Problem Solving Assignment #1

CIT 5940

January 21, 2025

DO NOT WRITE ANYTHING ON THIS PAGE! While you will have to hand it in at the end of recitation, we will not grade anything written here.

Question 1. Implementing `compareTo`

This question only requires elementary Java, so you should write Java that is mostly correct. Small syntax errors are OK.

Your job is write an implementation of the `compareTo` method for a DNA `Sequence` class. The class definition and its instance variables are provided below:

```java
 public class Sequence implements Comparable<Sequence> {
     private String basePairs;
     private int position;
     ...
 }

```

Implement the `compareTo` method so that calling `Collections.sort()` on a `List` of `Sequence` objects will place them in *ascending order of the length of `basePairs`*. Ties in the length of this property should be broken first by alphabetical ordering of the `basePairs`, and then by increasing value of the `position`. For example, the following is an example of the intended ordering of several `Sequence` objects, symbolized here as tuples of (`basePairs`, `position`):

```
[("ACG", 3), ("GCA", 2), ("ATCT", 1), ("TATA", 4), ("GATGA", 3), ("GATGA", 7)]
```

Solution:

```java
public int compareTo(Sequence other) {
    int lengthDiff = this.basePairs.length() - other.basePairs.length();
    if (lengthDiff == 0) {
        int alphaOrdering = this.basePairs.compareTo(other.basePairs);
        if (alphaOrdering == 0) {
            return this.position - other.position;
        }
        return alphaOrdering;
    }
    return lengthDiff;
}

```

Question 2. Repeatedly Checking for Membership

You should write Java that is mostly correct. Although you should attempt to follow proper syntax related to using Java Collection classes, syntax errors related to generics or Collection methods will not count against you.

You've taken a job working for airport security. Given an unsorted array representing the names of passengers with a ticket for a flight and an array representing the names of passengers who are banned from flying, return a `Collection` (a `List` or a `Set` of some kind) containing those names that appear in both arrays. It is possible that `passengers` and `banned` contain duplicate names, although your solution should only contain at most one copy of each name.

Your solution does not have to be the most efficient, although you will have to analyze the runtime complexity in the next question.

```
public static Collection<String> matches(String[] passengers, String[] banned) {}
```

Solution:

```
1  public static Collection<String> matches(String[] passengers, String[] banned) {
2      Set<String> bannedSet = new TreeSet<>();
3      for (String bannedPassenger : banned) {
4          bannedSet.add(bannedPassenger);
5      }
6      Set<String> matches = new TreeSet<>();
7      for (String passenger : passengers) {
8          if (bannedSet.contains(passenger)) {
9              matches.add(passenger);
10         }
11     }
12     return matches;
13 }
```

Question 3. Analyze the Runtime

Find the worst case big-O runtime of your implementation from the previous question. Express the length of the `passengers` array as n and the length of the `banned` array as k, where you can assume that $1 \leq k << n$. You may find it helpful to follow this procedure:

1. Ignore any `for` or `while` loops (the headers, not the bodies of the loops) and identify the lines (or blocks of lines) that represent operations that are not constant time. It may help to write line numbers for your previous solution.
2. Use the table below to identify the big-O runtime of these individual lines.
3. Identify for how many iterations each of your `for` or `while` loops will run for. Decide if the bodies of each of these loops run in constant time or not, and conclude what the big-O runtime of each loop, body included, will be. (*Remember: Multiplication*)
4. Sum up the runtime of all blocks of code in your solution, and conclude what the total runtime is in terms of k and n.

	LinkedList	**ArrayList**	**TreeSet**
add	**O(1)** to the head/tail, **O(n)** to the middle	**O(1)** to the end, **O(n)** elsewhere	**O(log n)**
get(int i)	**O(i)**	**O(1)**	**n/a**
remove	**O(1)** from the head/tail, **O(n)** from the middle	**O(1)** at the end, **O(n)** elsewhere	**O(log n)**
contains	**O(n)**	**O(n)**	**O(log n)**
size/clear	**O(1)**	**O(1)**	**O(1)**

Solution:

1. Line 2: $O(1)$
2. Line 4: $O(\log k)$ (adding to a `TreeSet`)
3. Line 6: $O(1)$
4. Line 8: $O(\log k)$ (checking membership in a `TreeSet`)
5. Line 9: $O(\log k)$ (adding to a `TreeSet` that will contain at most k elements)

Line 4 is in a loop that will run k times. Line 8 will run n times as part of the loop. Line 9 will run at most n times since there may be duplicate passenger names. The first loop therefore contributes $O(k \log k)$ and the second loop contributes $O(n \log k)$, so the final runtime is $O((k+n)\log k) \in O(n \log k)$ (either would be fine).