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

C212 Practice Midterm Exam Rubric

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 $4 \leq \texttt{oz} \leq 20$, their (current) score is multiplied by y such that $y = 1/16\texttt{oz} + 0.25$. 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.

Solution. *This is admittedly a pretty evil question and is harder than one I would put on an actual exam, but serves as great practice for writing comprehensive tests.*

Rubric:

- (1 pt) example when *type* is `"Cake"`.
- (1 pt) example when *type* is `"Pasta"`.
- (1 pt) example when *type* is `"Burger"`.
- (1 pt) example when *type* is `"Pie"`.
- (1 pt) example when *type* is not one of the four types.
- (1 pt) example when the score should be capped by the interval. Either side is fine.
- (2 pts) purpose statement sensible.
- (2 pts) signature is correct.
- (2.5 pts) initial score of the "type" is correct.
- (2.5 pts) weight calculation and conditions are correct.
- (2.5 pts) combined price score is correct.
- (1 pt) appealing flag correctly updates score.
- (1.5 pts) score is correctly capped by the interval.

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

class CookingScoreTester {

  @Test
  void testCookingScore() {
    assertAll(
      () -> assertEquals(0, cookingScore("Junk", 3, 1, 0, false)),
      () -> assertEquals(1.9, cookingScore("Cake", 3, 1, 0, false)),
      () -> assertEquals(3.4, cookingScore("Pasta", 3, 1, 0, true)),
      () -> assertEquals(1.28125, cookingScore("Burger", 5, 1, 0, false)),
      () -> assertEquals(0, cookingScore("Pasta", 5, 1000000000, 0, false)),
      () -> assertEquals(2.921875, cookingScore("Pie", 5, 1, 0, true)));
  }
}
```

```java
class CookingScore {

  /**
   * Computes the score of some food.
   * @param type        one of "Cake", "Pasta", "Pie", or "Burger".
   * @param oz          weight in oz
   * @param costDollars cost in whole dollars
   * @param costCents   cost in cents
   * @param isAppealing whether it's appealing
   * @return score
   */
  static double cookingScore(String type, double oz,
                             int costDollars, int costCents,
                             boolean isAppealing) {
    double score = 0;
    // Type
    if (type.equals("Cake") || type.equals("Pasta")) { score = 1; }
    else if (type.equals("Burger")) { score = 0.5; }
    else if (type.equals("Pie")) { score = 0.75; }
    else { return 0; }

    // Weight
    if (oz < 4) { score *= 0.9; }
    else if (oz >= 4 && oz <= 20) {
      double y = 1.0 / 16 * oz + .25;
      score *= y;
    } else { score *= 0.2; }

    // Price
    double combinedPrice = costCents / 100.0 + costDollars;
    if (combinedPrice <= 5) { score += combinedPrice; }
    else { score = score - (combinedPrice - 5); }

    // Score and max/min.
    score += (isAppealing ? 1.5 : 0);
    return Math.min(10, Math.max(0, score));
  }
}
```

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.

   Solution.

   Rubric:

(a)
- (2 pts) correct signature.
- (2 pts) correct base case.
- (5 pts) correctly finds the string inside the non-base case, and recurses correctly.

```
static List<String> collectParenthesizedStrings(String s) {
  if (!s.contains("(")) {
    return new ArrayList<>();
  } else {
    int l = s.indexOf("(");
    int r = s.indexOf(")");
    String inside = s.substring(l + 1, r);
    List<String> rest = collectParenthesizedStrings(s.substring(r + 1));
    List<String> all = new ArrayList<>();
    all.add(inside);
    all.addAll(rest);
    return all;
  }
}
```

(b) *Rubric:*

- (1 pt) correct driver method.
- (1 pt) tail recursive method uses `private` access modifier.
- (3 pts) correct conditionals.
- (3 pts) correctly updates accumulator.

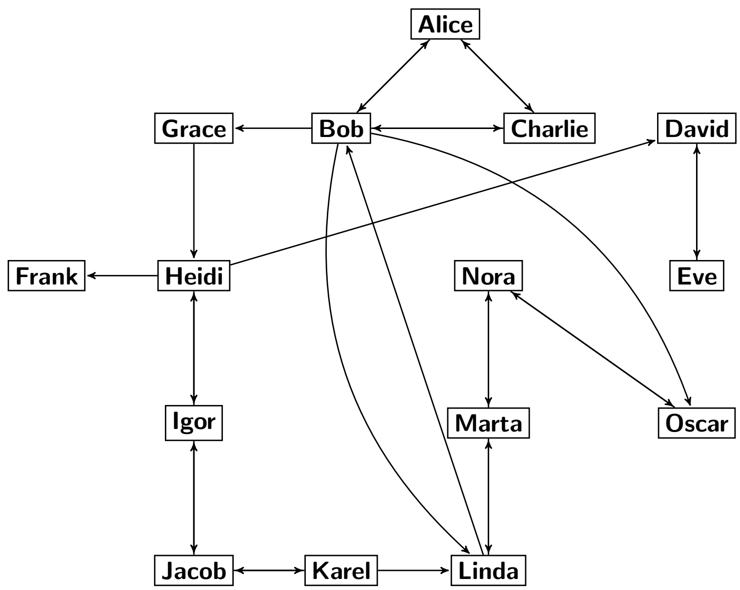
```
static List<String> collectParenthesizedStringsTR(String s) {
  List<String> acc = new ArrayList<>();
  return collectParenthesizedStringsTRHelper(s, acc);
}

