HOMEWORK PROBLEMS 14

1) What is wrong with the following recursive method:

```
public static void rp1(int x)
{
   System.out.println(x);
   rp1(x - 1);
}
```

2) What is displayed when the following method is passed 5:

```
public static void rp2(int x)
{
   if (x == -1)
   {
      System.out.println("A");
      rp2(6);
   }
   else
   if (x == 3)
      System.out.println("E");
   else
   {
      System.out.println("B");
      rp2(x - 3);
      System.out.println("C");
   }
   System.out.println("D");
}
```

3) What does the following method do?

```
public static int rp3(int x, int y)
{
   if (y == 0)
      return x;
   return rp3(x, y-1) + 1;
}
```

4) What does the following method do?

```
public static void rp4(int x)
{
   if (x == 0)
      return 0;
   return x + rp4(x-1);
}
```

5) When should you use recursion rather than a loop?

6) Write a program equivalent to `C14h6.java` (a copy of `Recursion3` in Fig. 14.5), but implement `countDown` using a `while` loop instead of recursion.

7) The following method returns the i^{th} number in the **Fibonacci sequence**: 1, 1, 2, 3, 5, 8, 13, ... Each number after the first two is the sum of the two preceding numbers. `fib(1)` returns 1; `fib(2)` returns 1; `fib(3)` returns 2, `fib(4)` returns 3, `fib(5)` returns 5, `fib(6)` returns 8, and so on.

```
public static long fib(long i)
{
   if (i < 1)
      return 0;
   else                                        // else here necessary?
   if (i <= 2)
      return 1;
   else                                        // else here necessary?
      return fib(i - 1) + fib(i - 2);          // double recursion
}
```

Each time you call this method and pass it a value greater than 2, it calls itself twice because it has two recursive calls (`fib(i - 1)` and `fib(i - 2)`). As a result, the total number of calls can become excessive, making this method extraordinarily inefficient. Write a program that includes this method. Your program should prompt for and read in an integer. It should then call `fib` passing it the integer read it. For what integer passed is the program's run time 3 seconds? Double the integer passed and re-run your program. Does the run time also double? Again double the integer passed and re-run. What is relation between the integer passed and run time? Can the `else` reserved words be omitted without affect the computation performed?

8) Implement the `fib` method in homework problem 7 using a loop instead of recursion. Perform the same type of timing measurements that you performed in homework problem 7. Is the loop implementation more efficient? *Rule*: If double recursion (i.e., two recursive calls performed for each call) is not intrinsically a necessary part of the computation, *don't use recursion*!!!

9) Rewrite the `reverse` method in `C14h9.java`. Reverse a string by placing its last character at the beginning of the reverse of the substring that contains the entire string except for its last character. Test your program.

10) Rewrite the `reverse` method in `C14h10.java` (Fig. 14.6). Use the `reverse` method in the `StringBuffer` class.

11) Write a method that does multiplication on non-negative integers using recursion. Test your method by using it to compute the product of 0 and 0, the product of 0 and 5, the product of 5 and 0, and the product of 5 and 10. *Hint*:

```
a×b = a×(b - 1) + a     for b > 0
a×b = 0                 for b == 0
```

12) Write a recursive method that returns the number of occurrences of `'A'` in the string it is passed. Test your method with the following strings: "", `"A"`, `"B"`, `"BCA"`, `"ABC"`, and `"ABACAD"`.

13) Write a recursive method that returns the maximum integer in the array it is passed. Test your method by passing it a 20-slot `int` array initialized with random integers. Display the maximum. Generate integers with `nextInt()` in the `Random` class. Use the seed 7 when you construct the `Random` object.

14) A recursive method does not have to first recurse "down" and then "up". It can first recurse "up" and then "down". Write a recursive method returns the sums the integers from 1 to `n` when passed `n`. Your method should first recurse "up", hit the "top" when the parameter is equal to the largest integer to be summed, and then recurse "down." *Hint*: Your method should have two parameters.

15) Write a method that implements **Ackermann's function**:

```
Ack(x, y) = y + 1                 when x == 0
Ack(x, y) = Ack(x-1,1)            when x > 0 and y == 0
Ack(x, y) = A(x-1, A(x,y-1))      when x > 0 and y > 0
```

Use type `long`. Write a program that displays the value of `Ack(n, n)` for `n` = 0, 1, 2, … For what value of `n` is the value of `Ack(n, n)` too large for type `long`? Ackermann's function is famous because `Ack(n, n)` grows so rapidly with increasing `n`.

16) Write a recursive method that determines if the first string it is passed is a *prefix* (i.e., starts with) of the second string it is passed. Test your method with the following pairs: `""` and `""`, `""` and `"A"`, `"AB"` and `"ABC"`, `"C"` and `"ABC"`, `"BC"` and "`ABCD`", `ABC"` and `"BCD"`, `"ABCD"` and `"ABCE"`.

17) Same as homework problem 16 but determine if the first string is a substring of the second string (i.e., appears anywhere in the second string).

18) Write two methods, one recursive, one using a loop, that compute `n!` *Hint*: Here is a recursive definition of `n!`:

```
n! = n×(n - 1)!    for n > 0
n! = 1             for n == 0
```

19) Write a program that reads in a file and outputs it to another file. The output file created should be identical to the input file but with its lines in reverse order. Use recursion. Obtain the names of the input and output files from the command line. For the input file, use the file `t1.txt`.

