C212 Midterm Exam (80 points)
Oct 9, 2024

PLEASE READ ALL DIRECTIONS BEFORE STARTING YOUR EXAM. DO NOT OPEN UNTIL YOU ARE TOLD TO DO SO.

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 out documentation comments and tests. Of course, doing so may aid you in creating 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.

The exam has 80 total points split across two parts. Part I contains two required problems. Part II contains two problems and you are required to choose and solve one of them. Cross out the pages of the problem you do not want graded. Answering both questions will result in only problem #3 being graded.

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.

Good luck!

Question	Points	Score
1	20	
2	25	
3	35	
4	35	
Total:	80	

Name: ______________________________

IU Email: ______________________________

Part I

Recommended Time: 35 minutes

2 Problems

1. (20 points) Design the `String determineRating(String genre, int timeLength, boolean hasViolence, boolean hasStrongLanguage, boolean hasAdultThemes)` method, which determines the parental rating of a movie. The method cannot return `null`, and must return a valid "rating" as defined below.

```
A rating is one of:
- "G"
- "PG"
- "PG-13"
- "R"
- "A"

A genre is one of:
- "Action"
- "Comedy"
- "Drama"
- "Horror"
```

The ratings are on an interval scale: `"G"` < `"PG"` < `"PG-13"` < `"R"` < `"A"`. You should correspond the ratings to the natural numbers $1, 2, 3, 4$, and 5 respectively.

Below are the criteria for rating a movie:

- If the `genre` is `"Horror"` or `"Action"`, the base rating is `"PG-13"`. If the `genre` is `"Drama"`, the base rating is `"PG"`. If the `genre` is `"Comedy"`, the base rating is `"G"`.
- If the movie runtime, `timeLength`, is less than 60, their (current) rating is lowered by one level. If $60 \leq$ `timeLength` ≤ 120, their (current) rating remains the same. Otherwise, their (current) rating is increased by one level.
- If the movie `hasViolence`, the rating is increased by one level.
- If the movie `hasStrongLanguage`, the rating is increased by one level.
- If the movie `hasAdultThemes`, the rating is increased by two levels.

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. Because there are several permutations of ratings and genres, you only need to write one test for each rating. None of your tests should produce a `null` rating.

The skeleton code is on the next page.

You may assume the existence of the following method:

`String rating(int val)` returns a rating, as described above, for a given integer in the interval $[1, 5]$. Any numbers outside of this interval return `null`.

```
rating(1) => "G"
rating(5) => "A"
rating(-1) => null
rating(8) => null
```

```
import static Assertions.assertEquals;

class MovieRatingTester {

  @Test
  void testMovieRating() {
    assertEquals(______, determineRating(____________________________________________));
    assertEquals(______, determineRating(____________________________________________));
    assertEquals(______, determineRating(____________________________________________));
    assertEquals(______, determineRating(____________________________________________));
    assertEquals(______, determineRating(____________________________________________));
  }
}

class MovieRating {

  /**
   *
   *
   * @param
   * @param
   * @param
   * @param
   * @param
   * @return
   */
  static String determineRating(String genre, int timeLength, boolean hasViolence,
                                boolean hasStrongLanguage, boolean hasAdultThemes) {













  }
}
```

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

   (a) *(9 points)* Design the *standard recursive* `findMaxWordLength` method, which receives a `String[]` and returns the length of the longest word in the array. You may assume that the array contains at least one string. Hint: you should design a helper method to recurse over the array. The helper method *must* be standard recursive!

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

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

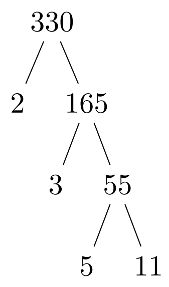
Part II

Recommended Time: 40 minutes

Part II contains two problems, but you should answer only one. *CROSS OUT THE ENTIRE PROBLEM THAT YOU DO NOT WANT GRADED.* Answering both questions will result in only #3 counting towards your grade.

3. (35 points) The *prime factorization* problem is about finding prime numbers that multiply to some positive integer. That is, given a positive integer n, we want to find its prime factors. It is an open mathematics and computer science question whether it is possible to find the prime factorization of a positive integer in *polynomial time.* The naïve algorithm is to iterate over the primes from $2, 3, ..., n$, find the lowest prime p that divides n, divide n by p, then repeat until n is prime.

