

4.13 Implementing 2D Array Algorithms

Self-Study Guide · About 75 minutes · With answer key

By the end you will be able to

- Compute a total or an average for a whole 2D array, a single row, a single column, or another region of it.
- Find the smallest or largest value in a 2D array or in part of it, and remember where it was found.
- Use a flag to decide whether at least one element, or every element, in a region has a property, and use a counter to find how many do.
- Visit every pair of neighboring elements in a row or a column without going past the edge.
- Decide whether a row, a column, or a whole 2D array contains a repeated value.
- Shift, rotate, and reverse the elements of one row or one column, and trace exactly what each step changes.

What to remember from earlier

- In `grid[r][c]` the first index is the row and the second is the column. ([Topic 4.11: 2D Array Creation and Access](#))
- Nested loops visit every element, usually with the outer loop on rows. ([Topic 4.12: 2D Array Traversals, 4.12.1 Row-Major Order](#))
- In a nested enhanced for loop the outer variable is a whole row. ([Topic 4.12: 2D Array Traversals, 4.12.4 Nested Enhanced for Loops](#))
- Standard 1D algorithms include a running total, a minimum or maximum, a flag, and a counter. ([Topic 4.5: Implementing Array Algorithms](#))
- Dividing two int values drops the fraction unless one is cast to double. ([Topic 1.5: Casting and Range of Variables](#))

4.13.1 Totals and Averages for the Whole Array

A teacher keeps quiz scores in a 2D array: one row per student, one column per quiz. At the end of the unit she wants the class's total points and the average score on a quiz. In Topic 4.5 you added up a 1D array with one loop and a running total. In Topic 4.12 you learned to visit every element of a 2D array with nested loops. This topic puts the two together: every algorithm you used on a 1D array has a 2D version, and the main new decision is **which elements the loops visit**.

The plan in plain words: start a total at 0, visit every element in row-major order, and add each one to the total. The total is an **accumulator**, a variable that collects a result a little at a time across the passes of a loop (Topic 2.9). For the average, divide the total by the number of elements. A rectangular 2D array has `grid.length` rows with `grid[0].length` elements in each, so it holds `grid.length * grid[0].length` elements in all.

KEY IDEA

To total a whole 2D array, set the total to 0 before the nested loops and add `grid[r][c]` in the inner loop. The average is the total divided by `grid.length * grid[0].length`, with a cast to `double` so the fraction is kept.

WORKED EXAMPLE Total and average of every quiz score

Three students took four quizzes, each worth 10 points. Watch where `total` is declared (line 4, before both loops) and where it is changed (line 9, inside both).

```

1  int[][] scores = { {8, 10, 7, 9},
2                    {6, 9, 10, 8},
3                    {10, 7, 7, 5} };
4  int total = 0;
5  for (int r = 0; r < scores.length; r++)
6  {
7      for (int c = 0; c < scores[0].length; c++)
8      {
9          total += scores[r][c];
10     }
11 }
12 int count = scores.length * scores[0].length;
13 double average = (double) total / count;
14 System.out.println("Total: " + total);
15 System.out.println("Average: " + average);

```

The outer loop picks a row; the inner loop walks across that row. The table shows `total` at the end of each row.

r	Row elements added	Row sum	total after the row
0	8, 10, 7, 9	34	34
1	6, 9, 10, 8	33	67
2	10, 7, 7, 5	29	96

Line 12 counts the elements: 3 rows times 4 columns is 12. Line 13 casts `total` to `double` before dividing, so the division is `96.0 / 12`, which is `8.0`.

```

Total: 96
Average: 8.0

```

Because the algorithm visits **every** element and the order of adding does not change a sum, nested enhanced `for` loops from Topic 4.12 work just as well. Here the data change too: two rows of three practice times, in minutes.

```
1 int[][] minutes = { {10, 5, 8},
2                     {7, 9, 6} };
3 int total = 0;
4 for (int[] row : minutes)
5 {
6     for (int value : row)
7     {
8         total += value;
9     }
10 }
11 int count = minutes.length * minutes[0].length;
12 double average = (double) total / count;
13 System.out.println("Total: " + total);
14 System.out.println("Average: " + average);
```

The outer variable `row` is a whole `int[]` row, and the inner variable `value` is one `int` from that row. The six values add to 45, there are $2 \times 3 = 6$ of them, and $(\text{double})\ 45 / 6$ is 7.5. The output is `Total: 45` and then `Average: 7.5`. Use enhanced loops when you need every value but not its position; use index loops when you need `r` or `c` (you will need them soon).

WATCH OUT

Dividing two `int` values. It is easy to think that storing the answer in a `double` is enough:

```
1 int[][] minutes = { {10, 5, 8},
2                     {7, 9, 6} };
3 int total = 0;
4 for (int[] row : minutes)
5 {
6     for (int value : row)
7     {
8         total += value;
9     }
10 }
11 int count = minutes.length * minutes[0].length;
12 double average = total / count;
13 System.out.println("Average: " + average);
```

This prints `Average: 7.0`, not 7.5. Java evaluates the right side first: `total / count` is $45 / 6$ with two `int` values, which is integer division (Topic 1.5), so the result is 7. Only then is 7 widened to 7.0 and stored. The fraction was thrown away before the `double` variable ever saw it. The fix is to cast one operand before dividing:

```
double average = (double) total / count;
```

PAUSE AND TRY

Question 1 Tracing

Write the exact output.

```

1  int[][] grid = { {3, -2},
2                  {5, 1},
3                  {4, 7} };
4  int sum = 0;
5  for (int[] row : grid)
6  {
7      for (int value : row)
8      {
9          sum += value;
10     }
11 }
12 System.out.println(sum);
13 System.out.println((double) sum / (grid.length * grid[0].length));

```

Question 2 Multiple choice

grid is a rectangular 2D array with 5 rows and 8 columns. Which expression gives the number of elements it holds?

- A. grid.length
- B. grid[0].length
- C. grid.length + grid[0].length
- D. grid.length * grid[0].length

4.13.2 One Row, One Column, or a Region

The teacher now asks smaller questions. What is one student's total? What was the class average on one quiz? Those answers come from a single row or a single column, not from the whole array. Visiting everything and then picking out the right values would work, but there is a simpler idea.

Fix one index and loop the other. Every element of row 1 has a first index of 1, so a loop that keeps the row at 1 and lets the column run from 0 to the end visits exactly row 1. Every element of column 2 has a second index of 2, so a loop that keeps the column at 2 and lets the row run visits exactly column 2. Either way you need **one** loop, not two.

Region	What stays fixed	Loop variable runs over	Element
Row row	the row index	c from 0 to grid[row].length - 1	grid[row][c]
Column col	the column index	r from 0 to grid.length - 1	grid[r][col]

The bound matters. A row has as many elements as there are **columns**, so a row loop stops at grid[row].length. A column has as many elements as there are **rows**, so a column loop stops at grid.length.

KEY IDEA

To work on one row, keep the row index fixed and loop the column index up to the row's length. To work on one column, keep the column index fixed and loop the row index up to grid.length.

WORKED EXAMPLE One student's total and one quiz's average

This is the same quiz array as in 4.13.1. Lines 4–10 work on row 1 (the second student); lines 12–19 work on column 2 (the third quiz).

```

1  int[][] scores = { {8, 10, 7, 9},
2                    {6, 9, 10, 8},
3                    {10, 7, 7, 5} };
4  int student = 1;
5  int rowTotal = 0;
6  for (int c = 0; c < scores[student].length; c++)
7  {
8      rowTotal += scores[student][c];
9  }
10 System.out.println("Student 1 total: " + rowTotal);
11
12 int quiz = 2;
13 int colTotal = 0;
14 for (int r = 0; r < scores.length; r++)
15 {
16     colTotal += scores[r][quiz];
17 }
18 double quizAverage = (double) colTotal / scores.length;
19 System.out.println("Quiz 2 average: " + quizAverage);

```

Row 1. `student` is 1 and never changes. Line 6 runs `c` from 0 while `c < scores[1].length`, which is 4.

c	scores[1][c]	rowTotal
0	6	6
1	9	15
2	10	25
3	8	33

Column 2. `quiz` is 2 and never changes. Line 14 runs `r` from 0 while `r < scores.length`, which is 3.

r	scores[r][2]	colTotal
0	7	7
1	10	17
2	7	24

The column has 3 elements, one per student, so line 18 divides by `scores.length`: `24.0 / 3` is `8.0`.

```

Student 1 total: 33
Quiz 2 average: 8.0

```

A region does not have to be a whole row or column. Any rectangle works: choose where each loop **starts** and where it **stops**. Here the teacher wants the points from the second half of the unit,

quizzes 2 and 3, for every student. The row loop still covers all rows; the column loop starts at 2 instead of 0.

```

1 int[][] scores = { {8, 10, 7, 9},
2                   {6, 9, 10, 8},
3                   {10, 7, 7, 5} };
4 int secondHalf = 0;
5 for (int r = 0; r < scores.length; r++)
6 {
7     for (int c = 2; c < scores[0].length; c++)
8     {
9         secondHalf += scores[r][c];
10    }
11 }
12 System.out.println("Quizzes 2 and 3 total: " + secondHalf);

```

The elements visited are 7 and 9 from row 0, 10 and 8 from row 1, and 7 and 5 from row 2. They add to 46, so the output is Quizzes 2 and 3 total: 46. Changing a loop's starting value or its bound is how you aim an algorithm at any rectangular part of the array.

