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

- 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.