, as Jackie recommended through the posted tutorial video, and identify when calls are pushed and popped.

Exponential-time sketch

Setup: The notes compare the work for `fib(n+2)` with the work for two separate `fib(n)` subtrees.

Main point: Increasing the input size by 2 more than doubles the work. Repeating that pattern gives exponential growth, which is why naive recursive Fibonacci is far too slow for large inputs.

What to practise: Reconstruct the inequality idea $RT(n+2) > 2 \cdot RT(n)$ from the recursion tree and explain in words where the two copies of `fib(n)` come from.

Dynamic-programming storage idea

Setup: The final notes page shows a current subproblem being computed using stored solutions from earlier subproblems, with lookup treated as constant time.

Main point: Dynamic programming reduces runtime by replacing recomputation with storage and lookup. Top-down memoization keeps recursion; bottom-up tabulation fills a table iteratively.

What to practise: Compare the ordinary recursive Fibonacci method with the memoized helper method from the posted code and identify exactly where the cached value is checked and where a newly computed value is stored.

Common Pitfalls and Misconceptions

Assuming that a correct recursive program is automatically efficient

Pitfall: Writing the direct recursive Fibonacci method and stopping because it returns the correct values.

Why it causes problems: The direct recursion recomputes the same subproblems many times, causing exponential running time.

How to avoid it: After writing a recursive solution, draw a small recursion tree and check whether the same state appears more than once.

Mixing up base cases and recursive cases

Pitfall: Treating F_0 and F_1 as recursive cases, or using the wrong base value such as returning 1 for F_0.

Why it causes problems: Incorrect base cases make every later computed value wrong and may also break termination reasoning.

How to avoid it: State the recurrence before coding: $F_0 = 0$, $F_1 = 1$, and only $i > 1$ uses $F_{i-1} + F_{i-2}$.

Calling every recursive split dynamic programming

Pitfall: Assuming that any recursive divide-and-conquer solution should automatically be called dynamic programming.

Why it causes problems: Dynamic programming is motivated by specific structure: overlapping subproblems and optimal substructure. Without those, memoization or tabulation may not be appropriate or useful.

How to avoid it: Before applying dynamic programming, explicitly identify the state, repeated subproblems, and the way subsolutions compose into a larger solution.

Confusing stack depth with total running time

Pitfall: Looking only at the longest path in the recursion tree and concluding that naive Fibonacci is linear time.

Why it causes problems: The longest path explains stack space, not total work. Total time depends on the number of calls across the whole recursion tree.

How to avoid it: Use the deepest path for space complexity and the whole recursion tree for time complexity.

Expecting the Fibonacci recursion tree height to be logarithmic

Pitfall: Thinking the tree height is $O(\log n)$ because the picture is a tree.

Why it causes problems: In Fibonacci, the recursive calls reduce the input by 1 or 2, not by a constant factor. The deepest path is therefore proportional to n.

How to avoid it: Follow the path $n, n-1, n-2, \ldots, 1$. Since the state decreases one step at a time along the deepest branch, the stack depth is $O(n)$.

Misspelling or misunderstanding memoization

Pitfall: Thinking of memoization as "memorization," or thinking it means precomputing everything before the algorithm starts.

Why it causes problems: Memoization stores results as they are computed on the fly and reuses them later. It is dynamic storage of subproblem answers, not human memorization and not necessarily full precomputation.

How to avoid it: In code, look for two actions: checking whether a subproblem result is already stored, and storing the result after computing it for the first time.

Confusing top-down and bottom-up DP

Pitfall: Treating memoization and tabulation as the same implementation style.

Why it causes problems: Top-down DP preserves recursive calls and uses memoization; bottom-up DP uses loops and tabulation. They may compute the same values but the control flow and tracing method differ.

How to avoid it: For each DP solution, ask: Does the code still recurse? If yes, it is top-down memoization. If it fills a table by loop order from smaller states, it is bottom-up tabulation.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain the general recursive pattern of base case, divide, recursive subproblem solving, and combine/conquer;
- distinguish overlapping subproblems from optimal substructure;
- identify the state, base cases, and recursive case in the Fibonacci recurrence;
- translate the Fibonacci recurrence into the ordinary recursive Java method;
- draw a recursion tree for a small Fibonacci input such as `fib(5)`;
- identify repeated subproblems in the Fibonacci recursion tree;
- explain why naive recursive Fibonacci uses $O(n)$ stack space;
- explain the intuition behind exponential running time for naive recursive Fibonacci;
- define memoization and tabulation in the context of dynamic programming;
- compare top-down DP with bottom-up DP at the level of recursion versus iteration;
- explain why a cached subproblem result can avoid recomputation;
- compare a naive recursive Fibonacci method with a memoized helper method and point out the efficiency-critical differences;
- locate and use the posted dynamic-programming slides, code, and recursion-tracing video as preparation for the next lecture and the final exam.

Suggested Practice

- Redraw the recursive-problem-solving diagram from the notes and label the base case, recursive case, divide step, assumed subsolutions, and combine step.
- Write the Fibonacci recurrence from memory, including the correct base cases $F_0 = 0$ and $F_1 = 1$.
- Implement the naive recursive `fib(i)` method in Java and test inputs 0 through 6 before trying larger inputs.
- Draw the full recursion tree for `fib(5)` and circle every repeated `fib(3)` and `fib(2)` occurrence.
- Follow the posted tutorial video and trace `fib(4)` using a call stack; note which calls are active at the deepest point.
- Reconstruct the runtime intuition: explain why `fib(n+2)` contains two `fib(n)` computations and why that indicates exponential growth.
- Compare the posted top-down Fibonacci helper with the ordinary recursive version, and mark the memoization lookup and the place where a newly computed result is stored.

- Preview the bottom-up Fibonacci code and ask what order the table entries must be filled in so that each entry can use earlier entries.
- Review the posted DP slides from slide 3 onward before the next class, since Jackie said the lecture resources are sufficient for the exam scope.
- When studying Assignment 3 solutions, pay attention to the written-part induction proof that Jackie highlighted as useful for learning and exam preparation.

Where to Review

- **iPad/PDF notes page 1**: Lecture title and scope: dynamic programming, recursive problem solving, recursive Fibonacci, and memoization versus tabulation.
- **iPad/PDF notes page 2**: Administrative reminders: final exam time/location, Assignment 3 written solution, and Wednesday office hours.
- **iPad/PDF notes page 3**: General recursive problem-solving diagram and the two DP conditions: overlapping subproblems and optimal substructure.
- **iPad/PDF notes page 4**: Fibonacci sequence, recurrence relation, recursive implementation, `fib(5)` recursion tree, and space/time complexity summary.
- **iPad/PDF notes page 5**: Jackie's exponential-runtime sketch for naive recursive Fibonacci.
- **iPad/PDF notes page 6**: Dynamic-programming storage idea, constant-time lookup, top-down memoization, and bottom-up tabulation.
- **Posted code**: Compare the ordinary recursive Fibonacci method with the top-down memoized helper method before the next lecture.
- **Posted tutorial video**: Use the Fibonacci stack-tracing video, especially for `fib(4)`, to practise call-stack tracing.
- **Lecture recording**: Revisit the segments on overlapping subproblems, optimal substructure, the `fib(5)` recursion tree, the $O(n)$ stack-space explanation, and the exponential-time sketch.
- **Dynamic-programming slides**: Jackie specifically directed students to study from slide 3 onward for the exam scope.

Closing Reminder

This digest is only a roadmap for Lecture 23. To learn the material, revisit the recording and Jackie's iPad/PDF notes, trace the Fibonacci examples yourself, compare the Java implementations, and practise explaining why memoization and tabulation remove repeated

work. Dynamic programming is not learned by reading definitions alone; you need to identify states, write recurrences, trace executions, and check that your code stores and reuses subproblem results correctly.

EECS3101 Algorithms (Summer'26)

Digest of Lecture 24, July 28

Fibonacci DP: Memoization, Tabulation, Min-Cost Climbing: Specification

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 9

Suggested Practice 9

Where to Review 10

Closing Reminder 11

Main Ideas

- This final lecture completed the Fibonacci dynamic-programming example by developing both the top-down memoization solution and the bottom-up tabulation solution.
- Jackie connected the new DP code back to Lecture 23: naive recursive Fibonacci is correct but has overlapping subproblems, linear stack space, and exponential running time.
- The top-down solution keeps recursion but adds a shared `memo` array so each Fibonacci state is computed once and reused later by constant-time array lookup.
- The bottom-up solution removes recursion and fills a `tab` array iteratively from the base cases upward, using the recurrence to decide the fill direction.
- A major lesson was that DP is not automatically worth using just because it is possible: it is most valuable when it gives a real asymptotic improvement, such as exponential time down to linear time.
- Jackie introduced the Minimum Cost Climbing Stairs problem as the next DP practice target, warned against a greedy interpretation, and assigned the Tutorial 9 recording plus the 0-1 Knapsack materials for exam preparation.
- Logistically, Jackie said this was the last lecture session, the Thursday meeting would be cancelled, an exam guide PDF would be released, and extra review/office-hour time would be arranged before the final exam.

Roadmap

Final lecture plan and quick recap

Jackie opened by framing this as the last lecture of the course. The plan was to finish the Fibonacci dynamic-programming techniques, assign important practice exercises, and give students guidance for exam preparation. Jackie also mentioned that an exam guide PDF would be released and that extra review or office-hour time would be arranged for the following week.

The lecture then briefly recalled Lecture 23. Recursive problem solving starts with base cases and recursive cases: a big problem is divided into strictly smaller subproblems, the subproblems are solved recursively, and their solutions are combined. Fibonacci satisfied the two conditions that made DP applicable: overlapping subproblems and optimal substructure. The naive recursive version had space complexity $O(n)$, because of the maximum call-stack depth, and time complexity $O(2^n)$, because repeated subproblems cause exponential recomputation.

Top-down Fibonacci: recursion with memoization

The notes first developed a top-down, recursive DP solution for Fibonacci. The public method still receives the state i, but it creates a one-dimensional array `memo` of size $i + 1$,

because the computation may need to store $F_0, F_1, \ldots, F_i$. The array is initialized with a sentinel value such as -1, and the base cases are stored as `memo[0] = 0` and, when the array is large enough, `memo[1] = 1`.

Jackie emphasized that the helper method receives the same array reference throughout the recursive calls. The important pattern is: check whether the desired entry is already stored; if so, return it immediately; otherwise, compute the value recursively, store it in `memo[i]`, and then return it. This is the "compute once, reuse many times" principle.

Tracing `fib(5)` with memoization

The trace of `fib(5)` showed why memoization changes the running time. The first time `fib(2)` is needed, it is computed from the base cases and stored. Later, when `fib(2)` is needed again, the helper does not expand a new subtree; it finds `memo[2]` and returns the stored value. The same idea applies to `fib(3)` and later entries.

This pruning of repeated recursive work reduces the time complexity to $O(n)$. The space complexity is also $O(n)$, now accounting for both the recursion stack and the `memo` array. Jackie explicitly treated this as a worthwhile time-space trade-off: using linear extra space is worth it when the time drops from exponential to linear.

Implementation details that matter

Several student questions focused on implementation details. Jackie explained why setting `memo[1]` should be guarded by a condition such as $i > 0$: if the user asks for F_0, the array has length 1, and trying to access index 1 would be out of bounds. Jackie also used this example to remind students about arrays passed by value as references in Java: recursive calls share the same array object.

Another important implementation point was the choice of data structure. The memoization table should support constant-time indexing. An array is appropriate for this Fibonacci example; replacing it with a linked list would make lookup linear and undermine the efficiency gain.

Bottom-up Fibonacci: iteration with tabulation

The next page developed the bottom-up, iterative DP solution. This version also creates an array, now called `tab`, and initializes the same base cases. However, it does not use recursive calls. Instead, it fills the table in a loop, starting from $j = 2$ and continuing until $j = i$.

For Fibonacci, the recurrence says

$$F(j) = F(j-1) + F(j-2).$$

Therefore, to compute entry j, entries $j-1$ and $j-2$ must already be available. This is why the table is filled from left to right. Jackie called this forward filling and contrasted it with the possibility of backward filling in problems whose recurrence depends on larger-index entries such as $f(x+1)$ and $f(x+2)$.

Tracing `fib(5)` with tabulation

The bottom-up trace is shorter than the top-down trace. After setting $F_0 = 0$ and $F_1 = 1$, the loop fills F_2, then F_3, then F_4, then F_5. Each step uses two entries already present in the table. The final returned value is the entry at index i.

The time complexity is $O(n)$, because the method fills the table once from left to right. The space complexity is $O(n)$, because of the table. Jackie noted that for Fibonacci specifically, one could optimize the array away and keep only two recent values, but this is a problem-specific optimization. The general DP skill students should practise is tabulation with a table.

When dynamic programming is worth using

Jackie then assigned factorial as a self-study exercise. Students should write the recurrence, trace `fac(5)`, compare the plain recursive, top-down, and bottom-up versions, and analyze their complexities. The main lesson is not that every recursive problem should be converted to DP. For factorial, the plain recursive version is already linear, so DP does not give the kind of asymptotic improvement that it gives for Fibonacci.

The study question is therefore: what improvement does DP actually buy? If DP reduces a runtime from exponential to linear, as in Fibonacci, it is highly valuable. If it only changes one linear method into another linear method with extra storage and code structure, it may not be worth the trouble.

Minimum Cost Climbing Stairs as the next DP practice problem

The final technical topic introduced the Minimum Cost Climbing Stairs problem. The input is an array `cost`, where `cost[i]` is the cost of stepping on stair i. A climb may start at stair 0 or stair 1. From stair i, one pays `cost[i]` and then moves to either $i+1$ or $i+2$. The "top" is the imaginary index n, where n is `cost.length`; an index beyond that can also be treated as already past the staircase.

The goal is to minimize the total cost needed to reach the top or beyond. Jackie walked through small examples such as $\{5\}$, $\{10, 15\}$, $\{15, 10\}$, and $\{5, 5, 3\}$, then sketched a recursion tree to show that the problem explores multiple possible paths. Jackie warned students not to treat this as a greedy problem: choosing a locally cheaper next stair can lead to a worse total cost later.

Assigned resources after the last lecture

Jackie directed students to the Tutorial 9 recording for the full Minimum Cost Climbing Stairs solution, including recursive, top-down, and bottom-up versions. Jackie also pointed to an extra 0-1 Knapsack exercise with a solution guide, Java starter/solution code, a notes template, and a guidance video. The knapsack problem is important because its DP solution uses a two-dimensional memoization/table structure rather than the one-dimensional arrays used in the Fibonacci example.

Key Terms, Notation, and Structures

- **Dynamic programming**: A technique for solving suitable recursive problems by storing and reusing subproblem results instead of recomputing them.
- **Overlapping subproblems**: The same state or subproblem appears more than once in the computation, as with repeated `fib(2)` and `fib(3)` calls in naive Fibonacci.
- **Optimal substructure**: The solution to a larger problem can be built from solutions to smaller subproblems.
- **Top-down DP**: A recursive DP style that starts from the requested input and recursively asks for smaller states, while caching results.
- **Memoization**: The top-down caching technique; in the Fibonacci code, `memo[i]` stores the result for F_i.
- **Sentinel value**: A default value, such as -1, used to mark an entry as not yet computed.
- **Recursive helper method**: A method that carries extra state, such as the shared `memo` array, while the public method exposes a simpler interface.
- **Call by value for object references**: In Java, the array reference is passed by value, so recursive calls can still access and modify the same array object.
- **Bottom-up DP**: An iterative DP style that starts from base cases and fills larger states using already-computed smaller states.
- **Tabulation**: The bottom-up table-filling technique; in the Fibonacci code, `tab[j]` stores the result for F_j.
- **Forward filling**: Filling a DP table from lower indices to higher indices because each entry depends on earlier entries, as in $F(j) = F(j-1) + F(j-2)$.
- **Backward filling**: Filling a DP table from higher indices to lower indices when the recurrence depends on future/larger-index entries.
- **Time-space trade-off**: The DP solution may use extra space, such as an array, to obtain a much better running time.
- **Minimum Cost Climbing Stairs**: A DP practice problem where `cost[i]` gives the cost of stepping on stair i, and the goal is a minimum total cost path to the top or beyond.
- **Top**: In the climbing-stairs problem, the imaginary index n = `cost.length`, one position after the last real stair.
- **0-1 Knapsack**: A more advanced DP exercise assigned for practice; Jackie indicated that its DP solution uses a two-dimensional array.

Key Examples and Demonstrations

Top-down Fibonacci trace

Setup: The lecture traced `fibTopDown(5)` using a `memo` array of length 6. The base cases were stored at indices 0 and 1, while the remaining entries initially held a sentinel value.

Main point: The first computation of a non-base state stores its result. Later calls to the same state return the cached value immediately instead of expanding another recursive subtree.

What to practise: Redraw the top-down trace for `fib(5)` and update the `memo` array after each newly computed entry. Mark where the trace reuses `memo[2]` and `memo[3]` instead of recomputing them.

Array choice for memoization

Setup: Jackie compared array indexing with linked-list access for the memoization table.

Main point: Memoization depends on fast lookup. With an array, `memo[i]` is constant time. With a linked list, reaching index i generally requires traversal, which would compromise the intended efficiency gain.

What to practise: Explain why the asymptotic benefit of memoization assumes fast access to stored subproblem results. Then state what would go wrong if the table were replaced by a linked list.

Bottom-up Fibonacci trace

Setup: The lecture traced `fibBottomUp(5)` using a `tab` array and a loop with $j = 2, 3, 4, 5$.

Main point: The table is filled in a dependency-respecting order. To compute `tab[j]`, the values `tab[j - 1]` and `tab[j - 2]` must already have been filled.

What to practise: Make a small table with columns for iteration number, j, the assignment to `tab[j]`, and the resulting array. Verify that the final value for `fib(5)` is obtained without recursion.

Forward filling versus backward filling

Setup: Jackie used the Fibonacci recurrence to show why `tab` is filled left to right, then described a hypothetical recurrence depending on $x + 1$ and $x + 2$ to motivate backward filling.

Main point: The fill direction is determined by dependencies, not by habit. A table entry can only be computed after the entries it depends on are already available.

What to practise: For any recurrence, underline the states on the right-hand side and decide whether they are smaller or larger than the current state. Use that to determine the table-filling direction.

Factorial as a DP-worthiness exercise

Setup: The notes include recursive, top-down, and bottom-up factorial pages, and Jackie asked students to complete the tracing and complexity analysis independently.

Main point: DP may be applicable but not useful. Factorial has a simple recursive structure, but the plain recursive method is already linear, so DP does not produce an asymptotic improvement.

What to practise: Compute `fac(5)` using the recursive, top-down, and bottom-up versions. Compare their time and space complexities and explain whether the DP versions are worth the extra machinery.

Minimum Cost Climbing Stairs specification

Setup: The final notes page defines the `cost` array, the start choices at stair 0 or 1, the move rule from stair i to $i+1$ or $i+2$, and the imaginary top at index n.

Main point: This is an optimization problem over possible paths, not a one-step greedy problem. The examples and recursion-tree sketch show why all valid future choices must be accounted for.

What to practise: Rework the small examples from the notes, especially $\{5\}$, $\{10, 15\}$, $\{15, 10\}$, and $\{5, 5, 3\}$. Then watch the Tutorial 9 recording to connect the specification to recursive, top-down, and bottom-up DP code.

Common Pitfalls and Misconceptions

Confusing memoization with tabulation because both use arrays

Pitfall: Thinking top-down memoization and bottom-up tabulation are the same because both store values in a one-dimensional array in the Fibonacci example.

Why it causes problems: The program flow is different. Memoization is recursive and computes states on demand; tabulation is iterative and deliberately fills entries in table order.

How to avoid it: When studying a DP solution, first ask whether the main computation is a recursive helper or a loop. Then ask when and why the table entry is read.

Using a slow data structure for the DP table

Pitfall: Replacing the `memo` or `tab` array with a linked list because both are linear collections.

Why it causes problems: DP efficiency relies on quickly reusing stored results. A linked list does not provide constant-time indexing, so each lookup could add unnecessary traversal cost.

How to avoid it: Use a structure with fast access by state, such as an array for integer-indexed states in these examples.

Ignoring edge cases in table initialization

Pitfall: Always assigning `memo[1] = 1` or `tab[1] = 1` without checking whether index 1 exists.

Why it causes problems: If the requested input is 0, the array may have length 1, and index 1 is out of bounds.

How to avoid it: Initialize only entries that are guaranteed to exist. For Fibonacci, store F_0 and then store F_1 only when the input is at least 1.

Treating sentinel initialization as meaningless

Pitfall: Assuming there is no reason to fill a table with a sentinel such as -1 because the table will eventually be filled.

Why it causes problems: A default value such as 0 might be a meaningful result in some problems, so it can hide whether an entry is actually computed. In top-down memoization, the sentinel check is especially important.

How to avoid it: Use a default value that cannot be confused with a valid computed result, and be clear about whether the code needs to check for it.

Choosing a table-filling direction by habit

Pitfall: Always filling a DP array left to right just because Fibonacci is filled that way.

Why it causes problems: Some recurrences depend on future or larger-index states. Filling in the wrong direction means the needed entries are not available when you compute the current one.

How to avoid it: Read the recurrence first. Fill entries only after all states they depend on have already been computed.

Assuming DP is always better than plain recursion

Pitfall: Applying DP to every recursive problem without checking whether it improves the asymptotic running time.

Why it causes problems: DP can add storage, code structure, and tracing overhead. For a problem such as factorial, the plain recursive solution is already linear, so DP does not provide the same payoff as it does for Fibonacci.

How to avoid it: Analyze the plain recursive solution first. Use DP when overlapping subproblems cause serious recomputation and the stored results genuinely improve complexity.

Solving Minimum Cost Climbing Stairs greedily

Pitfall: At each stair, choosing the immediate next step with the smaller visible cost and assuming that must lead to the minimum total cost.

Why it causes problems: A locally cheap step can force more expensive future choices. The problem requires comparing complete future costs, which is why DP is appropriate.