WATCH OUT

Using the wrong length for a column loop. It feels natural to write "columns" when you are working on a column, so a student writes `scores[0].length`:

```

1 int[][] scores = { {8, 10, 7, 9},
2                   {6, 9, 10, 8},
3                   {10, 7, 7, 5} };
4 int quiz = 2;
5 int colTotal = 0;
6 for (int r = 0; r < scores[0].length; r++)
7 {
8     colTotal += scores[r][quiz];
9 }
10 System.out.println("Quiz 2 total: " + colTotal);

```

`scores[0].length` is 4, the number of **columns**, but a column holds one element per **row**, and there are only 3 rows. The passes with `r = 0, 1, and 2` work. Then `r` becomes 3, `3 < 4` is true, and line 8 asks for `scores[3][2]`. There is no row 3, so nothing is printed and the program stops with an exception like this (wording can vary between Java versions):

```
java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
```

The fix is `r < scores.length`. Ask yourself which index the loop variable stands for: if it is the row index, its bound is the number of rows.

PAUSE AND TRY

Question 3 Tracing

Write the exact output.

```
1 int[][] grid = { {2, 4, 6},
2                 {1, 3, 5},
3                 {7, 8, 9} };
4 int down = 0;
5 for (int r = 0; r < grid.length; r++)
6 {
7     down += grid[r][1];
8 }
9 int across = 0;
10 for (int c = 0; c < grid[2].length; c++)
11 {
12     across += grid[2][c];
13 }
14 System.out.println(down);
15 System.out.println(across);
```

Question 4 Multiple choice

table is a rectangular 2D array of `int`, and `k` is a valid row index. Which loop adds every element of row `k` to `sum` (which starts at 0)?

- A. `for (int c = 0; c < table.length; c++) { sum += table[k][c]; }`
- B. `for (int c = 0; c < table[k].length; c++) { sum += table[k][c]; }`
- C. `for (int c = 0; c < table[k].length; c++) { sum += table[c][k]; }`
- D. `for (int r = 0; r < table.length; r++) { sum += table[r][k]; }`

4.13.3 Finding the Minimum or Maximum

A weather app stores daily high temperatures: one row per city, one column per day. Which reading was the hottest, and in which city on which day? This is the maximum algorithm from Topic 4.5, with two changes: it visits a 2D region, and it remembers a **row** and a **column** for the location instead of a single index.

The plan: start with a "best so far" equal to an element that really is in the region, then compare every element with it and replace it whenever you find something better. Starting with a real element guarantees the answer is a value from the array.

KEY IDEA

To find a maximum (or minimum), start the best-so-far at an element of the region you are searching, such as `grid[0][0]` for the whole array or `grid[0][col]` for one column, and replace it inside the loops whenever an element is larger (or smaller). To remember where it was, store `r` and `c` at the same moment.

WORKED EXAMPLE The hottest reading and where it happened

Line 4 starts `highest` at the first element. Lines 13–15 run only when a new record is found, and they update all three variables together.

```

1  int[][] temps = { {58, 61, 55, 60},
2                    {52, 49, 57, 63},
3                    {66, 59, 50, 62} };
4  int highest = temps[0][0];
5  int bestRow = 0;
6  int bestCol = 0;
7  for (int r = 0; r < temps.length; r++)
8  {
9      for (int c = 0; c < temps[0].length; c++)
10     {
11         if (temps[r][c] > highest)
12         {
13             highest = temps[r][c];
14             bestRow = r;
15             bestCol = c;
16         }
17     }
18 }
19 System.out.println("Highest: " + highest);
20 System.out.println("City " + bestRow + ", day " + bestCol);

```

Only the passes that change something are listed. Every other comparison on line 11 is false.

r	c	temps[r][c]	Bigger than highest?	highest	bestRow, bestCol
(start)				58	0, 0
0	1	61	yes	61	0, 1
1	3	63	yes	63	1, 3
2	0	66	yes	66	2, 0

```

Highest: 66
City 2, day 0

```

The first pass compares `temps[0][0]` with itself; `58 > 58` is false, so nothing happens. That is harmless. The comparison uses `>`, so if two readings tied for the highest, the location of the **first** one in row-major order would be kept.

Now vary the region and the direction: the coldest reading on day 2, which is column 2. The search starts at `temps[0][2]`, the first element **of that column**, and flips `>` to `<`.

```
1 int[][] temps = { {58, 61, 55, 60},
2                   {52, 49, 57, 63},
3                   {66, 59, 50, 62} };
4 int day = 2;
5 int lowest = temps[0][day];
6 for (int r = 1; r < temps.length; r++)
7 {
8     if (temps[r][day] < lowest)
9     {
10         lowest = temps[r][day];
11     }
12 }
13 System.out.println("Lowest on day 2: " + lowest);
```

lowest starts at 55 (city 0). Because row 0 has already been used to start, the loop begins at `r = 1`. City 1 has 57, which is not smaller. City 2 has 50, which is, so `lowest` becomes 50. The output is `Lowest on day 2: 50`. Starting the loop at 1 or at 0 gives the same answer; starting at 1 just skips comparing the first element with itself.

WATCH OUT

Starting the best-so-far at 0. Zero looks like a safe "nothing yet" value, and it often gives the right answer, which is why the mistake survives. Try it with winter temperatures:

```
1 int[][] winter = { {-5, -12, -3},
2                   {-8, -1, -6} };
3 int highest = 0;
4 for (int[] row : winter)
5 {
6     for (int temp : row)
7     {
8         if (temp > highest)
9         {
10             highest = temp;
11         }
12     }
13 }
14 System.out.println("Highest: " + highest);
```

This prints `Highest: 0`, but 0 is not in the array. Every reading is below 0, so no comparison `temp > 0` is ever true and `highest` never changes. The real answer is `-1`. The fix is to start at a real element: `int highest = winter[0][0];` makes the program print `Highest: -1`. (Starting at `Integer.MIN_VALUE` from Topic 1.5 also works for a maximum, because no `int` is smaller.)

PAUSE AND TRY

Question 5 Tracing

Each row holds one store's prices. Write the exact output.

```

1 int[][] prices = { {12, 15, 11, 14},
2                   {7, 4, 9, 4} };
3 int store = 1;
4 int cheapest = prices[store][0];
5 int where = 0;
6 for (int c = 1; c < prices[store].length; c++)
7 {
8     if (prices[store][c] < cheapest)
9     {
10         cheapest = prices[store][c];
11         where = c;
12     }
13 }
14 System.out.println(cheapest + " at " + where);

```

Question 6 Short answer

A student finds the cheapest price in the whole `prices` array from Question 5 with the code below. It prints `0`. Explain why, and state what the starting value should be.

```

1 int[][] prices = { {12, 15, 11, 14},
2                   {7, 4, 9, 4} };
3 int cheapest = 0;
4 for (int[] row : prices)
5 {
6     for (int price : row)
7     {
8         if (price < cheapest)
9         {
10             cheapest = price;
11         }
12     }
13 }
14 System.out.println(cheapest);

```

4.13.4 Checking Whether Any or All Elements Qualify

A theater stores its seating chart as a 2D array of names, with `"empty"` for an unsold seat. Is there at least one empty seat? Did every student score at least 7 on a quiz? Each question has a yes-or-no answer about many elements at once, and the program can only be sure after it has looked at all of them.

You met the tool for this in Topic 4.5 (and Topic 2.10): a **flag**, a `boolean` variable that starts with one value before the loop and is switched to the other value inside the loop when something you are watching for happens. In a 2D array the flag works the same way; it just sits outside **both** loops.

Question	Flag starts	Switch it when an element...	Switched back?
Does at least one have it?	<code>false</code>	has the property	Never
Do all of them have it?	<code>true</code>	does not have it	Never

KEY IDEA

Declare and set the flag once, before all the loops. Switch it inside the loops when one element decides the answer, and read it only after the loops end.

WORKED EXAMPLE Is there an empty seat?

The seats are `String` objects, so line 9 compares with `equals`, not `==` (Topic 2.6). Look for the single line that changes the flag.

```
1 String[][] seats = { {"Ana", "Raj", "Lee"},
2                     {"Mo", "empty", "Kai"},
3                     {"Sam", "Eve", "Zoe"} };
4 boolean anyEmpty = false;
5 for (String[] row : seats)
6 {
7     for (String name : row)
8     {
9         if (name.equals("empty"))
10        {
11            anyEmpty = true;
12        }
13    }
14 }
15 System.out.println("Any empty seat: " + anyEmpty);
```

The outer enhanced loop hands over one row at a time; the inner one hands over one name.

Row visited	Names checked	"empty" found?	anyEmpty after the row
0	Ana, Raj, Lee	no	false
1	Mo, empty, Kai	yes	true
2	Sam, Eve, Zoe	no	true

```
Any empty seat: true
```

In row 2 the condition is false for every name, but there is no `else`, so nothing touches the flag. Once it is true, it stays true.

