C212 Midterm Exam (80 points)
Oct 9, 2024

C212 Midterm Exam Rubric

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.

Solution.

Rubric:

- 5 points for tests, 1 point per DISTINCT rating.
- 5 points for the Javadoc. 2 points for the purpose, and 3 for the parameter and return tags. Take off at most 3 points, one for each missing.
- 4 points for checking for each rating.
- 1 point for validating the time length, violence, strong language, and adult themes.
- 2 points for correctly calling `rating`. No points are awarded for this portion if the method can return `null`.

```
import static Assertions.assertEquals;

class MovieRatingTester {

    void testDetermineRating() {
        assertEquals("G", MovieRating.determineRating("Comedy", 60, false, false, false));
        assertEquals("PG", MovieRating.determineRating("Action", 50, false, false, false));
        assertEquals("PG-13", MovieRating.determineRating("Horror", 30, true, false, false));
        assertEquals("R", MovieRating.determineRating("Drama", 121, false, true, false));
        assertEquals("A", MovieRating.determineRating("Horror", 300, true, true, true));
    }
}

class MovieRating {

  /**
   * Rates a movie according to a criteria.
   * @param genre the genre of the movie.
   * @param timeLength length of movie in minutes..
   * @param hasViolence whether the movie has violence.
   * @param hasStrongLanguage whether the movie has strong language.
   * @param hasAdultThemes whether thhe movie has adult themes.
   * @return a movie rating, either G, PG, PG-13, R, or A.
   */
  static String determineRating(String genre, int timeLength, boolean hasViolence,
                              boolean hasStrongLanguage, boolean hasAdultThemes) {
    int rating = 0;
    if (genre.equals("Action") || genre.equals("Horror")) { rating = 3; }
    else if (genre.equals("Comedy")) { rating = 1;  }
    else { rating = 2; }

    if (timeLength < 60) { rating -= 1; }
    else if (timeLength > 120) { rating += 1; }
    if (hasViolence) { rating += 1; }
    if (hasStrongLanguage) { rating += 1; }
    if (hasAdultThemes) { rating += 2; }
    return rating(Math.max(1, Math.min(rating, 5)));
  }
}
```

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

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)*. If they forgot the cast on `Math.max`, that's fine. +1 point for using recursion. +2 points for having correct conditionals or using `Math.max`. +1 for passing the correct values to the method.

```
static int findMaxWordLength(String[] arr) {
  return longestStringHelper(arr, 0);
}

private static int findMaxWordLengthHelper(String[] arr, int idx) {
  if (idx >= arr.length) return 0;
  else {
    return (int) Math.max(arr[idx].length(), findMaxWordLengthHelper(arr, idx+1));
  }
}
```

(b) *Rubric:*

- (1 pt) correct driver method.
- (1 pt) tail recursive method uses `private` access modifier.
- (3 pts) correct conditionals. +2 for a correct base case, +1 for a correct "else/else if" clause.
- (3 pts) correctly updates accumulator and n.

```
static int findMaxWordLengthTR(String[] arr) {
  return findMaxWordLengthTRHelper(arr, 0, 0);
}

private static int findMaxWordLengthTRHelper(String[] arr, int idx, int len) {
  if (idx >= arr.length) return len;
  else {
    return findMaxWordLengthTRHelper(arr, idx + 1, (int) Math.max(len, arr[idx].length()));
  }
}
```

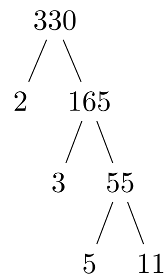
(c) *Rubric:*

- (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 int findMaxWordLengthLoop(String[] arr) {
  int idx = 0;
  int len = 0;
  while (!(idx >= arr.length)) {
    len = (int) Math.max(len, arr[idx].length());
    idx++;
  }
  return len;
}
```

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.

Solution.

Rubric:

- 5 tests, must be correct. 2 points for the "true" cases, and 3 for the "false" cases.

```
@Test
void testIsPrime() {
    assertTrue(isPrime(127));
    assertTrue(isPrime(13));
    assertFalse(isPrime(0));
    assertFalse(isPrime(1));
    assertFalse(isPrime(10));
}
```

(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.

Solution.

Rubric:

- 2 points for the initialization of the list.
- 2 points for the return.
- 4 points for a correct outer loop.
- 6 points for the condition check, adding numbers to the loop, and so forth. -3 points if they do not have an outer loop/do not reset i back to 2. Not sure how to divide partial points otherwise.

```
static List<Integer> primeFactors(int n) {
    List<Integer> ls = new ArrayList<>();
    while (n > 1) {
        for (int i = 2; i <= n; i++) {
            if (isPrime(i) && n % i == 0) {
                ls.add(i);
                n /= i;
                break;
            }
        }
    }
    return ls;
}
```

(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.

Solution.

Rubric:

- 2 points for calling `primeFactors` with n.
- 2 points for loop while not `isPrime(n)`. Not sure how they can solve it otherwise.
- 10 points for correctly adding factors to the list. Award partial points as needed but be consistent.

```
static List<Integer> primeFactorsTree(int n) {
    List<Integer> ls = new ArrayList<>();
    List<Integer> primeFactors = primeFactors(n);
    ls.add(n);
    while (!isPrime(n)) {
        int p = primeFactors.removeFirst();
        ls.add(p);
        n /= p;
        ls.add(n);
    }
    return ls;
}
```

4. (35 points) **Solution.**

Rubric:

- (2 points) Javadoc comment exists and all fields are full. This is an all-or-nothing point value.
- (3 points) The empty map test.
- (6 points) +2 points for testing a `LinkedHashMap`. +4 points for if it's correct.
- (6 points) +2 for testing a `TreeMap`. +4 points for if it's correct.
- (10 points) Correctly looping over the keys and concatenating them onto the string. +4 for adding the key as a string. +4 for adding the value as a string. +2 for adding the `=` character. +2 for adding the trailing comma. *Note: calling `toString` on the object is not required as Java coerces the type into a string and implicitly calls its `toString`. You can see I do this for the value.*
- (5 points) accounts for edge case of removing the last comma and space.
- (3 points) returning a string of any kind.

```java
/**
 * A sensible comment...
 * @param M
 * @return
 */
static <T, U> String convertToString(Map<T, U> M) {
  String s = "{";
  for (T t : M.keySet()) {
    s += t.toString() + "=" + M.get(t) + ", ";
  }
  if (!M.isEmpty()) {
    s = s.substring(0, s.length() - 2);
  }
  return s + "}";
}
```