How to avoid it: Define a recursive state for the remaining cost from a stair, compare both move options, and then convert the recursion to top-down or bottom-up DP.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain why naive recursive Fibonacci has overlapping subproblems and why that leads to exponential running time;
- describe the role of the `memo` array in top-down Fibonacci and trace how `fib(5)` fills and reuses it;
- describe the role of the `tab` array in bottom-up Fibonacci and trace how `fib(5)` is computed by a loop;
- distinguish top-down memoization from bottom-up tabulation in terms of program flow, not just storage;
- explain why array indexing is important for the efficiency of memoization and tabulation;
- handle base-case and edge-case initialization correctly for Fibonacci DP arrays;
- analyze the space and time complexity of naive recursive, top-down, and bottom-up Fibonacci solutions;
- determine a table-filling direction from a recurrence relation;
- evaluate whether DP is worth applying to a recursive problem by checking whether it improves asymptotic complexity;
- state the rules and goal of the Minimum Cost Climbing Stairs problem, including the meanings of stair 0, stair 1, top, and beyond;
- explain why the Minimum Cost Climbing Stairs problem should not be solved by a simple greedy rule; and
- locate and use the assigned Tutorial 9 and 0-1 Knapsack resources for additional DP practice before the exam.

Suggested Practice

- Download and run the posted Fibonacci and factorial source code; do not only read the code visually.
- Trace `fibTopDown(5)` by hand, updating the `memo` array after each computed value and marking every lookup that avoids recomputation.

- Trace `fibBottomUp(5)` by hand, listing the value of j, the assignment to `tab[j]`, and the resulting table after each iteration.
- Compare the naive recursive, top-down, and bottom-up Fibonacci methods and explain why the DP versions are $O(n)$ rather than $O(2^n)$.
- Complete the factorial self-study exercise: write the recurrence, trace `fac(5)`, analyze all three versions, and decide whether DP is worth using.
- Rework the Minimum Cost Climbing Stairs problem specification from the notes and draw the recursion tree for the small example discussed in class.
- Watch the Tutorial 9 recording for the full Minimum Cost Climbing Stairs solution, including recursive, top-down, and bottom-up versions.
- Work through the 0-1 Knapsack exercise using the provided guide, notes template, Java starter/solution files, and guidance video, paying attention to the two-dimensional table.
- Prepare questions for Jackie's review session or extra office hours after you have attempted the assigned DP exercises yourself.

Where to Review

- Review the Lecture 24 iPad/PDF notes page 1 for the lecture scope: Fibonacci memoization versus tabulation and Minimum Cost Climbing Stairs.
- Review page 2 for the top-down Fibonacci code, the `memo` array, the `fib(5)` trace, and the $O(n)$ space/time conclusions.
- Review page 3 for the bottom-up Fibonacci code, the `tab` array, forward filling, the `fib(5)` tabulation trace, and the $O(n)$ space/time conclusions.
- Review pages 4–6 for the factorial recursive, top-down, and bottom-up exercise prompts; use them to test whether DP is actually beneficial for that problem.
- Review page 7 for the Minimum Cost Climbing Stairs problem statement, movement rules, top/beyond notation, small examples, and recursion-tree sketch.
- Rewatch the lecture recording segments on memoization setup, the top-down trace, tabulation and fill direction, factorial as a DP-worthiness exercise, and the Minimum Cost Climbing Stairs specification.
- Use the Tutorial 9 recording for the complete Minimum Cost Climbing Stairs solution, and use the posted 0-1 Knapsack resources for the two-dimensional DP practice item.
- Watch for Jackie's exam guide PDF and review-session/office-hour announcement before finalizing your exam study plan.

Closing Reminder

This digest is only a roadmap for the final lecture. To prepare effectively, revisit Jackie's iPad/PDF notes, run the posted code, trace both Fibonacci DP versions by hand, complete the factorial exercise, and work through the assigned Minimum Cost Climbing Stairs and 0-1 Knapsack resources. Reading the digest alone is not enough: the key exam preparation is active tracing, coding, comparing implementations, and explaining why each DP table is filled the way it is.

EECS3101 Algorithms (Summer'26)

Digest of Tutorial 1, May 4

2D Arrays for Mileage Tables and Itinerary Logic

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 10

Closing Reminder 10

Main Ideas

- This tutorial used 2D arrays as a practical bridge from prerequisite Java knowledge to later EECS3101 topics such as graph implementations and dynamic programming tabulation.
- Jackie connected a rectangular distance table to a graph view: cities are vertices, direct flights are weighted edges, and an itinerary is a path through selected vertices.
- The central Java idea was that a 2D array is an array of arrays, so rows can be created, assigned, traversed, and reasoned about through references.
- Students saw two common initialization patterns: nested curly-brace initialization and creating a 2D array object with `new` before assigning rows.
- The live coding focused on translating between problem data and array indices, especially by mapping city names such as `Boston` and `Miami` to integer indices.
- A key boundary lesson was that an itinerary with `howMany` cities contains only `howMany - 1` adjacent city-pairs to process.
- The tutorial also introduced course routines that will matter later: importing zipped projects into Eclipse, practising with JUnit tests, and using test-driven development habits in assignments and programming tests.

Roadmap

Tutorial setup and course-material routine

Jackie began by situating this as the first EECS3101 tutorial, with the first official class and broader syllabus questions deferred to the next day. For this session, the target was deliberately narrower: review and practise 2D arrays before they become necessary in later course topics. Students were shown where to find the Tutorial 1 instructions, iPad notes, Java solution archive, and recording, and Jackie emphasized that this tutorial had no submission requirement.

A practical routine was demonstrated early: download the provided Java archive and import it into Eclipse. This matters because Eclipse will be the required IDE for programming tests, and starter projects will be distributed in a similar way.

Why 2D arrays matter in EECS3101

The first technical page connected 2D arrays to two future areas of the course. For graphs, a 2D array can implement an adjacency matrix. For dynamic programming, bottom-up tabulation often uses a table, which may be one-dimensional, two-dimensional, or higher-dimensional depending on the problem.

Jackie also introduced test-driven development as a course practice: JUnit test cases compare expected output with actual output, specifications sit above individual code examples, and regression testing means rerunning tests after changes. This was not developed into full JUnit coding in this tutorial, but it sets up expectations for assignments and programming tests.

Mileage table as a motivating graph example

The central example was an airline mileage table with seven cities: Chicago, Boston, New York, Atlanta, Miami, Dallas, and Houston. The rows represent departure cities and the columns represent arrival or destination cities, producing a 7×7 table with 49 entries. Diagonal entries are zero because a city-to-itself trip has no flight distance.

Jackie then used the table to preview graph vocabulary. Each city can be viewed as a vertex or node. A direct flight between two cities can be viewed as an edge or arc. The distance on that edge is its weight. The itinerary `<Boston, New York, Dallas, Houston>` is a path whose relevant edge weights are looked up in the table: Boston to New York, New York to Dallas, and Dallas to Houston. The table is symmetric in this example because the distance from one city to another is the same in the reverse direction, such as Atlanta to Miami and Miami to Atlanta.

2D array syntax: array of arrays

The syntax review began with the familiar one-dimensional declaration `int[] ia`. Jackie explained the type by separating the array marker from the element type: it is an array whose elements are `int`. The two-dimensional declaration `int[][] iaa` extends this idea: it is an array whose elements are themselves `int[]` arrays.

For a concrete rectangular example, the iPad notes used

```
{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}.
```

This produces three rows, each of length four. The number of rows is obtained from `list2.length`. Under the rectangular assumption, the number of columns can be obtained from any valid row, such as `list2[1].length`.

Creating a 2D array object and assigning rows

Jackie then moved from literal initialization to object creation with `new`. With `new int[3][4]`, Java creates three rows, each with four integer entries initially set to the default value `0`. Separate row arrays such as `row1`, `row2`, and `row3` can then be assigned into `list2a[0]`, `list2a[1]`, and `list2a[2]`.

The important reference idea is that `list2a[0]` is a row reference. Assigning `list2a[0] = row1` redirects that row reference to the array referred to by `row1`. A student question prompted Jackie to clarify that, for this course, it is reasonable to think of the row reference as storing the address of the beginning of that row array.

Jackie also showed the alternative `new int[3][]`. This creates the outer array of three row references, but each row reference initially stores `null`; rows can be attached later. This supports the broader point that Java 2D arrays do not have to be rectangular.

Traversing rows and columns

In the live coding, Jackie created a helper method such as `printMatrix(int[][] m)` and used nested loops: the outer loop walks through row indices and the inner loop walks through column indices. For a known rectangular matrix, using a fixed row length such as `m[0].length` can work.

Jackie then turned that simple version into a warning. If rows have unequal lengths, the inner loop should use the current row length, such as `m[r].length`. If a row can be `null`, code must check that before accessing its length or entries. This is the difference between writing a quick rectangular-array loop and writing code robust enough for jagged arrays.

Encoding the distance table by indices

The distance table was then encoded as a 2D integer array whose row and column indices correspond to the seven cities. For example, Boston has index 1 and Miami has index 4, so the distance from Boston to Miami can be found at `distance[1][4]`. Since the user types city names as strings, the program needs a utility method such as `indexOf` to convert a city name into its integer index.

This part of the tutorial separated the problem representation from the implementation: the graph view helps students understand vertices, edges, paths, and weights, while the Java program actually looks up entries in the 2D array.

Itinerary calculation and loop boundary

The final live-coded application read the number of cities, stored the typed city names into a `String[]` called something like `trip`, and then processed adjacent pairs. For the example `Boston`, `New York`, `Dallas`, `Houston`, the loop needs to process pairs at positions $(0,1)$, $(1,2)$, and $(2,3)$.

The boundary condition was the most important logic point: if there are `howMany` cities, the last valid value of `i` for accessing both `trip[i]` and `trip[i + 1]` is `howMany - 2`. Therefore the loop condition should be `i < howMany - 1`. Jackie also noted a Java console-input detail: after `nextInt()`, use `nextLine()` to consume the remaining newline before reading city names with `nextLine()`.

Key Terms, Notation, and Structures

- **2D array**: In Java, an array whose elements are themselves arrays; for example, `int[][]` is an array of `int[]` rows.
- **Row and column**: In the mileage table, rows represent departure cities and columns represent arrival or destination cities.
- **Rectangular 2D array**: A 2D array whose rows all have the same length, such as a 3×4 array.

- **Jagged 2D array**: A 2D array whose rows may have different lengths, or where some row references may be `null`.
- `list2.length`: The number of rows in `list2`, because the outer array stores row references.
- `list2[r].length`: The number of entries in row `r`; this is the safer inner-loop bound when rows may have different lengths.
- `new int[3][4]`: Creates an outer array of length 3, with each row initialized as an integer array of length 4 containing default zero values.
- `new int[3][]`: Creates only the outer array of three row references; each row starts as `null` until an actual row array is assigned.
- **Reference assignment**: An assignment such as `list2a[0] = row1` makes the first row reference point to the same row array as `row1`.
- **Vertex or node**: In the graph interpretation, each city is a vertex or node.
- **Edge or arc**: In the graph interpretation, a direct flight connection between two cities is an edge or arc.
- **Weight**: The distance associated with an edge; in the table, the weight is the mileage entry for a departure-destination pair.
- **Path**: A sequence of vertices, such as the itinerary `<Boston, New York, Dallas, Houston>`.
- **Adjacency matrix**: A graph implementation idea in which a 2D table stores information about pairs of vertices.
- **Dynamic programming tabulation**: A bottom-up dynamic programming approach that fills a table; Jackie previewed that such tables may be 1D, 2D, or higher-dimensional.
- **JUnit test case**: A test case that compares expected output with actual output.
- **Specification**: The general behaviour expected of the code, beyond one individual example.
- **Regression testing**: Rerunning tests after code changes to check that earlier behaviour still works.
- `Scanner`: A Java class used in the live coding to read console input from `System.in`.
- `indexOf(city)`: The utility-method idea used to convert a city name such as `Boston` into the integer index needed for 2D array lookup.
- **Adjacent-pair loop**: A loop over `trip[i]` and `trip[i + 1]`, where the loop must stop before `i + 1` leaves the array.

Key Examples and Demonstrations

Importing a zipped Java project into Eclipse

Setup: Jackie downloaded the provided tutorial solution archive and imported it into a fresh Eclipse workspace.

Main point: For a course-provided `.zip` project, the safe import path is `File -> Import -> General -> Existing Projects into Workspace`, then select `archive file`, browse to the `.zip`, and finish. This routine matters because programming tests will use starter projects in Eclipse.

What to practise: Download the Tutorial 1 archive, import it into Eclipse using the archive-file option, and confirm that you can run the provided Java application.

Mileage table and graph interpretation

Setup: The notes show a 7×7 mileage table for Chicago, Boston, New York, Atlanta, Miami, Dallas, and Houston, with the sample itinerary `<Boston, New York, Dallas, Houston>`.

Main point: The table can be read as a graph: cities are vertices, city-pairs with direct flights are edges, distances are weights, and an itinerary is a path. The sample path uses the table entries Boston to New York, New York to Dallas, and Dallas to Houston, giving total mileage $214 + 1548 + 239 = 2001$.

What to practise: Pick another itinerary, identify each adjacent city-pair, find the corresponding row and column entry, and add only the distances for those adjacent pairs.

Literal initialization of a rectangular 2D array

Setup: Jackie used a nested curly-brace expression with three rows and four columns:

```
{{1,2,3,4}, {5,6,7,8}, {9,10,11,12}}.
```

Main point: The outer braces describe the outer array of rows, while each inner brace describes one row array. For this example, `list2.length` is 3, and each row has length 4.

What to practise: Redraw the memory diagram as an outer array pointing to three row arrays, then explain why `list2[1].length` is 4.

Creating rows with `new` and assigning references

Setup: Jackie compared `new int[3][4]` with `new int[3][]`, then assigned row arrays using statements such as `list2a[0] = row1`.

Main point: `new int[3][4]` creates rows filled with default zero values, while `new int[3][]` creates row slots that initially hold `null`. Assigning a row changes which row array a row reference points to.

What to practise: Write a tiny program that creates both versions, prints the rows before and after assignment, and explains which rows exist and which row references are still `null`.

Printing a 2D array with nested loops

Setup: Jackie sketched a `printMatrix(int[][] m)` helper method with an outer row loop and an inner column loop.

Main point: The rectangular version can use a fixed column length, but robust traversal should account for rows of different lengths and possible `null` rows.

What to practise: Implement a version of `printMatrix` that works for a rectangular array, then revise it so that it handles jagged rows and skips or reports `null` rows safely.

The itinerary calculation loop

Setup: The final diagram represents a `trip` array of length 4 containing `Boston`, `New York`, `Dallas`, and `Houston`.

Main point: The loop should process `trip[i]` with `trip[i + 1]`. For 4 cities, `i` takes values 0, 1, and 2, not 3. This is why the condition is `i < howMany - 1`.

What to practise: Trace the loop for itineraries of length 2, 3, and 4, and identify the last valid value of `i` in each case.

Common Pitfalls and Misconceptions

Importing a `.zip` project as a root directory

Pitfall: Choosing the root-directory option when importing a zipped Java project into Eclipse.

Why it causes problems: A course-provided `.zip` archive is not an already-unzipped project folder, so Eclipse may not import it as intended.

How to avoid it: Use `Existing Projects into Workspace` and select the `archive file` radio button for the downloaded `.zip` file.

Treating every Java 2D array as rectangular

Pitfall: Assuming that every row has the same length, or that every row reference is non-`null`.

Why it causes problems: Java allows jagged arrays and `null` rows. Code that blindly uses one fixed column length can miss entries or cause run-time errors.

How to avoid it: In general-purpose traversal code, check whether `m[r]` is `null`; if it is not, use `m[r].length` as the inner-loop bound.

Confusing a row with a single cell

Pitfall: Trying to assign an entire row array into one integer cell, such as confusing the type of `list2a[0]` with the type of `list2a[0][1]`.

Why it causes problems: `list2a[0]` is an `int[]` row reference, but `list2a[0][1]` is a single `int`; Java requires compatible assignment types.

How to avoid it: Before assigning, say the type out loud: row access gives an array; row-and-column access gives one element.

Using the wrong loop boundary for adjacent pairs

Pitfall: Looping over all city indices while also accessing `trip[i + 1]`.

Why it causes problems: On the last city index, `i + 1` is outside the array, which can lead to an `ArrayIndexOutOfBoundsException`.

How to avoid it: When processing adjacent pairs in an array of length `howMany`, use the condition `i < howMany - 1`; the last processed `i` is `howMany - 2`.

Forgetting the `Scanner` newline after `nextInt()`

Pitfall: Reading an integer with `nextInt()` and then immediately trying to read a city name with `nextLine()`.

Why it causes problems: The leftover newline can be consumed as an empty line instead of the intended city name.

How to avoid it: After `nextInt()`, call `nextLine()` once to consume the remainder of the line before reading string input.

Relying on unsupported user input

Pitfall: Expecting the demo application to recognize misspelled or differently formatted city names.

Why it causes problems: Jackie noted that the demo does not include robust error checking, so city names must match the expected names used by the utility method.

How to avoid it: For this tutorial app, type the city names exactly as supported; for later practice, consider what validation would be needed for a more robust application.

What You Should Be Able to Do

After this lecture, you should be able to:

- import a provided Java `.zip` project into Eclipse using the archive-file option;
- explain why 2D arrays are relevant to graph adjacency matrices and dynamic programming tabulation;
- describe a 2D array in Java as an array of arrays;
- distinguish the type of a row expression such as `list2a[0]` from a cell expression such as `list2a[0][1]`;
- initialize a rectangular 2D array using nested curly braces;
- create a 2D array using `new int[rows][cols]` and explain the default values created;

- explain the difference between `new int[3][4]` and `new int[3][]`;
- draw a reference diagram showing an outer array pointing to row arrays;
- use `array.length` and `array[r].length` correctly for row and column traversal;
- identify when a 2D array traversal needs to handle jagged rows or `null` rows;
- interpret the mileage table as both a 2D array and a graph-like structure with vertices, edges, weights, and paths;
- map a city name to an integer index before using it to access a 2D array;
- trace the itinerary loop that processes `trip[i]` and `trip[i + 1]`;
- justify why the loop condition for `howMany` cities should be `i < howMany - 1`;
- explain the role of JUnit tests, expected versus actual output, specification, and regression testing in the course workflow.

Suggested Practice

- Import the Tutorial 1 Java archive into Eclipse from scratch, making sure you use the archive-file option rather than the root-directory option.
- Recreate the 3×4 example from the notes using both nested curly braces and `new int[3][4]` plus row assignment.
- Draw the reference diagram for `new int[3][]`, then mark which entries are initially `null` and how they change after assigning rows.
- Implement a simple `printMatrix` for rectangular arrays, then revise it to handle jagged arrays and `null` rows.
- Re-run the mileage example `<Boston, New York, Dallas, Houston>` and verify the total 2001 by manually looking up the three table entries.
- Try two new itineraries of your own and trace the row index, column index, looked-up distance, and accumulated total for each adjacent pair.
- Trace the loop condition `i < howMany - 1` for itineraries with 2, 3, and 4 cities; make sure you can explain the last valid value of `i`.
- Compare the 2D-array mileage solution with the posted 1D-array version, focusing on design clarity rather than asymptotic running time.
- Keep an eye on upcoming 2D-array exercises and future JUnit-based tutorials; use them to practise testing rather than only reading code.

Where to Review

- **iPad/PDF notes, page 1:** Tutorial title and scope: 2D arrays, syntax, and the motivating example.
- **iPad/PDF notes, page 2:** Why 2D arrays matter later: adjacency matrices for graphs, dynamic programming tabulation, and the TDD/JUnit/regression-testing preview.
- **iPad/PDF notes, page 3:** The mileage table, departure rows, destination columns, graph interpretation, symmetry, and the Boston-New York-Dallas-Houston path.
- **iPad/PDF notes, page 4:** 2D array initialization and indexing with the 3×4 rectangular example.
- **iPad/PDF notes, page 5:** Creating a new 2D array object, assigning row arrays, reference diagrams, default zero rows, and the `null`-row alternative.
- **iPad/PDF notes, page 6:** Encoding the seven-city distance table as a 2D array and using city-name-to-index conversion.
- **iPad/PDF notes, page 7:** The `trip` array, adjacent-pair processing, and the `howMany - 1` boundary.
- **Tutorial recording:** Rewatch the Eclipse import demonstration, the explanation of row references, the jagged-array traversal discussion, and the final itinerary loop.
- **Posted Java solution:** Review `MileageApp2DArray`, the utility method for city indices, the basic 2D-array app, and the optional 1D-array version for comparison.

Closing Reminder

Use this digest as a roadmap for review, not as a replacement for the tutorial recording, the iPad notes, the posted Java code, or your own programming practice. The important work after this tutorial is active: import the project, redraw the diagrams, trace the indices, run the code, modify the examples, and explain why each loop boundary is safe.

EECS3101 Algorithms (Summer'26)

Digest of Tutorial 2, May 14

JUnit Practice and Fixed-Increment Dynamic Array Analysis

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 10

Closing Reminder 10

Main Ideas

- This tutorial combined practical Java/JUnit setup with the main technical topic of dynamic arrays and amortized analysis.
- Jackie demonstrated the expected Eclipse workflow for importing starter code, running JUnit tests, and using red/green feedback to guide implementation.
- The 2D-array exercises were positioned as Java practice for programming-test preparation, especially for reading tests, creating missing model code, and checking edge cases.
- The written Big-O exercises reinforced that a proof of an asymptotic upper bound must exhibit concrete constants, not just name the target family.
- The main data-structure example was an array-backed generic stack whose internal array grows when the current capacity is exhausted.
- The constant-increment resizing strategy was analyzed carefully: some pushes cost $O(1)$, but a resizing push copies all existing elements and is $O(n)$ in the current size.
- Over a sequence of n pushes, fixed-increment resizing produces an arithmetic sequence of copy costs, giving aggregate resizing cost $O(n^2)$ and amortized cost $O(n)$ per push.
- The doubling strategy was previewed as the next comparison point: it still has linear-cost resize events, but it resizes less frequently and will lead to a better amortized result.