Now ask the "all" question about one column: did every student score at least 7 on quiz 3? The flag starts `true` ("no failure seen yet") and switches to `false` the moment one score is below 7. Because it is one column, a single loop over the rows is enough, as in 4.13.2.

```

1  int[][] scores = { {8, 10, 7, 9},
2                      {6, 9, 10, 8},
3                      {10, 7, 7, 5} };
4  int quiz = 3;
5  boolean allPassed = true;
6  for (int r = 0; r < scores.length; r++)
7  {
8      if (scores[r][quiz] < 7)
9      {
10         allPassed = false;
11     }
12 }
13 System.out.println("Everyone passed quiz 3: " + allPassed);

```

Column 3 holds 9, 8, and 5. The 9 and 8 leave the flag alone; at 5, `5 < 7` is true and `allPassed` becomes false. The output is `Everyone passed quiz 3: false`. With `quiz = 1`, the column holds 10, 9, and 7; no score is below 7, the flag is never switched, and the answer would be `true`.

WATCH OUT

Resetting the flag for every row. Putting `anyEmpty = false;` at the top of the outer loop looks like a fresh start for each row:

```

1  String[][] seats = { {"Ana", "Raj", "Lee"},
2                      {"Mo", "empty", "Kai"},
3                      {"Sam", "Eve", "Zoe"} };
4  boolean anyEmpty = false;
5  for (int r = 0; r < seats.length; r++)
6  {
7      anyEmpty = false;
8      for (int c = 0; c < seats[r].length; c++)
9      {
10         if (seats[r][c].equals("empty"))
11         {
12             anyEmpty = true;
13         }
14     }
15 }
16 System.out.println("Any empty seat: " + anyEmpty);

```

This prints `Any empty seat: false`, even though row 1 has an empty seat. The flag does become true during row 1, but line 7 runs again at the start of row 2 and wipes it out. Row 2 has no empty seat, so the final value describes **only the last row**. Students write this because resetting per row is correct when you want a separate answer for each row (you will see that in 4.13.5). For one answer about the whole array, delete line 7; the flag is set once, on line 4.

PAUSE AND TRY

Question 7 Tracing

Write the exact output.

```
1 int[][] grid = { {4, 7, 2},
2                 {6, 8, 10},
3                 {3, 12, 5} };
4 boolean rowCheck = true;
5 for (int c = 0; c < grid[1].length; c++)
6 {
7     if (grid[1][c] % 2 != 0)
8     {
9         rowCheck = false;
10    }
11 }
12 boolean colCheck = true;
13 for (int r = 0; r < grid.length; r++)
14 {
15     if (grid[r][0] % 2 != 0)
16     {
17         colCheck = false;
18     }
19 }
20 System.out.println(rowCheck + " " + colCheck);
```

Question 8 Multiple choice

You want `allSold` to be true only if **no** seat in the whole `seats` array holds "empty". Which plan is correct?

- A. Start `allSold` at false; set it to true when a seat does not equal "empty".
- B. Start `allSold` at true; set it to false when a seat equals "empty".
- C. Start `allSold` at true; set it to false when a seat equals "empty", and back to true otherwise.
- D. At the start of each row, set `allSold` to true; set it to false when a seat equals "empty".

4.13.5 Counting Elements in a Region

A flag tells you **whether** something happens. Often you need **how many**: how many days it rained each week, or how many X's sit on a diagonal of a game board. The **counter** from Topic 4.5 answers that: an `int` set to 0, increased by 1 each time an element meets the condition, and read after the loop.

In a 2D array, *where* you set the counter to 0 decides what it counts. Declared before both loops, it counts across the whole array. Declared at the top of the outer loop, it starts over for each row, so you get one count per row.

KEY IDEA

A counter set to 0 before all the loops counts the whole region. A counter set to 0 at the top of the outer loop gives a separate count for each row; print or use it before the next row starts.

WORKED EXAMPLE Rainy days in each week

Each row is a week and each column a day; the values are millimeters of rain. Line 7 creates a fresh counter for each week. Line 4 creates one grand total for the whole array.

```

1  int[][] rain = { {0, 3, 0, 0, 5},
2                  {2, 4, 1, 0, 0},
3                  {0, 0, 0, 0, 0} };
4  int total = 0;
5  for (int r = 0; r < rain.length; r++)
6  {
7      int rainyDays = 0;
8      for (int c = 0; c < rain[r].length; c++)
9      {
10         if (rain[r][c] > 0)
11         {
12             rainyDays++;
13         }
14     }
15     System.out.println("Week " + r + ": " + rainyDays);
16     total += rainyDays;
17 }
18 System.out.println("Total rainy days: " + total);

```

r	Row	Days with rain over 0	rainyDays at line 15	total after line 16
0	0, 3, 0, 0, 5	the 3 and the 5	2	2
1	2, 4, 1, 0, 0	the 2, 4, and 1	3	5
2	0, 0, 0, 0, 0	none	0	5

```

Week 0: 2
Week 1: 3
Week 2: 0
Total rainy days: 5

```

Resetting `rainyDays` on line 7 is correct here, unlike the flag in 4.13.4, because the question is asked about each week separately. Line 16 adds each week's count to `total` before the next week begins.

Some regions are not rectangles. In a **square** 2D array (same number of rows and columns), the **main diagonal** is the line of elements from the top-left corner to the bottom-right corner:

`board[0][0]`, `board[1][1]`, `board[2][2]`, and so on. Their row and column indices are equal, so a single loop variable `i` can serve as both.

```
1 String[][] board = { {"X", "0", "X"},
2                       {"0", "X", "0"},
3                       {"0", "X", "0"} };
4 int xCount = 0;
5 for (int i = 0; i < board.length; i++)
6 {
7     if (board[i][i].equals("X"))
8     {
9         xCount++;
10    }
11 }
12 System.out.println("X on the diagonal: " + xCount);
```

```
X | 0 | X
0 | X | 0
0 | X | 0
```

The loop checks `board[0][0]` (X), `board[1][1]` (X), and `board[2][2]` (O), so the output is X on the diagonal: 2. The other diagonal, from top-right to bottom-left, keeps the row index rising while the column index falls: its elements are `board[i][board.length - 1 - i]`, which are `board[0][2]`, `board[1][1]`, and `board[2][0]`. That diagonal holds X, X, and O, so it also has 2 X's.

WATCH OUT

Forgetting the - 1 on the other diagonal. The last column index is one less than the length, but it is easy to write just `board.length - i`:

```
1 String[][] board = { {"X", "0", "X"},
2                       {"0", "X", "0"},
3                       {"0", "X", "0"} };
4 int xCount = 0;
5 for (int i = 0; i < board.length; i++)
6 {
7     if (board[i][board.length - i].equals("X"))
8     {
9         xCount++;
10    }
11 }
12 System.out.println("X on the other diagonal: " + xCount);
```

On the very first pass `i` is 0, so line 7 asks for `board[0][3]`. A 3-column row has indices 0, 1, and 2, so the program crashes before printing anything, with an exception like `java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3`. The fix is `board[i][board.length - 1 - i]`; then the program prints X on the other diagonal: 2.

PAUSE AND TRY

Question 9 Tracing

Write the exact output. (Notice which loop is on the outside.)

```

1  int[][] marks = { {5, 2, 9},
2                    {8, 6, 1},
3                    {3, 7, 6} };
4  for (int c = 0; c < marks[0].length; c++)
5  {
6      int big = 0;
7      for (int r = 0; r < marks.length; r++)
8      {
9          if (marks[r][c] >= 6)
10         {
11             big++;
12         }
13     }
14     System.out.print(big + " ");
15 }
16 System.out.println();

```

Question 10 Multiple choice

grid is a square 2D array with 5 rows and 5 columns. On pass *i* of a loop from 0 to 4, which expression reaches the diagonal that runs from the **top-right** corner to the **bottom-left** corner?

- A. `grid[i][i]`
- B. `grid[i][grid.length - i]`
- C. `grid[i][grid.length - 1 - i]`
- D. `grid[grid.length - 1][i]`

4.13.6 Consecutive Pairs in a Row or Column

A fitness tracker stores each person's daily step counts in a row. On how many days did a person walk more than the day before? Each answer compares an element with its **neighbor**, the element right next to it. Elements that sit side by side are **consecutive**, and checking each neighbor pair is the "consecutive pairs" algorithm from Topic 4.5.

In a row, the neighbor of `grid[r][c]` is `grid[r][c + 1]`, one column to the right. In a column, the neighbor of `grid[r][c]` is `grid[r + 1][c]`, one row down. Because each pair reaches one step ahead, the loop must stop **one early**: a row of 5 elements has only 4 pairs, starting at columns 0 through 3.

Pairs run...	Pair	Loop bound
across row <i>r</i>	<code>grid[r][c]</code> and <code>grid[r][c + 1]</code>	<code>c < grid[r].length - 1</code>
down column <i>c</i>	<code>grid[r][c]</code> and <code>grid[r + 1][c]</code>	<code>r < grid.length - 1</code>

KEY IDEA

To visit every pair of neighbors, compare the element at the loop index with the one at index plus 1, and stop the loop one position early (subtract 1 from the bound) so the second element of the last pair is still inside the array.

