C212 Practice Midterm Exam (80 points)
Oct 9, 2024

Please read these directions before starting your exam.

This is a closed-note exam aside from your one page of notes, double-sided. You may not use any electronic devices to complete this exam, nor can you communicate with anyone besides the proctors and professor. *If you are caught cheating, you will receive an F in the course.*

For any question, unless specified otherwise, you may use any class without a corresponding `import`. E.g., if you want to use `HashMap`, you do not need to also import `java.util.HashMap`.

Unless otherwise stated, you do not need to spell out the "full design recipe", i.e., write the signature, documentation comments, and tests. Of course, doing so may aid you in your solution.

If you find a mistake, please raise your hand and let one of the proctors know; we will determine whether or not this is the case.

When you are finished, turn in your exam and notes sheet if you have one, then quietly exit.

You have 75 minutes to complete the exam, but it is designed to take only 60 minutes.

Good luck!

Question	Points	Score
1	20	
2	25	
3	35	
Total:	80	

Name: _____

IU Email: _____

1. (20 points) Design the `double cookingScore(String type, double oz, int costDollars, int costCents, boolean isAppealing)` method, which scores a culinary piece in a cooking contest. **The returned score is a value in the interval** $[0, 10]$.

   ```
   A type is one of:
   ```
   - `"Cake"`
   - `"Pasta"`
   - `"Pie"`
   - `"Burger"`

   Below are the criteria for scoring the piece:

   - If the `type` is `"Cake"` or `"Pasta"`, the base score is 1. If the `type` is `"Burger"`, the base score is 0.5. If the `type` is `"Pie"`, the base score is 0.75. Any other `type` is an automatic zero.

   - If the weight `oz` is less than 4, their (current) score is multiplied by 0.9. If $4 \leq$ `oz` ≤ 20, their (current) score is multiplied by y such that $y = 1/16$`oz` $+ 0.25$. Otherwise, their (current) score is multiplied by 0.2.

   - The *combined* price of the piece adds a fixed amount to the score up to a total of $\$5.00$. Anything beyond this subtracts that amount from the score. For example, if the combined cost of a piece is $\$1.25$, then its score is increased by 1.25. On the other hand, if the combined cost of a piece is $\$6.75$, then its score is decreased by $\$1.75$.

   - If the piece is appealing, add a constant factor of 1.5 to the piece.

   In designing this method, follow the design recipe from class; write the signature, purpose statement, testing, and *then* do the implementation. You should probably use simple numbers for the inputs so you can calculate the values in your head. As you can see, there are tons of possible inputs. You need to write at least four tests, one for each "type."

   The skeleton code is on the next page.

The following methods may be helpful.

`double Math.max(double a, double b)` returns the maximum of `a` and `b`.
`double Math.min(double a, double b)` returns the minimum of `a` and `b`.

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;

class CookingScoreTester {

  @Test
  void testCookingScore() {




  }
}

class CookingScore {

  /**
   *
   *
   * @param
   * @param
   * @param
   * @param
   * @param
   * @return
   */
  _____ _____ cookingScore(double type, double oz, int costDollars,
                                   int costCents, boolean isAppealing) {




