**Programming Fundamentals III**

**COSC 2436**

**Lab 1: Recursion**

# Lab 15.1 Designing a Recursive Algorithm for a Power Function

One way to solve problems in computer programs is to use looping, or iteration. You can also use recursion, which, like looping, is another method of repeating code. A recursive function is a function that calls itself. A recursive definition is one in which something is defined as a smaller version of itself. Recursion, like a loop, stops when a certain condition is met. The condition that causes the recursive function to stop is called the base case of the recursive function.

In recursion, a set of statements is repeated by having a function call itself. Moreover, a selection control structure is used to control the repeated calls in recursion.

Keep the following guidelines in mind as you work with recursive statements:

* Every recursive definition must have one or more base cases.
* The general case must eventually be reduced to a base case.
* The base case stops the recursion.

A function is called directly recursive if it calls itself. A function that calls another function and eventually results in the original function call is said to be indirectly recursive.

A recursive function in which the last statement executed is the recursive call is called a tail recursive function.

A program design for a recursive function is similar to a program design for a loop. You might find it easier to test your program design using a loop. Then, after you have determined that your design works, use the same design to implement the C++ code as a recursive function.

## Objectives

In this task, you design a program that uses a recursive function call to create a power function that raises a base value to a positive exponent.

## After completing this lab, you will be able to:

* Design an algorithm that uses a recursive call as part of a value-returning expression to a recursive call.

# Lab 15.1 Steps: Designing a Recursive Algorithm for a Power Function

In the following exercises, you answer questions to help design a recursive algorithm. Then you design and write a program that to raise a base value to a positive exponent.

1. How will the values for your base and exponent be input—by assignment or by interactive input?

interactive

1. The main program might restrict valid input. However, your function should accommodate all integer values. How will you accommodate negative exponents?

No changes should be needed

1. How will you accommodate zero exponents?

If statement

1. How will you accumulate the values returned from the recursive function?
2. How will you display the result from the function calls?

6a. *Critical Thinking Exercise*: Design a program that is a power function, which is used to raise a base value to a positive exponent. This power function should be a recursive function. You have already used the power function from the math header file. Design your own recursive function called power that takes integer arguments for base and exponent and returns an integer value. The exponent value must be greater than or equal to zero. The algorithm finds n raised to the power of x, or (nx).

Write your design in the following space. Your design should be a list of C++ comments without any code.

//create function power with a b

//if a ==0 return 0

//if b = 0 return 0

//else return a\*power s, b-1

6b. Write a C++ program based on the design you created in Exercise 6a, and name it

**power.cpp**. Step through the code by hand.

6c. Enter, compile, link, and execute **power.cpp**. Then copy the output and save it in a block comment at the end of your program. Demo your program to your instructor.

The following is a copy of the screen results that might appear after running your program, depending on the data entered. The input entered by the user is shown in bold.

Enter your base number and your exponent: **3 4**

The number 3 raised to the power of 4 is 81.

# Lab 15.2 Text processing using Recursion

Recursion and iteration (loops) both repeatedly perform the same set of statements. Most, but not all, problems using a recursive solution can also be written with an iterative solution, and vice versa. Iteration is more efficient than recursion in that it uses less memory and CPU operations. Consequently, it is used more often than recursion. There are, however, some problems that are more easily solved with a recursive solution.

## Objectives

In this lab, you will design both an iterative and a recursive solution for the same problem.

## After completing this lab, you will be able to:

* Design and code an iterative solution to a problem.
* Design and code a recursive solution to the same problem. Estimated completion time: **60–70 minutes**

# Lab 15.2 Steps: Displaying a Pattern

1a. Design a program that will print out a decreasing pattern of stars in each row for the number of rows input by a user. Use a loop in the design. For example, if a user input 4 for the number of rows, the output would look as follows:

\*\*\*\*

\*\*\*

\*\*

\*

Write your design in the following space.

1b. Write a C++ program based on the design you created in Exercise 1a and name it

## starsLoop.cpp.

1c. Enter, compile, link, and execute **starsLoop.cpp**. Demo your program to your instructor.

1d. Design a program using a recursive function to solve the problem set out in Exercise 1a.

Write your design in the following space.

1e. Write a C++ program based on the design you created in Exercise 1d and name it

## starsRecursion.cpp.

1f. Enter, compile, link, and execute **starsRecursion.cpp**. Demo your program to your instructor.