WORKED EXAMPLE Days when steps went up

Row 0 is one person's five days. Line 7 prints each pair so you can see which elements are compared.

```

1  int[][] steps = { {4200, 5100, 4900, 6000, 6000},
2                    {3000, 3500, 4100, 3900, 4500} };
3  int person = 0;
4  int increases = 0;
5  for (int c = 0; c < steps[person].length - 1; c++)
6  {
7      System.out.println(steps[person][c] + " -> " + steps[person][c + 1]);
8      if (steps[person][c + 1] > steps[person][c])
9      {
10         increases++;
11     }
12 }
13 System.out.println("Increases: " + increases);

```

The bound is `c < 5 - 1`, so `c` takes the values 0 through 3.

c	steps[0][c]	steps[0][c + 1]	Went up?	increases
0	4200	5100	yes	1
1	5100	4900	no	1
2	4900	6000	yes	2
3	6000	6000	no	2

```

4200 -> 5100
5100 -> 4900
4900 -> 6000
6000 -> 6000
Increases: 2

```

The last pair is equal, and `6000 > 6000` is false, so it does not count as an increase. Notice that each middle element appears in two pairs: once as the right side and once as the left.

Now turn the pairs sideways. In a tile-matching game, two equal tiles stacked one above the other form a match. Down column 1, the neighbor is one **row** down, so the loop runs over rows and stops at `board.length - 1`.

```

1  int[][] board = { {2, 5, 1},
2                    {2, 3, 1},
3                    {4, 3, 1},
4                    {4, 3, 2} };
5  int col = 1;
6  int matches = 0;
7  for (int r = 0; r < board.length - 1; r++)
8  {
9      if (board[r][col] == board[r + 1][col])
10     {
11         matches++;
12     }
13 }
14 System.out.println("Matching pairs in column 1: " + matches);

```

Column 1 holds 5, 3, 3, 3 from top to bottom. The pairs are (5, 3), (3, 3), and (3, 3), so the output is Matching pairs in column 1: 2. With 4 rows the loop runs $r = 0, 1, 2$; the last pair uses rows 2 and 3, the bottom row.

WATCH OUT

Forgetting to stop one early. Copying the usual "visit every element" header leaves the bound at the full length:

```

1  int[][] steps = { {4200, 5100, 4900, 6000, 6000},
2                  {3000, 3500, 4100, 3900, 4500} };
3  int person = 0;
4  int increases = 0;
5  for (int c = 0; c < steps[person].length; c++)
6  {
7      System.out.println(steps[person][c] + " -> " + steps[person][c + 1]);
8      if (steps[person][c + 1] > steps[person][c])
9      {
10         increases++;
11     }
12 }
13 System.out.println("Increases: " + increases);

```

The first four pairs print normally. Then c becomes 4, $4 < 5$ is true, and line 7 asks for `steps[0][5]`, one column past the end of the row. The program crashes with an exception like `java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5`, and `Increases:` is never printed. There is no fifth pair, because the last element has no neighbor to its right. The fix is $c < \text{steps}[\text{person}].\text{length} - 1$.

PAUSE AND TRY

Question 11 Tracing

Write the exact output.

```

1 int[][] levels = { {10, 4},
2                   {13, 9},
3                   {11, 2},
4                   {16, 8} };
5 for (int r = 0; r < levels.length - 1; r++)
6 {
7     System.out.println(levels[r + 1][0] - levels[r][0]);
8 }

```

Question 12 Multiple choice

A program must compare `grid[r][c]` with `grid[r + 1][c]` for **every** column of a rectangular 2D array. Which pair of loop headers visits every such vertical pair without an exception?

- A. `for (int r = 0; r < grid.length; r++)` and `for (int c = 0; c < grid[0].length; c++)`
- B. `for (int r = 0; r < grid.length - 1; r++)` and `for (int c = 0; c < grid[0].length; c++)`
- C. `for (int r = 0; r < grid.length; r++)` and `for (int c = 0; c < grid[0].length - 1; c++)`
- D. `for (int r = 0; r < grid.length - 1; r++)` and `for (int c = 0; c < grid[0].length - 1; c++)`

4.13.7 Detecting Duplicates

On a bingo card, no number may appear twice in a row. A class code list must not repeat a code anywhere. These are duplicate checks: does any value appear more than once?

Topic 4.5 solved this for a 1D array by comparing **every pair** of positions. The outer index `i` picks one element; the inner index `j` starts at `i + 1` and picks each element after it. Starting `j` at `i + 1` does two jobs: no element is compared with itself, and no pair is checked twice. A row or a column of a 2D array is a line of elements just like a 1D array, so the same two loops work on it once one index is fixed.

KEY IDEA

To check one row (or column) for a duplicate, fix the row (or column) index, loop `i` over its positions, loop `j` from `i + 1` to the end, and switch a flag to true when the elements at `i` and `j` are equal.

WORKED EXAMPLE Does row 0 of the bingo card repeat a number?

`row` is fixed at 0. Both loops run over column positions, so both stop at `card[row].length`.

```
1 int[][] card = { {3, 12, 7, 12},
2                 {5, 9, 1, 14} };
3 int row = 0;
4 boolean hasDuplicate = false;
5 for (int i = 0; i < card[row].length; i++)
6 {
7     for (int j = i + 1; j < card[row].length; j++)
8     {
9         if (card[row][i] == card[row][j])
10        {
11            hasDuplicate = true;
12        }
13    }
14 }
15 System.out.println("Row 0 has a duplicate: " + hasDuplicate);
```

Row 0 is {3, 12, 7, 12}. With 4 positions the pairs are:

i	j	Values compared	Equal?	hasDuplicate
0	1	3 and 12	no	false
0	2	3 and 7	no	false
0	3	3 and 12	no	false
1	2	12 and 7	no	false
1	3	12 and 12	yes	true
2	3	7 and 12	no	true

```
Row 0 has a duplicate: true
```

When `i` is 3, `j` would start at 4, which fails `j < 4` right away, so that pass of the outer loop does nothing. Row 1, {5, 9, 1, 14}, has no equal pair, so with `row = 1` the answer would be `false`.

Checking the **whole** 2D array is harder, because a repeated value can sit in two different rows. One clear way is to reuse what you know: copy every element into a 1D array in row-major order, then run the 1D duplicate check on that copy.

```

1  int[][] codes = { {41, 17, 30},
2                    {26, 8, 17} };
3  int[] all = new int[codes.length * codes[0].length];
4  int k = 0;
5  for (int[] row : codes)
6  {
7      for (int value : row)
8      {
9          all[k] = value;
10         k++;
11     }
12 }
13 boolean hasDuplicate = false;
14 for (int i = 0; i < all.length; i++)
15 {
16     for (int j = i + 1; j < all.length; j++)
17     {
18         if (all[i] == all[j])
19         {
20             hasDuplicate = true;
21         }
22     }
23 }
24 System.out.println("Any duplicate: " + hasDuplicate);

```

Lines 3–12 build `all`, a 1D array with room for $2 \times 3 = 6$ values, and fill it with an index `k` that moves forward once per element: {41, 17, 30, 26, 8, 17}. Lines 13–23 are the 1D duplicate check. The 17 at index 1 matches the 17 at index 5, so the output is `Any duplicate: true`. Notice that a row-by-row check would have said "no duplicate" for each row: {41, 17, 30} and {26, 8, 17} have no repeats on their own. The region you check changes the answer.

WATCH OUT

Starting `j` at 0. It seems thorough to compare every element with every element:

```

1  int[][] card = { {3, 12, 7, 12},
2                  {5, 9, 1, 14} };
3  int row = 1;
4  boolean hasDuplicate = false;
5  for (int i = 0; i < card[row].length; i++)
6  {
7      for (int j = 0; j < card[row].length; j++)
8      {
9          if (card[row][i] == card[row][j])
10         {
11             hasDuplicate = true;
12         }
13     }
14 }
15 System.out.println("Row 1 has a duplicate: " + hasDuplicate);

```

This prints `Row 1 has a duplicate: true`, but row 1 is {5, 9, 1, 14}, with no repeats. On the very first comparison `i` and `j` are both 0, so the code compares `card[1][0]` with **itself**, finds

them equal, and switches the flag. Every element equals itself, so this version says `true` for every row. The fix is `int j = i + 1;` with it the program prints `Row 1 has a duplicate: false.`

PAUSE AND TRY

Question 13 Tracing

Each column is a weekly schedule of who works one job. Write the exact output.

```

1 String[][] shifts = { {"Ava", "Ben"},
2                       {"Cal", "Ava"},
3                       {"Ben", "Dee"},
4                       {"Eli", "Ben"} };
5 int col = 1;
6 for (int i = 0; i < shifts.length; i++)
7 {
8     for (int j = i + 1; j < shifts.length; j++)
9     {
10         if (shifts[i][col].equals(shifts[j][col]))
11         {
12             System.out.println("rows " + i + " and " + j);
13         }
14     }
15 }
```

Question 14 Short answer

The bingo card's rows have 5 numbers each. How many times does the duplicate check in the worked example compare two elements when it checks one row? Explain how you counted.

