Indiana University
CSCI-C212: Introduction to Software Systems (F25)
Joshua Crotts

Problem Set 6
October 1, 2025

The Collections Framework

Important Dates:

- Assigned: October 1, 2025
- Deadline: October 7, 2025 at 11:59 PM EST,
  Note the date: this is one day earlier than normal!

Objectives:

- Students begin to understand the differences between data structures in the Java collections framework.

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. Anything in or before Chapter 3, *excluding streams*, is fair game for this problem set, **but** you must respect the rules enforced by any problem. Moreover, anything that trivializes the solution is disallowed. 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:

The problem sets are now getting difficult enough to where it is tempting to solve the problems with generative AI. We're approaching the middle point of the semester, and with fatigue setting in, the easy way out looks appealing. Do not fall into this trap. The TAs and I are, very easily, able to tell AI code from what you genuinely write. You will receive a 0 and be reported for academic dishonesty. (Moreover, this is a surefire way to not prepare yourself for the in-person, proctored, and written exams.) Please don't jeopardize your academic future.

Problem 1:

For this problem you are not allowed to use an `ArrayList` or any helper methods, e.g., `.contains`, or methods from the `Arrays` class. You *may* (and should) use a `Set<Integer>` to keep track of previously-seen peaks.

Joe the mountain climber has come across a large mountain range. He wants to climb only the tallest mountains in the range. Design the `int[] peakFinder(int[] H)` method that returns an array H' of all the peaks in an `int[]` of mountain heights H. A peak p is defined as an element of h at index i such that $p[i-1] < p[i]$ and $p[i] > p[i+1]$. If $i = 0$ or $i = |H| - 1$, Joe will not climb $p[i]$. Joe doesn't want to climb a mountain of the same height more than once, so you should not add any peaks that have already been added to H'. We present some test cases below.

```
peakFinder({9, 13, 7, 2, 8})                        => {13}
peakFinder({8, 7, 8, 7, 8, 7, 8, 7})                => {8}
peakFinder({111, 27, 84, 31, 5, 9, 4, 3, 2, 1, 64}) => {84, 9}
peakFinder({})                                      => {}
peakFinder({1})                                     => {}
peakFinder({1, 2})                                  => {}
peakFinder({1, 2, 1})                               => {2}
peakFinder({1, 2, 3, 2, 1})                         => {3}
```

Problem 2:

Design the `List<String> tokenize(String s, char d)` method that, when given a string *s* and a `char` delimiter *d*, returns an `ArrayList` of tokens split at the delimiter. You must do this by hand; you **cannot** call any `String` methods (except `.length` and `.charAt`). You may assume that delimiters are not side-by-side, that there is at least one delimiter in the string, and it is neither at the beginning nor the end of the string. (Therefore, the length of the input string is guaranteed to be at least 3.)

Problem 3:

Design the `Map<String, Integer> wordCount(String s)` method that, when given a string *s*, counts the number of words in the list, then stores the resulting frequencies in a `HashMap<String, Integer>`. Assume that *s* is not cleaned. That is, you should first remove all punctuation (periods, commas, exclamation points, question marks, semi-colons, dashes, hashes, ampersands, asterisks, and parentheses) from *s*, producing a new string *s′*. Then, split the string based on spaces (remember `tokenize`?), and produce a map of the words to their respective counts. Do not factor case into your total; e.g., `"fAcToR"` and `"factor"` count as the same word. The ordering of the returned map is not significant.

```
String s = "Hello world, the world is healthy, is
            it not? I certainly agree that the world
            is #1 and healthy."
wordCount(s) => [<"hello" : 1>, <"world" : 3>, <"the" : 2>
                 <"is" : 3>, <"healthy" : 2>, <"it" : 1>,
                 <"i" : 1>, <"certainly" : 1> <"agree" : 1>
                 <"that" : 1>, <"1" : 1>, <"and" : 1>, <"not" : 1>]
```

Problem 4:

Design the `double postfixEvaluator(List<String> l)` method that, when given a list of binary operators and numeric operands represents as strings, returns the result of evaluating the postfix-notation expression. You will need to write a few helper methods to solve this problem, and it is best to break it down into steps. First, write a method that determines if a given string is one of the four binary operators: `"+"`, `"-"`, `"*"`, or `"/"`. You may assume that any inputs that are not binary operators are operands, i.e., numbers. Then, write a method that applies a given binary operator to a list of operands, i.e., an `ArrayList<Double>`.

```
postfixEvaluator({"5", "2", "*", "5", "+", "2", "+"}) => 17
postfixEvaluator({"1", "2", "3", "4", "+", "+", "+"}) => 10
postfixEvaluator({"12", "3", "/"})                    => 4
```

Problem 5:

The *substitution cipher* is a text cipher that encodes an alphabet string A (also called the *plain-text alphabet*) with a key string K (also called the *cipher-text alphabet*). The A string is defined as `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`, and K is any permutation of A. We can encode a string s using K as a mapping from A. For example, if K is the string `"ZEBRASCDFGHIJKLMNOPQTUVWXY"` and s is `"WE MIGHT AS WELL SURRENDER!"`, the result of encoding s produces `"VN IDBCY JZ VNHH ZXRRNFMNR!"`

Design the `substitutionCipher` method, which receives a plain-text alphabet string A, a cipher-text string K, and a string s to encode, `substitutionCipher` should return a string s' using the aforementioned substitution cipher algorithm.

