Pset 1 Notes (CSCI E-10B)

General Information

a. Problem set 1 is due Monday, Sep 25, 2023 at 9:00 AM Eastern Time. You can resubmit your assignment as often as desired, but each submission's zip must include every file that you want to be graded.

b. We will deduct 10% for a homework assignment that is turned in up to 3 days late. 20% will be deducted if the homework is more than 3 days late. No homework will be accepted more than 7 days late. The last submission controls the late penalty for the entire assignment.

c. Don't procrastinate! Note, though, that the pset covers material from lecture 3 so you will probably need to defer some questions until you've viewed future lectures and attended or viewed future sections.

d. The course website pages from which problem sets can be retrieved and submitted are easily located via the home page's "Lecture …" buttons.

e. Your Java code must compile. Programs that do not compile will automatically have their score divided by two.

f. Your programs must behave as specified. Do everything the specs say to do. Do not do more than the specs say to do. Precisely follow the directions in the pset (e.g. use the exact filenames specified in the pset), except make adjustments specified by the staff in these notes or on the Ed discussion board.

g. Some of the work is designated as "extra credit." In this course, extra credit points are kept separate from regular credit points. They only come into play at the end of the semester when Dr. Leitner is assigning final grades. If you are on the cusp between, say, a B+ and an A-, extra credit points can influence that decision.[1]

   If the pset says "identify which problem(s) you want treated as extra credit," ignore that! We automatically allocate points to maximize your regular credit score because that contributes the most to your grade in the course.

   If your submission gets a late penalty, then you won't get any extra credit points for that submission.

h. This course emphasizes programming style. See the Ed posts titled *Reasonably-commented programs*, *Javadoc commenting* and *Java Style Guide*. Inadequately-commented or -styled programs will lose points.

Pencil-and-Paper Exercises

The Pencil-and-Paper Exercises must be submitted as text files. Text files can most easily be created by using a programmer's editor (VS Code, Notepad++, Sublime, Textpad, ...) or by using a text editor such as Windows Notepad. Answers submitted in another file format such as **.docx**, **.pages**, **.rtf**, **.pdf**, ... will not be accepted.[2]

Nobody should lose points on the Pencil-and-Paper Exercises programming questions! After you have thought through the problem and written down your answer, check your work by writing simple programs to verify your answer. Don't submit your test programs; only submit your answer for each problem.

a. Problem 1 (**`Mystery.txt`**) only needs to contain something like "After the code has been executed, the values of the elements in array a1 will be [whatever ...]."

b. Problem 2 (**`RecTriangle.java`**) must contain the slightly-modified recursive **`printTriangle()`** method.

[1] The decision is also influenced by your teaching assistant's feedback, so it's a good idea to get to know that person.

[2] Fyi, note that **`.java`** files are stored as text files.

c. Problem 3 (**`Enumeration.java`**) needs to contain the program in the pset, with the body of the **`switch`** statement (**`... // YOU NEED TO WRITE THIS PART`**) replaced with Java code that causes the program to behave as specified. You are not allowed to make any changes to the rest of the program.

d. Problem 4 (**`Power.java`**) must contain two things:
   1. The rewritten **`power()`** method. Your solution cannot use existing methods like **`Math.pow()`**.
   2. The answer to part ii only needs to be a comment in **`Power.java`** like **`// The modified power() method will be called a total of x times to compute power( foobar, 1024 ).`**

      `foobar` represents some **`double`** value. We'll discuss counting method calls in week 2's section.

Programming Problems

Submit complete and correct programs that behave exactly as specified, except make adjustments specified by the course's staff in these notes or on the Ed discussion board.

Your programs' interaction with the user should match what's in the assignment, and when sample cases are shown in the problem set, your program must produce the same output when given the sample case's input.

Whenever you are asked to write a specific method that's not **`main()`**, calling that method must not cause any output to be sent to the terminal unless the instructions say otherwise.

If you are told to provide code that demonstrates your method, you must follow these rules:

1. You must not require your TA to define the test cases.
2. You must not require your TA to read your source code in order to decipher your code's output.

Instead, you need to implement a robust set of self-documenting test cases that convincingly demonstrate your method. Here's how I would do that: For a well-chosen set of test cases, my demo would contain statements analogous to:

```
System.out.printf( "someMethod( %d, %d ) = %d\n", int1, int2,
                                                   someMethod( int1, int2 ) );
```

You will replace "**`someMethod`**" with the actual name of the method being demonstrated, and you'll adjust the number of arguments, their data types, the method's return type, etc.

a. Problem 5 (**`Palindrome.java`**) must contain a recursive method named **`isPalindrome()`** that accepts a **`String`** argument, and returns a **`boolean`** value indicating whether the **`String`** is a palindrome.

   Your **`isPalindrome()`** method can assume that the calling method converted the argument's letters into all-uppercase or all-lowercase. No other "pre-processing" is allowed. For instance, sometimes students' calling methods remove all punctuation before passing that sanitized **`String`** to their **`isPalindrome()`** method. That's not ok.

   Your **`isPalindrome()`**method itself must be recursive. For instance, some students sanitize the string in a non-recursive **`isPalindrome()`** and then pass the sanitized string to a recursive helper method. That's not ok. Instead, your recursive **`isPalindrome()`** method needs to handle punctuation as part of its scheme for recursing.

   The pset suggests a way to approach this problem, including what to treat as a base case and what to treat as the recursive case.