20) Write a program that prompts for and reads in a string. Your program should call a recursive method `stringLength`, passing it the string. `stringLength` should return the length of the string. Implement `stringLength` using recursion. Do *not* use the `length` method in the string object. Test your program with the strings `""` (the null string), `"A"`, and `"ABCDEF"`.

21) Write a program in which `main` prompts for and reads in a non-negative `int` constant into an `int` variable. `main` should then call a recursive method `reverseInt`, passing it the `int` value read in. `reverseInt` should return to `main` the `int` value obtained by reversing the digits of the number it is passed. For example, if `reverseInt` is passed 123, it should return (not display) the value 321. `main` should then display the value returned. Test your program with 0, 1, 12, and 123. *Hint*: Divide by 10 to strip off the rightmost digit. To isolate the rightmost digit, divide by 10 and take the remainder (use the `%` operator). `reverseInt` should *not* convert the `int` value it is passed to a string.

22) Write a method that is passed a string consisting of digits. Your method should return the sum of the digits. For example, if the string is `"135"`, your method should return the `int` value 9. If the null string is passed to your method, your method should return 0. Use recursion. Test your method by passing it `""`, `"0"`, `"00"`, and `"03405"`. The parameter type of your method should be `String`—not `int`.

23) Write a program in which `main` prompts for and reads in non-negative integers into an array. The user signals the end of the data by entering a negative number (which should not be entered into the array). `main` should then display the numbers in the array and call a `sort` method, passing it the array. Your `sort` method should sort the array *using a recursive implementation of the selection sort* (see Section 9.9). When your `sort` method returns to `main`, `main` should again display the numbers (now sorted) in the array.

24) Write a method that is passed non-negative integer. Your method should display the number of 1 bits in its binary representation. Use recursion. Test your method with 0, 4, 6, 7, 1000, and 1023.

25) Do homework problem 34 from Chapter 9 but implement the binary search method recursively.

26) Write a program that uses recursion to display all the solutions to the **eight queens problem**. A solution is a board configuration with eight queens only, each not attacking any other queen (i.e., each queen should not be in the same row, same column or same diagonal as another queen). Here is the basic structure of a recursive method that solve this problem:

```
public static void queens (int[] board, int row)
{
   if (row == 9)
      display board
   else
      for (int col = 1, column <= 8; col++)
      {
         Attempt queen placement on square (row, col)
         If successful
             queens(board, row + 1);
      }
}
```

`main` should call this method passing it an empty board (an 8 by 8 array) and 1 (the initial row number). *Hint*: a queen on square (r_1, c_1) and a queen on square (r_2, c_2) are attacking each other if any of the following conditions are true: $r_1 = r_2$ (same row), $c_1 = c_2$ (same column), $r_1+c_1 = r_2+c_2$ (same positive slope diagonal) or $r_1-c_1 = r_2-c_2$ (same negative slope diagonal). Because of the complexity of this problem, the method uses a loop in addition to recursion. Can it be done without using any loops?

27) Write a recursive method that performs a merge sort on the array of integers it is passed. A **merge sort** divides the numbers to be sorted into two halves, sorts each half, and then merges them. Test your method by sorting 20 random integers. Each time the number of numbers to be sorted is doubled, how it the sort time affected? The merge sort is an example of a **divide and conquer** algorithm. Divide and conquer algorithms break up a problem into smaller and easier sub-problems. These sub-problems are, in turn, broken up into even smaller and easier sub-problems. This process continues until we are left with sub-problems all of which are very small and trivially easy to solve. In a merge sort, the problem of sorting *n* items is broken up into two sorts, each of *n*/2 items. Each of these sorts of *n*/2 items is, in turn, broken up into two sorts, each of *n*/4 items. This process continues until we are left with sorts all of which are of one item only—a trivial problem, indeed (we don't have to do anything to sort one item).

28) Write a method that is passed a string which display all permutations of that string. Test your method with `""`, `"A"`, and `"ABC"`. *Hint*: `"ABC"` has six permutations. A string of length *n* has *n* factorial permutations.

29) Write a method that is passed a string which determines if the string is a palindrome (i.e., same spelling forwards and backwards). Test your method with `""`, `"A"`, `"ABBA"`, "`ABA`", `"ABAB".`

30) What does the following method return if it is passed 0, 3, 4, 32, 63, 64, 4095, or 4096? What mathematical function does it compute?

```
public static int f(int n)
{
    if (n <= 1)
        return 0;
    else
        return 1 + f(n/2);
}
```

31) Write a method that is passed a non-negative `int` value. Your method should display the binary representation of the `int` value. Test your method with 0, 1, 7, 63, and 4096. Use recursion. *Hint*: Do a recursive call in which you pass the `int` value with its rightmost bit chopped off (use the `/` operator). Then display the rightmost bit (use the `%` operator).

32) Same as homework problem 31, but display the `int` value in hex (base 16).

33) Same as homework problem 31, but display the `int` value in octal (base 8).

34) Same as homework problem 31, but specify the target number base on the command line when you invoke the program. How do the number of symbols used for each number compare for the number bases 2, 8, 10, and 16?

35) Redo homework problem 31, but do not use recursion.

36) What is displayed when the program in `C14h36.java` is executed?

For additional material on recursion (specifically, how to eliminate it), see the file `eliminateRecursion.pdf`.