Roadmap

Tutorial setup and what to work on independently

Jackie began by locating Tutorial 2 on the lecture site and distinguishing the parts to be done together from the parts students should complete on their own. Task 1 is Java/JUnit practice with 2D arrays, Task 2 is written Big-O proof practice, and Task 3 is the dynamic-array analysis developed in detail during the tutorial. Jackie also reminded the class that the first programming test is after reading week, that more Java practice from EECS2030/EECS2101 would be posted before the following Tuesday, and that Assignment 1 would later use similar project and submission workflows.

Importing the Java starter project into Eclipse

The first practical demonstration was how to import the Tutorial 2 starter project. Jackie showed the expected workflow: open Eclipse, use `File > Import > General > Existing Projects into Workspace`, choose `Select archive file`, browse to the downloaded zip file, and finish the import. The key warning was not to use the `root directory` option for this kind of starter archive, because that was a common source of wasted time in earlier offerings. The same kind of import workflow will matter for programming-test and assignment practice.

JUnit red/green workflow and 2D-array practice

After import, Jackie pointed out that the project may initially show red crosses or compilation errors; that is expected when starter code is incomplete. The project structure separates model code from tests, and students are expected to create or complete the required classes and methods so the JUnit tests pass. The notes include two 2D-array problems: returning the row with the maximum sum under the assumptions that the 2D array is non-empty and the maximum-sum row is unique, and checking whether a non-empty 2D array is rectangular. The main study habit is to run the tests, read the failures, fix the model code, and repeat until the JUnit bar is green.

Big-O proof practice with explicit witnesses

Task 2 was described as written-test or exam-style Big-O practice. The main requirement is that proving $f(n) \in O(g(n))$ means showing suitable constants, usually written as C and n_0, and justifying that the inequality holds for all sufficiently large n. Jackie emphasized that simply stating the target bound is not enough. Students should practise this separately from implementation so that mathematical reasoning and programming practice both develop.

Why fixed-capacity arrays are not flexible enough

The technical analysis began with a stack or queue implemented using an ordinary array of fixed capacity `MAX`. A fixed-capacity stack may work until exactly `MAX` pushes have filled indices 0 through `MAX` $- 1$, but one additional push can lead to an `ArrayIndexOutOfBounds Exception`. Jackie noted that a custom "stack full" exception may be a better API design than letting Java throw an array-bounds exception, but it still does not solve the flexibility problem. This motivates dynamic arrays: the array should grow internally when necessary.

Two resizing strategies: fixed increments and doubling

The notes and transcript distinguish two strategies for growing an internal array. In the fixed-increment strategy, whenever the array is full, the implementation creates a new array with C additional free slots. In the doubling strategy, the new array is about twice as large as the current one. Jackie stressed the average-case/amortized intuition: the strategy that requires less frequent resizing is asymptotically more efficient on average, even though an individual resize still requires copying existing elements. This tutorial focused on fixed increments; doubling was left for a later tutorial.

The array-backed generic stack

The main code example was a generic `ArrayStack<E>` backed by an array `data`. The important fields are the initial capacity I, the fixed increment C, the current `capacity`, the array reference `data`, and the top index `t`, initialized to -1. Jackie used concrete values such as $I = 1000$ and $C = 500$ to visualize the initial indices and the extra space made by resizing. The generic-array detail is part of the implementation context, and Jackie connected it to upcoming assignment work with generics.

Push with and without resizing

The push operation has two cases. If the number of stored elements is still below `capacity`, the actual push only increments `t` and stores the new element, so this case is $O(1)$. If the array is full, the method must first allocate a temporary array of size `capacity` + C, copy existing elements into it, redirect `data` to the new array, update `capacity`, and then perform the push. The copy loop is the expensive part, so a resizing push has linear worst-case cost in the number of existing elements.

Setting up amortized analysis over many pushes

Jackie then shifted from the worst-case cost of a single push to the amortized cost over n pushes. Amortized cost means total cost over the operation sequence divided by the number of operations. The analysis assumes, without loss of generality for the pattern being studied, that the n-th push triggers the last resize. The first resize copies I elements, the second copies $I + C$, the third copies $I + 2C$, and so on. Before the n-th push, there are $n - 1$ existing elements, so the last resize copies $n - 1$ elements.

Summing the resize costs

The resize costs form an arithmetic sequence:

$$I + (I + C) + (I + 2C) + \cdots + (n - 1).$$

If the last term is the k-th resizing cost, then

$$I + (k - 1)C = n - 1, \qquad k = \frac{n - I - 1}{C} + 1.$$

Using the arithmetic-series formula, the total resize cost has a dominant quadratic term, so the aggregate cost over the n pushes is $O(n^2)$. Dividing by n pushes gives amortized cost $O(n)$ per push for fixed-increment resizing. Jackie ended by emphasizing that students should review and redo this derivation independently, because the next resizing strategy will use a similar but more delicate analysis.

Key Terms, Notation, and Structures

- **JUnit red/green bar**: A red bar means at least one starter test fails; a green bar means all currently provided tests pass.
- **Model package vs. tests package**: The model package contains the code students complete; the tests package contains JUnit test cases used to check that code.
- **2D array**: An array of arrays; in this tutorial, it was used for the row-with-maximum-sum and rectangle-checking exercises.
- **Rectangular 2D array**: A 2D array whose rows all have the same length; the notes assume the input array is non-empty.

- **Row with maximum sum**: A 2D-array task asking for the one-dimensional row whose entries have the largest total; the notes assume the maximum-sum row is unique.
- **Big-O witnesses**: Constants C and n_0 used to prove that $f(n) \leq Cg(n)$ for all $n \geq n_0$.
- **LIFO stack**: A Last-In-First-Out structure; the tutorial's main implementation example was an array-backed stack.
- **FIFO queue**: A First-In-First-Out structure; Jackie mentioned queues as another structure that can be implemented with arrays, though the detailed example used a stack.
- **Fixed capacity**: An array design with a pre-set maximum number of slots, such as `MAX`, and no automatic resizing.
- **Dynamic array**: An array-backed structure that creates a larger internal array and copies elements when more capacity is needed.
- **Fixed-increment resizing**: A dynamic-array strategy where each resize adds a fixed number C of slots.
- **Doubling strategy**: A dynamic-array strategy where capacity is multiplied by about two; it was previewed for the next resizing analysis.
- I: The initial capacity of the internal array in the stack example.
- C: In the dynamic-array analysis, the fixed number of additional slots made during each resize; do not confuse this with the Big-O proof constant when the context changes.
- `capacity`: The current length or available slot count of the internal array.
- `data`: The internal array reference used to store stack elements.
- `t`: The top index of the stack; Jackie initialized it to -1 so the first push stores at index 0.
- **Copy loop**: The loop that copies all existing elements from the old array into the larger temporary array; this is the dominant cost during resizing.
- **Worst-case cost of** `push`: $O(n)$ when resizing occurs, because all existing elements must be copied.
- **No-resize cost of** `push`: $O(1)$, because the method only updates the top index and stores the new element.
- **Aggregate running time**: The total cost of a whole sequence of operations, such as n pushes.
- **Amortized running time**: Aggregate running time divided by the number of operations in the sequence.
- **Arithmetic sequence of resize costs**: Under fixed increments, resize costs grow as $I, I + C, I + 2C, \ldots$.
- **Number of resizes** k: If the last resizing copies $n - 1$ elements, then $I + (k - 1)C = n - 1$, so $k = \frac{n-I-1}{C} + 1$.

Key Examples and Demonstrations

Importing the starter project into Eclipse

Setup: The Tutorial 2 Java starter code is provided as a zip archive, and the Eclipse workspace begins empty or without this project.

Main point: Use `Existing Projects into Workspace` with `Select archive file`; do not switch to the root-directory option. This is a practical workflow skill for tutorials, assignments, and programming-test preparation.

What to practise: Download the starter zip, import it into a clean workspace, delete and re-import it once, and make sure you can find the model and tests packages without relying on Jackie doing the steps live.

Running JUnit tests to guide 2D-array implementation

Setup: The notes show JUnit tests for two 2D-array methods: one for returning the row with maximum sum and one for checking whether a 2D array is rectangular.

Main point: The starter project is expected to fail at first; the workflow is to run the tests, read the failures, implement the missing model code, and repeat until all starter tests pass.

What to practise: Implement the methods, then test cases with rectangular arrays, non-rectangular arrays, negative values, rows of different lengths, and the exact assumptions stated in the notes.

Fixed-capacity stack as motivation for dynamic arrays

Setup: A stack is backed by an array with indices 0 through `MAX` $- 1$, and repeated `push` operations fill every slot.

Main point: A fixed-capacity array either risks an `ArrayIndexOutOfBoundsException` or requires a custom full-stack exception; either way, it remains inflexible. Dynamic resizing is introduced to let the structure grow internally.

What to practise: Draw the fixed array, mark the top after `MAX` pushes, and explain why one more push cannot be handled without either an exception or resizing.

Push operation with fixed-increment resizing

Setup: The stack has current capacity I, fixed increment C, internal array `data`, and top index `t`. A push is called when the array may or may not be full.

Main point: Without resizing, push is constant time. With resizing, the method allocates a larger array, copies all existing elements, updates the array reference and capacity, then performs the actual push; the copy loop is the source of the linear cost.

What to practise: Trace one no-resize push and one resize-triggering push with small values such as $I = 4$ and $C = 3$, writing down the old capacity, new capacity, number of copied elements, and new value of `t`.

Amortized analysis of fixed-increment resizing

Setup: Consider n pushes, and assume the last push triggers the last resizing routine. The resize copy costs are $I, I + C, I + 2C, \ldots, n - 1$.

Main point: The resize costs form an arithmetic sequence. Solving $I + (k - 1)C = n - 1$ gives the number of resizing terms, and the arithmetic-series formula gives aggregate cost $O(n^2)$, hence amortized cost $O(n)$ per push.

What to practise: Re-derive the formula for k, plug it into the arithmetic-series sum, and explain in words why fixed increments do not give constant amortized push.

Fixed increment versus doubling

Setup: A student asked whether choosing $C = I$ means the implementation is doubling.

Main point: Jackie clarified that the first resize may look like doubling, but the later capacities are $I, 2I, 3I, 4I, \ldots$, so this is still fixed-increment growth, not doubling.

What to practise: Write the capacity sequences for fixed increment with $C = I$ and for true doubling, then compare how many resizes each strategy needs before reaching a large capacity.

Common Pitfalls and Misconceptions

Importing the starter code using the root-directory option

Pitfall: Choosing `root directory` instead of `Select archive file` when importing the starter zip into Eclipse.

Why it causes problems: Eclipse may not recognize the project correctly, and students can lose time before even starting the actual programming task.

How to avoid it: Practise the exact import sequence Jackie demonstrated and check that the project appears with the expected model and tests packages.

Treating initial red crosses as unexpected

Pitfall: Assuming that red crosses or failing JUnit tests mean the starter project is broken.

Why it causes problems: Starter projects are often incomplete by design; the red state tells you what needs to be implemented.

How to avoid it: Run the JUnit tests, inspect the failures, implement one requirement at a time, and rerun the tests until the bar becomes green.

Naming a Big-O family without proving the bound

Pitfall: Writing only the final answer, such as $O(n^2)$, when the question asks for a proof.

Why it causes problems: A formal Big-O proof requires explicit witnesses and an inequality argument; otherwise the answer is incomplete for written tests or exams.

How to avoid it: Always identify the running-time function, the reference function, a valid constant, and a valid threshold n_0.

Ignoring the copy loop during resizing

Pitfall: Treating resizing as just "creating a bigger array" and counting it as constant time.

Why it causes problems: The expensive part is copying all existing elements into the new array, which is linear in the current number of stored elements.

How to avoid it: When analyzing resize code, find the loop that copies old elements and count how many elements it copies.

Confusing fixed increments with doubling

Pitfall: Thinking that fixed-increment resizing becomes doubling whenever the first increment C equals the initial capacity I.

Why it causes problems: The capacity sequence and amortized conclusion are different: fixed increments grow linearly, while doubling grows multiplicatively.

How to avoid it: Write the first several capacities. Fixed increment with $C = I$ gives $I, 2I, 3I, 4I, \ldots$, while doubling gives $I, 2I, 4I, 8I, \ldots$.

Using n instead of $n - 1$ for the last resize

Pitfall: Saying the last resize copies n elements when the n-th push triggers that resize.

Why it causes problems: Before the n-th push inserts the new element, only $n - 1$ elements are already present, so the resize copies $n - 1$ elements.

How to avoid it: Separate the pre-push state from the post-push state whenever you analyze resizing.

Claiming fixed-increment push is amortized $O(1)$

Pitfall: Assuming that because most pushes do not resize, the amortized cost must be constant.

Why it causes problems: Under fixed increments, the resize copy costs form a growing arithmetic sequence whose aggregate cost is quadratic over n pushes.

How to avoid it: Sum the resize costs before dividing by the number of pushes; for fixed increments, $O(n^2)/n = O(n)$.

What You Should Be Able to Do

After this tutorial, you should be able to:

- import a Java starter project into Eclipse from a zip archive using `Select archive file`;
- locate the model and tests packages, run JUnit tests, and interpret red and green test results;
- implement small 2D-array utility methods that satisfy given JUnit tests;
- state the assumptions for the row-with-maximum-sum and rectangle-checking 2D-array tasks;

- prove a Big-O upper bound by giving concrete constants and a threshold n_0;
- explain why a fixed-capacity array-backed stack is not flexible enough once more than `MAX` elements are pushed;
- distinguish fixed-increment resizing from doubling by writing their capacity sequences;
- identify the roles of I, C, `capacity`, `data`, and `t` in the array-backed stack example;
- trace a push operation in both cases: no resizing and resizing;
- justify why a no-resize push is $O(1)$ while a resize-triggering push is $O(n)$;
- derive the resizing cost sequence $I, I + C, I + 2C, \ldots, n - 1$ for fixed-increment growth;
- solve $I + (k - 1)C = n - 1$ for the number of resizes k;
- use the arithmetic-series formula to explain why fixed-increment resizing has aggregate cost $O(n^2)$ and amortized cost $O(n)$ per push;
- explain, at a high level, why a less frequent resizing strategy should improve amortized performance.

Suggested Practice

- Download the Tutorial 2 starter project, import it into Eclipse, and repeat the import once from a clean workspace to make the workflow automatic.
- Complete the 2D-array methods, then add your own small tests for negative numbers, duplicate row lengths, different row lengths, and boundary cases allowed by the stated assumptions.
- Run the JUnit tests after each small implementation step instead of waiting until the end.
- Practise exporting the completed Eclipse project as a zip file, since Jackie mentioned similar submission guidance for upcoming assignments.
- Complete the Task 2 Big-O proof questions by writing the constants and threshold explicitly.
- Draw a fixed-capacity stack of size `MAX`, fill it with pushes, and explain exactly why the next push fails without resizing.
- Trace fixed-increment resizing with small numbers, such as $I = 4$, $C = 3$, and about ten pushes; record each resize, capacity, and copy cost.
- Re-derive the amortized analysis from scratch: first three resize costs, last resize cost $n-1$, formula for k, arithmetic-series sum, and final amortized bound.

- Compare the capacity sequences for fixed increment and doubling before the next dynamic-array tutorial.
- Try the same starter project or a small practice project on the remote lab Eclipse environment so the programming-test environment feels familiar.

Where to Review

- **Tutorial 2 instructions and starter code:** Use these for the exact Eclipse/JUnit workflow and the tasks Jackie assigned for independent practice.
- **Tutorial 2 iPad/PDF notes, page 1:** Overall tutorial focus: 2D arrays, JUnit testing, deriving asymptotic upper bounds, and Dynamic Arrays Part 1.
- **Tutorial 2 iPad/PDF notes, pages 4–5:** The 2D-array problems and sample JUnit tests for row with maximum sum and rectangle checking.
- **Tutorial 2 iPad/PDF notes, page 2:** Fixed-capacity versus dynamic arrays, including fixed increments and doubling.
- **Tutorial 2 iPad/PDF notes, page 3:** The array-backed stack code and the full fixed-increment amortized analysis; this is the main page to redraw and re-derive.
- **Asymptotic-analysis slides around dynamic arrays:** Review the fixed-array, constant-increment, doubling, and summary slides Jackie referenced.
- **Recording for Tutorial 2:** Rewatch the Eclipse import demonstration if your project setup is not automatic, and rewatch the dynamic-array derivation if the formula for k or the $n-1$ last-copy count is unclear.
- **Earlier Big-O proof material:** Return to the lecture notes on witnesses C and n_0 before attempting the written proof questions.
- **Upcoming Java practice resources:** Use the EECS2030/EECS2101-style exercises Jackie said would be posted before the following Tuesday to rebuild comfort with Java, Eclipse, and JUnit.

Closing Reminder

This tutorial digest is only a roadmap. To benefit from the tutorial, you still need to import and run the Java project yourself, read the JUnit failures, implement the 2D-array methods, write the Big-O proofs, and re-derive the dynamic-array amortized analysis on paper. The most important technical review is to redraw the fixed-increment resizing diagram and explain why the aggregate cost is $O(n^2)$ but the amortized cost per push is $O(n)$. Use Jackie's notes, the recording, the starter code, and the posted solutions or walkthroughs to check your work, not to replace doing the work.

EECS3101 Algorithms (Summer'26)

Digest of Tutorial 3, May 25

Doubling Dynamic Array and Generic BSTs in Java

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 9

Suggested Practice 9

Where to Review 10

Closing Reminder 11

Main Ideas

- Tutorial 3 finished the dynamic-array story by comparing fixed-increment resizing with the doubling strategy, with the main payoff being the amortized cost of `push`.
- The key technical result is that doubling does not improve the worst-case cost of one resizing `push`; that worst case remains $O(n)$, because existing elements still must be copied.
- The improvement appears only over a sequence of pushes: the resize costs form a geometric sequence $i, 2 \cdot i, 4 \cdot i, \ldots, n-1$, whose total is $O(n)$, so the amortized cost per push is $O(1)$.
- The second half shifted from analysis to implementation, introducing the generic `BSTNode<E>` structure used for binary search trees in Java.
- Jackie used in-order traversal and a small manually constructed BST to preview the Java/JUnit work that supports Assignment 1 and Programming Test 1.
- Jackie also pointed students to optional EECS2101 pre-study assignments on recursion, singly linked lists, and trees; these are practice resources, not submissions.

Roadmap

Tutorial logistics and preparation resources

Jackie began by situating the session as Tutorial 3. The tutorial had two main parts: dynamic-array amortized analysis with doubling, and a first look at generic BST programming in Java. Jackie also announced optional pre-study materials on e-class: three past EECS2101 programming assignments covering recursion, singly linked lists, and trees. These are intended as extra Java practice before Assignment 1 and Programming Test 1, not as required submissions. Jackie emphasized that reading solution code is not enough preparation; students need to practise writing and testing code.

Assignment 1 was planned for release the next Wednesday, and Programming Test 1 was still a few weeks away. Jackie described Assignment 1 as the most relevant study material for the programming test, while the optional EECS2101 materials can help students rebuild prerequisite Java fluency.

From constant increments to doubling

The first technical segment revisited the dynamic-array stack from Tutorial 2. In the fixed-increment strategy, the array capacity grew by adding a constant C. In the doubling strategy, the key code change is that the new capacity is based on `capacity * 2`; after copying, the stored capacity is also doubled. Jackie highlighted that this small implementation change leads to a major amortized-time difference.

The code was split into two conceptual regions. The unconditional push part increments the top index and stores the new element. The conditional resizing part runs only when `size() == capacity`. That resizing block creates a larger array, copies the existing elements, redirects `data`, and updates `capacity`.

Worst-case cost of one push

Jackie first separated worst-case analysis from amortized analysis. If a `push` triggers resizing, the copy loop runs once for each currently stored element. Therefore, the worst-case running time for a single push is still $O(n)$, even under doubling. The benefit of doubling is not that copying becomes cheaper; the benefit is that copying happens less often.

This distinction was important enough that Jackie returned to it later when answering a student question: $O(1)$ amortized does not mean every individual push takes constant time.

Resize costs as a geometric sequence

The iPad notes then visualized the resizing events. Starting with initial capacity i, the first resize copies i elements, the second copies $2 \cdot i$, the third copies $4 \cdot i$, and so on. Jackie rewrote these as

$$2^0 \cdot i, \quad 2^1 \cdot i, \quad 2^2 \cdot i, \quad \dots$$

to make the geometric pattern visible. The common factor is 2, unlike the constant-increment case, where the costs formed an arithmetic sequence.

A key counting point was reused from Tutorial 2: assume, without loss of generality for the analysis pattern, that there are n pushes and the last push triggers the last resize. Immediately before that n-th push, only $n-1$ elements already exist, so the last resize copies $n-1$ elements, not n.

Solving for the number of resizes

To apply the geometric-series formula, Jackie derived the number of resizing events. The k-th resizing cost follows the pattern

$$2^{k-1} \cdot i.$$

Since the last resizing cost is $n-1$, the analysis sets

$$2^{k-1} \cdot i = n - 1.$$

After dividing by i and applying logarithms, Jackie obtained

$$k = \log_2\left(\frac{n-1}{i}\right) + 1.$$

This step is where the logarithm facts matter: logarithms let us move an exponent down, and $2^{\log_2 x} = x$.

Total resizing cost and amortized push cost

With i as the initial term, common factor 2, and the derived value of k, Jackie applied the geometric sum formula. The resizing total simplifies to

$$2n - 2 - i,$$

so the aggregate resizing cost over n pushes is $O(n)$. Dividing by the number of push operations gives

$$\frac{O(n)}{n} = O(1).$$

Thus, with doubling, `push` has $O(1)$ amortized running time, even though a resizing push still has $O(n)$ worst-case running time.

Tutorial materials and Eclipse setup

After the break, Jackie pointed students to the Tutorial 3 materials and the short solution video for the comparison task. Jackie then demonstrated the familiar Eclipse import workflow: import an existing project into the workspace and select the archive file. The starter project was expected not to compile immediately, because students must implement missing methods and satisfy the JUnit tests.

