

Unit 5 Problem Set

part I

I. Overview

Electronic submission of this problem set is due prior to midnight on Monday, February 12.

Browse through chapters 1-5, 7 and 12 in the Reges text, and refer as needed to Oracle's on-line documentation at <http://docs.oracle.com/javase/7/docs/api/>.

This assignment is designed to reinforce your understanding of elementary Java language constructs (such as *array* and *String* objects), the *enumeration* types, 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 that are done on the computer will be graded partly on the basis of *style* as well as *correctness*.

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

[1] 3 points

use file *Mystery.java*

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 has been 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 instead output the following:

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

Your solution must still be *recursive*.

[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 + " 2017 has " +
                                daysInMonth(m, 2017) + " days!");
        }
    }
}
```

Write the missing code from the **switch** statement so that this program outputs

```
JAN 2017 has 31 days!
FEB 2017 has 28 days!
MAR 2017 has 31 days!
APR 2017 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. (No need to call on **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 all the 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."*
- ◆ *Marge let a moody baby doom a telegram.*

Obviously, the above sentences are all considered palindromes because only alphabetic 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. 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 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**. Once you have that working, then you should make your method work for the more complex cases that containing punctuation characters and spaces. (You might want to utilize the **isLetter()** 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, and then prints out whether or not the sentence is a palindrome by calling on **isPalindrome**. 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:

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

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

Note that 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 **[]** . Therefore, your solution 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
 $(1 + 1/2 + 1/3 + 1/4 + \dots + 1/n)$

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 larger than one million. Test your method out using a main program of your own design.

[9] 10 points

use file *Index.java*

Write a *recursive* method named **indexOf** that accepts two *Strings* as parameters and returns the starting index of the first occurrence of the second *String* inside the first *String* (or -1 if not found). For example, the call of `indexOf ("Barack Obama", "bam")` would return 8.

Note that *Strings* already have an **indexOf** method, but you may not use it in your solution!

[10] 10 points

use file *Binomial.java*

The numbers **C(n, k)** are defined for all **n, 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 $n \geq k > 0$

Write a recursive method that computes **C(n, k)**, where **n** and **k** are formal parameters.

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 $x^3 + 3x^2y + 3xy^2 + y^3$, and 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 *Java'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 returns the most frequently occurring element in an array of integers that is passed as the only argument. Assume that the array has at least one element and that there will be no “ties” for the most frequently occurring value.

Be sure you thoroughly test your method by writing a convincing 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.