Problem Set 1

I. Overview

Browse through chapters 1-5, 7 and 12 in the Reges & Stepp text. Refer as needed to Oracle's on-line documentation at https://docs.oracle.com/en/java/javase/20/docs/api/index.html. Java version 20 is the version currently running in cs50.dev

This assignment is designed to reinforce your understanding of elementary Java language constructs (such as *array* and *String* objects) and the *enumeration* type, while also introducing the important concept of *recursion*. One of the learning objectives is to ensure you understand the "flow of control" in a recursive computation and that you can also appreciate the relative efficiency of a recursive vs. an iterative solution; sometimes a recursive solution is easier to implement but may run more slowly than an iterative implementation.

Unless otherwise indicated, everything the *user would type* to the computer has been **underlined**. Everything the computer types is not.

Work carefully! The programming problems will be graded partly on the basis of *style* as well as *correctness*.

II. Pencil-and-Paper Exercises (25 points total)

Submit your answer to each problem in the named file. The file must be a plain text file (not a PDF file, a Word file, an RTF file, or any other non-text-file format.

[1] 3 points — use file *Mystery.txt*

Consider the following method:

```
public static void mystery (int [] a, int [] b)
{
   for (int i = 0; i < a.length; i++) {
      a [i] += b [b.length - 1 -i];
   }
}
```

What are the values of the elements in array **a1** after the following code is executed?

```
int [] a1 = {1, 3, 5, 7, 9};
int [] a2 = {1, 4, 9, 16, 25};
mystery (a1, a2);
```

[2] 7 points — use file *RecTriangle.java*

Consider the following method named **printTriangle** that *recursively* outputs a triangular pattern:

```
public static void printTriangle (int s)
{
       if (s < 1) return;
       printTriangle (s-1);
       for (int i = 0; i < s; i++)
       {
           System.out.print( "[]");
       }
       System.out.println ();
}
```

For example, **printTriangle (4)** produces the output

```
[]
[][]
[][][]
[][][][]
```

Make minor modifications to **printTriangle** so that the lines of output are *reversed* from what the original method produces. For example, **printTriangle (4)** should still be *recursive*, but instead output the following:

```
[][][][]
[][][]
[][]
[]
```

[3] 6 points — use file *Enumeration.java*

The following *Java* program is supposed to print out how many days are in each of the 12 months for a given year. The one special case is February, which has 29 days in "leap years" and 28 days during other years.[1]

```
import java.util.*;

enum Months { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG,
                SEP, OCT, NOV, DEC} ;

class Enumeration
{
   public static int daysInMonth (Months m, int year)
   {
       switch (m)
       {
             // YOU NEED TO WRITE THIS PART
       }
   }

   public static void main (String [] args)
   {
        for (Months m : Months.values() )
        {
           System.out.println (m + " 2023 has " +
                        daysInMonth(m, 2023) + " days!");
        }
    }
}
```

Add the missing code to the **switch** statement so that this program outputs

```
JAN 2023 has 31 days!
FEB 2023 has 28 days!
MAR 2023 has 31 days!
APR 2023 has 30 days!
 …
 …
```

[1] A *leap year* is any year that can be **evenly divided by 4** (such as 2016, 2020, etc), except if it can be **evenly divided by 100**, then it isn't (such as 2100, 2200, etc) ***except if*** it can be **evenly divided by 400**, then it ***is*** (such as 2000, 2400)

[4] 9 points use file *Power.java*

Below is a Java method that was discussed during lecture. The method *recursively* computes x^n .

```
public static double power (double x, int n)
{
    if (n == 0) return 1.0;
    else if (n > 0) return x * power(x, n-1);
    else return 1.0 / power(x, -n);
}
```

The **power** method can be made more efficient by taking advantage of the mathematical fact that $x^n = (x^{n/2})^2$ when the exponent n is an even number. For example, x^{10} is really $(x^5)^2$.

i. *(5 points)* Modify the **power** method to implement the above idea of more efficiently computing x^n when **n** is even. Your solution should save the result of raising x to the n/2 power when n is even, and then return that result multiplied by itself. (Don't user **power** in order to square the result.)

In other words, compute $(x^{n/2})$ only once.

ii. *(4 points)* How many total calls (including the initial one) on the modified **power** in *part i* above will occur when computing **`power (foobar, 1024)`** ?

III. Programming Problems (75 points)

[5] 25 points — use file *Palindrome.java*

A fun problem that can be solved recursively involves the technique of determining whether or not a sentence is a *palindrome* — a **String** that is equal to itself when you reverse its characters.