4.13.8 Shifting and Rotating a Row or Column

A scrolling sign moves its letters one spot to the left every second, and the letter that falls off the left edge reappears on the right. In a falling-blocks game, a column of blocks drops down one spot and an empty space (0) appears at the top. Both programs move every element of one row or column by one position.

Two words describe these moves:

Move	What happens to the element pushed off the end	Example on {1, 2, 3, 4}, moving left
shift	It is lost; the empty spot at the other end gets a fill value, such as 0	{2, 3, 4, 0}
rotate	It wraps around into the empty spot at the other end	{2, 3, 4, 1}

The steps are the same as for a 1D array in Topic 4.5, applied to one fixed row or column:

1. If you are rotating, **save** the element that will be pushed off, because the first copy will overwrite it.

- Copy each element into its neighbor's spot, **starting at the end the elements move toward**, so that each element is read before it is overwritten.
- Put the saved element (rotate) or the fill value (shift) into the spot left open.

KEY IDEA

To move a row or column by one position, copy each element into the spot next to it, starting at the end the elements move toward. Save the element that falls off first if it must wrap around (rotate); otherwise fill the empty spot (shift).

WORKED EXAMPLE Rotating the sign one spot to the left

Row 0 holds S A L E !. Line 4 saves the first letter. The loop moves left to right, copying each letter from the spot on its right.

```

1 String[][] sign = { {"S", "A", "L", "E", "!"},
2                     {"0", "P", "E", "N", "?"} };
3 int row = 0;
4 String first = sign[row][0];
5 for (int c = 0; c < sign[row].length - 1; c++)
6 {
7     sign[row][c] = sign[row][c + 1];
8 }
9 sign[row][sign[row].length - 1] = first;
10 for (String letter : sign[row])
11 {
12     System.out.print(letter);
13 }
14 System.out.println();

```

Step	sign[0] afterward
line 4: first = "S"	S A L E !
c = 0: copy from 1	A A L E !
c = 1: copy from 2	A L L E !
c = 2: copy from 3	A L E E !
c = 3: copy from 4	A L E ! !
line 9: last spot = first	A L E ! S

ALE!S

Each copy reads the spot to its right **before** anything has written there, so no letter is lost, except S, which was saved on line 4. The loop stops at `c < 4` because the last spot has no right-hand neighbor to copy from. Row 1 is never touched.

Now move a column the other way and shift instead of rotate. The blocks in column 0 fall **down** one row. The new values arrive from above, so the copying starts at the **bottom** and works up; the top spot becomes 0.

```

1  int[][] grid = { {7, 1, 4},
2                  {2, 8, 5},
3                  {9, 3, 6} };
4  int col = 0;
5  for (int r = grid.length - 1; r > 0; r--)
6  {
7      grid[r][col] = grid[r - 1][col];
8  }
9  grid[0][col] = 0;
10 for (int[] row : grid)
11 {
12     for (int value : row)
13     {
14         System.out.print(value + " ");
15     }
16     System.out.println();
17 }

```

Column 0 starts as 7, 2, 9 from top to bottom. When `r` is 2, `grid[2][0]` gets the 2 from above; when `r` is 1, `grid[1][0]` gets the 7 from above. The loop stops at `r > 0`, because the top row has nothing above it. Line 9 puts 0 in the top spot. The 9 that was at the bottom is gone, which is what makes this a shift and not a rotate.

```

0 1 4
7 8 5
2 3 6

```

WATCH OUT

Copying in the wrong direction. To rotate the sign to the **right**, a student saves the last letter and copies each letter from its left neighbor, but loops left to right as in the worked example:

```

1  String[][] sign = { {"S", "A", "L", "E", "!"},
2                      {"O", "P", "E", "N", "?"} };
3  int row = 0;
4  int len = sign[row].length;
5  String last = sign[row][len - 1];
6  for (int c = 1; c < len; c++)
7  {
8      sign[row][c] = sign[row][c - 1];
9  }
10 sign[row][0] = last;
11 for (String letter : sign[row])
12 {
13     System.out.print(letter);
14 }
15 System.out.println();

```

This prints `!ssss`. At `c = 1`, the spot gets the `s` from spot 0. At `c = 2`, it copies spot 1, which **already** holds `s`, and so on down the row: the first letter is copied over every other one

before they are read. Whenever elements move right, the copying must start at the right end. The fix is `for (int c = len - 1; c > 0; c--)`, which prints `!SALE`.

PAUSE AND TRY

Question 15 Tracing

Write the exact output.

```
1 int[][] grid = { {1, 5},
2                 {2, 6},
3                 {3, 7},
4                 {4, 8} };
5 int col = 1;
6 int top = grid[0][col];
7 for (int r = 0; r < grid.length - 1; r++)
8 {
9     grid[r][col] = grid[r + 1][col];
10 }
11 grid[grid.length - 1][col] = top;
12 for (int r = 0; r < grid.length; r++)
13 {
14     System.out.println(grid[r][0] + " " + grid[r][1]);
15 }
```

Question 16 Multiple choice

What does this code print?

```
1 int[][] lights = { {0, 0, 0, 0},
2                   {6, 2, 8, 3} };
3 for (int c = 0; c < lights[1].length - 1; c++)
4 {
5     lights[1][c] = lights[1][c + 1];
6 }
7 lights[1][lights[1].length - 1] = 0;
8 for (int value : lights[1])
9 {
10     System.out.print(value + " ");
11 }
12 System.out.println();
```

- A. 2 8 3 0
- B. 2 8 3 6
- C. 0 6 2 8
- D. 3 3 3 0

4.13.9 Reversing a Row or Column

A relay coach wants to run a team's order backward, last runner first. That is a reversal: the first element trades places with the last, the second with the second-to-last, and so on, until the two ends meet in the middle.

Each trade is a **swap**, from Topic 4.5: save one value in a temporary variable, copy the other value over it, then put the saved value in the second spot. Without the temporary variable, the first copy would destroy one of the two values.

The partner of position `c` in a line of `len` elements is position `len - 1 - c`: 0 pairs with `len - 1`, 1 pairs with `len - 2`, and so on. Only the first **half** of the positions start a swap, so the loop runs while `c < len / 2`. With an odd length, integer division rounds down and the middle element stays where it is, which is exactly right.

KEY IDEA

To reverse a row or column of length `len`, swap the element at position `i` with the one at `len - 1 - i`, using a temporary variable, for `i` from 0 while `i < len / 2`.

WORKED EXAMPLE Reversing row 0 of the lineup

Row 0 holds 4 9 2 7 5, so `len` is 5 and `len / 2` is 2. Lines 7–9 are the swap.

```

1 int[][] lineup = { {4, 9, 2, 7, 5},
2                   {1, 6, 3, 8, 0} };
3 int row = 0;
4 int len = lineup[row].length;
5 for (int c = 0; c < len / 2; c++)
6 {
7     int temp = lineup[row][c];
8     lineup[row][c] = lineup[row][len - 1 - c];
9     lineup[row][len - 1 - c] = temp;
10 }
11 for (int value : lineup[row])
12 {
13     System.out.print(value + " ");
14 }
15 System.out.println();

```

c	Partner <code>len - 1 - c</code>	Values swapped	Row 0 afterward
(start)			4 9 2 7 5
0	4	4 and 5	5 9 2 7 4
1	3	9 and 7	5 7 2 9 4

5 7 2 9 4

The loop stops when `c` is 2, the middle position, whose partner is itself. Row 1 is untouched.

A column works the same way, with the loop running over rows. Here column 1 of a 4-row array is reversed, so `len` is `grid.length` and the loop index goes in the **row** position.

```

1  int[][] grid = { {1, 10},
2                  {2, 20},
3                  {3, 30},
4                  {4, 40} };
5  int col = 1;
6  int len = grid.length;
7  for (int r = 0; r < len / 2; r++)
8  {
9      int temp = grid[r][col];
10     grid[r][col] = grid[len - 1 - r][col];
11     grid[len - 1 - r][col] = temp;
12 }
13 for (int r = 0; r < len; r++)
14 {
15     System.out.println(grid[r][0] + " " + grid[r][1]);
16 }

```

With 4 rows, `len / 2` is 2, so there are two swaps: rows 0 and 3 trade 10 and 40, then rows 1 and 2 trade 20 and 30. An even length has no middle element; every element moves. Column 0 is unchanged.

```

1 40
2 30
3 20
4 10

```

WATCH OUT

Swapping all the way across. It seems natural to let the loop run over every position:

```

1  int[][] lineup = { {4, 9, 2, 7, 5},
2                    {1, 6, 3, 8, 0} };
3  int row = 0;
4  int len = lineup[row].length;
5  for (int c = 0; c < len; c++)
6  {
7      int temp = lineup[row][c];
8      lineup[row][c] = lineup[row][len - 1 - c];
9      lineup[row][len - 1 - c] = temp;
10 }
11 for (int value : lineup[row])
12 {
13     System.out.print(value + " ");
14 }
15 System.out.println();

```

This prints `4 9 2 7 5`, the original order. The first two passes reverse the row. Then, from `c = 3` on, each pass swaps a pair that was **already** swapped, which puts it back. Every pair is swapped twice, and two swaps cancel out. The fix is `c < len / 2`, so each pair is swapped exactly once.

