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

- 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.