Some examples of palindromes:

- *"My gym tasks are too lonely?" a Jay Leno looter asks at my gym.*
- *Cigar? Toss it in a can, it is so tragic!*
- *"Ed, I saw Harpo Marx ram Oprah W. aside."*
- *12321*

Obviously, the above sentences are all considered palindromes because only alphanumeric characters are compared (i.e., punctuation characters and spaces are ignored), and the uppercase vs. lowercase differences are also ignored (in other words, 'm' and 'M' are considered to be equal).

An iterative solution for determining whether a sentence is a palindrome was written in CSCI E-10a last semester. [2] Now we want you to write a *recursive* solution. Specifically, define a method

```
public static boolean isPalindrome (String s)
```

that returns **true** or **false**, depending on whether the actual argument is, in fact, a palindrome.

Here's a big hint: the *base case* is going to be if the length of the argument **s** is ≤ 1. The *recursive case* is going to involve comparing the first

[2] See the files *Palindrome.java* and *PalindromeImproved.java* in the Unit 5 Lecture Files section of the course website's Bulletin Board page

and last characters in **s** — and if they are equal, returning the result of determining whether a substring of **s** is a palindrome. (That substring contains all characters of **s** except for the first and last characters that were just compared.)

In order to get the recursion working correctly, you may wish to begin by only considering sentences that contain no punctuation or space characters. In other words, start by taking into account only simple palindromes such as **madam** or **1x1**. Once you have that working, then you should make your method work for the more complex cases that contain punctuation characters and spaces. (You might want to utilize the **isLetterOrDigit()** method that is part of the **Character** class.)

Write a main method that allows the user to input a single **String** value on the keyboard, then prints out whether or not the sentence is a palindrome by calling on `isPalindrome`, then exits. Prior to calling method `isPalindrome` from `main()` you *may* convert the entire input *String* into all uppercase or all lowercase; no other "pre-processing" is allowed.

[6] 30 points use file *LowestGrade.java*

It is common in some courses for an instructor to not count the lowest exam score or homework score for each student.

Define a *static* method named **removeLowest** that accepts a *variable number of integer arguments* that represent the homework scores of a single student. Your method should *return an integer array* that contains all of the values passed to the method EXCEPT for the lowest score. To illustrate how this method ought to work, consider the following main method: [3]

```
public static void main (String [] args)
{
   int [] a = removeLowest ( 23, 90, 47, 55, 88);
```

[3] This code only shows the correct **`removeLowest()`** operation. Your **`main()`** program's demonstration logic must behave as described in the Programming Problems section of the Pset's "Notes" PDF on the course website's Bulletin Board page.

```
        int [] b = removeLowest ( 85 );
        int [] c = removeLowest ();
        int [] d = removeLowest (59, 92, 93, 47, 88, 47);

        System.out.println ("a = " + arrayPrint(a));
        System.out.println ("b = " + arrayPrint(b));
        System.out.println ("c = " + arrayPrint(c));
        System.out.println ("d = " + arrayPrint(d));
    }
```

The correct output for this code would be

```
a = [90, 47, 55, 88]
b = [85]
c = []
d = [59, 92, 93, 88, 47]
```

In the case of **a**, the lowest score, **23**, was removed. In the case of **b**, nothing was removed because it would be cruel to eliminate the ONLY grade that a student had obtained (this is a special case). The third case, **c**, is also special: when no values are passed. And the fourth case, **d**, shows what happens when the lowest score appears twice: it is removed only once from the array that is constructed.

In class we demonstrated how the method **`Arrays.toString()`** could be used to easily print an entire single-dimensioned array. For example,

```
int [] a = {3, 4, 9, -2, 5};
System.out.println (Arrays.toString (a) );
```

would output **`[3, 4, 9, -2, 5]`**

In this problem you will write your own *static* method named **`arrayPrint`**, so that **`System.out.println (arrayPrint (a) );`**
would also output **`[3, 4, 9, -2, 5]`**

Note that the **arrayPrint** method returns a **String** value, and should accept an array of integer values as its only argument. If the argument passed contains no values (i.e., the array has a length of 0), then your method should return "[]". Your **`arrayPrint`** method may NOT use the **Arrays.toString** method!

Your solution to this problem does NOT involve recursion!

[7] 20 points — use file *RecursiveSum.java*

Write a recursive method **`sumTo`** that takes an integer parameter **`n`** and returns the sum of the first **`n`** *reciprocals*. In other words, **`sumTo(n)`** returns the value of the expression

```
(1 + 1/2 + 1/3 + 1/4 + ... + 1/n)
```

For example, **sumTo (2)** should return the value **1.5**. The method should return **0.0** if passed the value **0** and should *throw* an **IllegalArgumentException** if passed a value less than 0.