PAUSE AND TRY**Question 17** Tracing

Write the exact output.

```
1 String[][] team = { {"P", "Q", "R"},
2                     {"S", "T", "U"},
3                     {"V", "W", "X"} };
4 int col = 2;
5 int len = team.length;
6 for (int r = 0; r < len / 2; r++)
7 {
8     String temp = team[r][col];
9     team[r][col] = team[len - 1 - r][col];
10    team[len - 1 - r][col] = temp;
11 }
12 for (String[] row : team)
13 {
14     for (String name : row)
15     {
16         System.out.print(name);
17     }
18     System.out.println();
19 }
```

Question 18 Debugging

This code should reverse row 0, printing 5 7 2 9 4. It prints 5 7 2 7 5. Find the error and write the corrected loop body.

```
1 int[][] lineup = { {4, 9, 2, 7, 5},
2                     {1, 6, 3, 8, 0} };
3 int row = 0;
4 int len = lineup[row].length;
5 for (int c = 0; c < len / 2; c++)
6 {
7     lineup[row][c] = lineup[row][len - 1 - c];
8     lineup[row][len - 1 - c] = lineup[row][c];
9 }
10 for (int value : lineup[row])
11 {
12     System.out.print(value + " ");
13 }
14 System.out.println();
```

Putting it together

A snack bar records daily sales in a 2D array: one row per week, one column per weekday, Monday (column 0) through Friday (column 4). The manager wants three answers: which week sold the most, whether Friday beat Thursday every single week, and how many days sold more

than 20 items. The program below answers all three, and it puts the row total in its own method (Topic 3.5) so the "best week" search can call it.

SnackSales.java

```
1 public class SnackSales
2 {
3     public static int rowTotal(int[][] grid, int row)
4     {
5         int total = 0;
6         for (int c = 0; c < grid[row].length; c++)
7         {
8             total += grid[row][c];
9         }
10        return total;
11    }
12
13    public static void main(String[] args)
14    {
15        // rows are weeks; columns are Monday through Friday
16        int[][] sales = { {12, 15, 9, 20, 31},
17                          {14, 11, 18, 22, 27},
18                          {10, 16, 13, 25, 35} };
19
20        int bestWeek = 0;
21        int bestTotal = rowTotal(sales, 0);
22        for (int r = 1; r < sales.length; r++)
23        {
24            int total = rowTotal(sales, r);
25            if (total > bestTotal)
26            {
27                bestTotal = total;
28                bestWeek = r;
29            }
30        }
31        System.out.println("Best week: " + bestWeek + " with " + bestTotal);
32
33        int thursday = 3;
34        boolean fridayAlwaysUp = true;
35        for (int r = 0; r < sales.length; r++)
36        {
37            if (sales[r][thursday + 1] <= sales[r][thursday])
38            {
39                fridayAlwaysUp = false;
40            }
41        }
42        System.out.println("Friday beat Thursday every week: " + fridayAlwaysUp);
43
44        int busyDays = 0;
45        for (int[] week : sales)
46        {
47            for (int amount : week)
48            {
49                if (amount > 20)
50                {
51                    busyDays++;
52                }
53            }
54        }
55    }
56 }
```

```

52         }
53     }
54 }
55     System.out.println("Days over 20: " + busyDays);
56 }
57 }

```

Lines 3–11: a row total as a method. `rowTotal` is the one-row algorithm from 4.13.2: the row stays fixed and `c` runs across it. It receives a reference to the array (Topic 3.6) and a row number, and returns that row's sum. It is `static` so `main` can call it without an object (Topic 3.7).

Lines 20–31: the best week. This is the maximum algorithm from 4.13.3, but the values being compared are **row totals**, not single elements. `bestTotal` starts at a real total, week 0's, and the loop starts at week 1.

r	rowTotal(sales, r)	Bigger?	bestTotal	bestWeek
(start)	12 + 15 + 9 + 20 + 31 = 87		87	0
1	14 + 11 + 18 + 22 + 27 = 92	yes	92	1
2	10 + 16 + 13 + 25 + 35 = 99	yes	99	2

Lines 33–42: Friday versus Thursday. This is an "all" flag from 4.13.4 applied to a neighbor pair from 4.13.6: in each row it compares column 3 with column 4. The flag switches to `false` only if Friday is not greater. The pairs are 20 and 31, 22 and 27, 25 and 35; Friday wins all three, so the flag stays `true`.

Lines 44–55: busy days. A counter from 4.13.5 over the whole array with enhanced loops, since positions do not matter here. The values over 20 are 31, 22, 27, 25, and 35, so `busyDays` is 5. (The 20 in week 0 does not count; `20 > 20` is false.)

```

Best week: 2 with 99
Friday beat Thursday every week: true
Days over 20: 5

```

Every part of the program comes down to the same two decisions you made throughout this chapter: **which elements** to visit (all, one row, one column, or a pair of columns) and **what to do** with each one (add it, compare it with a best-so-far, switch a flag, or count it).

Vocabulary

Term	Definition	Example
accumulator	A variable that collects a result, such as a running total, across the passes of a loop	<code>total += grid[r][c];</code>
flag	A <code>boolean</code> set before the loops and switched inside them when one element decides the answer	<code>anyEmpty = true;</code>
counter	An <code>int</code> set to 0 and increased by 1 each time an element meets a condition	<code>rainyDays++;</code>

main diagonal	In a square 2D array, the elements whose row and column indices are equal	<code>board[i][i]</code>
consecutive	Side by side: neighbors in the same row (<code>c</code> and <code>c + 1</code>) or the same column (<code>r</code> and <code>r + 1</code>)	<code>steps[0][c]</code> and <code>steps[0][c + 1]</code>
shift	Moving every element of a row or column one spot, losing the end element and filling the open spot	<code>{6, 2, 8, 3} → {2, 8, 3, 0}</code>
rotate	Moving every element one spot, with the element pushed off one end wrapping around to the other end	<code>S A L E ! → A L E ! S</code>
swap	Exchanging two elements by saving one in a temporary variable first	<code>int temp = row[0];</code>

Chapter summary

- A 2D algorithm makes two choices: **which elements** to visit and **what to do** with each. The 1D algorithms from Topic 4.5 supply the "what"; the loops supply the "which."
- The whole array needs nested loops. One row keeps the row index fixed and loops the column up to the row's length; one column keeps the column index fixed and loops the row up to `grid.length`. Changing a loop's start or bound aims the algorithm at any rectangle, and `[i][i]` reaches the main diagonal of a square array.
- Totals start at 0, and an average casts to `double` before dividing. A minimum or maximum starts at an element of the region being searched, never at 0, and can store `r` and `c` to remember where the best value was.
- A **flag** answers "at least one?" (start `false`) or "all?" (start `true`) and is never switched back; a **counter** answers "how many?". Where you set either one to its starting value decides whether you get one answer for the whole array or one per row.
- Neighbor pairs stop one early. Duplicate checks compare each position with every later one (`j` starts at `i + 1`). Shifting and rotating start copying at the end the values move toward; reversing swaps with a temporary variable for only the first half of the positions.

Review questions

Question 19 Tracing

Write the exact output.

```
1 int[][] grid = { {4, 1, 6},
2                 {2, 8, 3},
3                 {5, 0, 7} };
4 int bestCol = 0;
5 int bestSum = Integer.MIN_VALUE;
6 for (int c = 0; c < grid[0].length; c++)
7 {
8     int sum = 0;
9     for (int r = 0; r < grid.length; r++)
10    {
11        sum += grid[r][c];
12    }
13    System.out.println("Column " + c + ": " + sum);
14    if (sum > bestSum)
15    {
16        bestSum = sum;
17        bestCol = c;
18    }
19 }
20 System.out.println("Largest: column " + bestCol);
```

Question 20 Multiple choice

After this code runs, what does `count` represent, and what is printed?

```
1 int[][] grid = { {3, -1, 4},
2                 {5, 9, 2},
3                 {-6, 5, -3} };
4 int count = 0;
5 for (int r = 0; r < grid.length; r++)
6 {
7     boolean found = false;
8     for (int c = 0; c < grid[r].length; c++)
9     {
10        if (grid[r][c] < 0)
11        {
12            found = true;
13        }
14    }
15    if (found)
16    {
17        count++;
18    }
19 }
20 System.out.println(count);
```

- A. The number of negative elements in the array; 3 is printed
- B. The number of rows that contain at least one negative element; 2 is printed
- C. The number of rows in which every element is negative; 0 is printed
- D. Whether the last row contains a negative element; 1 is printed

Question 21 Debugging

This code should print the largest value in column 1, which is 12. It prints 50. Find the error and write the corrected line.

```

1  int[][] grid = { {50, 3, 8},
2                  {6, 12, 4},
3                  {9, 7, 1} };
4  int col = 1;
5  int largest = grid[0][0];
6  for (int r = 0; r < grid.length; r++)
7  {
8      if (grid[r][col] > largest)
9      {
10         largest = grid[r][col];
11     }
12 }
13 System.out.println(largest);

```

Question 22 Tracing

Write the exact output.

```

1  int[][] grid = { {1, 2, 3},
2                  {4, 5, 6},
3                  {7, 8, 9} };
4  int first = grid[1][0];
5  for (int c = 0; c < grid[1].length - 1; c++)
6  {
7      grid[1][c] = grid[1][c + 1];
8  }
9  grid[1][grid[1].length - 1] = first;
10
11 int len = grid.length;
12 for (int r = 0; r < len / 2; r++)
13 {
14     int temp = grid[r][2];
15     grid[r][2] = grid[len - 1 - r][2];
16     grid[len - 1 - r][2] = temp;
17 }
18 for (int[] row : grid)
19 {
20     for (int value : row)
21     {
22         System.out.print(value + " ");
23     }
24     System.out.println();
25 }

```

Question 23 Short answer

A student wants to know whether a 2D array contains any repeated value. She checks every row for duplicates and every column for duplicates, finds none, and concludes that no value repeats. Explain why her conclusion can be wrong, using the array `{ {1, 2}, {2, 1} }`, and describe a check that would be correct.