The starter project focuses on simple BST utilities, especially in-order traversal and searching. Jackie positioned these exercises as preparation for later assignment work and programming-test practice.

Generic `BSTNode<E>` and internal/external nodes

The BST programming segment introduced the generic `BSTNode<E>` class. Each node stores a key, a value of generic type E, and references to its parent, left child, and right child. Jackie connected this to earlier linked-node structures: a tree node has one parent reference but multiple child references, so it naturally represents a nonlinear structure.

Jackie also distinguished internal and external nodes. A constructor with no arguments creates an external node, which acts as a placeholder and does not contribute meaningful key-value data to search or traversal results. A constructor with key and value creates an internal node. The notes use circles for internal nodes and squares for external nodes.

Recursive in-order traversal and call-by-value

Jackie then showed a recursive helper-method pattern for in-order traversal. The public traversal method creates an `ArrayList` called `order`, passes it to a private recursive helper, and returns it after the helper finishes. Inside the helper, internal nodes contribute to the traversal, while external nodes do not.

The implementation relies on Java call-by-value for object references: each recursive call receives a copy of the reference to the same `ArrayList` object. Mutating the list through any recursive frame accumulates results in the same list. Jackie pointed students back to EECS2030 call-by-value materials and the optional EECS2101 recursion assignment for extra practice with this idea.

Constructing and traversing a small BST

The final trace manually constructed a BST with internal nodes $(28, \text{“alan”})$, $(21, \text{“mark”})$, and $(35, \text{“tom”})$, plus four external nodes. Jackie emphasized the need to set both child and parent links: for example, setting `n28`'s left child to `n21` should be paired with setting `n21`'s parent to `n28`.

The purpose of the trace was not just to memorize this tree, but to understand object references at run time. Once the links are correct, in-order traversal starting from `n28` should return the internal nodes in sorted-key order: $21, 28, 35$.

Key Terms, Notation, and Structures

- **Dynamic array**: An array-backed structure that can allocate a larger array and copy existing elements when more capacity is needed.
- **Fixed-increment strategy**: A resizing strategy that grows capacity by adding a fixed constant C; from Tutorial 2, its amortized push cost remains linear.
- **Doubling strategy**: A resizing strategy that replaces capacity by $2 \cdot$ `capacity`; this tutorial showed that it gives $O(1)$ amortized push.
- **Worst-case running time**: The cost of one operation in its most expensive case; for a resizing `push`, this is $O(n)$ because all current elements are copied.
- **Amortized running time**: The total cost of a sequence of operations divided by the number of operations; for n pushes under doubling, this is $O(1)$ per push.
- **Initial capacity** i: The starting size of the dynamic array in the analysis.
- `capacity`: The current length of the underlying array; this is doubled when resizing occurs.
- **Resize cost sequence**: Under doubling, resize costs follow $i, 2 \cdot i, 4 \cdot i, \ldots$, a geometric sequence with common factor 2.
- **Last resize cost** $n - 1$: If the n-th push triggers the last resize, the array contains $n - 1$ existing elements before that push is stored.
- **Number of resizes** k: The analysis solves $2^{k-1} \cdot i = n - 1$, giving $k = \log_2(\frac{n-1}{i}) + 1$.
- **Geometric sum**: The summation tool used for doubling costs; here it leads to an aggregate resizing cost of $2n - 2 - i = O(n)$.
- `BSTNode<E>`: A generic binary-tree node class with a key, a value of type E, and parent/left/right references.
- **Generic parameter** E: A type parameter allowing the node value to be a `String`, `Integer`, `Person`, or another object type.

- **Recursive type**: A class design where fields have the same class type, such as `BSTNode<E>` fields inside `BSTNode<E>`.
- **Internal node**: A BST node with meaningful key-value data and child references.
- **External node**: A placeholder node with no meaningful key-value data; external nodes do not contribute to the in-order traversal sequence.
- **In-order traversal**: For a BST, the recursive traversal of left subtree, node, then right subtree; it produces keys in sorted order when the tree is a valid BST.
- **Recursive helper method**: A private method used to carry extra state, such as an accumulating `ArrayList`, through recursive calls.
- **Call by value for references**: Java copies the reference value when passing an object parameter; recursive calls can still mutate the same referenced object.
- **JUnit test**: A test case used to check whether the implemented traversal or search method returns the expected result.

Key Examples and Demonstrations

Worst-case `push` under doubling

Setup: The dynamic-array stack is full, so `size() == capacity` and the next `push` triggers the resizing block.

Main point: Even with doubling, resizing requires copying all current elements into the new array. The larger future capacity does not remove the current copy cost, so this single operation has worst-case cost $O(n)$.

What to practise: Trace a small array with capacity 4: fill it, trigger a resize, and identify exactly which lines correspond to allocation, copying, pointer update, capacity update, and the actual push.

Amortized derivation for the doubling strategy

Setup: Start with initial capacity i, assume n pushes, and assume the last push triggers the last resizing routine. The resize costs are $2^0 \cdot i, 2^1 \cdot i, 2^2 \cdot i, \ldots, n-1$.

Main point: The costs form a geometric sequence. Solving $2^{k-1} \cdot i = n-1$ gives the number of resize terms, and the geometric sum simplifies to $2n - 2 - i$, which is $O(n)$. Therefore the amortized push cost is $O(1)$.

What to practise: Re-derive the formula without looking at the notes. In particular, justify why the last copy cost is $n-1$, and explain why $2^{\log_2 x} = x$ is used in the simplification.

Importing and using the Tutorial 3 starter project

Setup: Jackie demonstrated downloading the Tutorial 3 starter code and importing it into Eclipse as an existing project from an archive file.

Main point: The starter project may have compilation errors until the required classes or methods are implemented. The goal is to use the JUnit tests to drive the implementation of BST operations such as traversal and search.

What to practise: Repeat the import process yourself, run the tests, inspect the failing methods, and implement the missing code until the JUnit results match the expected behaviour.

Recursive in-order traversal with a shared `ArrayList`

Setup: The traversal method creates an empty `ArrayList<BSTNode<E>>` and passes it to a recursive helper. The helper recursively visits left subtree, current node, and right subtree for internal nodes.

Main point: The same list object accumulates results across recursive calls because Java passes a copy of the reference value, not a new list object. External nodes are base cases and do not add anything to the traversal sequence.

What to practise: Trace the helper on a three-node BST and mark when each recursive call receives the same list reference. Then modify the example mentally: what would go wrong if each recursive call created a fresh list and did not combine the results correctly?

Manually constructing the $21, 28, 35$ BST

Setup: The JUnit example creates internal nodes for $(28,$ "alan"), $(21,$ "mark"), and $(35,$ "tom"), then creates four external nodes and links everything into a small BST.

Main point: Correct tree construction requires setting references in both directions: parent to child and child to parent. In-order traversal then verifies the structure by returning keys in sorted order $21, 28, 35$.

What to practise: Redraw the object diagram and label each `parent`, `left`, and `right` reference. Then write the corresponding setup code without looking at the solution.

Common Pitfalls and Misconceptions

Confusing worst-case and amortized claims

Pitfall: Saying that doubling makes every `push` operation $O(1)$.

Why it causes problems: A push that triggers resizing still copies all current elements, so one operation can still be linear.

How to avoid it: Always state the type of analysis: worst-case single push is $O(n)$, while amortized push over a long sequence is $O(1)$.

Counting every push as a resize

Pitfall: Summing costs as if every push operation executes the resizing block.

Why it causes problems: Amortized analysis works because resizing happens only occasionally. Treating every push as a resize destroys the intended analysis.

How to avoid it: Separate ordinary push cost from resizing cost. In this tutorial's derivation, focus on the sequence of resize events: first resize, second resize, third resize, and so on.

Using n instead of $n-1$ for the last resize

Pitfall: Assuming the last resizing routine copies n elements when the n-th push triggers it.

Why it causes problems: The copy happens before the new element is stored. At that moment, there are only $n-1$ existing elements.

How to avoid it: Draw the array state immediately before the triggering push: $n-1$ occupied slots are copied, then the n-th element is added afterward.

Losing the geometric-sequence pattern

Pitfall: Treating the doubling costs as if they formed an arithmetic sequence.

Why it causes problems: Fixed increments lead to an arithmetic sequence; doubling leads to a geometric sequence. Using the wrong summation formula gives the wrong conclusion.

How to avoid it: Rewrite the resize costs as $2^0 \cdot i, 2^1 \cdot i, 2^2 \cdot i, \ldots$. The exponent pattern is the clue that the common factor is 2.

Reading code solutions without practising

Pitfall: Looking at posted Java solutions and assuming recognition is enough preparation.

Why it causes problems: Programming tests require writing, tracing, debugging, and adapting code, not just understanding a finished solution when it is already in front of you.

How to avoid it: Use the optional EECS2101 materials and Tutorial 3 starter project actively: import, code, run tests, fix errors, and repeat.

Forgetting parent links in tree construction

Pitfall: Setting `left` or `right` links but forgetting the corresponding `parent` link.

Why it causes problems: The object structure no longer matches the intended tree, and later operations that depend on upward navigation can fail.

How to avoid it: Treat each connection as a pair of updates: after `parent.setLeft(child)`, also check `child.setParent(parent)`.

Treating external nodes as meaningful data nodes

Pitfall: Adding external nodes to the in-order traversal sequence or treating their placeholder values as searchable keys.

Why it causes problems: External nodes are placeholders for missing children or future insertions; including them corrupts the traversal result.

How to avoid it: In traversal and search code, explicitly distinguish `isInternal()` from `isExternal()` and only collect meaningful internal nodes.

What You Should Be Able to Do

After this lecture, you should be able to:

- identify the code-level difference between fixed-increment resizing and doubling in a dynamic-array stack;
- explain why a single resizing `push` is still $O(n)$ in the worst case under doubling;
- derive the resize-cost sequence $i, 2 \cdot i, 4 \cdot i, \ldots, n-1$ and explain why it is geometric;
- justify why the last resize copies $n-1$ elements when the n-th push triggers it;
- solve $2^{k-1} \cdot i = n-1$ for k using logarithms;
- apply the geometric-sum formula to show that total resizing cost under doubling is $O(n)$;
- explain why $O(n)$ total resizing cost over n pushes gives $O(1)$ amortized cost per push;
- compare the amortized behaviour of fixed increments and doubling without confusing amortized cost with worst-case cost;
- import the Tutorial 3 starter project into Eclipse and use JUnit tests to guide implementation;
- describe the fields of `BSTNode<E>` and explain why it is a recursive type;
- distinguish internal nodes from external nodes in the BST representation used in the tutorial;
- implement or trace a recursive in-order traversal helper using a shared `ArrayList`;
- explain how Java call-by-value for object references allows recursive calls to accumulate results in the same list;
- manually construct a small BST by correctly setting `left`, `right`, and `parent` links.

Suggested Practice

- Redo the full doubling derivation from scratch: draw the first few resizing events, write the geometric sequence, solve for k, sum the series, and divide by n.
- Trace a dynamic-array stack with a small initial capacity such as $i = 2$ or $i = 4$; record exactly which pushes trigger resizing and how many elements are copied.

- Compare Tutorial 2 and Tutorial 3 side by side: explain why fixed increments give linear amortized cost while doubling gives constant amortized cost.
- Watch the posted short solution video for Tutorial 3 Task 2, then write your own comparison of the two resizing strategies in words.
- Import the Tutorial 3 starter project into Eclipse using `Select archive file`, run the JUnit tests, and implement the missing traversal and search methods.
- Recreate the small BST with keys $21, 28, 35$ in Java, including all external nodes and parent links; then verify the in-order traversal result.
- Review Java generics and call-by-value using the optional EECS2030/EECS2101 review materials Jackie pointed to, especially if recursive helper methods still feel unclear.
- Work through at least one optional EECS2101 pre-study assignment actively rather than only reading its solution.

Where to Review

- **Tutorial 3 PDF/iPad notes, page 1:** overall structure of the tutorial: Dynamic Arrays Part 2 and testing generic BSTs using JUnit.
- **Tutorial 3 PDF/iPad notes, page 2:** main derivation for amortized analysis of dynamic arrays with doubling, including the resize sequence, the k-th resize, and the final $O(1)$ amortized result.
- **Tutorial 3 PDF/iPad notes, page 3:** visual reminder that worst-case cost applies to a particular resizing push, while amortized cost averages over n push operations.
- **Tutorial 3 PDF/iPad notes, page 4:** structure of a generic binary tree node, including `parent`, `key`, `value`, `left`, and `right`.
- **Tutorial 3 PDF/iPad notes, page 5:** internal versus external nodes in the BST representation.
- **Tutorial 3 PDF/iPad notes, page 6:** recursive in-order traversal helper and the shared `ArrayList` accumulation pattern.
- **Tutorial 3 PDF/iPad notes, page 7:** JUnit-style trace for constructing and traversing the small BST with keys $21, 28, 35$.
- **Tutorial 3 instructions and starter code:** use these for the actual Eclipse/JUnit practice; the digest is not enough to prepare for programming work.
- **Posted Task 2 solution video:** review it after attempting the strategy-comparison task yourself.

- **Optional e-class pre-study materials:** past EECS2101 assignments on recursion, singly linked lists, and trees, plus starter projects and solutions.
- **EECS2030 call-by-value refresher:** revisit this if the recursive traversal helper with a shared list is confusing.

Closing Reminder

Tutorial 3 connects a precise mathematical analysis with concrete Java implementation practice. Use this digest to find the main ideas, but do the real work in the official notes, the recording, the tutorial starter code, the JUnit tests, and the optional practice materials. In particular, do not stop at recognizing the amortized derivation or reading the BST solution code: redraw the resize diagrams, re-derive the formulas, import the project, write the methods, trace the object references, and debug your own mistakes.

EECS3101 Algorithms (Summer'26)

Digest of Tutorial 4, June 1

Trinode Restructuring Steps After Insertions and Deletions

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 9

Closing Reminder 10

Main Ideas

- Tutorial 4 focused on tree rotations as the practical mechanism for restoring the height-balance property in self-balancing BSTs.
- Jackie connected the paper rotation exercises directly to Assignment 1, which is already released, due Wednesday, June 17, and intended as strong preparation for Programming Test 1.
- The tutorial reviewed how to compute node heights bottom-up and use height differences to decide whether a subtree violates the height-balance property.
- The abstract A, B, C trinode restructuring pattern was used to explain why rotations preserve the BST sorting property through the same in-order traversal.
- A left-left imbalance was repaired by a single right rotation on the middle node B; the same-slant versus different-slant distinction determines single versus double rotations.
- A right-left insertion example showed how to identify the lowest unbalanced node A, choose B and C along the insertion ancestor path, and promote the lower node C through a double rotation.
- Deletion rebalancing was treated as more subtle than insertion rebalancing: after deletion, B and C are chosen using subtree heights, and multiple restructuring steps may be required.
- Jackie emphasized implementation discipline for Assignment 1: do not modify the provided `BSTNode` class, use the designated model package, and treat the JUnit tests as the specification.
- Programming Test 1 will have a similar format and submission workflow to Assignment 1, and Jackie stated that it will not cover graphs.

Roadmap

Assignment 1 and Programming Test 1 logistics

Jackie began by showing Assignment 1, which had been released earlier than scheduled to give students more preparation time. The assignment is due on Wednesday, June 17, is purely programming-based, and asks students to implement core self-balancing BST functionality such as insertion, height calculation, ancestor paths, and rotations. Jackie emphasized that the assignment format is similar to Programming Test 1, that Programming Test 1 will not cover graphs, and that students should not wait for solutions before making a serious attempt.

The starter project should be imported into Eclipse as an existing project from an archive file. The provided `BSTNode` class will also be given in the programming test and should not be modified. The empty `model` package is where students should put their implementation

work. The starter JUnit test class is the complete specification for the assignment, and the tests are organized to support gradual development from easier pieces to more advanced ones.

Tutorial 4 plan: rotations on paper

The tutorial then moved to paper exercises on rotations. The notes list three major pieces: abstracting the trinode restructuring step, doing a right-left restructuring after insertion, and understanding restructuring after deletion. Jackie explained that some exercises would be completed together and that other rotation exercises would be left for students to attempt with posted solutions or walkthroughs later.

Restoring balance in the abstract left-left case

The first technical exercise used an abstract tree with nodes A, B, and C, and subtrees T_1, T_2, T_3, T_4 all of height h. In the starting shape, A has left child B, and B has left child C. Students computed heights bottom-up: C has height $h+1$, B has height $h+2$, and A has height $h+3$. Since the two child heights at A differ by 2, node A violates the height-balance property.

This setup also illustrates why the repair is a single right rotation on the middle node B. Before rotation, the in-order traversal has the order

$$T_1, C, T_2, B, T_3, A, T_4.$$

After the right rotation, B becomes the local root, C becomes its left child, A becomes its right child, and the same in-order sequence is preserved. Jackie used this to show both parts of the repair: the sorting property remains valid and the height-balance property is restored.

Single versus double trinode restructuring

Jackie summarized the general trinode restructuring distinction. If A, B, and C slant the same way, such as left-left or right-right, the repair is a single rotation on the middle node B. If they slant differently, such as left-right or right-left, the repair is a double rotation on the lower node C.

For paper exercises, Jackie recommended thinking of a double rotation as a single reconstruction guided by the in-order traversal, rather than drawing every intermediate step. For implementation, however, students may either call two single-rotation helper methods in sequence or implement a direct restructuring method, provided the references and resulting tree are correct.

Right-left rotation after inserting 85

The main insertion example began from the sequence of keys

$$\langle 44, 17, 78, 32, 50, 88, 82, 95 \rangle$$

inserted into an empty BST. After drawing the balanced tree and annotating heights, students inserted key 85 as the right child of 82. Jackie highlighted the ancestor path of the inserted node and checked balance bottom-up. Node 78 is the lowest unbalanced node, so $A = 78$; the child on the insertion path is $B = 88$; and the next node on the path is $C = 82$.

Because the path goes right from A to B and left from B to C, this is a right-left case. The double rotation promotes $C = 82$ to become the local root. The left side of the rebuilt local subtree contains the part before 82 in the in-order traversal, and the right side contains the part after 82. Jackie emphasized that the Java implementation should not create new nodes for this; it should rearrange parent and child references, which is a constant-time local operation.

Restructuring after deletion

The final technical part shifted from insertion to deletion. Jackie pointed students to the deletion slides and highlighted that after deletion the rebalancing logic is different. If no ancestor becomes unbalanced, no rotation is needed. If an ancestor becomes unbalanced, choose A as the lowest unbalanced node. Then choose B as A's taller child. To choose C, inspect B's children: if their heights are different, choose the taller child; if their heights are the same, choose C so that A, B, C slant the same way.

The major difference is that deletion may require more than one trinode restructuring step. After one local rotation, students must continue checking upward along the ancestor path. Since the tree is balanced, the number of possible restructuring steps is $O(h) = O(\log n)$.

Deletion example: removing 80

The deletion example used the key sequence

$$\langle 50, 25, 10, 30, 5, 15, 27, 1, 75, 60, 80, 55 \rangle$$

inserted into an empty BST. After deleting 80, the first lowest unbalanced node is $A = 75$. Its taller child is $B = 60$, and B's taller child is $C = 55$. These nodes slant left-left, so the first repair is a single right rotation on the middle node $B = 60$. Jackie left the rest as an exercise: after that first rotation, continue upward to check whether another unbalanced node remains.

Key Terms, Notation, and Structures

- **Height-balance property**: At every internal node, the heights of the left and right children differ by at most 1.
- **Node height**: Computed bottom-up as $1 + \max(\text{height of left child}, \text{height of right child})$ for an internal node.
- **External node**: A placeholder child in the BST model; in the height calculations used here, external nodes have height 0.

- **Ancestor path**: The path from an inserted or affected node upward toward the root; balance checking after an update is driven by this path.
- **Lowest unbalanced node**: The first unbalanced node encountered when checking bottom-up from the affected node; this node is labeled A.
- **Trinode restructuring**: The rotation-based repair using nodes A, B, and C and the four surrounding subtrees T_1, T_2, T_3, T_4.
- **Single rotation**: A left or right rotation used when A, B, and C slant the same way.
- **Double rotation**: A left-right or right-left rotation used when A, B, and C form a zig-zag shape.
- **Middle node**: In a single rotation, the node B is promoted and becomes the local root after restructuring.
- **Lower node**: In a double rotation, the node C is promoted two levels and becomes the local root after restructuring.
- **In-order traversal invariant**: Rotations must preserve the same in-order traversal sequence, which is what preserves the BST sorting property.
- **Right rotation**: The single rotation used for a left-left shape; in the abstract example, it promotes B above A and C.
- **Right-left rotation**: A double rotation for a right-left shape; in the tutorial example, it promotes $C = 82$ after inserting 85.
- **Deletion rebalancing rule for B**: After deletion, choose B as the taller child of A, not merely the next node on the deletion path.
- **Deletion rebalancing rule for C**: Choose C as B's taller child; if the two child heights tie, choose C so that A, B, C slant the same way.
- $O(1)$ **local rotation**: A single local restructuring changes only a constant number of references in Java.
- $O(h) = O(\log n)$ **deletion restructuring**: Deletion can require continuing upward through the height of a balanced tree.
- `BSTNode`: The provided node class for Assignment 1; Jackie stated that students should not modify it.
- `model` **package**: The package where students should implement the required assignment logic.
- **JUnit tests**: The assignment tests act as the complete specification and should be followed in order as a development guide.
- **Java generics**: The assignment uses generic node/tree structures; students who are rusty should review the generics section and optional pre-study materials.

Key Examples and Demonstrations

Assignment 1 starter project and JUnit specification

Setup: Jackie imported the Assignment 1 starter project into Eclipse, showed the provided `BSTNode` class, the empty `model` package, and the JUnit test class.

Main point: The assignment is not just extra practice; it is a structured rehearsal for Programming Test 1. The provided tests describe what the code must do, and the intended workflow is to implement gradually until the tests pass.

What to practise: Import the starter project, run the JUnit tests before writing code, read the tests carefully, and then implement the required methods in the order suggested by the test structure.

