Arrays and ArrayLists

Important Dates:

- Assigned: September 24, 2025
- Deadline: October 1, 2025 at 11:59 PM EST

Objectives:

- Students write methods that operate recursively or iteratively over both one/two-dimensional arrays and `ArrayList` objects.
- Students see the advantages and disadvantages over static arrays versus `ArrayList` objects.

What To Do:

For each of the following problems, create a class named `ProblemX`, where `X` is the problem number. E.g., the class for problem 1 should be `Problem1.java`. Write (JUnit) tests for each method that you design in corresponding test files named `ProblemXTest`, where `X` is the problem number. Additionally, write Javadoc comments explaining the purpose of the method, its parameters, and return value. **Do not round your solutions!**

What You Cannot Use:

You cannot use any content beyond Chapter 3.2. More importantly, for this problem set, you can only use `List`, `ArrayList`, and arrays. You cannot use any of the "complex" data structures from later in section 3.2. Please contact a staff member if you are unsure about something you should or should not use. Any use of anything in the above-listed forbidden categories will result in a **zero** (0) on the problem set.

Warning:

This is a long problem set, considerably longer than the previous ones. We recommend starting this one early!

Problem 1:

Design the `String[] fizzBuzz(int min, int max)` method that iterates over the interval $[min, max]$ (you may assume $max \geq min$) and returns an array containing strings that meet the following criteria:

- If i is divisible by 3, insert `"Fizz"`.
- If i is divisible by 5, insert `"Buzz"`.
- If i is divisible by both 3 and 5, insert `"FizzBuzz"`.
- Otherwise, insert `"i"`, where `i` is the current number.

```
fizzBuzz(1, 12)  => {"1", "2", "Fizz", "4", "Buzz",
                     "Fizz", "7", "8", "Fizz", "Buzz",
                     "11", "Fizz"}
fizzBuzz(15, 18) => {"FizzBuzz", "16", "17", "Fizz"}
```

Problem 2:

Design the `boolean canSum(int[] A, int t)` method that, when given an array of integers A and a target t, determines whether or not there exists a group of numbers in A that sum to t. For example, if $A = \{2, 4, 10, 8\}$ and $t = 9$, then `canSum` returns false because there is no possible selection of integers from A that sum to 9. On the other hand, if $A = \{3, 7, 4, 5, 9\}$ and $t = 8$, then we return true because $3 + 5 = 8$. If $A = \{2, 4, 2, 1, 5, 4\}$ and $t = 9$, then we return true because $4 + 1 + 4$, but also $4 + 5 = 9$, $5 + 4 = 9$, and $4 + 1 + 2 + 2 = 9$.

Hint: approach this problem (standard) recursively; do not try and solve it with a loop or tail recursively. Our solution is four lines long—yours should be around the same length.

Problem 3:

The correlation coefficient r is a measure of the strength and direction of a linear relationship between two variables x and y. The value of r is always between -1 and 1. When $r > 0$, there is a positive linear relationship between x and y. When $r < 0$, there is a negative linear relationship between x and y. When $r = 0$ or is approximately zero, there is no (or little) linear relationship between x and y. The formula for the correlation coefficient is as follows:

$$r = \frac{1}{n-1} \cdot \frac{1}{S_x \cdot S_y} \cdot \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$$

Where n is the number of data points, x_i and y_i are the i^{th} data points, $\bar{x}$ and $\bar{y}$ are the means of x and y respectively, and S_x and S_y are the sample standard deviations of x and y respectively. To compute the sample standard deviation of a set of values S, we use the formula:

$$S_x = \sqrt{\frac{\sum_{i=1}^{|S|} (x_i - \bar{x})^2}{|S| - 1}}$$

Design the `double correlationCoefficient(double[] xs, double[] ys)` method that, when given two arrays of doubles *xs* and *ys*, returns the correlation coefficient between the two arrays. Assume $|xs| = |ys|$, and that $|xs|, |ys| \geq 2$.

Problem 4:

This problem has three parts.

(a) Design the standard recursive `int findMaxWordLength(String[] S)` method, which receives a `String[]` and returns the length of the longest word in the array. You may assume that the array contains at least one string. Hint: you should design a helper method to recurse over the array. The helper method *must* be standard recursive!

(b) Design the `int findMaxWordLengthTR(String[] S)` method that uses tail recursion to solve the problem. You will need to design a helper method. Remember to include the relevant access modifiers!

(c) Design the `int findMaxWordLengthLoop(String[] S)` that solves the problem using a loop.

Problem 5:

Consider the following data definition:

```
A ValidPolynomial is one of:
 - Var
 - PositiveInteger
 - PositiveInteger Var "^" PositiveInteger
 - ValidPolynomial " + " ValidPolynomial
 - ValidPolynomial " - " ValidPolynomial
```

Suppose we want to write the `int evalPolynomial(String p, char v, int n)` method that, when given a "valid polynomial", a variable v, and a value n, evaluates the polynomial with respect to v at n. Design this method, but do so in a piecemeal fashion:

(a) Design the `boolean isPositiveInteger(String s)` method that returns whether s is a string that represents a positive integer, i.e., an integer $n \geq 0$.[1] You may assume that the integer is always within the bounds of a valid 32-bit integer. Do not use any casting, exceptions, or other helper methods; you must handle the logic manually.