   Include demonstration code that reads a **`String`** from the user's keyboard, prints out whether or not the **`String`** is a palindrome as determined by **`isPalindrome()`**, and then exits.

b. Problem 6 (**LowestGrade.java**) must contain a non-recursive method named **removeLowest()** that accepts a variable number of **int** arguments and returns an **int** array that contains all of the values passed to the method except for the lowest score. As shown with the pset's sample cases, the returned array's values must be in the same relative order in which they were passed to **removeLowest()**.

   You must also create a method named **arrayPrint()** that accepts an **int** array as its only argument and returns [3] a **String** value that represents that array as shown in the problem.

   Note that **removeLowest()** sees the variable number of **int** arguments as an array. You are not allowed to sort or otherwise modify the array or a copy of the array to solve this problem. Also, your **arrayPrint()** method can't cheat by using built-in Java methods like **Arrays.toString()**.

   LowestGrade.java must contain a program that convincingly demonstrates your **removeLowest()** method.[4]

c. Problem 7 (**RecursiveSum.java**) must contain a recursive method named **sumTo()** that takes an **int** argument **n** and returns the sum of the first **n** reciprocals.

   E.g., **sumTo(2)** returns the value of the expression **1 + ½**, or **1.5**.

   The method should return **0.0** if passed the value **0** and should execute the statement **throw new IllegalArgumentException()** if passed a negative value.[5]

   sumTo() must be recursive; it cannot compute the answer using a loop.

   RecursiveSum.java must contain a program that convincingly demonstrates your **sumTo()** method.[4]

Supplementary Problems

Graduate students are required to solve one Supplementary problem, and can optionally solve a second Supplementary problem for extra credit. Undergraduate students are not required to solve any Supplementary problems, but can optionally solve one Supplementary problem for extra credit. Students do not have to specify which problem is extra-credit; we will automatically treat the highest-scoring problem(s) as regular credit.

The guidelines discussed in the beginning of the **Programming Problems** section also apply to this section.

a. Problem 8 (**RecursivePrint.java**) must contain a recursive method named **printNumber()** that accepts an **int** argument and prints [6] the value of the argument using standard English words. You can assume that the argument is less than one million.[7] E.g., **printNumber(123456)** will print "**one hundred twenty three thousand four hundred fifty six**".

   I suggest that you figure out how to process the argument in groups of three digits. For instance, with **n** = **123456** we need to translate **123** into "**one hundred twenty three**" for the "**thousands**", and we need to translate **456** into "**four hundred fifty six**" for the **units**.

   RecursivePrint.java must contain a program that convincingly demonstrates your **printNumber()** method.[4]

---

[3] You'll lose points if your method also prints the string. You'll lose more points if your method only prints the string.

[4] You must obey the demonstration guidelines at the beginning of the Programming Problems section. For methods like problem 8's **printNumber()** that print their result instead of returning it, you'll need to modify the "Here's how I would do that …" suggestion.

[5] We'll learn much more about exceptions in future lectures.

[6] You'll lose points if your method also returns the string. You'll lose more points if your method only returns the string.

[7] For 2 points of extra credit, write your method to work with numbers ≤ **Integer.MAX_VALUE**.

b. Problem 9 (**Permutations.java**) must contain a method named **listPermutations()** that accepts a **String** argument, and recursively prints [6] the complete set of permutations of that **String**. E.g., **listPermutations("ABCD")** would print the 24 (4-factorial) unique combinations of those letters.

   You can put all of the permutations on one line, or you can put each permutation on a separate line. You don't need to recreate the multi-row, multi-column output shown in the problem set.

   As with every recursive problem, you need to figure out what is the base case and what is the recursive case. Perhaps the base case is when the **String** argument's length is <2. Now consider the **String** argument **"ABC"**. You would print three sets of output. The first set starts with **"A"** prepended to all permutations of **"BC"**. The second set starts with **"B"** prepended to all permutations of **"AC"**. The third set starts with **"C"** prepended to all permutations of **"AB"**.

   This is a challenging problem. Normally we would require that **listPermutations()** itself be recursive, but for this problem it's okay for **listPermutations()** itself not to be recursive but instead to call a recursive helper method.

   Permutations.java must contain a program that convincingly demonstrates your **listPermutations()** method.[4]

c. Problem 10 (**Binomial.java**) must contain a recursive method, preferably named **C()**,[8] as described in the problem set, that accepts two **int** arguments **n** and **k**, and returns their binomial coefficient.

   Include a program that leverages your **C** method to print the first **x** rows of Pascal's Triangle, where **x** is read from the user's keyboard. For instance, if **x** is **3**, the program prints 3 rows of numbers:

   ```
   1
   1 1
   1 2 1
   ```

   Note that the 3rd row's numbers are **c(2,0)**, **c(2,1)** and **c(2,2)**.

d. Problem 11 (**AlaMode.java**) must contain a **static** non-recursive method named **mode()** that accepts one argument, an array of **int**s, and returns the most frequently occurring value in the array. Assume that the array has at least one element and that there are no "ties" for the most frequently occurring value.

   You are not allowed to sort or otherwise modify the array or a copy of the array to solve this problem.

   AlaMode.java must contain a program that convincingly demonstrates your **mode()** method.[4]

e. Problem 12 (**Anagram.java**) must contain a method, preferably named **isAnagram()**,[8] that accepts two **String** arguments and returns **true** or **false** according to whether they are case-insensitive [9] anagrams.

   Anagram.java must contain a program that convincingly demonstrates your **isAnagram()** method.[4]

---

[8] The problem doesn't specify the name of the method that you create, but my test harness assumes that the method has the suggested name so I would be grateful if you named your method as suggested.

[9] Case-insensitive anagrams contain the same letters without regard to case and the same number of each letter. For instance, **"Stop!"** and **"pots"** are case-insensitive anagrams.