Unit 5 Problem Set

part 1

1. Overview

Browse through chapters 1-5, 7 and 12 in the Reges text, and refer as needed to Sun Microsystem's on-line documentation at `http://download.oracle.com/javase/6/docs/api/` .

This assignment is designed to reinforce your understanding of elementary Java language constructs (such as array and String objects) 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*.

Electronic submission of this problem set is due prior to the start of lecture on Monday, February 11.

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

[1] 3 points

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

The following *recursive* method is supposed to add up the digits of the argument **n**. For example, `digitSum(3456)` returns 3+4+5+6 = **18**. The code has two omissions, indicated by ??? . Replace each ??? with the correct Java code. You may assume the argument **n** is non-negative.

```
public static int digitSum (int n)
{
     if ( n < 10) { return ???  ; }   // base case
      else return  ???   ;
}
```

## [3] 6 points

The following *Java* program is supposed to print out how many days are in each of the 12 months (we will simplify a bit and assume February always has 28 days):

```
import java.util.*;

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

class Prob3
{
   public static int daysInMonth (Months m)
   {
       switch (m)
       {
             // YOU NEED TO WRITE THIS PART
       }
   }

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

What Java code is missing from the **switch** statement so that this program produces the output

```
JAN has 31 days!
FEB has 28 days!
MAR has 31 days!
APR has 30 days!
MAY has 31 days!
JUN has 30 days!
 ...
 ...
```

## [4] 9 points

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.

ii. *(2 points)* How many recursive calls will occur when computing **power(foobar, 1023)** ?

iii. *(2 points)* How many recursive calls 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-50a 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.`

[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 "[]" .

Your solution to this problem does NOT involve recursion!

[7] 20 points

use *file Rabbits.java*

There is an interesting sequence of numbers known as the *Fibonacci Sequence*, named after Leonardo Fibonacci (born in Pisa around 1170 and depicted in the above portrait). The first two terms of the sequence are given as 0 and 1. After that, the terms are constructed according to the rule that each number in the list is the sum of the preceding two numbers. The first few terms are shown below:

```
0,  1,  1,  2,  3,   5,  8,  13,  21,  34,  55, 89,  ...
```

This sequence was introduced in 1202 A.D. by the Italian mathematician, Fibonacci, to provide a model of population growth in rabbits! His main assumptions were

- it takes rabbits one month from birth to reach maturity;
- one month after reaching maturity, and every month thereafter, each pair of mature rabbits will produce another *pair* of rabbits; and
- rabbits never die!

One senses that this model may not be completely realistic. But the essence of mathematical modeling is to start with a crude model that emphasizes the important aspects of the situation and suppresses less important information. A more refined model can be developed later on, profiting from the experience with the crude model. Thus we might eventually improve the Fibonacci model by obtaining more accurate figures on the birth rate, taking mortality into account, considering the limitations of food supply, the effects of predators, disease, overcrowding, and the like.

Despite its frivolous origins, the Fibonacci sequence has some fascinating properties, and does play an important role in the solution of a number of seemingly unrelated mathematical problems. There is even a quarterly journal entirely devoted to the properties and applications of the Fibonacci sequence: `http://www.fq.math.ca/`

After this long digression, let us see how the rabbit-pair population model gives rise to the Fibonacci sequence. Fibonacci starts with one pair of newborn rabbits at the beginning of month #1, and then he lets nature take its course. This is shown below:

Beginning of Month #	1	2	3	4	5	6	7	8
Infant Rabbit Pairs	1	0	1	1	2	3	5	8
Mature Rabbit Pairs	0	1	1	2	3	5	8	13
Total # of Rabbit Pairs	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**21**

You are to define a *Java* method named **`fibo`**, which takes one argument (a month number), and which returns the number of mature rabbit pairs at the beginning of the month. Essentially, you are just writing a function to compute the *n'th* Fibonacci number. This **`fibo`** method can be a static member of a class named **`Rabbits`**.

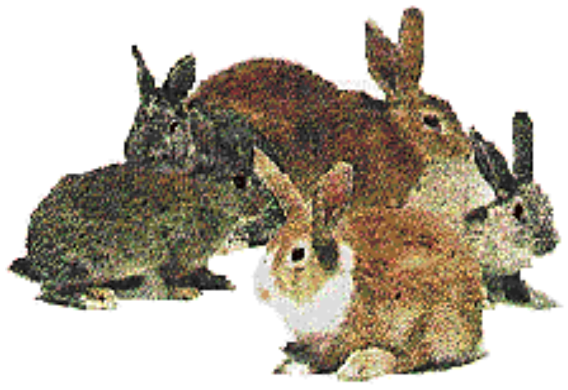
You will also need to write a main method in the **`Rabbits`** class that uses a **`for`** statement to print out the number of mature rabbits at the beginning of month #1 through month #12 (or some other limit). The output from your main program should be something like

```
At the end of month #1, there are 0 mature rabbit pairs.
At the end of month #2, there are 1 mature rabbit pairs.
At the end of month #3, there are 1 mature rabbit pairs.
At the end of month #4, there are 2 mature rabbit pairs.
 ...
 ...
```

Your method **fibo** must be *recursive*. Here is a big, big hint: The n'th Fibonacci number is the sum of the *n-1st* Fibonacci number added to the *n-2nd* Fibonacci number! (We won't tell you how to make the recursion stop — but it should be fairly obvious.)

A fascinating website that describes applications of the Fibonacci sequence is located at

http://library.thinkquest.org/27890/theSeries1.html



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 *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
```

while **`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 *file GCD.java*

Write a *recursive* **method** to compute the Greatest Common Divisor (GCD) of two numbers. Write a main program that demonstrates convincingly that your GCD method works correctly.

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

You may solve this problem only if you did NOT solve this as a supplementary problem in the Unit 4 problem set last semester in CSCI E-50a.

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 ≥ 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 *file RecursiveSum.java*

Write a *static* method that accepts an array of integers and the size of the array as its only two arguments, and returns the sum of all the values. The sum must be computed *recursively*.

Remember to think about the base case: when have all the values in the array been summed up? Then think about how to go from a larger size toward that base case — this is your recursive case. Do not construct a new array when your method calls on itself. Be sure to provide a main program that demonstrates your recursive method works correctly.

## [12] 10 points *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.