C212 Midterm Exam (80 points)
Feb 28, 2024

C212 Midterm Exam Rubric

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.

Solution.

Rubric:

- (1 pt) example when $noHrs \leq 40$ and not on vacation
- (1 pt) example when $noHrs \leq 40$ and on vacation
- (1 pt) example when $noHrs > 40$ and not on vacation
- (1 pt) example when $noHrs > 40$ and on vacation
- (1 pt) example when $noHrs > 70$ and not on vacation
- (1 pt) example when $noHrs > 70$ and on vacation
- (2 pts) purpose statement sensible
- (2 pts) signature is correct
- (2 pts) simple case when no OT pay
- (4 pts) correctly computes the new pay rate
- (2 pts) correctly computes the gross pay when number of hours > 70
- (2 pts) correctly applies the tax amt

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

class ComputeOvertimePayTester {

  @Test
  void computeOvertimePay() {
    assertAll(
      () -> assertEquals(0, Problem1.computeOvertimePay(10, 35, false, 10)),
      () -> assertEquals(0, Problem1.computeOvertimePay(10, 40, true, 10)),
      () -> assertEquals(67.5, Problem1.computeOvertimePay(10, 45, false, 10)),
      () -> assertEquals(90, Problem1.computeOvertimePay(10, 45, true, 10)),
      () -> assertEquals(543.375, Problem1.computeOvertimePay(10, 75, false, 10)),
      () -> assertEquals(724.4999999999999, Problem1.computeOvertimePay(10, 75, true, 10)));
  }
}

class ComputeOvertimePay {

  static double computeOvertimePay(double hrRate, double noHrs,
                                   boolean onVacation, double taxRate) {
    if (noHrs <= 40) return 0;
    else {
      double newRate = onVacation ? hrRate * 2 : hrRate * 1.5;
      double grossPay = (noHrs - 40.0) * newRate;
      if (noHrs > 70) {
        grossPay *= 1.15;
      }
      return grossPay * ((100 - taxRate) / 100);
    }
  }
}
```

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.

   Solution.

   Rubric:

(a)
- (2 pts) correct signature.
- (3 pts) correct return values in non-helper (¡ 0)
- (4 pts) uses standard recursion and accumulates the index *(if it's not private that's fine)*

```
static double score(String[] E, String[] A) {
  return Math.max(0, scoreHelper(E, A, 0)) / (A.length * 3) * 100;
}

private static double scoreHelper(String[] E, String[] A, int idx) {
  if (idx >= E.length) {
    return 0;
  } else if (E[idx].equals(A[idx])) {
    return scoreHelper(E, A, idx + 1) + 3;
  } else if (A[idx].equals("?")) {
    return scoreHelper(E, A, idx + 1);
  } else {
    return scoreHelper(E, A, idx + 1) + -1;
  }
}
```

(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 and n.

```
static double scoreTR(String[] E, String[] A) {
  return Math.max(0, scoreTRHelper(E, A, 0, 0)) / (A.length * 3) * 100;
}

private static double scoreTRHelper(String[] E, String[] A, int idx, int score) {
  if (idx >= E.length) {
    return score;
  } else if (E[idx].equals(A[idx])) {
    return scoreTRHelper(E, A, idx + 1, score + 3);
  } else if (A[idx].equals("?")) {
    return scoreTRHelper(E, A, idx + 1, score);
  } else {
    return scoreTRHelper(E, A, idx + 1, score + -1);
  }
}
```

(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 double scoreLoop(String[] E, String[] A) {
  double score = 0;
  int idx = 0;
  while (!(idx >= E.length)) {
    if (E[idx].equals(A[idx])) {
      idx = idx + 1;
      score = score + 3;
    } else if (A[idx].equals("?")) {
      idx = idx + 1;
    } else {
      idx = idx + 1;
      score = score + -1;
    }
  }
  return Math.max(0, score) / (A.length * 3) * 100;
}
```

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'
```

Solution.

Rubric:

- (4 pts) at least two coherent examples.
- (2 pts) sensible purpose statement.
- (9 pts) definition works as expected. *As I said, this is a pretty hard problem, so be lenient when grading. If they populate the frequency map correctly, give them 4 points. The rest can be awarded as needed.*

```
// Note: this is the naive solution that I imagine
// most students will give. The next page contains a more
// optimal yet complex solution.
static char nthMostFrequentChar(char[] A, int n) {
  Map<Character, Integer> M = new HashMap<>();
  for (char c : A) { M.put(c, M.getOrDefault(c, 0) + 1); }
  char mc = '\0';
  while (n > 0) {
    int max = -1;
    for (char c : M.keySet()) {
      if (M.get(c) > max) {
        max = M.get(c);
        mc = c;
      }
    }
    M.remove(mc);
    n--;
  }
  return mc;
}
```

```
static char nthMostFrequentCharPq(char[] A, int n) {
  Map<Character, Integer> M = new HashMap<>();
  for (char c : A) { M.put(c, M.getOrDefault(c, 0) + 1); }
  // Create a comparator that compares values based off their frequency.
  Queue<Map.Entry<Character, Integer>> pq = getEntries();
  M.entrySet().forEach(pq::add);
  while (n > 1) {
    pq.poll();
    n--;
  }
  return pq.peek().getKey();
}

private static Queue<Map.Entry<Character, Integer>> getEntries() {
  Comparator<Map.Entry<Character, Integer>> cmp = (t0, t1) -> {
    // If the VALUES are the same, then we need to return the one with the lowest character.
    if (t0.getValue().equals(t1.getValue())) {
      return Integer.compare(0, t1.getKey().compareTo(t0.getKey()));
    } else {
      // Otherwise, they are not the same.
      return t1.getValue() - t0.getValue();
    }
  };
  // Add each entry to the pq.
  return new PriorityQueue<>(cmp);
}
```

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.

Solution.

Rubric:

- (2 pts) example where there are n common elements.
- (2 pts) example where there are not n common elements.
- (2 pts) example where there are no common elements.
- (3 pts) `<T>`
- (1 pt) `int counter = 0`
- (3 pts) `addAll`, `s1`, `addAll(s2)`
- (5 pts) `(s1.contains(v) && s2.contains(v))`
- (1 pt) `counter = counter + 1` or some equivalent
- (1 pt) `return counter == n` or some equivalent

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

class ContainsNCommonValuesTester {

  @Test
  void testContainsNCommonValues() {
    assertAll(
      () -> assertTrue(Problem4.containsNCommonValues(
                       Set.of(3, 4, 2, 1, 9, 10, 20),
                       Set.of(5, 10, 2, 44, 56, 23, 3),
                       3)),
      () -> assertFalse(Problem4.containsNCommonValues(
                        Set.of(3, 4, 2, 1, 9, 10, 20),
                        Set.of(5, 10, 2, 44, 56, 23, 3),
                       5)),
      () -> assertTrue(Problem4.containsNCommonValues(
                       Set.of(1,2,3,4,5,6,7,8),
                       Set.of(10,20,30),
                       0)));
  }
}

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

class FindMaxString {

  /**
   * 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 <T> boolean containsNCommonValues(Set<T> s1,
                                           Set<T> s2,
                                           int n) {
    int counter = 0;
    Set<T> unionedSet = new HashSet<>();
    unionedSet.addAll(s1);
    unionedSet.addAll(s2);
    for (T v : unionedSet) {
      if (s1.contains(v) && s2.contains(v)) {
        counter++;
      }
    }
    return counter == n;
  }
}
```