Problem 6:

Design the `List<List<String>> lex(String e)` method that, when given an expression written in prefix fashion, returns a list of *tokens* and their identifiers.[1]

The `lex` method returns a list of two-element lists. The first is the "tag" of the token, and the second is the token itself. Tokens in this language are `"L_PAREN"`, `"R_PAREN"`, `"NUMBER"`, and `"SYMBOL"`. Left and right parentheses are straightforward, as are numbers (you may assume that all numbers are positive integers). Anything else should be regarded as a symbol. Note that symbols are to be considered space-separated, e.g., `"HELLO"` is one `"SYMBOL"`; not five.

For example, consider the input `(+ (- 43 5) 42)`. The `lex` method therefore returns:

```
[
 ["L_PAREN", "("],
 ["SYMBOL", "+"],
 ["L_PAREN", "("],
 ["SYMBOL", "-"],
 ["NUMBER", "43"],
 ["NUMBER", "5"],
 ["R_PAREN", ")"],
 ["NUMBER", "42"]
 ["R_PAREN", ")"]
]
```

[1] It should be noted that this method could *also* be called "tokenize," but that method name has a particular meaning with respect to the problem set.

Problem 7:

Design the `Map<String, Set<String>> trending(Map<String, Set<String>> regionTopics, Set<String> gTrending)` method that, when given a map of regions to topics that are trending in that region, returns a map of topics to regions such that the keys are topics and the values are regions where that topic is trending. A topic is trending in a region if it is not a globally-trending topic and it is trending in at least two regions simultaneously. We provide an example below.

```
trending([<"North America" : {"Tech", "Comedy", "Sports"}>,
          <"Europe" : {"Comedy", "Music"}>,
          <"Asia" : {"Fashion, "Music"}>], {"Tech"})
  => [<"Comedy" : {"North America", "Europe"}>, <"Music" : {"Europe, "Asia"}>]
```

Problem 8:

Design the `Set<List<Integer>> selectPairs(int[] A, int t)` method that, when given an array of integers A and a target t, returns all possible pairs of numbers in A that sum to t. For example, if $A = \{2, 2, 4, 10, 6, -2\}$ and $t = 4$, we return a set containing two two-element lists: $\{2, 2\}$ and $\{6, -2\}$. Do not add a pair that already exists in the set or a pair that, by reversing the pair, we get a pair in the existing set. E.g., $\{-2, 6\}$ should not be added to the set.

There is a simple brute-force algorithm to solve this problem via two loops, but by incorporating a second set for lookups, we can do much better: for every number n in A, add n to a set S, and if $t - n = m$ for some $m \in S$, then we know that $m + n$ must equal t, therefore we add $\{n, m\}$ to the resulting set of integer arrays. Walking through this with the example from before, we get the following sequence of actions:

- Initialize $S = \{\}$ and $L = \{\}$. We know that $t = 4$.
- We add 2 to S. $S = \{2\}$.
- Because $4 - 2 \in S$, the two-element array $\{2, 2\}$ is added to L. 2 is not re-added to S.
- Because $4 - 4 \notin S$, we only add 4 to S. $S = \{2, 4\}$.
- Because $4 - 10 \notin S$, we only add 10 to S. $S = \{2, 4, 10\}$.
- Because $4 - 6 \notin S$, we only add 6 to S. $S = \{2, 4, 10, 6\}$.
- Because $4 - (-2) \in S$, the two-element array $\{6, -2\}$ is added to L. We add -2 to S. $S = \{2, 4, 10, 6, -2\}$.

Problem 9 Extra Credit (10 points):

You are designing a system with querying functionality similar to a database language such as SQL. In particular, we have a 2D array of strings whose first row contains column headers to a database. Examples of such columns may be `"ID"`, `"Name"`, `"Age"`, `"Salary"`, and so forth.