You may NOT use a *while* loop, *for* loop or *do-while* loop to solve this problem; you MUST use ***recursion***. Write a main method that convincingly demonstrates your method in action.

IV. Supplementary Problems (10-20 points)

Graduate-credit students must answer ONE of the following problems, and may answer at most one additional problem for up to 10 points of "extra credit." Undergraduate-credit students may answer at most one of the following for up to 10 points of "extra credit."

[8] 10 points — use file *RecursivePrint.java*

Define a *recursive* method named **`printNumber (int n)`** that takes a single integer argument, **`n`**, and prints the value of **`n`** using standard English words. You can assume **`n`** is less than **one million**.

For example, **`printNumber (143)`** should produce the output

```
one hundred forty three
```

whereas **`printNumber (-24549)`** should produce the output

```
minus twenty four thousand five hundred forty nine
```

For 2 points of additional credit, write your method to work with numbers up to **`Integer.MAX_VALUE`**. Demonstrate your **printNumber** method using a main program of your own design.

[9] 10 points use file *Permutations.java*

Write a *recursive* method named **listPermutations** that accepts a *String* as its only parameter and prints the complete set of permutations from that *String*. For example, given the string "ABCD", there are 24 (4-factorial) different permutations, as follows:

```
ABCD    BACD    CABD    DABC
ABDC    BADC    CADB    DACB
ACBD    BCAD    CBAD    DBAC
ACDB    BCDA    CBDA    DBCA
ADBC    BDAC    CDAB    DCAB
ADCB    BDCA    CDBA    DCBA
```

To display these permutations, you would simply call
`listPermutations("ABCD");`

Here's an important observation to help you out with the recursive strategy. Starting with the string "ABCD", for example, there are four possible ways to choose the first letter. Therefore, we can divide the permutations into the following four distinct groups

- the set of permutations starting with **A**
- the set of permutations starting with **B**
- the set of permutations starting with **C**
- the set of permutations starting with **D**

If you think a bit about these 4 individual subproblems, you will make an important discovery, namely: the set of all permutations starting with the letter **A** consists of the letter **A** added to the front of all permutations of the remaining letters "BCD". Similarly the set of all permutations starting with **B** can be generated by adding the prefix **B** to the front of all permutations of other remaining letters "ACD", and so on. Of course, you will need a main method to convincingly demonstrate our your **listPermutations** method.

[10] 10 points use file *Binomial.java*

The numbers **C(n, k)** are defined for all **n** ≥ **0**, **k** ≥ **0** by the following three rules:

- C(n, 0) = 1
- C(n, k) = 0, if k > n
- C(n, k) = C (n-1, k) + C(n-1, k-1), for 0 < k ≤ n

Write a recursive method that computes **C(n, k)**.

These numbers are called the *binomial coefficients*, and appear in a number of areas. For example, they count the number of arrangements in a row that one can make from **n** objects, **k** of which are red, and **n-k** of which are green.

They also are the coefficients of $x^n y^{n-k}$ in the expansion of $(x+y)^n$. For instance, $(x + y)^3$ may be written $x^3 + 3x^2y + 3xy^2 + y^3$. Note that 1, 3, 3, 1 are **C(3,0)**, **C(3,1)**, **C(3,2)**, and **C(3,3)**. If we write the binomial coefficients in a table, with **k** increasing from left to right, and **n** increasing as we go down the table, we produce what is known as *Pascal's Triangle*. Below are the first four rows. Notice that it mirrors the definition, since each term is the sum of the one above it and the one above and to the left.

```
1
1     1
1     2     1
1     3     3     1
```

Write a complete *Java* program that prints the first **n** rows of *Pascal's Triangle*, where **n** is input by the user from the keyboard. Use your recursive method for computing **C(n, k)** in writing the program.

[11] 10 points use file *AlaMode.java*

Write a *static* ***NON****-recursive* method named **mode** that accepts one parameter, an array of integers, and returns *the most frequently occurring element in that array of integers.* Assume that the array has at least one element and that there will be no "ties" for the most frequently occurring value.

You CANNOT sort or otherwise modify the original array.

Be sure you convincingly demonstrate your method by writing a main method that calls on **mode** with a variety of different arrays.

[12] 10 points use file *Anagram.java*

Two character strings are *anagrams* if they contain the same letters and the same number of each letter. For example, "stop" is an anagram of "pots", and "gin and vermouth" is an anagram of "hung over, damn it".

Define a method that takes two *String* objects as arguments, and returns a *boolean* that indicates whether they are anagrams of each other. Write a main method that convincingly demonstrates your method.