Abstract left-left imbalance and right rotation

Setup: The notes use a tree with A above B above C, all slanting left, and subtrees T_1, T_2, T_3, T_4 all of height h.

Main point: Computing heights bottom-up shows that A is unbalanced. A right rotation on the middle node B restores the height-balance property while preserving the in-order traversal $T_1, C, T_2, B, T_3, A, T_4$.

What to practise: Recompute the heights without looking at the annotations, then redraw the rotated tree and verify both the in-order traversal and the child-height differences.

Right-left double rotation after inserting 85

Setup: Starting from the tree built from $\langle 44, 17, 78, 32, 50, 88, 82, 95\rangle$, insert 85 and update heights along the new node's ancestor path.

Main point: The lowest unbalanced node is $A = 78$, with $B = 88$ and $C = 82$. Since the shape is right-left, the repair is a double rotation on the lower node $C = 82$.

What to practise: Identify T_1, T_2, T_3, T_4 in the concrete tree, reconstruct the subtree with 82 as the local root, and then verify the final heights.

Deletion example: first restructuring after deleting 80

Setup: The notes build a larger BST from $\langle 50, 25, 10, 30, 5, 15, 27, 1, 75, 60, 80, 55\rangle$ and then delete 80.

Main point: After deletion, choose A as the lowest unbalanced node, then choose B and C by height. Here $A = 75$, $B = 60$, and $C = 55$, leading to a single right rotation as the first restructuring step.

What to practise: Finish the example from the notes: perform the right rotation, then continue checking upward to find whether another unbalanced node remains.

Common Pitfalls and Misconceptions

Modifying the provided `BSTNode` class

Pitfall: Changing `BSTNode` to make the assignment feel easier.

Why it causes problems: Jackie stated that this class is provided and should not be modified. Changing it can break the expected test environment and will not prepare you for Programming Test 1, where the same kind of provided class will be fixed.

How to avoid it: Put your logic in the required model class or helper methods indicated by the JUnit tests, and treat `BSTNode` as part of the fixed interface.

Waiting for solutions instead of practising

Pitfall: Looking briefly at the assignment, waiting for solutions, and hoping the solution will be enough preparation.

Why it causes problems: Assignment 1 is meant to expose where you struggle before Programming Test 1. Reading a solution after the fact does not build the tracing and coding habits needed under test conditions.

How to avoid it: Attempt the assignment early, record which methods or tests block you, and use any later solution only to repair specific misunderstandings.

Rotating when no rotation is needed

Pitfall: Assuming every insertion or deletion must trigger a rotation.

Why it causes problems: Rotations are only needed when the height-balance property is violated. If the ancestor path remains balanced, changing the tree structure can create errors instead of fixing anything.

How to avoid it: First compute or update heights and check balance along the relevant ancestor path; rotate only after identifying a lowest unbalanced node.

Forgetting that rotations must preserve in-order traversal

Pitfall: Treating rotations as arbitrary shape changes that only aim to make the tree look balanced.

Why it causes problems: A balanced-looking tree is still wrong if it violates the BST sorting property. The same in-order traversal before and after rotation is the key check.

How to avoid it: For every paper rotation, write or mentally check the in-order sequence. For code, think about how each child and parent reference preserves the same left-root-right order.

Confusing insertion and deletion rules for choosing B and C

Pitfall: Using the insertion-path rule for a deletion case.

Why it causes problems: After insertion, B and C come from the ancestor path of the inserted node. After deletion, the height loss means the repair is driven by taller subtrees, so B and C are chosen by height.

How to avoid it: Say the update type out loud before labeling A, B, C: after insertion, follow the insertion path; after deletion, choose the taller child, and then the taller child again unless there is a tie.

Assuming deletion always needs only one restructuring step

Pitfall: Stopping immediately after the first deletion rotation.

Why it causes problems: Deletion can reduce subtree height in a way that causes another imbalance higher up. Jackie emphasized that multiple trinode restructuring steps may be needed.

How to avoid it: After a deletion rotation, continue checking upward until the remaining ancestor path is balanced.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain the height-balance property and apply it by comparing child-node heights;
- compute heights bottom-up for abstract trees and concrete BST diagrams;
- identify when a node violates the height-balance property and label the lowest unbalanced node as A;
- explain why the in-order traversal sequence must be preserved by any BST rotation;
- perform a single right rotation for a left-left trinode shape and verify that the result is balanced;
- distinguish single rotations from double rotations using the slant pattern of A, B, C;
- perform a right-left double rotation on the insertion example where key 85 is inserted;
- identify the four surrounding subtrees T_1, T_2, T_3, T_4 and reattach them correctly after restructuring;
- state the different rules for choosing B and C after insertion versus after deletion;
- explain why insertion requires at most one trinode restructuring step, while deletion can require multiple steps;
- justify why deletion restructuring may take $O(h) = O(\log n)$ steps in a balanced tree;
- import Assignment 1, run the JUnit tests, and use the test order as a development plan;
- explain why `BSTNode` should not be modified and why the `model` package is where your implementation belongs;
- connect the paper rotation diagrams to Java reference updates involving parent, left, and right links.

Suggested Practice

- Start Assignment 1 early: import the starter project, run the tests, read the instructions, and begin with the first failing test cases.
- Review the generics section of the assignment instructions if `BSTNode<E>` and generic helper classes feel unfamiliar.
- Redo the abstract left-left example from page 2 of the notes: compute heights, identify the imbalance, perform the right rotation, and verify in-order traversal.
- Redo the right-left insertion example from page 5 of the notes without looking at the final diagram; focus on labeling $A = 78$, $B = 88$, and $C = 82$ correctly.
- Complete at least one other insertion rotation exercise from the Tutorial 4 instructions, especially a left-right or single-right case not fully worked in class.
- Finish the deletion example from page 9 of the notes after the first right rotation, then continue searching upward for any remaining imbalance.
- Practise deletion cases from the self-balancing BST slides even though deletion is not required in Assignment 1, because Jackie connected it to Programming Test 1 preparation.
- For each rotation you practise, write down the in-order sequence before and after the rotation to check that the BST property is preserved.
- When coding, map every paper-level child/parent change to the actual Java reference updates needed in your implementation.

Where to Review

- **Tutorial 4 notes, page 1**: overview of the tutorial focus: tree rotations, abstract trinode restructuring, right-left insertion restructuring, and deletion restructuring.
- **Tutorial 4 notes, page 2**: abstract left-left imbalance, right rotation on B, and the in-order traversal check $T_1, C, T_2, B, T_3, A, T_4$.
- **Tutorial 4 notes, page 3**: summary of single versus double rotations based on whether A, B, C slant the same way or differently.
- **Tutorial 4 notes, page 4**: right-left double rotation, including the two approaches: sequential single rotations or simultaneous reconstruction using in-order traversal.
- **Tutorial 4 notes, page 5**: the insertion of 85 and the right-left restructuring example.
- **Tutorial 4 notes, pages 6–7**: deletion rebalancing rules, including how to choose A, B, and C and why multiple restructuring steps may be needed.

- **Tutorial 4 notes, pages 8–9**: deletion restructuring exercises, including the partially worked multiple-rotation example after deleting 80.
- **Assignment 1 instructions and starter project**: review the generics explanation, requirements, submission instructions, provided `BSTNode`, empty `model` package, and JUnit tests.
- **Self-balancing BST slides**: revisit the rotation and deletion slides Jackie pointed to, especially the deletion and post-deletion rotation rules.
- **Posted tutorial solutions or walkthrough videos**: use them after attempting the paper exercises yourself, not as a substitute for first-pass practice.

Closing Reminder

This tutorial digest is only a roadmap for review. To learn the material, you still need to redraw the rotation diagrams, recompute heights, trace ancestor paths, finish the deletion example, and connect each paper restructuring step to Java reference updates in Assignment 1. Reading the digest or waiting for solutions will not be enough preparation for Programming Test 1; use the official notes, recording, starter code, JUnit tests, and your own practice to check that you can actually perform and implement the rotations.

EECS3101 Algorithms (Summer'26)

Digest of Lecture Tutorial 5, June 8

Induction Proofs: Proper Binary Trees and Spanning Trees

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 8

Suggested Practice 8

Where to Review 9

Closing Reminder 9

Main Ideas

- The tutorial focused on how to use mathematical induction for structured objects, first proper binary trees and then spanning trees.
- The first proof showed that for any non-empty proper binary tree, the number of external nodes is exactly one more than the number of internal nodes: $N_E = N_I + 1$.
- The key tree-proof idea was to choose a structure-preserving minimal extension: convert one external node into an internal node and add two new external children.
- The second proof showed that if a graph is a spanning tree, then its number of edges satisfies $m = n - 1$, where $n = |V|$ and $m = |E|$.
- The graph-proof idea was to extend a spanning tree by adding one new vertex and exactly one new edge, because zero edges breaks connectedness and two or more edges create a cycle.
- Jackie emphasized that implication direction matters: G being a spanning tree implies $m = n - 1$, but $m = n - 1$ alone does not imply that G is a spanning tree.
- Assignment 1 was briefly connected to reading one-line `BSTNode` construction expressions in JUnit-style tests; a short guide was to be posted for students who need help parsing those expressions.

Roadmap

Tutorial agenda and where this fits

Jackie began by locating Tutorial 5 under the graph category on the lecture site and explaining that Tutorials 5 through 8 would continue graph-related work, while Tutorial 9 would be for dynamic programming. The main goal of this tutorial was not to introduce a new algorithm, but to practise proof structure: one induction proof on trees and one induction proof on graphs. Jackie also noted that, time permitting, Assignment 1's one-line tree construction style would be discussed because it appears in the starter tests and is useful for later programming tests.

Warm-up: induction pattern

The tutorial reused the induction template from the earlier graph lecture: check small cases, state an inductive hypothesis, and then prove a strictly larger case by making the smallest valid extension. Jackie stressed that the extension must preserve the defining structure. For graphs this may mean adding one vertex; for proper binary trees, adding just one node is not valid because it can break properness.

Proper binary trees and the target property

The first problem concerned a non-empty proper binary tree. Proper means that every internal node has exactly two children; every node therefore has either zero children or two children. The target property was written using N_I for the number of internal nodes and N_E for the number of external nodes:

$$N_E = N_I + 1.$$

Jackie first checked the property on a concrete tree in the notes, where counting gave $N_I = 5$ and $N_E = 6$. This example was used only to build intuition before the proof.

Tree proof: base cases and inductive hypothesis

The smallest non-empty proper binary tree has one external node and no internal nodes, so $N_E = 1$ and $N_I = 0$, and the property holds. Jackie also checked the next smallest proper binary tree: one internal root with two external children, so $N_E = 2$ and $N_I = 1$. The inductive hypothesis then assumed that the property holds for a proper binary tree larger than these base cases.

Tree proof: the structure-preserving extension

For the inductive step, Jackie used the smallest valid way to make a larger proper binary tree: choose an external node, convert it into an internal node, and attach two new external children. This increases the total size by two, not one. The before/after count is the core of the proof: $N'_I = N_I + 1$, because the chosen external node becomes internal, and $N'_E = N_E - 1 + 2 = N_E + 1$, because one external node is lost and two new ones are gained. Substituting the inductive hypothesis $N_E = N_I + 1$ gives $N'_E = N'_I + 1$, which completes the step.

Translating the spanning-tree statement

The second problem was the first property from the graph-property slide: the number of edges is the number of vertices minus one if the graph is a spanning tree. Jackie paused to translate this English sentence into a logical implication. The direction is

$$G \text{ is a spanning tree} \Rightarrow m = n - 1.$$

This is not the same as the converse. A graph can have $n - 1$ edges and still fail to be a spanning tree if the chosen edges create a cycle or leave a vertex disconnected.

Reviewing what a spanning tree must preserve

Before proving the property, Jackie reviewed the defining properties of a spanning tree: it is undirected, it is a spanning subgraph, it is connected, and it is acyclic. These are not decorative words; they control what a valid inductive extension may do. Any extension used in the proof must still cover the relevant vertices, remain undirected, stay connected, and avoid cycles.

Spanning-tree proof: base cases

The proof was by induction on the number of vertices. The empty case was not used because the formula would give $0 = 0 - 1$, which is false. Jackie then checked small spanning trees: for $n = 1$, $m = 0$; for $n = 2$, $m = 1$; for $n = 3$, $m = 2$. These examples support the pattern $m = n - 1$ and also show why the graph has to be connected and acyclic.

Spanning-tree proof: exactly one new edge

For the inductive step, Jackie drew the old spanning tree as a cloud with n vertices and $m = n - 1$ edges. To make a strictly larger spanning tree, add one new vertex x. The new vertex cannot be left isolated, because the result would not be connected. It also cannot be connected to two or more existing vertices, because the old tree already has a path between those existing vertices; adding two edges through x would create a cycle. Therefore the valid minimal extension adds exactly one new edge.

Spanning-tree proof: completing the count

After the extension, the new graph has $|V'| = n + 1$ vertices and $|E'| = |E| + 1$ edges. By the inductive hypothesis, $|E| = n - 1$, so $|E'| = (n - 1) + 1 = n$. Since $|V'| - 1 = (n + 1) - 1 = n$, the larger graph also satisfies $|E'| = |V'| - 1$. Jackie emphasized that this proof works because the inductive hypothesis is explicitly used.

Assignment 1 connection

Near the end, Jackie pointed to the Assignment 1 starter tests and the large one-line expression used to construct a binary tree with nested `BSTNode` constructor calls. The tutorial did not fully walk through that code, but Jackie noted that students who find it hard to parse should review the short guide to be posted before the next lecture. The point is to be able to read the tree shape encoded by nested constructor calls, not merely to copy the expression.

Key Terms, Notation, and Structures

- **Mathematical induction**: A proof method with base cases, an inductive hypothesis, and an inductive step that proves a strictly larger case by using the hypothesis.
- **Structure-preserving extension**: The operation used in the inductive step must keep the object inside the same category, such as proper binary tree or spanning tree.
- **Proper binary tree**: A non-empty binary tree in which every internal node has exactly two children.
- **Internal node**: A node with children; in a proper binary tree, it has exactly two children.
- **External node**: A leaf node with zero children.
- N_I: The number of internal nodes in a binary tree before extension.

- N_E: The number of external nodes in a binary tree before extension.
- N_I', N_E': The corresponding internal-node and external-node counts after an extension.
- **Proper binary tree property**: For any non-empty proper binary tree, $N_E = N_I + 1$.
- **Spanning tree**: An undirected, connected, acyclic spanning subgraph.
- $n = |V|$: The number of vertices in a graph.
- $m = |E|$: The number of edges in a graph.
- **Spanning-tree edge property**: If G is a spanning tree, then $m = n - 1$, equivalently $|E| = |V| - 1$.
- **Implication direction**: The statement proved was G is a spanning tree $\Rightarrow m = n - 1$; the reverse direction was explicitly not being proved.
- **Cloud diagram**: Jackie's abstraction for an already-valid tree or graph instance whose exact internal shape is not important, only its properties and counts.
- `BSTNode`: The Assignment 1 node class used in nested constructor expressions for building binary trees in starter tests.

Key Examples and Demonstrations

Checking the proper binary tree property on a concrete tree

Setup: The notes showed a proper binary tree and counted internal and external nodes.

Main point: The example had $N_I = 5$ and $N_E = 6$, matching $N_E = N_I + 1$. This does not prove the property, but it gives the right target before writing the induction proof.

What to practise: Redraw the example tree from the notes, count internal and external nodes without looking at the answer, and then explain why every internal node satisfies the proper-tree requirement.

Induction proof for $N_E = N_I + 1$

Setup: Start with a non-empty proper binary tree and prove the relationship between external and internal nodes.

Main point: The inductive step must increase the tree by two nodes, not one. Converting an external node n into an internal node and adding two external children changes the counts to $N_I' = N_I + 1$ and $N_E' = N_E + 1$, which preserves $N_E = N_I + 1$ after using the inductive hypothesis.

What to practise: Reproduce the proof from the notes with the before/after variables. Then try to prove the equivalent statement from the tutorial prompt about the total number of nodes being odd.

Formulating the spanning-tree property

Setup: The graph-property slide stated that the number of edges is the number of vertices minus one if the graph is a spanning tree.

Main point: The correct implication is G is a spanning tree $\Rightarrow m = n-1$. Jackie separated this from the false converse and reminded students to pay attention to "if", "only if", and "if and only if".

What to practise: Write the statement in both English and symbolic form. Then draw a small graph with n vertices and $n-1$ edges that is not a spanning tree because it is disconnected or cyclic.

Induction proof for spanning trees

Setup: Assume an existing spanning tree has n vertices and, by the inductive hypothesis, $n-1$ edges. Add a new vertex x.

Main point: A valid extension to a larger spanning tree must add exactly one edge from x to the old tree. Zero edges makes x disconnected. Two or more edges create a cycle because the old tree already has a path between any two existing vertices.

What to practise: Redraw the cloud diagram from the notes. Explain why one new edge preserves connectedness and acyclicity, then redo the final count $|E'| = |V'| - 1$ without looking.

Assignment 1 one-line `BSTNode` construction

Setup: The notes showed the `BSTNode` constructors and a starter-test expression with deeply nested constructor calls.

Main point: The expression encodes a tree recursively: each internal-node constructor contains a key/value and two child subtrees, while each external-node constructor marks a leaf placeholder. Reading this expression is part of understanding the test cases.

What to practise: Take a small nested `BSTNode` expression and draw the corresponding tree. Then reverse the exercise: draw a small tree and write the corresponding constructor expression.

Common Pitfalls and Misconceptions

Extending a proper binary tree by only one node

Pitfall: Adding a single child to an external node and treating the result as a proper binary tree.

Why it causes problems: The resulting internal node would have one child, violating the definition of proper binary tree.

How to avoid it: In the inductive step for proper binary trees, convert an external node into an internal node by adding exactly two external children.

Using node names as if they were numeric counts

Pitfall: Writing an expression such as subtracting node n or adding nodes x, y as if the symbols themselves were numbers.

Why it causes problems: Nodes are objects or positions; counts must change by numeric amounts such as -1, $+1$, or $+2$.

How to avoid it: Use count variables such as N_E, N_I, N'_E, and N'_I, and explain the numerical change: lose one external node, gain two external nodes.

Forgetting to use the inductive hypothesis

Pitfall: Completing an inductive step without ever substituting or invoking the assumed property for the smaller case.

Why it causes problems: The proof no longer connects the larger case to the smaller case, so it is not an induction proof.

How to avoid it: In each inductive step, identify the exact line where the inductive hypothesis is used, such as substituting $N_E = N_I + 1$ or $|E| = |V| - 1$.

Reversing the implication for spanning trees

Pitfall: Concluding that any graph with $m = n - 1$ must be a spanning tree.

Why it causes problems: The edge count alone does not guarantee connectedness or acyclicity. Jackie explicitly treated the converse as not being proved and generally false.

How to avoid it: Remember the proven direction: spanning tree $\Rightarrow m = n-1$. To identify a spanning tree, also check spanning, connectedness, acyclicity, and undirectedness.

Adding too many edges in the spanning-tree induction step

Pitfall: Adding a new vertex to the old spanning tree and connecting it to two or more old vertices.

Why it causes problems: The old tree is connected, so there is already a path between those old vertices; the two new edges through the new vertex create a cycle.

How to avoid it: For the spanning-tree proof, add exactly one edge from the new vertex to the existing tree.

Treating nested constructor code as unreadable text

Pitfall: Skipping the large one-line `BSTNode` expression because it looks too long.

Why it causes problems: The starter tests specify tree shapes using those expressions, so misunderstanding them can lead to misunderstanding what the test expects.

How to avoid it: Parse the expression recursively: identify the root constructor first, then the left subtree, then the right subtree, and draw the tree as you go.

What You Should Be Able to Do

After this tutorial, you should be able to:

- explain what makes a binary tree proper and distinguish internal nodes from external nodes;
- count N_I and N_E on a concrete proper binary tree and check whether $N_E = N_I + 1$ holds;
- reproduce the induction proof that every non-empty proper binary tree satisfies $N_E = N_I + 1$;
- explain why the proper-binary-tree inductive step must add two nodes rather than one;
- translate "$m = n - 1$ if G is a spanning tree" into the implication G is a spanning tree $\Rightarrow$ $m = n - 1$;
- list the properties that define a spanning tree: undirected, spanning, connected, and acyclic;
- justify why extending a spanning tree by one new vertex requires exactly one new edge;
- reproduce the induction proof that a spanning tree with n vertices has $n - 1$ edges;
- give a counterexample showing that $m = n - 1$ alone does not guarantee a spanning tree;
- read a nested `BSTNode` constructor expression well enough to draw the intended binary tree.

Suggested Practice

- Redraw the proper binary tree example from the notes and recount N_I and N_E before reviewing the proof.
- Reproduce the proof of $N_E = N_I + 1$ on paper, making the two before/after equations $N'_I = N_I + 1$ and $N'_E = N_E + 1$ explicit.
- Try the tutorial prompt's equivalent claim that the total number of nodes in a non-empty proper binary tree is odd, using the same induction idea.
- Redraw the spanning-tree cloud diagram and explain in words why zero new edges and two new edges are both invalid extensions.
- Reproduce the proof that a spanning tree satisfies $|E| = |V| - 1$, and mark the exact line where the inductive hypothesis is used.
- Complete the graph taxonomy table from the previous lecture before the next class, since Jackie planned to continue from it.

- For Assignment 1, take the one-line `BSTNode` expression from the starter tests and draw the tree it constructs.
- Create a small false-converse example: a graph with n vertices and $n-1$ edges that is not a spanning tree, and explain which defining property fails.

Where to Review

- Review the Tutorial 5 iPad/PDF notes page 1 for the tutorial scope: mathematical induction, Assignment 1, induction on trees and graphs, and the one-line BST construction topic.
- Review page 2 for the graph property slide and the annotated proof sketch for spanning trees, including implication direction, base cases, and the before/after edge count.
- Review page 3 for the cloud diagram showing why adding two edges from a new vertex creates a cycle and is not a valid spanning-tree extension.
- Review page 5 for the proper binary tree proof, especially the conversion of an external node into an internal node with two new external children.
- Review page 6 for the `BSTNode` constructors and the starter-test one-line construction style for Assignment 1.
- Revisit Lecture 9's mathematical induction template and Lecture 10's definitions of spanning subgraph, connected graph, forest, tree, and spanning tree.
- Review the math review slide on implication language if "if", "only if", and "if and only if" are still confusing.