  }
}
```

2. (25 points) This question has three parts.

   A *parenthesized string* is a string enclosed by parentheses. For example, the string `"(abc)pqr(de)"` contains two parenthesized strings: `"abc"`, and `"de"`.

   For the following problems, you may assume that there are no nested parentheses, all parentheses are balanced, and if there is a parenthesized string, it contains at least one character.

   (a) *(9 points)* Design the *standard recursive* `collectParenthesizedStrings` method, which receives a `String` S and returns a `List` of all the parenthesized strings of S.

   Hint: what is your base case?

   (b) *(8 points)* Design the `collectParenthesizedStringsTR` method and its accompanying helper method `collectParenthesizedStringsTRHelper`. The former acts as the driver to the latter; the latter solves the same problem as `collectParenthesizedStrings` does, but it instead uses tail recursion. Remember to include the relevant access modifiers!

(c) *(8 points)* Design the `collectParenthesizedStringsLoop` method, which solves the problem using either a `while` or `for` loop.

3. (35 points) Consider a network of friends, as follows. An arrow from one name A to another B means that A is friends with B.
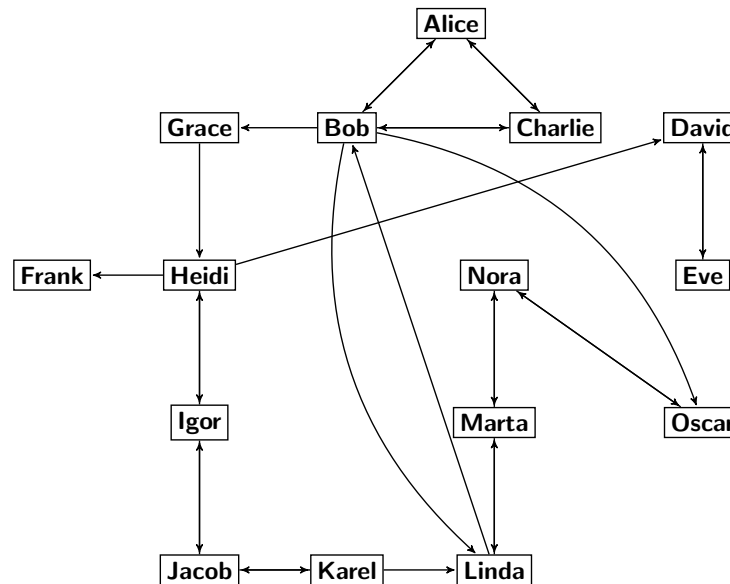
We want to find the *longest contiguous friend sequence.* That is, given a name, we want to find the length of the chain of friends that is the longest. In the above diagram, this is the path from Alice to Oscar, with a length of 11.
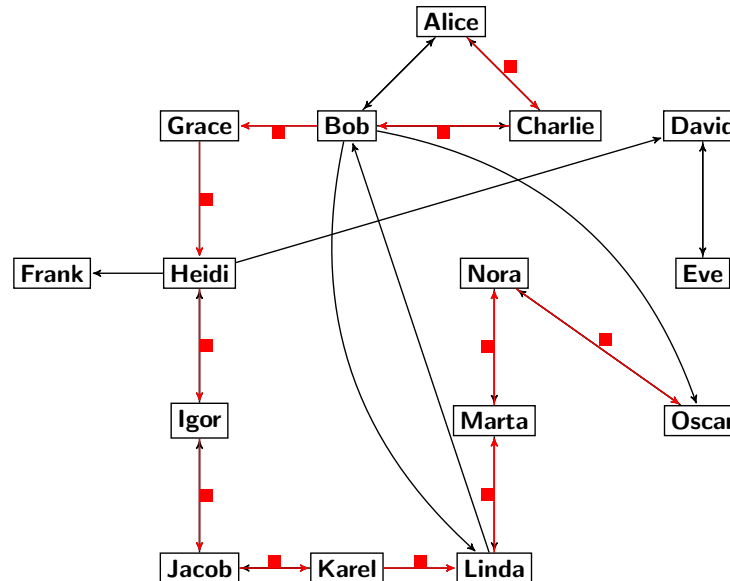
Here's the idea: we need a recursive algorithm to traverse the friend relationship. Each time we run into a new friend, we want to add one to a counter, and if we encounter a cycle, we stop recursing. To do so, let's design two methods: `int longestFriendSequence(String s, Map<String, List<String>> friendList)` and an accompanying helper method.

The helper method receives three arguments: the name of the friend that we're recursing on, the friend list, and a set of names that we have visited thus far. The `friendList` is nothing more than a map of names to who their friends are, according to the relationship diagram. For example, one such entry is `"Alice"` that maps to `["Bob", "Charlie"]`.

As we said, the helper method receives a friend name f and adds it to the set of visited friends S. Then, it loops over their friends according to the map. For every friend f', we invoke the helper method on f', which returns a length l. If $l > m$, where m is the maximal length found thus far, it is updated accordingly. After the loop, we remove f from S and return $m+1$ to designate that this path contains f.

Fill in the following code to complete this algorithm.

```java
import java.util.*;

class FriendPath {

  /**
   * Find the longest path of friends from a friend.
   * @param f friend to start from.
   * @param friends map of friends.
   * @return the longest path from the friend.
   */
  static int longestFriendPath(String f, Map<String, List<String>> friends) {
    Set<String> visited = new HashSet<>();
    return longestFriendPathHelper(f, friends, visited);
  }

  /**
   * Helper method to recursively find the longest path from a friend.
   * @param f friend to start from.
   * @param friendsList map of friends.
   * @param visited set of visited friends.
   * @return the longest path from the friend.
   */
  private static int longestFriendPathHelper(String f,
                                     Map<String, List<String>> friendsList,
                                     Set<String> visited) {
    if (_____) {
      return 0; // If visited, no length should be added from this path.
    } else if (_____) {
      return 0; // If no friends are listed.
    } else {
      _____.add(_____);
      int max = 0;
      for (_____ friend : friendsList._____(_____)) {
        int pathLength = _____;
        if (_____) {
          _____;
        }
      }
      visited.remove(_____);
      return _____;
    }
  }
}
```

Scratch work

Scratch work