C212 Midterm Exam Makeup (80 points)
Mar 06, 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 computeBonusPay(double baseSalary, int yearsOfService, boolean achievedTarget, double salesAmount, double targetSales)` method that calculates the amount of bonus pay (**only the bonus pay**) given to an employee under the following conditions:

   - A base bonus rate of 10% of the base salary is given to employees who have achieved their sales target.
   - Employees with more than 5 years of service receive an additional 5% bonus of their base salary.
   - If the sales amount exceeds the target sales by more than 50%, the employee receives an additional bonus of 25% of the base salary.
   - If the employee has not achieved their sales target, they receive a flat bonus of 2% of their base salary, regardless of sales amount or years of service.
   - The total bonus amount is reduced by a flat tax rate of 25%.

   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 ComputeBonusPayTester {

  @Test
  void testComputeBonusPay() {









  }
}

class ComputeBonusPay {

  /**
   *
   *
   * @param
   * @param
   * @param
   * @param
   * @return
   */
  static double computeBonusPay(double baseSalary, int yearsOfService,
                                double salesAmount, double targetSalesAmount) {
```

```
  }
}
```

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

   You are developing a system to help monitor and reduce electricity usage in households. The system calculates the total energy consumption based on the number of hours each appliance is used daily. Different appliances have different power ratings, which affects their energy consumption.

   (a) *(9 points)* Design the *standard recursive* `calculateEnergy` method, which takes two arrays as inputs: one representing the power ratings of various appliances P (in watts) and the other representing the hours each appliance is used daily H. The method should return the total energy consumed in a day by all appliances in kilowatt-hours (kWh). Appliances not used (indicated by 0 hours) should not contribute to the total. *Hint: designing a helper method is a good idea!*

   ```
   double[] P = new double[]{100, 1500, 300, 1200, 60}; // Power ratings
   calculateEnergy(P, new double[]{2, 4, 3, 0, 5}) => 8.46 kWh
   calculateEnergy(P, new double[]{0, 2, 0, 1, 0}) => 3.3 kWh
   calculateEnergy(P, new double[]{1, 0, 2, 3, 0}) => 2.16 kWh
   ```

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

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

3. (15 points) Design the `nthCommonWordLength` method that, when given a *non-empty* list of strings W and an `int` n, returns the length of the n^{th} most common word length in the list. Assume $1 \leq n \leq$ the number of unique word lengths in W. We provide four 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 at first glance. As a hint, compute a map of word lengths to a list of words that have that length. Then, find the most-common word length and remove that key/value pair. Keep doing this until n is zero.

```
nthCommonWordLength(["apple", "bat", "cat", "dog", "elephant", "fish", "goat"], 2) => 4
nthCommonWordLength(["hello", "world", "java", "python", "code"], 1) => 4
nthCommonWordLength(["one", "two", "three", "four", "five", "six"], 3) => 5
nthCommonWordLength(["cb", "bc", "a", "abc", "abc", "abc", "abc", "abc"], 2) => 2
```

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

class NthCommonWordLengthTester {

  @Test
  void testNthCommonWordLength() {









  }
}

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

class NthCommonWordLength {

  /**
   *
   * @param
   * @param
   * @return
   */
  ____________ ____________________ nthCommonWordLength(________________, _______) {

















  }
}
```

4. (20 points) Oh no! Joshua's cat, Nebraska, scratched away 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 MergeKSortedListsTester {

  @Test
  void testMergeKSortedLists() {




  }
}
```

```
import java.util.List;
import java.util.ArrayList;
import java.util.PriorityQueue;

class MergeKSortedLists {

  /**
   * Merges k sorted lists into one sorted list and returns it.
   * The input is a list of lists, where each inner list is sorted in ascending order.
   * @param lists - a list containing k sorted lists of integers.
   * @return a single list containing all elements from the k lists in sorted order.
   */
  static _____ __________ mergeKSortedLists(List<List<___>> lists) {
    PriorityQueue<___> pq = ____________________;
    for (_____________ list : lists) {
      for (___ value : list) {
        pq.___(_______);
      }
    }

    List<Integer> result = new ArrayList<>();
    while (_____________) {
      result.add(pq._____);
    }
    return result;
  }
}
```

Scratch work

Scratch work