   We can visualize this algorithm via a *prime factor tree.* For example, let's find the prime factorization of 330. The smallest prime starting from 2 that divides 330 is 2. So, the root of the tree is 330, the left branch leads to a prime factor, and the right is a smaller sub-problem, that being $330/2 = 165$. The smallest prime that divides 165 is 3, so we get 3 in the left branch and 55 in the right branch. Repeat once more to get 5 and 11, and we stop because 11 is prime.



```
class PrimeFactorTree {

  /**
   * Returns whether a positive integer is prime.
   * @param n integer > 0.
   * @return true if prime, false otherwise.
   */
  static boolean isPrime(int n) { /* Implementation not shown. */ }

  /**
   * Creates a list of all the prime factors of n. The resulting list
   * should contain only prime numbers and have a product equal to the input.
   * @param n integer >= 2.
   * @return list of prime factors.
   */
  static List<Integer> primeFactors(int n) { /* To be implemented in part (b). */ }

  /**
   * Creates the prime factor tree from a given integer. The
   * "tree" is a list of prime factors where the ith item is
   * the root of a tree, the (i + 1)th branch is the prime factor,
   * and the (i + 2)th prime factor tree.
   * @param n integer >= 2.
   * @return a list representing the prime factor tree as described.
   */
  static List<Integer> primeFactorsTree(int n) { /* To be implemented in part (c). */ }
}
```

(a) *(7 points)* Write JUnit test cases for the `isPrime` method. You do not need to know how it works, only that it receives a positive integer n and returns `true` if it is prime and `false` otherwise.

```
import static Assertions.*;

class PrimeFactorTreeTester {

  @Test
  void testIsPrime() {
    assertTrue(isPrime(_______________));
    assertTrue(isPrime(_______________));
    assertFalse(isPrime(_______________));
    assertFalse(isPrime(_______________));
    assertFalse(isPrime(_______________));
  }
}
```

(b) *(14 points)* Design the `static List<Integer> primeFactors(int n)` that, when given a positive integer $n \geq 2$, returns a list of all the prime factors of n. For example, `primeFactors(330)` returns `[2, 3, 5, 11]` because all numbers in the returned list are prime and their product equals the input 330. Your definition must call `isPrime` in order to receive full credit.

```
/**
 * Creates a list of all the prime factors of n. The resulting list
 * should contain only prime numbers and have a product equal to the input.
 * @param n integer >= 2.
 * @return list of prime factors.
 */
static List<Integer> primeFactors(int n) {



}
```

(c) *(14 points)* Design the `static List<Integer> primeFactorsTree(int n)` method that creates a "factor tree" as a list. That is, consider once again the prime factorization of 330. The returned list should be $[330, 2, 165, 3, 55, 5, 11]$, because the left branch of 330 leads to the prime factor 2, and the right branch leads to a factoring of 165. Your definition must call both `isPrime` and `primeFactors` in order to receive full credit.

```
/**
 * Creates the prime factor tree from a given integer. The
 * "tree" is a list of prime factors where the ith item is
 * the root of a tree, the (i + 1)th branch is the prime factor,
 * and the (i + 2)th prime factor tree.
 * @param n integer >= 2.
 * @return a list representing the prime factor tree as described.
 */
static List<Integer> primeFactorsTree(int n) {







}
```

4. (35 points) Design the generic `static <T, U> String convertToString(Map<T, U> M)` method that, when given a map of keys to values of type T and U respectively, returns a “stringified” version of the map. A stringified version of the map is enclosed by braces, and key/value pairs are separated by commas and spaces. Consider the following example:

```
Map<String, Integer> M1 = new LinkedHashMap<>();
M1.put("Siobahn", 35000);
M1.put("Gilmore", 16000);
M1.put("Jack", 90);
M1.put("Jeremy", 80000);
```

Calling `convertToString(M1)` produces the string:

```
"{Siobahn=35000, Gilmore=16000, Jack=90, Jeremy=80000}"
```

The exact returned string depends on the instantiated map. So, writing tests for a `HashMap` is problematic, but writing tests for `TreeMap` and `LinkedHashMap` is simple. Note: you cannot simply call `.toString` on the map object; you must do the bulk of the work yourself. As a hint: call `toString` on each key/value pair in the map to receive its string representation.

The tester and implementation skeleton code is on the next page. You can use math notation for your test cases and not full declarations of lists, but you must specify what kind of list you are creating with the appropriate constructor(s). Write three tests. It is up to you to figure out what are the correct tests!

```
import static Assertions.assertEquals;

class ConvertToStringMapTester {

  @Test
  void testConvertToString() {
    assertEquals(__________________________________________________,
      convertToString(__________________________________________________);
    assertEquals(__________________________________________________,
      convertToString(__________________________________________________);
    assertEquals(__________________________________________________,
      convertToString(__________________________________________________);
  }
}

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

class ConvertToStringMap {

  /**
   *
   *
   * @param
   * @return
   */
  static <T, U> String convertToString(Map<T, U> M) {










  }
}
```

Scratch work

Scratch work