Closing Reminder

This tutorial digest is a roadmap for the two induction proofs and the Assignment 1 code-reading connection. It is not enough to read the final equations: you need to redo the proofs, justify the extension steps, and practise converting between diagrams, formulas, and code. Revisit Jackie's notes and recording, complete the taxonomy table, and work through the Assignment 1 tree-construction examples actively before relying on the digest for review.

EECS3101 Algorithms (Summer'26)

Digest of Tutorial 6, June 15

Tracing DFS on a Directed Graph: Colours, Time Stamps, Edge Types

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 10

Closing Reminder 10

Main Ideas

- Tutorial 6 was a practice-focused directed-graph DFS trace, continuing the directed DFS notation from Lecture 12.
- The main skill was to trace DFS from a specified start vertex while simultaneously tracking the stack, vertex colors, discovery times, finish times, the DFS tree, and edge classifications.
- The example used a directed graph with vertices A, B, C, D, E, F, $|V| = 6$, $|E| = 11$, start vertex A, and alphabetical order for choosing outgoing edges.
- Before tracing, Jackie emphasized graph visualization: draw the directed graph from V and E, check every vertex and edge, and avoid a messy layout that makes arrows difficult to follow.
- The trace illustrated the expected counts: a single DFS tree on six reachable vertices has $|V| - 1 = 5$ discovery edges, the remaining six edges are non-tree edges, and the final timestamp is $2 \cdot |V| = 12$.
- The tutorial reinforced that only push and pop events increment the global timer t; classifying a back, forward, or cross edge does not change the stack or the timer.
- Jackie connected this skill to assessment preparation: similar edge-classification and tracing tasks may appear on tests or exams, and Programming Test 1 is on June 22 after reading week.

Roadmap

Reviewing the directed DFS notation from Lecture 12

Jackie began by reminding students that this tutorial continues the directed DFS material from Thursday. The relevant background is the notation page from the directed DFS lecture: discovery time d, finish time f, vertex colors white/grey/black, DFS trees, and edge categories. A specific notation overloading warning was repeated: the DFS d for discovery time is not the same as the d used earlier in the context of self-balancing BSTs.

Course logistics before the trace

Jackie reminded students that reading week starts on Tuesday, Assignment 1 is due on Wednesday, and Programming Test 1 is on Monday, June 22. Students were told to read the posted programming-test guide and follow Jackie's later reminders about room-entry procedure and test expectations. Jackie also mentioned a Wednesday 1:00 p.m. Zoom office hour and explained that Assignment 2 would involve DFS on an undirected graph, while this tutorial gives one more directed DFS practice example.

Reading a directed graph from V and E

The tutorial exercise did not begin with only a finished drawing. It specified the directed graph by a vertex set and an edge set, so the first task was to visualize the graph. Jackie emphasized that this is an assessment-relevant skill: before starting a trace, verify that all vertices are present, every directed edge has the correct direction, and no extra edge has been inserted. The specific graph had

$$V = \{A, B, C, D, E, F\}$$

and directed edges are

$$\{(A,B),(A,C),(A,D),(B,C),(C,A),(D,E),(D,F),(E,A),(E,F),(F,B),(F,C)\}$$

The pair (A, C) and (C, A) was discussed as allowed in a simple directed graph: they are opposite-directed (bi-directional) edges, not a self-loop and not parallel copies of the same directed edge.

Pre-trace checks: counts and expectations

Before performing the trace, Jackie used the graph size to predict what the completed trace should look like. Since $|V| = 6$, a DFS tree covering all six reachable vertices should have $6 - 1 = 5$ discovery edges. Since $|E| = 11$, the remaining $11 - 5 = 6$ edges must be non-tree edges, classified as back, forward, or cross edges. Since each vertex is pushed once and popped once, the timer should end at $2 \cdot |V| = 12$, with $t = 0$ reserved for the initial all-white state. These counts are not just facts to memorize; they are error checks during tracing.

Setting up the trace tables

Jackie set up several records at once: a push/pop table, a table of $d(v)$ and $f(v)$, a time-indexed color table for white/grey/black vertices, a stack, and a DFS tree diagram. The start vertex was A, so A becomes the root of the DFS tree. The initial row $t = 0$ has all vertices white, the stack empty, and no discovery or finish times defined. Jackie also posed review questions for each color: whether d is defined, whether f is defined, and whether the vertex is currently in the stack.

Beginning the trace: A, B, and C

The trace starts by pushing A, making A grey, and setting $d(A) = 1$. From A, the outgoing edges are processed in alphabetical order, so $A \to B$ is the first discovery edge. Then $B \to C$ is the next discovery edge. At this point the DFS tree contains $A \to B \to C$, the stack top is C, and $d(A) = 1$, $d(B) = 2$, $d(C) = 3$. Jackie stressed that the current decision point is always the top of the stack, not an older vertex lower in the stack.

First non-tree edge: $C \to A$ as a back edge

From C, the only outgoing edge to consider is $C \to A$. Since A is an active ancestor still in the stack, this edge is classified as a back edge. This classification happens while $t = 3$, and because it is not a discovery edge, the timer and stack do not change. Jackie highlighted this as a likely test-style question: students may be asked not only what type of edge it is, but also at what time t it was identified.

Backtracking from C and B

After $C \to A$ is classified, C has no remaining outgoing edge to process, so C is popped, colored black, and assigned $f(C) = 4$. Then B also has no remaining outgoing edge, so B is popped, colored black, and assigned $f(B) = 5$. This part of the trace illustrated the difference between edge-identification labels and timestamps: edge labels record the chronological order of edge consideration, while d and f record push/pop times.

Returning to A: forward edge then new branch

With A again at the top of the stack, Jackie considered the remaining outgoing edges from A in order. The edge $A \to C$ is a forward edge because C is already a finished descendant of A in the DFS tree; this is identified while $t = 5$, with no push or pop. Then $A \to D$ discovers a new vertex, so D is pushed, colored grey, and assigned $d(D) = 6$. Jackie also pointed out that D should be attached as a child of A in the DFS tree, because A is the vertex from which D was discovered.

Exploring the D, E, F branch

From D, alphabetical order chooses $D \to E$ before $D \to F$, so E is discovered with $d(E) = 7$. From E, alphabetical order considers $E \to A$ before $E \to F$, so $E \to A$ is a second back edge: it points to the active ancestor A and is identified while $t = 7$. Then $E \to F$ discovers F, assigning $d(F) = 8$. At this point all vertices have been discovered, but the trace is not finished because several outgoing edges and finish times still need to be handled.

Cross edges from F, then finishing the branch

From F, the outgoing edges $F \to B$ and $F \to C$ are processed alphabetically and classified as cross edges. In the DFS tree, F lies on the branch A – D – E – F, while B and C lie on a different branch under A. Neither target is an ancestor or descendant of F. Both cross edges are identified while $t = 8$, and neither changes the stack. After that, F is popped with $f(F) = 9$, and E is popped with $f(E) = 10$.

Final forward edge and completing the trace

After returning to D, the remaining edge $D \to F$ is a forward edge because F is a finished descendant of D. Jackie noted this as analogous to $A \to C$. Then D is popped with $f(D) = 11$,

and finally A is popped with $f(A) = 12$. The completed trace matches the earlier predictions: six pushes, six pops, five discovery edges, six non-tree edges, and final timestamp 12.

Key Terms, Notation, and Structures

- **DFS run**: One execution of depth-first search starting from a designated start vertex.
- **Start vertex**: The vertex where the DFS run begins; in this tutorial the start vertex is A, which becomes the root of the DFS tree.
- **DFS tree**: The tree formed by the discovery edges in one DFS run; in this example it spans all six vertices reached from A.
- **Discovery edge / tree edge**: An edge that leads to a white vertex and expands the DFS tree.
- **Back edge**: A non-tree edge pointing to an active ancestor on the stack, such as $C \to A$ or $E \to A$ in this trace.
- **Forward edge**: A non-tree edge pointing to a finished descendant in the DFS tree, such as $A \to C$ or $D \to F$.
- **Cross edge**: A non-tree edge between vertices with no ancestor/descendant relation in the DFS tree, such as $F \to B$ and $F \to C$ in this trace.
- **White vertex**: A vertex not yet discovered; its discovery and finish times are not yet defined, and it is not in the stack.
- **Grey vertex**: A discovered but unfinished vertex; its discovery time is defined, its finish time is not yet defined, and it is in the stack.
- **Black vertex**: A finished vertex; both discovery and finish times are defined, and it has been popped from the stack.
- t: The global DFS timer; in this tutorial $t = 0$ is the initial state and the first discovery event has time 1.
- $d(v)$ or $d(v)$: The discovery time of vertex v, assigned when v is pushed and turns grey.
- $f(v)$ or $f(v)$: The finish time of vertex v, assigned when v is popped and turns black.
- **Stack discipline**: At each step, decide what to do from the top of the stack; do not jump back to an older vertex unless backtracking has actually popped vertices above it.
- **Alphabetical adjacency order**: The visiting rule used in the tutorial; if several outgoing edges are available, consider their destination vertices alphabetically unless Jackie specifies a different order.
- **Simple directed graph with bidirectional edges**: Opposite directed edges such as (A, C) and (C, A) are allowed because they are not self-loops and not duplicate parallel copies of the same directed edge.

Key Examples and Demonstrations

Drawing the graph from vertex and edge sets

Setup: The tutorial exercise provided $V = \{A, B, C, D, E, F\}$ and a set of eleven directed edges rather than asking students to trace from a pre-digested answer.

Main point: Before any DFS trace, the graph itself must be correct. Jackie emphasized checking every vertex, every directed edge, and every arrow direction. A wrong diagram makes the entire trace unreliable.

What to practise: Recreate the graph from the listed V and E without looking at the final iPad diagram, then check it edge by edge against the notes before starting DFS.

Precomputing trace checks from $|V|$ and $|E|$

Setup: The graph has $|V| = 6$ and $|E| = 11$, and the DFS starts at A and reaches all vertices.

Main point: A completed single DFS tree on six vertices should contain 5 discovery edges. Since there are eleven total edges, six edges remain to be classified as non-tree edges. Since every vertex is pushed and popped once, the final timer should be 12.

What to practise: Before retracing the example, write down the expected values 5, 6, and 12, then use them to check whether your trace missed an edge, a push, a pop, or a timestamp.

The full directed DFS trace from A

Setup: Start at A, use alphabetical order for outgoing edges, and maintain the stack, colors, discovery/finish times, DFS tree, and edge labels.

Main point: The trace produces discovery times

$$d(A) = 1, d(B) = 2, d(C) = 3, d(D) = 6, d(E) = 7, d(F) = 8.$$

The finish times are

$$f(A) = 12, f(B) = 5, f(C) = 4, f(D) = 11, f(E) = 10, f(F) = 9.$$

The discovery edges are $A \to B$, $B \to C$, $A \to D$, $D \to E$, and $E \to F$. The non-tree edges are $C \to A$ and $E \to A$ as back edges, $A \to C$ and $D \to F$ as forward edges, and $F \to B$ and $F \to C$ as cross edges.

What to practise: Do not merely copy the final table. Reproduce the trace step by step and say, at each edge, whether it pushes a new vertex or only classifies a non-tree edge.

Classifying edges at a specific time

Setup: Jackie highlighted that a test question may ask when an edge is identified, not only what type it is.

Main point: In the trace, $C \to A$ is identified as a back edge when $t = 3$; $A \to C$ is identified as a forward edge when $t = 5$; $E \to A$ is identified as a back edge when $t = 7$;

$F \to B$ and $F \to C$ are identified as cross edges when $t = 8$; and $D \to F$ is identified as a forward edge after E and F have finished.

What to practise: For each non-tree edge, write the current stack, current t, target color, and tree relationship. This forces you to justify the label rather than guess from the final drawing.

Review questions on vertex colors

Setup: The final notes page asks, for each color, whether d is defined, whether f is defined, and whether the vertex is in the stack.

Main point: The color table is not decoration. White, grey, and black encode the vertex's exact DFS status: undiscovered, active, or finished.

What to practise: Fill in the answers for white, grey, and black vertices, then verify them against the trace at several different time rows.

Common Pitfalls and Misconceptions

Starting the trace before checking the graph drawing

Pitfall: Reading V and E quickly, drawing an incomplete or wrong directed graph, and then tracing from that drawing.

Why it causes problems: A missing arrow, reversed arrow, or extra edge changes the stack evolution, edge classifications, and timestamps.

How to avoid it: Before tracing, check the diagram against V and E exactly: all six vertices, all eleven directed edges, correct directions, no extras.

Confusing the initial state $t = 0$ with a discovery time

Pitfall: Assigning $d(A) = 0$ because the table begins at $t = 0$.

Why it causes problems: The tutorial uses $t = 0$ for the all-white initial state. The first actual event is pushing A, so $d(A) = 1$.

How to avoid it: Treat $t = 0$ as setup only. Start discovery and finish times at 1, and check that the final time is $2 \cdot |V|$.

Changing focus after a non-tree edge

Pitfall: After identifying $C \to A$ as a back edge, jumping to A's outgoing edges because the edge points to A.

Why it causes problems: Non-tree edges do not push, pop, or change the current stack top. Jumping focus breaks DFS order and produces wrong finish times.

How to avoid it: Always ask, "What is currently on top of the stack?" Continue processing that vertex until all of its outgoing edges are handled, then pop it.

Popping a vertex just because an edge points to it

Pitfall: Seeing $E \to A$ and trying to pop A, or assuming A is finished because it was discovered earlier.

Why it causes problems: A vertex can be active for a long time. A stays on the stack until all branches below it have been completed.

How to avoid it: Pop only the stack-top vertex, and only after all of its outgoing edges have been processed.

Treating edge labels as timestamps

Pitfall: Mixing up the chronological edge-identification numbers with d, f, or t.

Why it causes problems: Edge labels count the order in which edges are considered. The timer increments only when vertices are pushed or popped.

How to avoid it: Keep separate columns or annotations: one for edge identification order, one for the global time t, and one for $d(v)/f(v)$.

Misclassifying forward and cross edges

Pitfall: Calling every edge to a black vertex a forward edge, or treating all non-tree edges as back edges.

Why it causes problems: Directed DFS has more categories than undirected DFS. A black descendant gives a forward edge, while a black vertex in another branch gives a cross edge.

How to avoid it: Use the DFS tree: descendant already finished means forward; different branch with no ancestor/descendant relation means cross.

Forgetting the specified adjacency order

Pitfall: Choosing $D \to F$ before $D \to E$, or choosing $E \to F$ before $E \to A$, despite the alphabetical-order assumption.

Why it causes problems: Different visitation orders can produce different traces, so tests must specify an order and students must follow it exactly.

How to avoid it: List each vertex's outgoing neighbors in the required order before tracing, then mark them off as they are considered.

What You Should Be Able to Do

After this tutorial, you should be able to:

- draw a directed graph correctly from a given set of vertices and directed edges;
- explain why opposite-directed (bi-directional) edges such as (A, C) and (C, A) do not by themselves violate the simple-graph assumption;
- determine the start vertex and root of a DFS tree for a specified DFS run;

- precompute expected trace checks such as $|V| - 1$ discovery edges and final timestamp $2 \cdot |V|$;
- maintain a stack-based directed DFS trace using the top-of-stack rule;
- update vertex colors correctly from white to grey to black;
- assign $d(v)$ when a vertex is pushed and $f(v)$ when a vertex is popped;
- distinguish edge-identification order from timestamp values;
- construct the DFS tree from discovery edges only;
- classify directed edges as discovery, back, forward, or cross using color, stack status, and ancestor/descendant relationships;
- reproduce the main Tutorial 6 trace from A and justify why $C \to A$, $A \to C$, $E \to A$, $F \to B$, $F \to C$, and $D \to F$ receive their classifications;
- explain why a non-connected graph may require more than one DFS run to visit all vertices and edges;
- use the posted Programming Test 1 guide and Jackie's office-hour support to prepare for the June 22 test.

Suggested Practice

- Review Lecture 12's directed DFS notation before redoing this tutorial trace; especially review colors, d, f, and the four directed edge types.
- Redraw the Tutorial 6 graph from the given V and E sets without looking at the final diagram, then check all eleven edges.
- Recreate the full trace table with rows $t = 0, 1, \ldots, 12$, including white/grey/black sets, stack contents, push/pop order, and d/f values.
- For each edge in the graph, write its edge-identification order and type; then separately write the value of t when each non-tree edge is identified.
- Answer the page-5 review questions for each color: whether d is defined, whether f is defined, and whether the vertex is in the stack.
- Redo the trace with a 10- to 15-minute timer after you have understood it once; the first goal is correctness, then speed and tidiness.
- Make a small variant by changing the start vertex, or by changing the adjacency order, such as reverse alphabetical order, and observe which parts of the trace change.
- Read the Programming Test 1 guide, prepare questions during reading week, and use the Wednesday Zoom office hour if any assignment or test-preparation issue remains unclear.

Where to Review

- Review the Tutorial 6 iPad/PDF notes page 1 for the topic: DFS run on a directed graph.
- Review page 2 for the vertex and edge sets, the directed graph drawing, the start vertex A, the counts $|V| = 6$, $|E| = 11$, five discovery edges, six non-tree edges, and final timestamp 12.
- Review page 3 for the reminder that more than one DFS run may be needed when a graph is not connected; Jackie will revisit the directed/undirected details after reading week.
- Review page 4 for the completed trace: stack push/pop order, d/f table, white/grey/black table, DFS tree, and the classification of back, forward, and cross edges.
- Review page 5 for the vertex-color review questions; use them as a compact self-test before attempting another DFS trace.
- Rewatch the Tutorial 6 recording around the parts where Jackie explains why $C \to A$ is back, why $A \to C$ is forward, why $F \to B$ and $F \to C$ are cross, and why non-tree edges do not change the stack.
- Revisit Lecture 12 slides 31–32 and the Lecture 12 digest for the directed DFS terminology that this tutorial assumes.
- Check the posted Programming Test 1 guide and Jackie's course announcements for test logistics, allowed procedures, and reminders.

Closing Reminder

This tutorial digest is a roadmap for a directed DFS trace, not a substitute for doing the trace yourself. The skill only becomes reliable when you draw the graph, maintain the stack, update colors and timestamps, classify each edge, and check your result against the expected counts. Revisit Jackie's notes and recording, redo the Tutorial 6 table actively, and use reading week to turn this example into a repeatable tracing method before Programming Test 1.

EECS3101 Algorithms (Summer'26)

Digest of Tutorial 7, July 2

Iterative Algorithms: Precondition, Postcondition, Loop Invariants

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 3

Key Examples and Demonstrations 4

Common Pitfalls and Misconceptions 5

What You Should Be Able to Do 6

Suggested Practice 6

Where to Review 7

Closing Reminder 7

Main Ideas

- Iterative algorithms can be specified and reasoned about using three contracts: a precondition Q, a postcondition R, and a loop invariant I.
- The loop invariant must be *established* after initialization and *preserved* after every loop iteration; it is the key bridge from the loop's intermediate state to the final result.
- When a loop exits, its correctness argument must show that $\neg B \wedge I \Rightarrow R$, where B is the stay condition of the loop.
- The `findMax` example showed how to write a complete postcondition: the result must dominate every array element and must itself be an array element.
- A plausible-looking invariant can still be wrong. Tracing the proposed invariant on $[20, 10, 40, 30]$ exposed a violation caused by incrementing the loop counter before the invariant is checked.
- Tutorial 7 prepares you to reason about the loop invariant for Dijkstra's algorithm in the next lecture. Assignment 2 is due July 8; the Written Test is July 13 and Programming Test 2 is July 20.

Roadmap

Course timeline and the purpose of this tutorial

Jackie situated the tutorial between the introduction to Dijkstra's algorithm and the upcoming graph material. The immediate technical goal was not another graph trace: it was to build the correctness language needed for iterative algorithms, especially for Dijkstra next week. Continue working on Assignment 2, which is due July 8, and use the test-free week to prepare for the Written Test on July 13 and Programming Test 2 on July 20.

The contract structure of an iterative algorithm

The notes introduced the general pattern $Q\ \{S_{init};\ \texttt{while}(B)\ \{S_{body}\}\}\ R$. Here, Q is the input precondition, S_{init} prepares the state for iteration, B is the boolean stay condition, and R is the promised input–output relation after termination. The additional contract I, the loop invariant, is checked after initialization and after each execution of the loop body.

How the flowchart locates correctness failures

The flowchart separates several ways an execution can fail. Illegal input violates Q before the algorithm begins. After initialization, I must already be true; otherwise it was not established. At the end of every iteration, I must still be true; otherwise it was not preserved. When B becomes false, the loop exits and the final state must satisfy R. The intended safe state has $\neg B$, I, and R all true.

A small trace: `testLI`

The counter example used precondition $i = 1$, invariant $1 \le i \le 6$, stay condition $i \le 5$, loop body `i = i + 1`, and postcondition $i = 6$. It illustrates two important habits: the invariant remains true even after the final iteration, and a postcondition may be false during intermediate iterations without being a postcondition violation. The postcondition is checked only after exit.

Deriving public contracts for `findMax`

For a non-null, non-empty array `a`, `findMax` returns `result`. Jackie first developed the requirement that `result` is at least every array entry, then showed why that alone is insufficient: a faulty implementation could return an arbitrary value larger than every element. A complete postcondition must also require that `result` occurs in the input array. Express array properties by quantifying over valid indices, not by treating an array as a mathematical set.

Tracing a candidate loop invariant

The proposed invariant said that `result` is at least every `a[j]` for $0 \le j \le i$. On $[20, 10, 40, 30]$, it is established initially and survives the first iteration, but fails after the second iteration: `i` becomes 2 while `result` remains 20, so the condition wrongly includes `a[2]`$= 40$ before that entry has been processed. The point is to trace a candidate invariant carefully rather than trust its wording. The follow-up tutorial solution gives the corrected invariant and the formal correctness argument.