Question 24 Multiple choice

Assume `grid` is a rectangular 2D array of `int` and `col` is a valid column index. Which segments reverse the order of the elements in column `col`?

```
1 // I
2 for (int r = 0; r < grid.length / 2; r++)
3 {
4     int temp = grid[r][col];
5     grid[r][col] = grid[grid.length - 1 - r][col];
6     grid[grid.length - 1 - r][col] = temp;
7 }
8
9 // II
10 int[] copy = new int[grid.length];
11 for (int r = 0; r < grid.length; r++)
12 {
13     copy[r] = grid[grid.length - 1 - r][col];
14 }
15 for (int r = 0; r < grid.length; r++)
16 {
17     grid[r][col] = copy[r];
18 }
19
20 // III
21 for (int r = 0; r < grid.length; r++)
22 {
23     int temp = grid[r][col];
24     grid[r][col] = grid[grid.length - 1 - r][col];
25     grid[grid.length - 1 - r][col] = temp;
26 }
```

- A. I only
- B. II only
- C. I and II only
- D. I, II, and III

Question 25 Multiple choice

Assume `grid` is a rectangular 2D array of `int`. After this segment runs, when is `inOrder` true?

```
1 boolean inOrder = true;
2 for (int r = 0; r < grid.length; r++)
3 {
4     for (int c = 0; c < grid[r].length - 1; c++)
5     {
6         if (grid[r][c] > grid[r][c + 1])
7         {
8             inOrder = false;
9         }
10    }
11 }
```

- A. When every row's values never decrease from left to right
- B. When at least one row's values never decrease from left to right
- C. When every column's values never decrease from top to bottom
- D. When the last row's values never decrease from left to right

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can total and average every element of a 2D array without losing the fraction. Reread: 4.13.1
Totals and Averages for the Whole Array
- ☐ I can aim an algorithm at one row, one column, or a rectangle by choosing which index stays fixed and where each loop starts and stops. Reread: 4.13.2 One Row, One Column, or a Region
- ☐ I can find a minimum or maximum in any region, start it at the right element, and remember its row and column. Reread: 4.13.3 Finding the Minimum or Maximum
- ☐ I can use a flag to decide whether at least one element, or every element, in a region has a property. Reread: 4.13.4 Checking Whether Any or All Elements Qualify
- ☐ I can count matching elements for the whole array, for each row, or along a diagonal. Reread: 4.13.5 Counting Elements in a Region
- ☐ I can visit every neighbor pair across a row or down a column without going out of bounds. Reread: 4.13.6 Consecutive Pairs in a Row or Column
- ☐ I can check a row, a column, or a whole 2D array for a repeated value. Reread: 4.13.7 Detecting Duplicates
- ☐ I can shift or rotate a row or column and choose the direction to copy so nothing is overwritten too early. Reread: 4.13.8 Shifting and Rotating a Row or Column
- ☐ I can reverse a row or column with swaps and explain why the loop stops halfway. Reread: 4.13.9 Reversing a Row or Column

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

18
3.0

The rows add to $3 + (-2) = 1$, then $5 + 1 = 6$, then $4 + 7 = 11$, so `sum` is $1 + 6 + 11 = 18$. The array has 3 rows and 2 columns, so there are 6 elements. The cast makes `(double) sum` equal to 18.0, and $18.0 / 6$ is 3.0. Because the division is done with a `double`, Java prints 3.0, not 3.

Reread 4.13.1 *Totals and Averages for the Whole Array* if your output differed.

Question 2

D. Each of the 5 rows holds 8 elements, so there are $5 \times 8 = 40$ elements, which is `grid.length * grid[0].length`.

- **A** is wrong: `grid.length` is the number of rows, 5. A 2D array is an array of rows, so its own length counts rows, not elements.
- **B** is wrong: `grid[0].length` is the length of one row, the number of columns, 8.
- **C** is wrong: adding gives 13, which counts one row plus one column. Every row contains a full set of columns, so you multiply.

Reread 4.13.1 *Totals and Averages for the Whole Array* if you chose differently.

Question 3

15
24

The first loop keeps the column at 1 and runs `r` through 0, 1, 2, so it adds `grid[0][1]`, `grid[1][1]`, and `grid[2][1]`: $4 + 3 + 8 = 15$. That is column 1, read down. The second loop keeps the row at 2 and runs `c` through 0, 1, 2, so it adds $7 + 8 + 9 = 24$. That is row 2, read across.

Reread 4.13.2 *One Row, One Column, or a Region* if your output differed.

Question 4

B. Row `k` keeps the first index fixed at `k`, and the column index runs from 0 up to the row's length, `table[k].length`.

- **A** is wrong: `table.length` is the number of rows. If there are more rows than columns it crashes with an `ArrayIndexOutOfBoundsException`; if there are fewer, it skips the end of the row.
- **C** is wrong: `table[c][k]` puts the loop variable in the row position, so it walks down column `k` instead of across row `k` (and can go out of bounds).
- **D** is wrong: it adds column `k`, not row `k`. The first index is the row, so fixing the **second** index picks a column.

Reread 4.13.2 *One Row, One Column, or a Region* if you chose differently.

Question 5

4 at 1

The search looks only at row 1: {7, 4, 9, 4}. `cheapest` starts at 7 and `where` at 0.

c	prices[1][c]	Smaller than cheapest?	cheapest	where
1	4	yes	4	1
2	9	no	4	1
3	4	no (4 < 4 is false)	4	1

The 4 at index 3 ties but is not **smaller**, so the first location, 1, is kept.

Reread 4.13.3 *Finding the Minimum or Maximum* if your output differed.

Question 6

Every price is greater than 0, so the condition `price < cheapest` is never true and `cheapest` keeps its starting value, 0, which is not a price in the array. The correct answer is 4.

A complete answer says:

- The best-so-far must start at a value that is actually in the region, such as `prices[0][0]`.
- Starting at 0 only works by luck: for a minimum it fails whenever all values are positive (for a maximum, whenever all are negative).
- With `int cheapest = prices[0][0]`; the program prints 4.

Reread 4.13.3 *Finding the Minimum or Maximum* if your explanation differed.

Question 7

true false

`rowCheck` asks whether every element of row 1 is even. Row 1 is {6, 8, 10}; none of them has a remainder when divided by 2, so the flag is never switched and stays `true`.

`colCheck` asks the same about column 0, which holds 4, 6, and 3. At 3, `3 % 2 != 0` is true, so `colCheck` becomes `false`.

Reread 4.13.4 *Checking Whether Any or All Elements Qualify* if your output differed.

Question 8

B. "Every seat is sold" is an "all" question. Assume yes (`true`) and let one empty seat switch it to `false` for good.

- **A** is wrong: the first sold seat switches it to `true`, and nothing switches it back, so one sold seat would make the answer true even with empty seats elsewhere. That is the "at least one" pattern aimed at the wrong question.
- **C** is wrong: the `else` makes the flag describe only the last seat checked.
- **D** is wrong: resetting at each row means the final value describes only the last row.

Reread 4.13.4 *Checking Whether Any or All Elements Qualify* if you chose differently.

Question 9

1 2 2

The outer loop runs over columns, so this is column-major order (Topic 4.12), and `big` starts over for each column.

c	Column values	Values of at least 6	big printed
0	5, 8, 3	8	1
1	2, 6, 7	6, 7	2
2	9, 1, 6	9, 6	2

The condition is `>= 6`, so the 6 values count. Each count is printed with `print` and a space, so they appear on one line.

Reread 4.13.5 *Counting Elements in a Region* if your output differed.

Question 10

C. Going down this diagonal, the row index rises 0, 1, 2, 3, 4 while the column index falls 4, 3, 2, 1, 0. Since `grid.length - 1` is 4, the column is `4 - i`.

