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.