Key Terms, Notation, and Structures

- **Precondition Q:** A boolean condition on valid input values that must hold before the algorithm begins.
- **Postcondition R:** A boolean input–output relation promised when the algorithm terminates normally.
- **Loop invariant I:** A boolean correctness property that must hold after initialization and after every completed loop iteration.
- **Initialization S_{init}:** Statements before the loop that prepare the first iteration and must establish I.
- **Stay condition B:** The boolean condition in `while (B)`. When B is true, another iteration runs; when $\neg B$ is true, the loop exits.
- **Loop body S_{body}:** The statements executed in one iteration. They must preserve I.
- **Established / preserved:** I is *established* if it holds after initialization; it is *preserved* if every loop body execution keeps it true.

- **Safe final state:** A correct exit state satisfies $\neg B$, I, and R. The central proof obligation is $(\neg B) \wedge I \Rightarrow R$.
- **Loop variant:** An integer-valued measure used to prove termination. It is different from a loop invariant, which is a boolean predicate used for correctness.
- **Universal and existential quantifiers:** $\forall$ means "for every" and $\exists$ means "there exists." For arrays, quantify over legal indices such as $0 \leq j <$ `a.length`.
- **Public contracts vs. internal state:** Q refers to inputs; R refers to inputs and outputs. A loop invariant may additionally refer to loop counters and local variables because it describes intermediate progress.

Key Examples and Demonstrations

The `testLI` contract trace

Setup: Start with $i = 1$, use an empty initialization, repeatedly increment `i` while $i \leq 5$, and maintain $1 \leq i \leq 6$.

Main point: After the last iteration, $i = 6$: the invariant still holds, the stay condition is false, and the postcondition $i = 6$ holds. This is the intended exit pattern, not an exception to the invariant.

What to practise: Fill in the trace table yourself for initialization and all five iterations. At each row, evaluate I, B, and R, and distinguish "R is currently false" from a genuine postcondition violation.

A complete postcondition for `findMax`

Setup: For a valid array `a`, `findMax` returns `result`. The notes used $[20, 10, 40, 30]$ as the running example.

Main point: A complete postcondition combines the following two claims:

$$R_1: \quad \forall j \bullet \big(0 \leq j < \texttt{a.length} \Rightarrow \texttt{result} \geq \texttt{a[j]}\big)$$
$$R_2: \quad \exists k \bullet \big(0 \leq k < \texttt{a.length} \wedge \texttt{result} = \texttt{a[k]}\big)$$

R_1 says the result is at least every array value; R_2 rules out values that are not actually in the array.

What to practise: Explain why R_1 alone would incorrectly allow a result such as $+\infty$, then rewrite R_2 using an index variable rather than informal set-membership notation.

Finding the flaw in a proposed invariant

Setup: Consider the candidate $\forall j \bullet (0 \leq j \leq i \Rightarrow \texttt{result} \geq \texttt{a[j]})$ for the given `findMax` loop, where `i` is incremented at the end of the loop body.

Main point: The invariant's range is one step too large after `i` is incremented. On the lecture array, it fails when `i` = 2 and `result` = 20, because it now requires $20 \geq$ `a[2]` $= 40$.

What to practise: Recreate the initialization, first iteration, and second iteration in a table. State exactly which value of j witnesses the violation, then use the posted exercise solution to compare against the correct invariant and proof.

Common Pitfalls and Misconceptions

Confusing the stay condition with the loop invariant

Pitfall: Treating B as the invariant because both are boolean expressions checked during execution.

Why it causes problems: B only decides whether another iteration occurs. It does not usually provide the stronger progress information needed to derive the postcondition at exit.

How to avoid it: Ask whether $\neg B$ together with the proposed I is strong enough to imply R. If not, the proposed invariant is not useful for the correctness proof.

Expecting the postcondition to hold throughout the loop

Pitfall: Declaring an error because R is false during an intermediate iteration.

Why it causes problems: R describes the final output after the loop exits, not a promise about partial progress.

How to avoid it: Check R only on the exit path, after B becomes false. During iterations, focus on whether I is established or preserved.

Writing an incomplete postcondition for `findMax`

Pitfall: Specifying only that `result` is at least every array entry.

Why it causes problems: A value not stored in the array could still satisfy that condition, so the contract would permit an incorrect implementation.

How to avoid it: Combine a universal dominance condition with an existential membership condition over valid array indices.

Using loop counters in public contracts

Pitfall: Defining Q or R using an internal loop counter such as `i`.

Why it causes problems: Preconditions and postconditions should state the public input and output behaviour, independent of one particular implementation.

How to avoid it: Use input variables in Q, input and output variables in R, and reserve counters or local variables for I.

Treating an array as a set

Pitfall: Writing an informal membership assertion such as `result` $\in$ `a`.

Why it causes problems: An array is an indexed sequence, so the condition does not state which array positions are being considered.

How to avoid it: Use $\exists k$ with the bounds $0 \leq k <$ `a.length`, then compare `result` with `a[k]`.

Trusting a plausible invariant without tracing it

Pitfall: Assuming the proposed $j \leq i$ invariant is correct because it resembles the final maximum condition.

Why it causes problems: The invariant is checked after `i` changes, so it can include an element that the loop body has not processed yet.

How to avoid it: Trace the exact state after initialization and after each iteration, paying attention to the order of assignments and the value of each quantified bound.

What You Should Be Able to Do

After this tutorial, you should be able to:

- identify Q, R, I, S_{init}, B, and S_{body} in a simple iterative algorithm;
- explain the difference between a loop invariant used for correctness and a loop variant used for termination;
- trace a loop and determine whether an invariant is established, preserved, or violated;
- explain why a correct exit state must satisfy $\neg B$, I, and R, and apply $(\neg B) \wedge I \Rightarrow R$;
- write the precondition for a non-null, non-empty input array;
- formulate the two essential parts of a complete `findMax` postcondition using $\forall$ and $\exists$ over array indices;
- distinguish public contracts from implementation-specific variables; and
- diagnose the failure of a candidate loop invariant by giving a concrete trace and counterexample state.

Suggested Practice

- Rewatch the Tutorial 7 flowchart discussion and redraw the paths for: precondition violation, invariant not established, invariant not preserved, postcondition violation, and safe termination.
- Recreate the `testLI` trace with columns for `i`, I, B, and R. Explain why R may be false before termination.
- For `findMax`, write Q, R_1, and R_2 formally, then test whether each condition rejects deliberately faulty implementations.

- Trace the proposed $j \leq i$ invariant on $[20, 10, 40, 30]$ and identify the exact iteration and array index at which it fails.
- Watch the Tutorial 7 exercise-solution video for the correct `findMax` invariant and the proof using the exit condition and invariant.
- Review predicate and propositional logic, especially $\forall$, $\exists$, $\Rightarrow$, $\wedge$, and negation, before the Written Test.
- Review the Lecture 16 Dijkstra trace so that you are ready to apply the loop-invariant framework to Dijkstra's algorithm in Lecture 17.
- Continue Assignment 2 before its July 8 deadline; its work also supports Programming Test 2 preparation.

Where to Review

- **Tutorial 7 iPad/PDF notes, page 2:** the general syntax Q, R, I, S_{init}, B, and S_{body}, plus the flowchart of contract checks and violations.
- **Tutorial 7 notes, page 3:** the `testLI` trace and the exit reasoning $(\neg B) \wedge I \Rightarrow R$.
- **Tutorial 7 notes, pages 4–6:** `findMax`, the precondition, the two-part postcondition, index-based quantification, and the public/private distinction for contract variables.
- **Tutorial 7 notes, page 7:** the proposed incorrect invariant and its failure on $[20, 10, 40, 30]$.
- **Graph lecture slides 62–67:** the formal loop-correctness material, including the flowchart on slide 65.
- **Tutorial 7 page on the lecture site:** the session recording and the posted exercise-solution video for the corrected loop invariant and formal proof.
- **Lecture 16 notes and recording:** weighted graphs and the Dijkstra trace that this tutorial prepares you to revisit in Lecture 17.

Closing Reminder

Loop correctness is not learned by memorizing names such as precondition or invariant. Revisit the notes and recording, trace the examples carefully, write the predicates yourself, and check the exact state at initialization, after each iteration, and at exit. Use the posted Tutorial 7 solution to verify your reasoning, then apply the same habits when Dijkstra's algorithm returns next week.

EECS3101 Algorithms (Summer'26)

Digest of Tutorial 8, July 9

Dijkstra's Algorithm: Time Complexity, Graph Representation Tradeoffs

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 10

Closing Reminder 10

Main Ideas

- This tutorial completed the heap-based running-time analysis of Dijkstra's algorithm and explained why the final bound is $O((n+m)\cdot\log n)$ in terms of vertices $n = |V|$ and edges $m = |E|$.
- The most important counting idea was that the inner neighbour-processing work is counted by summing vertex degrees, giving m for directed graphs and $2m$ for undirected graphs, not by blindly multiplying loop headers.
- Heap implementation details matter: reading the minimum at the root is $O(1)$, but insertion, deletion, and decrease-key can require bubbling and therefore cost $O(\log n)$.
- Jackie emphasized that auxiliary checks, such as testing whether a vertex is still in the priority queue, must be implemented efficiently; an $O(n)$ scan inside the inner loop would damage the intended analysis.
- The second half of the tutorial compared graph implementation strategies: edge list, adjacency list, and adjacency matrix.
- The adjacency-list strategy extends the edge-list idea by storing incident-edge lists at vertices and position links at edges, allowing efficient edge removal when all related lists are updated correctly.
- The adjacency-matrix strategy stores edge references in a graph-level 2D array indexed by vertices, making edge lookup direct but making dynamic vertex addition expensive.
- For Written Test 2, the graph lecture coverage was confirmed as up to and including the graph-representation comparison table on slide 75; later graph topics next week were excluded from that test coverage.

Roadmap

Dijkstra analysis setup

The tutorial began by returning to the graph lecture and using an alternative pseudocode form of Dijkstra's algorithm because it is easier to analyze line by line. The priority queue Q was assumed to be implemented by a heap, so the analysis reused the heap facts from Lecture 18: heap height is $O(\log n)$, the minimum is found at the root, and bubbling after insertion, deletion, or key change takes logarithmic time. Jackie also reminded students that the graph may be directed or undirected, but Dijkstra still requires non-negative edge weights.

Initialization and heap construction

The first part of the pseudocode initializes the distance array or map D and the ancestor array or map a. These are simple assignments over the vertices, so the initialization lines for

D and a contribute $O(n)$. The next step inserts all n vertices into the heap-backed priority queue. Since one insertion can require up-heap bubbling, a single insertion is $O(\log n)$, and inserting all vertices costs $O(n \cdot \log n)$.

The outer loop and the minimum vertex

The main loop continues while Q is not empty. Since all vertices are inserted and each vertex is eventually removed or finalized once, the outer loop runs n times. Jackie separated two related operations: simply reading the minimum at the heap root is $O(1)$, but removing the minimum and restoring the heap structure is $O(\log n)$. Therefore, across the whole run, the removals contribute $O(n \cdot \log n)$.

Counting the inner loop correctly

The key technical point was the inner loop over the vertices adjacent to the current vertex u. A tempting but misleading approach is to treat the nested loop as automatically n^2. Jackie instead counted how often the inner body runs over the entire algorithm: for each finalized vertex u, the loop visits the incident or outgoing edges of u. Summing these degrees over all vertices gives m in a directed graph, or $2m$ in an undirected graph, which is still $O(m)$. This is the reason the edge term appears in the final bound.

Decrease-key and membership checks

Inside the inner loop, updating $D[v]$ is not just a constant-time assignment if $D[v]$ is the key stored in a heap. Decreasing a key may break the Heap Order Property, so the heap may need up-heap bubbling, giving $O(\log n)$ for that update. Jackie also discussed the cost of checking whether v is still in Q: this membership test must be $O(\log n)$ or better, for example by maintaining an appropriate auxiliary balanced search structure, rather than by doing a linear scan.

Putting the Dijkstra bound together

Combining the pieces gives $O(n)$ for simple initialization, $O(n \cdot \log n)$ for building the heap, $O(n \cdot \log n)$ for removals, and $O(m \cdot \log n)$ for edge relaxations and related efficient checks. The resulting general bound is

$$O((n + m) \cdot \log n).$$

Jackie then explained that if the graph is dense, so m can be on the order of n^2, this may be written as $O(n^2 \cdot \log n)$ for that special case. The more accurate general form remains $O((n + m) \cdot \log n)$.

Moving from Dijkstra to graph representations

After finishing the Dijkstra analysis, Jackie introduced the main exercise for Tutorial 8: compare the running times of graph operations under three implementation strategies. Students had already worked with edge lists through the assignments. The tutorial then introduced

adjacency lists and adjacency matrices and pointed students to the feature summary and time-complexity table in the graph slides.

Adjacency list strategy

The adjacency-list strategy keeps the familiar lists of vertices and edges, but each vertex additionally stores a list of incident edges. Each edge also stores positions in the main edge list and in the incident-edge lists of its endpoints. The visualization with vertices A, B, C and edges O and P showed how vertex objects, edge objects, list positions, and incident-edge lists point to one another at run time. This is the hardest diagram in the tutorial, and Jackie explicitly encouraged students to map each arrow back to a Java attribute.

Removing an edge from an adjacency list

For an edge such as $O = (B, A)$ in the adjacency-list diagram, removal must happen in three places: the origin vertex's incident-edge list, the destination vertex's incident-edge list, and the main edge list. Because stored positions point directly to the relevant list nodes, each removal can be $O(1)$. The main lesson is not only the final bound, but also the need to update every related structure so the graph representation does not leave stale references behind.

Adjacency matrix strategy

The adjacency-matrix strategy stores a graph-level 2D array of edge references. Each vertex has an index used to locate the corresponding row and column. In the undirected example, the entries for (A, B) and (B, A) point to the same edge object, so the matrix is symmetric through aliasing; a null entry means that no edge is currently present. Rows or columns can be used to inspect incident edges, and edge existence checks can be direct.

Tradeoffs and the Tutorial 8 comparison table

Jackie closed by directing students to slides 74 and 75: slide 74 summarizes the ingredients of the three graph strategies, and slide 75 gives the comparison table for operations such as `numVertices()`, `vertices()`, `getEdge(u,v)`, `insertVertex(x)`, `removeVertex(v)`, `insertEdge(u,v,x)`, and `removeEdge(e)`. Students should not only memorize the table; they should be able to derive entries from the underlying representation.

Key Terms, Notation, and Structures

- $n = |V|$ **and** $m = |E|$: n denotes the number of vertices and m denotes the number of edges in the input graph.
- **Heap-backed priority queue** Q: The priority queue used by Dijkstra stores vertices using their current D values as keys.

- **Find minimum versus remove minimum**: Reading the minimum at the heap root is $O(1)$, while removing it and restoring the heap is $O(\log n)$.
- **Decrease-key**: The operation of lowering a vertex's key $D[v]$ in the heap; it may require up-heap bubbling and therefore costs $O(\log n)$ in this analysis.
- **Degree**: The number of incident edges of a vertex. In this tutorial, a lowercase d in the graph-representation table means degree, not Dijkstra's true-distance notation from previous lectures.
- **Sum of degrees**: For directed graphs, summing the relevant outgoing degrees across all vertices gives m; for undirected graphs, summing degrees gives $2m$, which is still $O(m)$.
- **Dense graph**: A graph whose number of edges can be on the order of n^2; in this case $O((n+m)\cdot\log n)$ can specialize to $O(n^2\cdot\log n)$.
- **Edge list**: A graph representation with a list of vertices and a list of edges; it is the strategy students had already used in the assignments.
- **Adjacency list**: A graph representation that stores, for each vertex, a list of its incident edges, in addition to global vertex and edge lists.
- **Incident-edge list**: The per-vertex list that stores the edges touching that vertex in the adjacency-list strategy.
- **Position pointers**: Stored references to positions in linked lists; they allow constant-time removal once the relevant object is known.
- **Adjacency matrix**: A graph representation using a graph-level 2D array of edge references, with vertex indices used as row and column indices.
- **Aliasing in an undirected matrix**: In an undirected graph, `matrix[i][j]` and `matrix[j][i]` may refer to the same edge object.
- **Null matrix entry**: A null entry in the adjacency matrix means there is no edge between the corresponding pair of vertices.

Key Examples and Demonstrations

Line-by-line Dijkstra running-time analysis

Setup: The pseudocode uses a graph $G = (V, E)$, a source vertex s, distance values D, ancestor values a, and a heap-backed priority queue Q.

Main point: The final complexity comes from combining several different costs: simple vertex-wide initialization, heap insertions, n removals, and edge-based inner-loop work. The important result is $O((n+m)\cdot\log n)$, not a generic nested-loop guess.

What to practise: Recreate the analysis line by line: label which lines are $O(n)$, which are $O(n\cdot\log n)$, where $O(m)$ inner-loop executions come from, and why each key update can add a $\log n$ factor.

Counting inner-loop work by degrees

Setup: For each removed vertex u, the algorithm examines vertices or edges adjacent to u.

Main point: Across the whole algorithm, these examinations correspond to the graph's edges. In a directed graph the total is m; in an undirected graph it may be $2m$, which is still asymptotically $O(m)$.

What to practise: Draw a small directed graph and a small undirected graph. List each vertex degree and sum the values. Then connect that sum to how many times Dijkstra's inner-loop body can execute.

Adjacency-list object-and-pointer diagram

Setup: The notes use vertices A, B, C and edges O and P to show global vertex and edge lists, vertex objects, edge objects, incident-edge lists, and stored positions.

Main point: Adjacency lists are faster than pure edge lists for many local operations because each vertex directly knows its incident edges. The price is a more sophisticated object structure with multiple pointers that must remain consistent.

What to practise: Redraw the diagram with one additional edge. For each edge object, mark its origin, destination, main edge-list position, and positions in the endpoint incident-edge lists.

Constant-time edge removal in an adjacency list

Setup: To remove an edge, the representation must remove it from the origin's incident-edge list, the destination's incident-edge list, and the global edge list.

Main point: With stored positions and doubly linked lists, each of the three removals can be $O(1)$. The correctness of the representation depends on doing all three updates, not just deleting the edge from one visible place.

What to practise: Pick one edge in the adjacency-list diagram and write the three conceptual removal actions. Then explain why each one is constant time only because a position is already stored.

Adjacency matrix visualization and vertex insertion

Setup: The matrix example maps vertices such as B, A, and C to indices 0, 1, and 2, and stores edges in a 2D array of edge references.

Main point: Matrix entries make edge existence checks direct, but adding a new vertex to a primitive array-based matrix requires allocating a larger matrix and copying the old entries, giving $O(n^2)$ work.

What to practise: Draw the 3×3 matrix for the example, then add a fourth vertex. Mark which entries are copied, which entries are new nulls, and why the copy step is quadratic.

Graph representation comparison table

Setup: The tutorial exercise asks students to compare operations under edge list, adjacency list, and adjacency matrix strategies.

Main point: Each table entry should be derived from the representation, not memorized. For example, `getEdge(u,v)` depends on whether you scan all edges, scan a smaller incident list, or index directly into a matrix.

What to practise: Choose three operations from slide 75 and write the step-by-step procedure for each representation before checking the table or the solution walkthrough.

Common Pitfalls and Misconceptions

Blindly treating the nested loops in Dijkstra as n^2

Pitfall: Assuming that an outer loop over vertices and an inner loop over adjacent vertices automatically costs $O(n^2)$.

Why it causes problems: This ignores the graph structure. The inner loop work is governed by the total number of incident edges examined, which sums to m or $2m$, not necessarily n^2.

How to avoid it: Whenever a graph algorithm loops over neighbours, try to rewrite the total count as a sum of degrees before deciding the asymptotic bound.

Forgetting that decrease-key is not just an assignment

Pitfall: Treating a change to $D[v]$ as $O(1)$ even though $D[v]$ is the heap key.

Why it causes problems: Changing a heap key can violate the Heap Order Property, so the heap may need up-heap bubbling to restore its relational structure.

How to avoid it: Each time an algorithm changes a value used as a priority-queue key, ask what repair operation the data structure needs.

Confusing root access with heap deletion

Pitfall: Saying that `extractMin` is always $O(1)$ or always $O(\log n)$ without clarifying whether it only reads the root or removes it.

Why it causes problems: The root can be read in constant time, but removing it and restoring completeness and the Heap Order Property takes logarithmic time.

How to avoid it: In complexity explanations, state the operation semantics: reading or peeking at the minimum is $O(1)$; deleting or finalizing it is $O(\log n)$.

Using a linear membership check inside Dijkstra's inner loop

Pitfall: Checking whether v is still in Q by scanning a linear collection.

Why it causes problems: An $O(n)$ membership check inside edge processing can dominate the intended heap-based analysis.

How to avoid it: Use an auxiliary structure with $O(\log n)$ or better membership checks, and include that cost explicitly in the reasoning.

Memorizing the graph-representation table without derivation

Pitfall: Treating slide 75 as a table to memorize mechanically.

Why it causes problems: Jackie may ask for the reasoning behind a complexity, not just the final entry. Memorization also makes it easier to mix up edge list, adjacency list, and adjacency matrix costs.

How to avoid it: For each operation, first write what the representation must touch: all vertices, all edges, one incident list, one matrix row or column, or the whole matrix.

Removing an adjacency-list edge from only one place

Pitfall: Deleting an edge from the global edge list but forgetting its endpoint incident lists, or deleting it from one incident list but not the other.

Why it causes problems: The representation will contain dangling or inconsistent references, so later degree, incident-edge, or traversal operations can return wrong results.

How to avoid it: For edge removal, use the checklist: origin incident list, destination incident list, and global edge list.

Assuming the adjacency matrix is always the best representation

Pitfall: Choosing the adjacency matrix only because edge existence checks are direct.

Why it causes problems: A primitive array-based matrix can be expensive for dynamic vertex operations, especially adding a vertex, which requires creating a larger matrix and copying n^2 entries.