Design the `List<String> query(String[][] db, String cmd)` method that, when given a "database" and a "Command", returns the data from the rows that satisfy the criteria enforced by the command.

```
A Command is "SELECT <count> <header> WHERE <predicate>"
```

The `SELECT` command receives a `<count>`, which is a number between 1 and *n*, or the asterisk to indicate everyone in the database. The `WHERE` clause receives a "Predicate". The `SELECT` command receives a `<count>` and a `<header>` to designate that the command should return `<count>` rows with data from the `<header>` column. An asterisk can be used to select all rows in the database. Your implementation should be flexible enough to work with any arbitrary column over the database. (You may assume that the input `<header>` is a valid column in the database, but it cannot be hard-coded to fit only a particular set of database columns, e.g., `"ID"`, `"Name"`, and so forth.)

```
A Predicate is "<header> <comparator> <value>"
```

Headers are one of the column headers of the database, and comparators are either `=`, `!=`, `<`, `<=`, `>`, or `>=`, or `LIKE`. Values are either numbers, floats, or strings.

Parsing a `LIKE` command is more complicated. There are four possible types of values:

```
'S'
'%S'
'S%'
'%S%'
```

The first matches an exact string, namely `S`. The second matches any string that ends with `S`. The third matches any string that begins with `S`. The fourth matches any string that contains `S`.

You may assume all commands are well-formed. However, it is possible that a command returns no results, e.g., `SELECT * Name WHERE Salary >= '10000000'`

Problem 10 Extra Credit (20 points):

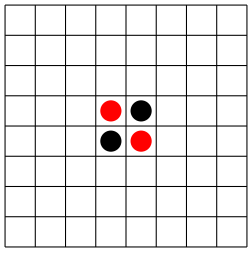
Be warned: this problem is pretty tough. If you enjoyed the Minesweeper and Game of Life problems from Problem Set 5, you might also enjoy this.

Reversi is a "pebble-based," grid-style, turn-based, two-player game where the goal is to have as many pebbles of your color on the board as possible. If you're unfamiliar with the game, play it here until you understand how it works: There are a few rules to placing pebbles on the board:

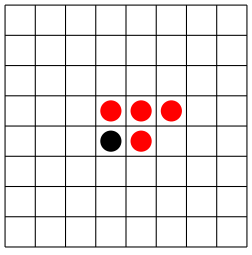
1. For a player to place a pebble of color P at position (r, c), it must make a valid move. A move is *valid* if and only if there is at least one pebble of the opposite color Q in between (r, c) and some other pebble of color P.

2. When making a move at position (r, c) by pebble of color P, we flip all Q-colored pebbles in between (r, c) and other P-colored pebbles along the grid.

For example, consider the following board configuration, where one pebble is colored `R` (red) and the other is colored `B` (black).



If it is `R`'s move, then they must make a move that flips at least one `B`, according to rule (1). So, we can place a pebble colored `R` at the following positions (denoted by '(row, column)'): (3, 5), (5, 3), (2, 4), and (4, 2). If we choose to place a `R`-colored pebble at position (3, 5), we flip the `B`-colored pebble at position (3, 4) because a line forms between (3, 3) and (3, 5), according to rule (2):



From here, it is `B`'s turn, and they can move to positions (2, 3), (4, 5), and (2, 5). In this exercise you will implement Reversi as a series of piecemeal-designed methods.

(a) Design the `boolean isBlankCell(char[][] B, int r, int c)` method that, when given a Reversi board B and a position at row r and column c, returns whether that position is blank. A cell is blank if it is denoted by the `'.'` character.

(b) Design the `boolean isValidMove(char[][] B, int r, int c)` method that returns whether the position (r, c) is in the bounds of the array.

(c) Design the `Set<List<Integer>> getValidMovesForPebble(char[][] B, char P, int r, int c)` method that returns all valid moves from the pebble colored as P located at (r, c). Follow the rules from above. Hint: it may be helpful to have an array of the eight directional offsets (similar to the Minesweeper presentation), and expand outward.

(d) Design the `Set<List<Integer>> getPebblePositions(char[][] B, char P)` method that returns all positions where there is a pebble colored as P.

(e) Design the `Set<List<Integer>> getValidMoves(char[][] B, char P)` method that returns all valid moves from *all* positions with a pebble colored as P. You must use `getPebblePositions` and `getValidMovesForPebble` in your definition of this method.

(f) Design the `Set<List<Integer>> getFlippablePebbles(char[][] B, char P, int r, int c)` method that returns a set of all the positions that will be flipped by placing a pebble P at position (r, c). Do *not* assume that position (r, c) starts off as valid.

(g) Design the `char[][] flipPositions(char[][] B, char P, char Q, int r, int c)` method that returns a new board after flipping all of the pebbles of color Q that, if we start from position (r, c) with a pebble colored P, we expand out until we hit another pebble that is also colored P. Don't forget to place a P-colored pebble at position (r, c)!