CSC 2111 **Spring 2013**

Name(s): ________________________________

Lab 10: More Recursion and Algorithm Efficiency

Reference: *C++ Plus Data Structures*
Chapter 7.14 - "Deciding Whether to Use a Recursive Solution"
Chapter 2.6 –"Comparison of Algorithms"

Objective: To gain a better understanding of how recursion works and the efficiencies (or inefficiencies) of a recursive algorithm.

Files: *Timer.h, Hanoi.cpp and Fibonacci.cpp*

Introduction

Efficiency is an important property of algorithms and, by extension, an important property of programs. As described in Section 2.6 of the text *C++ Plus Data Structures*, there are a number of issues involved in the computation of efficiency. Efficiency takes many forms including:

- space efficiency
- time efficiency
- resource efficiency, such as use of disk drives and processors

Time efficiency—that is, speed of execution—is a common one. One function is more efficient than another in solving a given problem if it solves the problem in less time than the other function. In this lab exercise we will actually time some algorithms to gain information about their efficiency.

Measuring Execution Time

To measure execution time, you will make use of a Timer class to time sections of code. It acts like a stopwatch for system time. Take a look at the supplied class in *Timer.h* and check out the operations that are provided.

To use the Timer class in your program, you must include its header file:

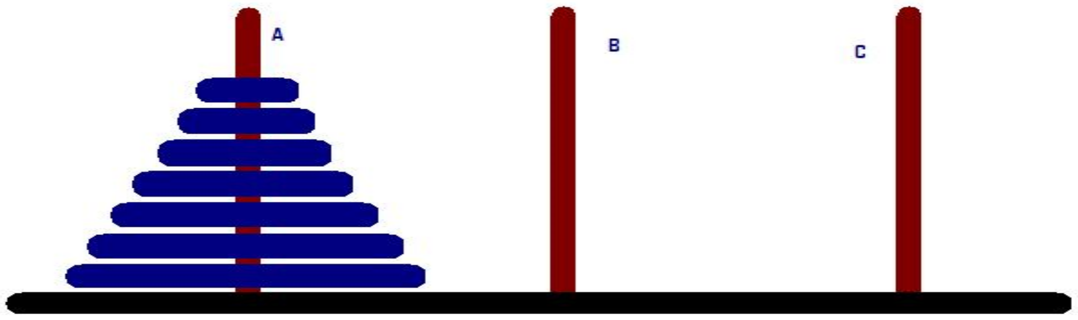
```
#include "Timer.h"
```

You can then declare and use a Timer object as follows:

```
Timer t;  // Declare a stopwatch object.

t.start()

// do some processing -- normally, call a function

t.stop()

cout << "The number of seconds processing was " << t << endl;
```

Example: The Towers of Hanoi Problem

The Towers of Hanoi is a classic problem in Computer Science. (It is described in detail at http://en.wikipedia.org/wiki/Tower_of_Hanoi.) It illustrates dramatically the fact that for some kinds of problems the solution using a recursive algorithm is very easy to find, but a non-recursive one is considerably more difficult.



The problem involves disks on three pegs, and to solve it you must move the disks from one peg to another according to the following rules:

1. *When moving a disk, you must put it on a peg— you may not simply set it to the side to use later.*
2. *You may move only one disk at a time, and it must be the top disk on one of the pegs.*
3. *You may never put a larger disk on top of a smaller one.*

Consider how to solve this problem recursively. Remember that, as described in your previous lab, there are two parts to a recursive solution: the *base* case and the *general* or *inductive case.*

The Base Case

We begin by deciding what the base case is. It is usually a simple case, and here the simplest case is when there is only one disk left on the source peg. Then the solution is trivial. You move the single remaining disk from the source peg to the destination peg, and the problem is solved.

The Inductive Case

For more than one disk, we have to use the inductive case, which should look something like this:

1. Move the topmost n — 1 disks from peg A to peg B, using peg C for temporary storage.
2. Move the final disk remaining on peg A to peg C (the base case).
3. Move the n — 1 disks from peg B to peg C, using peg A for temporary storage.

As you can see by looking at the function `move()` in the program *hanoi.cpp*, it is easy to implement this algorithm in code.

Compile and execute *hanoi.cpp*. Does it compile without errors and warnings? Y/N

How many moves does it take to solve the problem with 3 disks? __________

How many moves does it take to solve the problem with 5 disks? __________

Now modify *hanoi.cpp* as follows:

1. Comment out the output statement in function `move()`.

2. Use the ideas described earlier to add statements in `main()` to time the call to function `move()`.

3. Compile and execute the program repeatedly until you find a value for the number of disks for which the execution time is <u>at least 1/2 second</u>. Use it as the first entry in the following table and then find the execution times for the next 5 values. **Record the results in the following table:**

Number of disks	Execution time

What pattern do you see between the computing time for *n* disks and that for *n* + 1 disks?

You should see behavior indicating that the time required to solve this problem grows exponentially as the number of disks increases.

Modify *hanoi.cpp* to display the number of moves by adding an output statement after the call to `move()` in `main()` to display the value of `moveNumber`, which will be the number of the last move.

Compile this modified program. **Execute it several times to find the following**:

Number of disks	1	2	3	4	5	6	7	8	9	10
Number of moves										

In general, for *n* disks, how many moves are required? _____________

We say that this algorithm for solving the Tower of Hanoi problem is $O(2^n)$, or "of order 2 to the nth" or it is of "exponential order." *Algorithms of exponential order clearly do not scale well to large inputs!*