# Lab 15.2 Steps: Recursive Palindrome

2a. Design a recursive algorithm to determine whether a string entered by a user is a palindrome (a string that reads the same if its characters are reversed; e.g., “toot,” is a palindrome). . You can use either a c-string or an object of the string class.

Write your design in the following space.

2b. Write a C++ program based on the design you created in Exercise 2a and name it

## palindromeRecursive.cpp.

2c. Enter, compile, link, and execute **palindromeRecursive.cpp**. Demo your program to your instructor.

The following is a copy of the screen results that might appear after running your program, depending on the data entered. The input entered by the user is shown in bold.

Enter a word to see if it is a palindrome: **carat**

The word you entered is carat. It is not a palindrome.

Enter a word to see if it is a palindrome: **radar**

The word you entered is radar. It is a palindrome.

# Lab 15.3: Designing a Recursive Algorithm to calculate the nth Fibonacci number

Recursion and iteration (loops) both repeatedly perform the same set of statements. Most, but not all, problems using a recursive solution can also be written with an iterative solution, and vice versa. Iteration is more efficient than recursion in that it uses less memory and CPU operations. Consequently, it is used more often than recursion. There are, however, some problems that are more easily solved with a recursive solution.

## Objectives

In this lab, you will design both an iterative and a recursive solution for the same problem.

## After completing this lab, you will be able to:

* Design and code an iterative solution to a problem.
* Design and code a recursive solution to the same problem.

# Lab 15.3 Steps: Recursion and Iteration

Consider the following sequence of numbers:

1, 1, 2, 3, 5, 8, 13, 21, 34,….

This sequence is called the Fibonacci sequence. After the first two numbers (Fib(1) = 1, and Fib(2) = 1), then the nth value of this sequence ( when n > 2) can be obtained by summing the values its two predecessors (i.e. Fib(n) = Fib(n-1) + Fib(n-2)). Therefore, we can write the definition of the nth Fibonacci number recursively as follows:

Fib(n) = **1 if n= 1**

**2 if n = 1**

**Fib(n-1)+Fib(n-2) if n > 2**

1. Write and evaluate a recursive function that generate the nth Fibonacci number
   1. Write a definition of a recursive function that takes a non-negative integer n and returns its Fibonacci number (i.e. its value in the sequence).
   2. Write a driver program to test your function. Test your function with few integer values, including the base cases.
   3. Enter the output you obtained for the following input numbers:

|  |  |
| --- | --- |
| n | Fib(n) |
| 1 |  |
| 2 |  |
| 3 |  |
| 6 |  |
| 10 |  |

* 1. What’s your observation about the performance of your program? Demo your program.
  2. Add statements to your driver program to measure the execution time of computing each Fibonacci number. To do that you can call function ***clock()*** and record the time before and after the call then find the difference between them. Note that **clock()** returns the time in clock cycles, so you need to divide it by the frequency of your CPU (which is in variable CLOCKS\_PER\_SEC). So your driver should look as follows:

**#include <time.h>**

int main() {

int num;

cout<<“enter a positive integer: “<<endl;

cin >> num;

clock\_t start = clock();

cout<<“The fibonacci of “<<num<<“ is: “<< **fib(num)**;

cout <<endl;

clock\_t end = clock();

cout<<"Time elapsed: "<<

((double)end - start) / CLOCKS\_PER\_SEC;

* 1. Enter the output and execution time for each of the following cases:

|  |  |  |
| --- | --- | --- |
| n | Fib(n) | Execution Time(T) |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 6 |  |  |
| 10 |  |  |

1. Write and evaluate an iterative version of the Fibonacci function
   1. Defined a new function that takes a non-negative integer n and uses a loop to compute the nth value in the Fibonacci sequence. *Hint: in this case, you need 2 temporary variables temp1 and temp2 to save the previous two Fib numbers at each iteration.*
   2. Use the driver program from task1 above to test this version of the Fibonacci function.
   3. Test your program with the following numbers and compare the execution of the two versions of your function. Demo your program.

|  |  |  |  |
| --- | --- | --- | --- |
| n | Fib(n) | Execution Time Recursive | Execution Time Iterative |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |
| 6 |  |  |  |
| 10 |  |  |  |