- **A** is wrong: `grid[i][i]` is the main diagonal, from top-left to bottom-right.
- **B** is wrong: when `i` is 0 it asks for column 5, which does not exist, so it throws an `ArrayIndexOutOfBoundsException`. The last index is one less than the length.
- **D** is wrong: the row is always `grid.length - 1`, so this walks across the bottom row.

Reread 4.13.5 *Counting Elements in a Region* if you chose differently.

Question 11

```
3
-2
5
```

The loop goes down column 0 (10, 13, 11, 16) and prints each element minus the one above it. There are 4 rows, so `r` runs from 0 to 2: $13 - 10 = 3$, $11 - 13 = -2$, $16 - 11 = 5$. Column 1 is never used.

Reread 4.13.6 *Consecutive Pairs in a Row or Column* if your output differed.

Question 12

B. Only the **row** index reaches ahead (`r + 1`), so only the row loop stops one early. The column index is used as it is, so the column loop covers every column.

- **A** is wrong: when `r` is the last row, `grid[r + 1][c]` is past the bottom and throws an `ArrayIndexOutOfBoundsException`.
- **C** is wrong: it stops the wrong loop early. It skips the last column and still goes past the bottom row.
- **D** is wrong: it runs safely but skips every pair in the last column. Subtract 1 only from the bound of the index that has `+ 1` added.

Reread 4.13.6 *Consecutive Pairs in a Row or Column* if you chose differently.

Question 13

```
rows 0 and 3
```

`col` is fixed at 1, and both loops run over row positions. Column 1 holds Ben, Ava, Dee, Ben. The pairs checked are (0,1), (0,2), (0,3), (1,2), (1,3), (2,3). Only rows 0 and 3 both hold "Ben". Ava appears twice in the array, but once in each column, so a check of column 1 does not see it.

Reread 4.13.7 *Detecting Duplicates* if your output differed.

Question 14

10 comparisons. When `i` is 0, `j` runs 1–4 (4 comparisons); when `i` is 1, `j` runs 2–4 (3); when `i` is 2, 2 comparisons; when `i` is 3, 1; when `i` is 4, none. $4 + 3 + 2 + 1 = 10$.

A complete answer says:

- `j` starts at `i + 1`, so each pass of the outer loop compares with only the elements after position `i`.
- Each pair of positions is compared exactly once, and no element is compared with itself.

Reread 4.13.7 *Detecting Duplicates* if your count differed.

Question 15

```
1 6
2 7
3 8
4 5
```

Only column 1 moves. It starts as 5, 6, 7, 8. Line 6 saves `top = 5`. The loop copies up: row 0 gets 6, row 1 gets 7, row 2 gets 8. Then the bottom row gets the saved 5. This is a rotate **up**: the top element wraps around to the bottom. Column 0 is unchanged.

Reread 4.13.8 *Shifting and Rotating a Row or Column* if your output differed.

Question 16

A. The loop copies each element of row 1 from its right-hand neighbor, moving left to right, so {6, 2, 8, 3} becomes {2, 8, 3, 3}. Then the last spot is set to 0. The 6 that was pushed off the left end is lost: this is a shift left.

- **B** is wrong: that is a rotate. The code never saves the 6, so it cannot wrap around.
- **C** is wrong: that would be a shift right. Each spot copies from `c + 1`, the spot to its **right**, so the values move left.
- **D** is wrong: copying left to right reads each right-hand neighbor before it is overwritten, so the values are not smeared. Smearing happens when the copying starts at the end the values move away from.

Reread 4.13.8 *Shifting and Rotating a Row or Column* if you chose differently.

Question 17

```
PQX
STU
VWR
```

Column 2 holds R, U, X. `len` is 3, so `len / 2` is 1 and there is one swap: rows 0 and 2 trade R and X. The middle element, U, stays. The other columns are unchanged.

Reread 4.13.9 *Reversing a Row or Column* if your output differed.

Question 18

The error is in lines 7–8: the swap has no temporary variable. Line 7 copies the partner's value over `lineup[row][c]`, so the original value is gone. Line 8 then copies that same value back, so both spots end up with the partner's value. On the first pass both ends become 5; on the second, both inner spots become 7.

Corrected loop body:

```
1 int temp = lineup[row][c];
2 lineup[row][c] = lineup[row][len - 1 - c];
3 lineup[row][len - 1 - c] = temp;
```

Saving the value first means it still exists when the second spot needs it. With this body the program prints 5 7 2 9 4.

Reread 4.13.9 *Reversing a Row or Column* if you missed the error.

Question 19

```
Column 0: 11
Column 1: 9
Column 2: 16
Largest: column 2
```

The outer loop runs over columns, and `sum` starts over for each one, so this computes a total for each column: $4 + 2 + 5 = 11$, $1 + 8 + 0 = 9$, $6 + 3 + 7 = 16$. At the same time it runs the maximum algorithm on those totals. `bestSum` starts at `Integer.MIN_VALUE`, so the first column total always replaces it: after column 0 it is 11 (column 0). Column 1's 9 is not bigger. Column 2's 16 is, so `bestCol` becomes 2.

Reread 4.13.2 *One Row, One Column, or a Region* and 4.13.3 *Finding the Minimum or Maximum* if your output differed.

Question 20

B. `found` is reset to `false` at the start of each row, switched to `true` if that row has a negative, and checked after the inner loop. So `count` goes up once for each **row** that has at least one negative. Rows 0 and 2 do; row 1 does not, so 2 is printed.

- **A** is wrong: there are 3 negative elements, but row 2 has two of them and still adds only 1, because `count++` runs once per row, after the inner loop.
- **C** is wrong: the flag starts `false` and is switched to `true` by a negative, which is the "at least one" pattern, not the "all" pattern.
- **D** is wrong: `count` is increased inside the outer loop, right after each row is checked and before the next reset, so every row with a negative is counted, not only the last.

Reread 4.13.4 *Checking Whether Any or All Elements Qualify* and 4.13.5 *Counting Elements in a Region* if you chose differently.

Question 21

The error is on line 5. The search is over column 1 (3, 12, 7), but `largest` starts at `grid[0][0]`, which is 50, an element of column 0. No element of column 1 is bigger than 50, so `largest` never changes and the program prints a value that is not in the column at all.

Corrected line:

```
1 int largest = grid[0][col];
```

Now `largest` starts at 3, the first element of the column being searched, becomes 12 at row 1, and the program prints 12. The best-so-far must start at an element of the **region you are searching**, not just any element of the array.

Reread 4.13.3 *Finding the Minimum or Maximum* if you missed the error.

Question 22

```
1 2 9
5 6 4
7 8 3
```

Step 1, lines 4–9: rotate row 1 left. `first` saves 4, the row copies left to become 5, 6, 6, and the last spot gets 4: row 1 is now 5 6 4.

Step 2, lines 11–17: reverse column 2. Column 2 is now 3, 4, 9 (the 4 arrived in step 1). `len / 2` is 1, so rows 0 and 2 swap: column 2 becomes 9, 4, 3.

Reread 4.13.8 *Shifting and Rotating a Row or Column* and 4.13.9 *Reversing a Row or Column* if your output differed.

Question 23

Row 0 is {1, 2} and row 1 is {2, 1}: neither has a repeat. Column 0 is 1, 2 and column 1 is 2, 1: neither has a repeat. Yet 1 and 2 each appear twice in the array. The repeats sit in **different rows and different columns** (diagonally from each other), so no row check or column check ever compares them.

A complete answer says:

- A row or column check compares only elements that share that row or column.
- Duplicates in the whole array can be in different rows and columns.
- A correct check compares every element with every other element of the whole array, for example by copying all elements into a 1D array and running the duplicate check (`j` from `i + 1`) on it.

Reread 4.13.7 *Detecting Duplicates* if your explanation differed.

Question 24

C. Segment I is the swap-to-the-middle reversal. Segment II builds a reversed copy of the column in a 1D array (reading the column from the bottom up) and then writes it back from the top down. Both leave the column reversed; on a column of 1, 2, 3, 4 each gives 4, 3, 2, 1.

- **A** and **B** are wrong because each leaves out a correct segment. Making a reversed copy is a different method from swapping, but just as correct.
- **D** is wrong: segment III swaps across the whole column, so every pair is swapped twice and the column ends in its original order (1, 2, 3, 4 stays 1, 2, 3, 4).

Reread 4.13.9 *Reversing a Row or Column* if you chose differently.

Question 25

A. The inner loop visits every neighbor pair across a row (stopping one early), and the outer loop does that for every row. The flag starts `true` and is switched to `false` by any pair whose left value is bigger than its right value. So it stays `true` only if, in every row, no value is followed by a smaller one.

- **B** is wrong: the flag is set once, before both loops, and never reset, so one bad pair in any row makes it `false`. "At least one row" would need a flag per row.
- **C** is wrong: the pairs are `c` and `c + 1` in the same row, so they are side by side, not stacked. Columns are never compared top to bottom.
- **D** is wrong: there is no reset at the start of each row, so earlier rows still count.

Reread 4.13.4 *Checking Whether Any or All Elements Qualify* and 4.13.6 *Consecutive Pairs in a Row or Column* if you chose differently.