(b) Design the `String[] extract(String s)` method that retrieves the components of a valid polynomial of the form ax^n, where a and n are positive integers, and x is a variable, but not necessarily the character `'x'`.

```
extract("5x^4")    => ["5", "4"]
extract("9x^1")    => ["9", "1"]
extract("102x^10") => ["102", "10"]
```

(c) Finish the `evalPolynomial` method. You may assume that the given variable v is always the sole variable used in the given expression.

```
evalPolynomial("3x^1 + 3", 'x', -3)                        => -6
evalPolynomial("4x^4 + 7x^3 + 21x^2 - 65x^1 + 3", 'x', 0)  => 3
evalPolynomial("4x^4 + 7x^3 + 21x^2 - 65x^1 + 3", 'x', 16) => 295155
```

[1] Yes, I am including 0 in this definition of a positive integer. Mathematicians, be salty.

Problem 6:

Design the `int countAdjacentZeroSums(int[][] A)` method that receives a two-dimensional array A and returns the number of row-adjacent cells that sum to zero. By row-adjacent, we mean two cells that are ordered one after the other in a row-major order traversal over the array. Consider the following 4×3 array. We see that $A[0][0]$ and $A[0][1]$ are side-by-side and sum to zero. Additionally, $A[2][4]$ and $A[3][0]$ are side-by-side when considering a row-major traversal and sum to zero. There are no other adjacent zero sums, so we return 2.

$$\begin{bmatrix} -5 & 5 & 1 & 3 & 0 \\ 9 & 3 & 12 & -3 & 17 \\ 23 & 31 & -42 & -8 & 16 \\ -16 & -23 & 18 & -8 & -7 \end{bmatrix}$$

Problem 7:

John Conway designed the "Game of Life," which is a cellular automaton. In essence, the game is a grid of cells, where each cell is either "alive" or "dead." The neighbors of a cell are the (up to eight) cells that surround a cell. Rules advance/guide the "game" from one state to the next. These rules are as follows:

1. If a cell c is alive and has between two and three alive neighbors, then it remains alive.
2. If a cell c is alive and has less than two alive neighbors, then it dies.
3. If a cell c is alive and has more than three alive neighbors, then it dies.
4. If a cell c is dead and has three alive neighbors, then it becomes alive.

Design the `boolean[][] gameOfLife(boolean[][] B)` method that, when given an initial board configuration B, returns the next state of the game after applying the preceding list of rules. Each element in B is a boolean value, where `true` means the cell at i, j is alive and `false` means the cell at i, j is dead. The `gameOfLife` method should *not* update the given board. Instead, return a new board with the updated values. It may be helpful to design the following auxiliary methods:

- `int[][] getAdjacentCells(boolean[][] B, int i, int j)`
- `int[][] getAliveCells(boolean[][] B, int i, int j)`
- `int[][] getDeadCells(boolean[][] B, int i, int j)`

Problem 8:

Minesweeper is a simple strategy game where the objective is to uncover all spaces on a board without running into mines. If you are not familiar with the mechanics, we encourage you to find a version online and play it for a bit to understand its gameplay. In this exercise you will implement the minesweeper game as a series of methods. **Note: you only need to officially test `play`, but it may help you to test other methods.R**

(a) First, design the static `boolean isValidMove(char[][] board, int mx, int my)` method that receives a board and a move position, and determines whether the move is valid. A move is valid if it is located within the bounds of the board.

(b) Design the static `List<int[]> getValidNeighbors(char[][] board, int mx, int my)` method that receives a board and a move position, and returns a list of all the immediate neighbors to the cell (mx, my). Each element of the list is a two-element integer array containing the x and y coordinates of the neighbor. Consider the diagram below, where $(0, 0)$ is the move position, and the surrounding cells are its neighbors, represents as offsets. Note that `getValidNeighbors` should only return neighbors that are *in bounds*. Hint: use `isValidMove`.

(-1, 1)	(0, 1)	(1, 1)
(-1, 0)	(0, 0)	(1, 0)
(-1, -1)	(0, -1)	(1, -1)

(c) Design the static `List<int[]> getNonMineNeighbors(char[][] board, int mx, int my)` method that receives a board and a move position, and returns a list of all the neighbors that are not mines. A non-mine neighbor is denoted by the character literal `'-'`. You *must* use `getValidNeighbors` in your definition.

(d) Design the static `List<int[]> getMineNeighbors(char[][] board, int mx, int my)` method that receives a board and a move position, and returns a list of all the neighbors that are mines. Mines are denoted by the character literal `'B'`. You *must* use `getValidNeighbors` in your definition.