private static List<String> collectParenthesizedStringsTRHelper(String s,
                                    List<String> acc) {
  if (!s.contains("(")) {
    return acc;
  } else {
    int l = s.indexOf("(");
    int r = s.indexOf(")");
    String inside = s.substring(l + 1, r);
    List<String> newAcc = new ArrayList<>();
    newAcc.addAll(acc);
    newAcc.add(inside);
    return collectParenthesizedStringsTRHelper(s.substring(r + 1), newAcc);
  }
}
```

(c) *Rubric:*

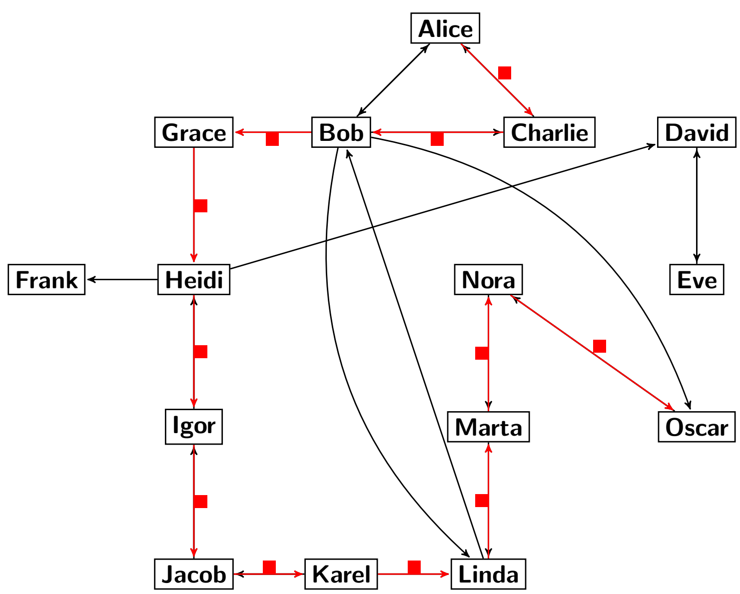
- (1 pt) correct signature.
- (1 pt) localized accumulators.
- (2 pts) correct loop condition.
- (2 pts) correctly updates local variables.
- (2 pt) correct return value.

```
static List<String> collectParenthesizedStringsLoop(String s) {
  List<String> acc = new ArrayList<>();
  while (s.contains("(")) {
    int l = s.indexOf("(");
    int r = s.indexOf(")");
    String inside = s.substring(l + 1, r);
    List<String> newAcc = new ArrayList<>();
    newAcc.addAll(acc);
    newAcc.add(inside);
    acc = newAcc;
    s = s.substring(r + 1);
  }
  return acc;
}
```

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.

Solution.

Rubric:

- (35 pts) 12 blanks, each is worth 2.5 points. The one that makes the recursive call is worth 7.5 points. These are all-or-nothing points.

```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 (visited.contains(f)) {
      return 0; // If visited, no length should be added from this path.
    } else if (friendsList.get(f).isEmpty()) {
      return 0; // If no friends are listed.
    } else {
      visited.add(f);
      int max = 0;
      for (String friend : friendsList.get(f)) {
        int pathLength = longestFriendPathHelper(friend, friendsList, visited);
        if (pathLength > max) {
          max = pathLength;
        }
      }
      visited.remove(f);
      return max + 1;
    }
  }
}
```