C212 Midterm Exam (80 points)
Feb 28, 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	15	
4	20	
Total:	80	

Name: ____________________

IU Email: ____________________

1. (20 points) Design the `double computeOvertimePay(double hrRate, double noHrs, boolean onVacation, double taxRate)` method that computes the amount of overtime pay (note: **only** the overtime pay) given to an employee under the following conditions:

   - An employee's base overtime pay rate is 1.5 times their hourly rate.
   - If the employee is on vacation, their pay rate is 2 times their hourly rate rather than 1.5.
   - If the number of hours is less than or equal to 40, then no overtime pay is given.
   - If the number of hours is greater than 70, then the resulting gross pay (before taxes) is increased by 15%.
   - The gross pay is subject to the tax *percentage* passed to the method.

   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. You must write tests that *fully cover* all possible kinds of inputs.

   The skeleton code is on the next page.

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

class ComputeOvertimePayTester {

  @Test
  void testComputeOvertimePay() {









  }
}

class ComputeOvertimePay {

  /**
   *
   *
   * @param
   * @param
   * @param
   * @param
   * @return
   */
  _________ _________ computeOvertimePay(double hrRate, double noHrs,
                                         boolean onVacation, double taxRate) {













  }
}
```

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

   You are writing a multiple choice question exam score calculator. Correct answers award three points, incorrect answers remove one point, and a `"?"` represents a guess, which neither awards nor removes points.

   (a) *(9 points)* Design the *standard recursive* `score` method, which receives two `String` arrays representing the expected answers E and the actual answers A respectively. The `score` method should return the score of the student as a percentage. If the raw score is less than zero, then return a zero. *Hint: designing a helper method is a good idea!*

```
String[] E = new String[]{"A", "C", "D", "A", "B", "B", "D", "C", "C"};
score(E, new String[]{"A", "C", "D", "C", "B", "B", "C", "C", "C"} => 70.3
score(E, new String[]{"A", "C", "D", "A", "B", "B", "D", "C", "C"} => 100.0
score(E, new String[]{"A", "C", "?", "C", "?", "B", "?", "C", "C"} => 51.8
```

   (b) *(8 points)* Design the `scoreTR` and `scoreTRHelper` methods. The former acts as the driver to the latter; the latter solves the same problem as `score` does, but it instead uses tail recursion. Remember to include the relevant access modifiers!

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

3. (15 points) Design the `nthMostFrequentChar` method that, when given a *non-empty* `char[]` A and an `int` n, returns the n^{th} most frequent character from the array. You may assume that $1 \leq n \leq |A|$. We provide three examples, but you **cannot** use these in your tests. **You cannot use the Stream API.** In designing this method, follow the template from class; write the signature, purpose statement, testing, and *then* do the implementation. You can use abbreviated array notation, e.g., $[1, 2, 3, ..., n]$, in your tests instead of Java code.

   The skeleton code is on the next page.

   This problem is harder than it looks. As a hint, first compute the frequencies of each character, then traverse the map and look for the n^{th} most frequent. This can be achieved by removing keys from the map via `.remove`. A more clever (and performant) solution can be achieved with a priority queue!

```
nthMostFrequentChar(['a', 'c', 'c', 'a', 'c', 'b', 'c', 'd', 'f', 'b'], 2) => 'a'
nthMostFrequentChar(['a', 'a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd'], 1) => 'a'
nthMostFrequentChar(['d', 'c', 'd', 'b', 'd', 'c', 'd', 'c', 'b', 'a'], 3) => 'b'
```

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

class nthMostFrequentCharTester {

  @Test
  void testnthMostFrequentChar() {






  }
}

import java.util.*; // Import all necessary collections.

class nthMostFrequentChar {

  /**
   *
   * @param
   * @param
   * @return
   */
  ____________ ___________________ nthMostFrequentChar(__________ __________) {













  }
}
```

4. (20 points) Oh no! Hemeshwar's Shih Tzu, Tipsy, ate part of this exam and we need you to add the missing code. Fill in the blanks to complete this method implementation. Additionally, write at least three examples, one for each case (you need to figure out what the cases are). You can use abbreviated set notation, e.g., $\{1, 2, 3, ..., n\}$, in your tests instead of Java code.

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

class ContainsNCommonValuesTester {

  @Test
  void testContainsNCommonValues() {





  }
}

import java.util.Set;
import java.util.HashSet;

class ContainsNCommonValues {

  /**
   * Determines whether or not the given sets contains exactly n common elements.
   * These elements are compared via their .equals method.
   * @param s1 - first set.
   * @param s2 - second set.
   * @param n - an integer 1 <= n <= min(|s1|, |s2|)
   * @return true if they contain common values and false otherwise.
   */
  static _____ __________ containsNCommonValues(Set<_____> s1,
                                                Set<_____> s2,
                                                int n) {
    int counter = ________;
    Set<___> unionedSet = ____________________;
    unionedSet.addAll(____);
    unionedSet.________(____);
    for (_____ v : unionedSet) {
      if (____________________________) {
        counter = _______________________;
      }
    }
    return counter ____ ____;
  }
}
```

Scratch work

Scratch work