(e) Design the static `int countAdjacentMines(char[][] board, int mx, int my)` method that receives a board and a move position, and returns the number of mines that are adjacent to the given position. This method should be one line long and contain a call to `getMineNeigbors`.

(f) With the helper methods complete, we now need a method that searches through a position and reveals all non-mine adjacent positions. In general, this is a *traversal* algorithm called *depth-first search*. The idea is to recursively extend out the path until we hit a mine, at which point we unwind the recursive calls to extend another path.

Design the static `void extPath(char[][] board, int mx, int my)` method that receives a board and a move position, and extends the path from the given position using the following rules:

(i) If the given move position is invalid, then return.

(ii) If the character at `board[mx][my]` is not a dash, `'-'`, then return.

(iii) Otherwise, determine the number of adjacent mines to the move position. If the number of adjacent mines is non-zero, assign to `board[mx][my]` the number of mines at that move position.

(iv) If the number of adjacent mines *is* zero, then we can extend out the path to all non-mine neighbors. First, assign to `board[mx][my]` the character literal `'0'`, then loop over all non-mine neighbors to the move position. In the loop body, call `extPath` on each neighbor.

(g) Minesweeper board generation is an algorithmic problem in and of itself, and as such our implementation will be simple. Design the static `char[][] makeBoard(int N, int M, int B)` method that receives a board size of *N* rows, *M* columns, and *B* mines to place. To randomly place mines, create a `List<int[]>` of all the possible cells on the board, shuffle the list, retrieve the first *B* cells, and assign the character literal `'B'` to them. Assign the character literal `'-'` to all other cells.

(h) Finally, design the static `char[][] play(char[][] board, int mx, int my)` method that receives a board and a move position, and attempts to play the given move position on the board. If, at that position on the board, there is a mine, return `null`. Otherwise, call `extPath` on the board and position, then return `board`. In essence, `play` receives one game state and transitions it to the next state.

Problem 9:

Design the List<Integer> exptIntermediates(int n, imt m) method that returns a List<Integer> of the intermediate values of n^m. The first element of the resulting list is n^m, and the second element is n^0, the third is n^1, the fourth is n^2, and so forth. For example, exptIntermediates(3, 4) returns [81, 1, 3, 9, 27].

Problem 10:

Design the `List<Boolean> areValidNames(List<String> L)` method that, when given a list of names L, returns a list of booleans where the i^{th} boolean denotes whether the i^{th} name is valid. A name is valid if it is all upper-cased, and contains only letters and dashes.

Problem 11:

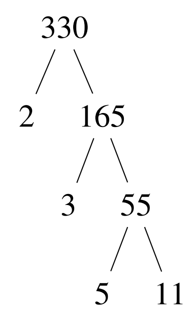
Design the `int[] minDistancePoint(List<int[]> L, int x, int y)` method that, when given a non-empty list of two-element integer arrays L and a coordinate pair (x, y), returns the coordinate pair in L that is the closest to (x, y). The input list, as described, contains two-element arrays, where index 0 is its x-coordinate and index 1 is its y-coordinate.

For example, if $L = [[1, 2], [-2, 3], [2, 0]]$ and $(x, y) = (0, 0)$, the returned pair is $[2, 0]$ because its distance of 2 to the point $(0, 0)$ is the smallest out of all three points.

Problem 12:

The *prime factorization* problem is about finding prime numbers that multiply to some positive integer. That is, given a positive integer n, we want to find its prime factors. It is an open mathematics and computer science question whether it is possible to find the prime factorization of a positive integer in *polynomial time*.[2] The naive algorithm is to iterate over the primes from $2, 3, \ldots, n$, find the lowest prime p that divides n, divide n by p, then repeat until n is prime.

We can visualize this algorithm via a *prime factor tree*. For example, let's find the prime factorization of 330. The smallest prime starting from 2 that divides 330 is 2. So, the root of the tree is 330, the left branch leads to a prime factor, and the right is a smaller sub-problem, that being $330/2 = 165$. The smallest prime that divides 165 is 3, so we get 3 in the left branch and 55 in the right branch. Repeat once more to get 5 and 11, and we stop because 11 is prime.



(a) Design the `static List<Integer> primeFactors(int n)` that, when given a positive integer $n \geq 2$, returns a list of the prime factors of n. To test a positive integer for primality, use the method that we provide as an example in Chapter 2, or design your own version.

(b) Design the `static List<Integer> primeFactorsTree(int n)` method that creates a "factor tree" as a list. That is, consider once again the prime factorization of 330. The returned list should be $[330, 2, 165, 3, 55, 5, 11]$, because the left branch of 330 leads to 2, and the right branch leads to factoring 165. The `primeFactorsTree` method must call `primeFactors`.

[2]The fundamental theorem of arithmetic states that every positive integer has a unique prime factorization. So we can always *find* a prime factorization, but the issue is that, as the input grows large, the naive algorithm becomes intractable.