How to avoid it: Choose the graph representation according to the operations your application performs most frequently.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain why Dijkstra's algorithm with a heap-backed priority queue has running time $O((n+m)\cdot\log n)$;
- identify where the $O(n\cdot\log n)$ and $O(m\cdot\log n)$ terms come from in the pseudocode;
- justify why the total neighbour-processing count is $O(m)$ using the sum of degrees;
- distinguish reading the minimum heap root from removing the minimum and restoring the heap;
- explain why decreasing a heap key requires up-heap bubbling in the worst case;
- describe why a membership check such as $v \in Q$ must not be implemented by an $O(n)$ scan inside Dijkstra's inner loop;
- draw the main object structure of an adjacency-list graph, including vertex lists, edge lists, incident-edge lists, and position links;

- explain the three updates needed to remove an edge from an adjacency-list representation;
- draw an adjacency matrix for a small undirected graph and explain how symmetric entries can alias the same edge object;
- explain why adding a vertex to an array-based adjacency matrix costs $O(n^2)$;
- compare edge list, adjacency list, and adjacency matrix strategies by deriving selected operation costs rather than only recalling the final table;
- locate the graph-lecture coverage for Written Test 2 and use slide 75 as a derivation target, not just a memorization target.

Suggested Practice

- Reproduce the Dijkstra running-time calculation from the pseudocode without looking at the digest: initialization, heap construction, outer loop count, inner-loop count, key updates, removals, and final simplification.
- Draw one directed and one undirected graph, compute each vertex degree, and verify the appropriate sum-of-degrees count used in the Dijkstra analysis.
- Review heap insertion, deletion, and decrease-key until you can explain which operation uses up-heap bubbling and which uses down-heap bubbling.
- Complete the Tutorial 8 graph-representation exercise: for each operation, write the steps under edge list, adjacency list, and adjacency matrix before checking slide 75.
- Redraw the adjacency-list runtime diagram with vertices and edges as objects. Label which arrows are ordinary object attributes and which are stored positions for efficient removal.
- Practise removing one edge from an adjacency-list diagram by crossing it out from all three required places.
- Redraw the adjacency-matrix example and add a new vertex. Show the new matrix size and explain where the $O(n^2)$ copy cost appears.
- Use the Assignment 2 solution and the newly released Assignment 3 material to connect the graph representations back to the programming-test preparation path.
- For Written Test 2, practise explaining at least a few table entries in words, especially `getEdge(u,v)`, `removeVertex(v)`, `insertVertex(x)`, and `removeEdge(e)`.

Where to Review

- Review the iPad notes page on Dijkstra's time complexity, especially the annotated pseudocode, the $n = |V|$, $m = |E|$ notation, the heap facts, and the final $O((n + m) \cdot \log n)$ bound.
- Revisit the part of the recording where Jackie explains why the inner loop contributes $O(m)$ work. This is the key reasoning step students found less intuitive during class.
- Review the adjacency-list strategy pages in the notes, including the Java class attributes and the runtime object diagram with vertices A, B, C and edges O, P.
- Review the adjacency-matrix strategy pages, especially the graph-level 2D edge array, vertex index attributes, null entries, and undirected aliasing.
- Study slides 74 and 75 of the graph slides: slide 74 summarizes the ingredients of the graph strategies, and slide 75 is the comparison table for operation running times.
- Use the Tutorial 8 instruction and solution walkthrough video if your derived table entries do not match slide 75.
- Review the heap insertion-versus-deletion exercises from Lecture 18, because the Dijkstra analysis depends on the $O(\log n)$ heap-operation costs.
- For Written Test 2, use Jackie's posted guide and the confirmed graph-lecture coverage up to and including slide 75; topics introduced after this point next week were not part of that written-test coverage.

Closing Reminder

This digest gives you the roadmap, but the learning work is in the tracing and derivation. Revisit the Dijkstra pseudocode, redo the degree-sum argument, and practise explaining why each heap operation costs what it costs. Then use the graph-representation diagrams and slide 75 table as active exercises: for each operation, identify what data structure is touched and derive the time bound yourself before checking the posted solution or recording.

EECS3101 Algorithms (Summer'26)

Digest of Tutorial 9, July 30

Min-Cost Climbing: Recursion, Memoization, and Tabulation

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 9

Closing Reminder 10

Main Ideas

- This final tutorial session completed the Minimum Cost Climbing Stairs example by moving from the problem specification to the recursive, top-down memoized, and bottom-up tabulated solutions.
- The central state was the minimum cost to finish from stair i, with terminal states at the top n and beyond $n+1$ contributing zero additional cost.
- The key recurrence was M_i = `cost[i]` + $\min(M_{i+1}, M_{i+2})$ for valid stair indices, and the final answer was $\min(M_0, M_1)$ because climbing may start from stair 0 or stair 1.
- The recursion tree for $\{4, 3, 5, 7\}$ showed both optimal substructure and overlapping subproblems, so the plain recursive solution is correct but inefficient.
- Top-down memoization kept the recursive structure but used a `memo` array to compute each subproblem once and reuse it later by constant-time lookup.
- Bottom-up tabulation used a `tab` array of length $n+2$, stored the top and beyond base cases, and filled the table backward because each entry depends on larger indices.
- Jackie connected this tutorial directly to exam preparation: study the provided Java projects, reproduce the DP traces, attempt the knapsack exercise with a 2D DP table, and note that Kruskal's running-time derivation is out of scope for the exam.

Roadmap

Course wrap-up, code projects, and the final tutorial exercise

Jackie opened by framing this as the final session of the course and pointed students to two Java projects on the lecture site. The first project contains complete dynamic-programming examples for Fibonacci, factorial, and Minimum Cost Climbing Stairs, including JUnit tests. The second project is the Java starter for the final tutorial exercise on the knapsack problem.

The knapsack exercise is an important post-lecture practice task. Jackie provided a working recursive implementation and asked students to implement the top-down memoization and bottom-up tabulation methods. Unlike the 1D DP examples used in lecture and tutorial, knapsack requires a 2D DP table, so students should use the instruction PDF, hints, starter code, and JUnit tests rather than waiting passively for the solution.

Revisiting the Minimum Cost Climbing Stairs problem

The notes first restated the problem. The input is an array `cost`, where `cost[i]` is the cost paid when moving from stair i. If n = `cost.length`, then the valid stair indices are $0, 1, \ldots, n-1$. The imaginary index n is the top, and $n+1$ is beyond the top.

A climb may start at stair 0 or stair 1. From stair i, one pays `cost[i]` and then moves to either $i+1$ or $i+2$. Once the climb reaches the top or beyond, no further cost is needed. This distinction matters: the cost is tied to the current stair, not to the destination stair.

Exploring $\{4, 3, 5, 7\}$ with a recursion tree

The main running example was `cost` = $\{4, 3, 5, 7\}$. Jackie built a recursion tree to represent all legal choices. Starting from stair 0 branches to stairs 1 and 2; starting from stair 1 branches to stairs 2 and 3; and so on until the top or beyond is reached.

The tree was then evaluated backward. From stair 3, the only cost paid is `cost[3]` = 7, followed by zero additional cost from top or beyond, so the minimum from stair 3 is 7. From stair 2, the computation uses the cheaper of the already-understood futures from stairs 3 and 4, giving $5 + \min(7, 0) = 5$. Continuing backward gives minimum costs 8 from stair 1 and 9 from stair 0, so the final answer is $\min(9, 8) = 8$.

Deriving the recurrence and dependency direction

Jackie generalized the local question: if you are currently at stair i, what is the minimum total cost to reach the top or beyond? The answer must include the cost of leaving stair i, plus the cheaper of the two possible future subproblems. This gives

$$M_i = \begin{cases} 0, & i \geq n, \\ \texttt{cost[i]} + \min(M_{i+1}, M_{i+2}), & 0 \leq i < n. \end{cases}$$

The overall answer is $\min(M_0, M_1)$, because the rules allow two starting positions.

This recurrence also tells students how to fill a bottom-up table. Since M_i depends on M_{i+1} and M_{i+2}, the entries with larger indices must already be known before M_i can be computed. Therefore this problem uses backward filling, in contrast to Fibonacci's forward filling.

Why the recursive solution motivates DP

The recursive implementation in the notes mirrors the recurrence: handle terminal states with a base case, then return the current cost plus the minimum of the two recursive calls. This is a clean way to express the problem, but it recomputes the same subproblems repeatedly.

Jackie highlighted the two DP criteria. The problem has optimal substructure because the optimal cost from a stair is built from optimal costs of later stairs. It also has overlapping subproblems because states such as M_1, M_2, and M_3 reappear in the recursion tree. Therefore, the naive recursive version has exponential time complexity, while its stack space is linear in the input length.

Top-down DP: memoization

The top-down version keeps the recursive structure but adds a shared `memo` array. In the shown implementation, `memo` stores the minimum cost to finish from a stair index. Entries begin with a sentinel value such as -1, meaning the subproblem has not been computed yet.

When a recursive helper reaches a valid stair index, it first checks whether `memo[i]` already contains a computed value. If so, it returns that value in $O(1)$ time. If not, it computes the value once using the recurrence, stores it in `memo[i]`, and returns it. In the

$\{4, 3, 5, 7\}$ trace, the memo entries are filled backward as 7, 5, 8, and 9 for indices 3, 2, 1, and 0.

The result is still found by comparing the two valid starting states, M_0 and M_1. The memoized solution has time complexity $O(n)$, because each state is computed once, and space complexity $O(n)$, because of the memo array and recursion stack.

Bottom-up DP: tabulation

The bottom-up solution removes recursion and fills an array explicitly. The notes used a `tab` array of length $n + 2$, which gives slots for the two terminal base states: `tab[n] = 0` for the top and `tab[n+1] = 0` for beyond.

The loop starts at the last valid stair index, $n - 1$, and moves backward to 0. At each index i, it applies

$$\texttt{tab[i]} = \texttt{cost[i]} + \min(\texttt{tab[i+1]}, \texttt{tab[i+2]}).$$

For $\{4, 3, 5, 7\}$, this fills `tab[3]`, then `tab[2]`, then `tab[1]`, then `tab[0]`. The final answer is again $\min(\texttt{tab[0]}, \texttt{tab[1]}) = 8$.

Exam-scope and review guidance

Jackie told students to be able to reproduce the tracing and explain what happens at each step. The exam may not require grading a full hand-drawn tree exactly like the lecture, but the understanding behind the trace is examinable.

Jackie also said the Kruskal running-time derivation would be skipped for this course and would not be asked directly on the exam. Students who want to review heap operations can use the earlier heap lecture independently, but the immediate DP priorities are Minimum Cost Climbing Stairs and knapsack.

Key Terms, Notation, and Structures

- `cost` **array**: The input array where `cost[i]` is the cost paid when moving from stair i.
- **State** i: The current stair index used to define the subproblem "minimum cost to finish from stair i".
- **Top and beyond**: If n = `cost.length`, then index n is the top and index $n + 1$ is beyond; both are terminal states with zero additional cost.
- **Minimum-cost state** M_i: The minimum total cost to reach the top or beyond starting from stair i.
- **Recurrence**: For valid stair indices, M_i = `cost[i]` + $\min(M_{i+1}, M_{i+2})$; for $i \geq n$, $M_i = 0$.
- **Final answer**: The answer is $\min(M_0, M_1)$, because climbing may start from stair 0 or stair 1.

- **Optimal substructure**: The optimal answer for a current stair is built from optimal answers for the two possible next stairs.
- **Overlapping subproblems**: The recursion tree reaches the same stair states multiple times, so plain recursion repeats work.
- **Memoization**: The top-down DP technique that keeps recursion but stores computed values in a `memo` array for reuse.
- **Tabulation**: The bottom-up DP technique that fills a `tab` array iteratively according to the recurrence dependencies.
- **Backward filling**: Filling larger indices before smaller ones because M_i depends on M_{i+1} and M_{i+2}.
- `Math.min`: A Java library method used in the implementation to choose the cheaper of two recursive or tabulated futures.
- **2D DP table**: The next practice step in the knapsack exercise, where two state variables require a two-dimensional array rather than the 1D arrays used in this tutorial.

Key Examples and Demonstrations

Problem specification and edge cases

Setup: The notes define Minimum Cost Climbing Stairs using a `cost` array, valid indices 0 to $n-1$, top at index n, and beyond at index $n+1$. Small examples include $\{5\}$, $\{10, 15\}$, and $\{15, 10\}$.

Main point: The climb may start at stair 0 or stair 1, and reaching the top or beyond ends the game with zero additional cost. These rules explain why very small arrays can have answers that initially feel unintuitive.

What to practise: Rework the small examples from the notes and justify each answer using the rule "pay at the current stair, then move one or two steps."

Recursion tree for $\{4, 3, 5, 7\}$

Setup: The running example uses `cost = {4,3,5,7}`, with top at index 4 and beyond at index 5. The recursion tree branches from each stair to the next one or two stairs.

Main point: Evaluating the tree backward gives minimum costs from later stairs first. The important values are $M_3 = 7$, $M_2 = 5$, $M_1 = 8$, and $M_0 = 9$, so the final answer is $\min(9, 8) = 8$.

What to practise: Redraw the tree and annotate each node with the minimum cost from that stair to the top or beyond. Mark every place where a subtree would be recomputed by the plain recursive method.

Recurrence relation

Setup: Jackie generalized the local question from the tree: what is the minimum cost to finish starting from stair i?

Main point: The current move costs `cost[i]`, and then the algorithm should choose the cheaper of the two future states. This gives the recurrence M_i = `cost[i]` + $\min(M_{i+1}, M_{i+2})$, with terminal value 0 at and beyond the top.

What to practise: Derive the recurrence yourself before looking at the code. Then explain why the final answer compares M_0 and M_1 rather than returning only one of them.

Recursive implementation and complexity

Setup: The recursive implementation checks input validity, handles terminal states, and recursively computes the cheaper of the two possible moves.

Main point: The recursive code matches the recurrence and is easy to reason about, but it is not efficient because the same states are recomputed. Its space complexity is linear because of stack depth, while its time complexity is exponential because of repeated branching.

What to practise: Trace the recursive method on a smaller input and identify the base case, the two recursive calls, and the repeated states.

Top-down memoization trace

Setup: The top-down implementation adds a `memo` array initialized with -1. For $\{4, 3, 5, 7\}$, entries for valid stair indices are filled as the recursion unfolds.

Main point: Memoization keeps the recursive flow but replaces later visits to a computed state with an $O(1)$ lookup. In the trace, values are stored once and then reused, producing the same final answer 8.

What to practise: Recreate the memoization trace and write down the order in which `memo[3]`, `memo[2]`, `memo[1]`, and `memo[0]` become available.

Bottom-up tabulation trace

Setup: The bottom-up implementation uses `tab` of length $n + 2$, initializes `tab[n]` and `tab[n+1]` to 0, and loops backward from $n - 1$ to 0.

Main point: Each assignment uses two already-filled larger-index entries. The tabulation array ends with the same computed state values as the memoization array, but it reaches them through a loop rather than recursion.

What to practise: Make a table with columns for iteration number, loop index i, assignment to `tab[i]`, and the updated array. Check that the loop direction is forced by the recurrence.

Knapsack as the next DP step

Setup: Jackie showed the final tutorial exercise on the knapsack problem. The recursive version is provided, while students must implement the top-down and bottom-up DP ver-

sions.

Main point: Knapsack extends the same DP ideas to a problem with two state variables, so the auxiliary table is 2D rather than 1D.

What to practise: Read the knapsack background PDF, study the provided recursive method, identify the two state variables, and start designing the 2D memoization/tabulation table before checking the eventual solution.

Common Pitfalls and Misconceptions

Charging the destination stair instead of the current stair

Pitfall: Treating a move to $i+1$ or $i+2$ as if it costs `cost[i+1]` or `cost[i+2]`.

Why it causes problems: The recurrence becomes wrong because the problem charges the stair you are leaving, not the stair you are entering.

How to avoid it: State the subproblem as "starting from stair i." Then write `cost[i]` before comparing the two future costs.

Forgetting the top and beyond base cases

Pitfall: Trying to access `cost[n]` or `cost[n+1]` as if they were ordinary array elements.

Why it causes problems: Those are invalid indices in `cost`. They are conceptual terminal states with zero additional cost, not real cost entries.

How to avoid it: In recursion, return 0 when $i \geq n$. In tabulation, create slots such as `tab[n]` and `tab[n+1]` and initialize both to 0.

Returning M_0 instead of $\min(M_0, M_1)$

Pitfall: Assuming the climb must start at stair 0.

Why it causes problems: The problem explicitly allows starting at stair 0 or stair 1, so considering only one starting state can give the wrong answer.

How to avoid it: After computing the state values, finish with $\min(M_0, M_1)$ or `Math.min(tab[0], tab[1])`.

Filling the tabulation table in the wrong direction

Pitfall: Filling the table forward from 0 to $n-1$, by analogy with Fibonacci.

Why it causes problems: M_i depends on M_{i+1} and M_{i+2}, so a forward fill tries to use values that have not yet been computed.

How to avoid it: Read the recurrence before writing the loop. If the right-hand side uses larger indices, fill backward.

Thinking every recursive problem deserves DP

Pitfall: Believing that a recurrence relation alone is enough reason to use dynamic programming.

Why it causes problems: Some recursive problems, such as factorial, do not have overlapping subproblems. DP then adds machinery without improving the asymptotic running time.

How to avoid it: Check both DP criteria: optimal substructure and overlapping subproblems. Then compare the complexity of the plain recursive version with the DP version.

Treating the recursion tree as an arbitrary graph

Pitfall: Drawing a general graph with cycles to model the recursion.

Why it causes problems: Cycles suggest infinite recursion or unclear termination. The recursive call structure should be understood as a tree of calls, even if different branches represent the same state value.

How to avoid it: Draw the unfolding of recursive calls as a tree, then separately mark duplicated states as overlapping subproblems.

Waiting for the knapsack solution before trying the exercise

Pitfall: Treating the provided solution or review session as a substitute for attempting the DP design independently.

Why it causes problems: The exam preparation value comes from identifying states, array dimensions, and recurrence dependencies yourself.

How to avoid it: Study the provided recursive knapsack method first, then attempt the top-down and bottom-up methods using the starter code and JUnit tests before looking for the solution.

What You Should Be Able to Do

After this tutorial, you should be able to:

- explain the Minimum Cost Climbing Stairs rules, including valid starts, the top, beyond, and when costs are paid;
- define the state M_i as the minimum cost to finish from stair i;
- derive and explain the recurrence M_i = `cost[i]` + $\min(M_{i+1}, M_{i+2})$ with terminal value 0 for $i \geq n$;
- justify why the final answer is $\min(M_0, M_1)$;
- draw and evaluate a recursion tree for a small cost array such as $\{4, 3, 5, 7\}$;
- identify optimal substructure and overlapping subproblems in the recursion tree;
- analyze why the plain recursive solution has exponential time and linear stack space;
- implement or explain the top-down memoized solution using a shared `memo` array and constant-time lookup;

- implement or explain the bottom-up tabulated solution using a `tab` array, top/beyond base cases, and a backward loop;
- compare memoization and tabulation in terms of control flow, table initialization, and complexity;
- use the provided Java projects and JUnit tests to check your DP implementations; and
- begin the knapsack exercise by identifying why it needs a 2D DP table.

Suggested Practice

- Redraw the $\{4, 3, 5, 7\}$ recursion tree from the notes, label top and beyond, and compute the minimum cost at each node backward.
- Re-derive the recurrence before looking at the code: write the current cost, then the minimum of the two future subproblems.
- Trace the plain recursive method on a small input and mark every repeated state.
- Trace the top-down solution by updating the `memo` array in the order values become available.
- Trace the bottom-up solution with a table of iterations, loop index i, assignment, and updated `tab` entries.
- Run the provided JUnit tests for the dynamic-programming examples, then add at least one small edge-case test of your own.
- Compare Minimum Cost Climbing Stairs with Fibonacci: decide for each one whether the table is filled forward or backward, and explain why.
- Revisit the factorial self-study exercise from Lecture 24 and explain why it is not a strong DP example despite having a recurrence.
- Start the knapsack tutorial exercise: read the background PDF, study the recursive implementation, identify the two state variables, and attempt the top-down and bottom-up methods before the review session.

Where to Review

- **iPad/PDF notes, page 1:** Topic overview for Tutorial 9: Minimum Cost Climbing by recursion, memoization, and tabulation.
- **iPad/PDF notes, page 2:** Problem statement, rules, top/beyond notation, and small examples that clarify the start choices and edge cases.

- **iPad/PDF notes, page 3:** Full worked trace for $\{4, 3, 5, 7\}$, including the recursion tree, backward cost calculations, recurrence idea, and final answer 8.
- **iPad/PDF notes, page 4:** Formal recurrence relation, recursive implementation, DP suitability criteria, and complexity summary for the plain recursive solution.
- **iPad/PDF notes, page 5:** Top-down memoization implementation and trace, including the `memo` array and the order in which entries are filled.
- **iPad/PDF notes, page 6:** Bottom-up tabulation implementation and trace, including the $n + 2$ table size, top/beyond base cases, backward filling, and linear complexity.
- **Lecture recording/tutorial recording:** Review the explanations of the $\{4, 3, 5, 7\}$ tree, the derivation of the recurrence, and the comparison between memoization and tabulation.
- **Example dynamic-programming Java project:** Study the Fibonacci, factorial, and Minimum Cost Climbing solutions plus their JUnit tests.
- **Java starter for the final tutorial exercise:** Use the knapsack starter, instruction PDF, hints, and tests to practise a 2D DP problem.
- **Exam preparation notes:** Follow Jackie's upcoming review-session and exam-guide announcements; Kruskal's running-time derivation is not a direct exam target for this course.

Closing Reminder

This digest is only a roadmap for the final tutorial session. To make the DP material usable for the exam, you still need to revisit the recording, study Jackie's iPad/PDF notes, run and trace the Java code, check behaviour with JUnit tests, and attempt the knapsack exercise yourself. The important work is not memorizing the final arrays; it is being able to derive the state, recurrence, base cases, fill direction, and complexity for a new problem.