# CSCE 155 - C

**Lab 12.0 - Recursion**

## Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Read Chapters 11 and 24 of the Computer Science I textbook
3. Watch Videos 12.1 thru 12.3 of the Computer Science I video series

## Peer Programming Pair-Up

**For students in the online section:** you may complete the lab on your own if you wish or you may team up with a partner of your choosing, or, you may consult with a lab instructor to get teamed up online (via Zoom).

**For students in the face-to-face section:** your lab instructor will team you up with a partner.

To encourage collaboration and a team environment, labs are be structured in a *peer programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

## 1 Lab Objectives & Topics

At the end of this lab you should be familiar with the following

- How to use recursion to solve a problem
- Understanding the pitfalls of recursion

## 2 Background

In a programming language, recursion is a mechanism by which a function or method repeatedly calls itself. Recursion can be a powerful tool in a programming language, lending itself to a divide-and-conquer strategy to problem solving using very simple code. However, there are draw backs as we have previously demonstrated. Moreover, a non-recursive solution is always possible and may be far more efficient when the proper data structures are used. In general, a recursive function should:

1. Define a *base case* which does not call the function again, but provides a direct solution.
2. Otherwise, recursively call itself on a smaller input that works toward the base case.

In this lab, you will use recursion to solve a couple of problems.

## 3 Activities

Clone the project code for this lab from GitHub using the following URL: `https://github.com/cbourke/CSCE155-C-Lab12`

## 3.1 Solving a Problem Using Recursion

A *palindrome* is a string that is the same backward and forward: “rats live on no evil star” or single words such as “civic” or “deed” are palindromes. In this activity you will design and implement a recursive function that determines whether or not a given string is a palindrome. The recursive function should work as follows.

- The function is given a string, a left-index, and a right-index.
- If the characters at the left index and right index are not equal then we can stop, it is not a palindrome.
- Otherwise, we need to check the substring not including the (equal) characters at the left/right indices.
- Until the string is “empty” (the left/right indices are the same or have crossed each other) at which point we can say, yes it is a palindrome

For example, given the word “civic” we would check the first and last characters. Finding them the same, we would recursively call the function on the substring “ivi”. However, with the proper function parameters, we don’t create a new substring; we simply pass the relevant indices that define the substring.

#### Instructions

1. Complete the `isPalindrome` function in the `palindrome.c` source file as specified above.
2. Compile and test your program from the command line. Note: you can test a phrase containing whitespace by encapsulating it in double quotes; example: `./a.out "rats live on no evil star"`
3. Answer the questions on your worksheet and demonstrate your working program to a lab instructor

## 3.2 A Recursive Function

We have provided you a program, `recursiveFunction.c` that computes a recursive function similar to the Fibonacci sequence. This function, however, is defined as follows:

$$f(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ f(n-1) + \lfloor f(n-2)/2 \rfloor & \text{otherwise} \end{cases}$$

Examine at the contents of this source file and understand what is going on. Compile and run this program which accepts $n$ as a command line argument. Answer the questions on your worksheet.

### 3.3 The Jacobsthal Function

The Jacobsthal sequence is very similar to the Fibonacci sequence in that it is defined by its two previous terms. The difference is that the second term is multiplied by two.

$$J_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ J_{n-1} + 2J_{n-2} & \text{otherwise} \end{cases}$$

Write a recursive function to compute the $n$-th Jacobsthal number. Write a `main` function that reads in an integer $n$ and outputs $J_n$. Test your program for $n = 32$, the largest Jacobsthal number expressible by a 32-bit signed 2-s complement integer. Answer the questions on your worksheet.

### 3.4 String Suffixes

A suffix tree is a data structure that holds representations of all the suffixes of a particular string. For example, "computer" has eight suffixes, "computer", "omputer", ..., "ter", "er", "r". Suffix tree data structures are used extensively in DNA analysis and other pattern matching algorithms. For this exercise, we will not be generating an actual suffix tree. Rather, we will simply write a recursive function that, given a string outputs all the suffixes. Your program should take a string from the command line and produce a list of suffixes, one per line. An example run:

```
>a.out computer
computer
omputer
mputer
puter
uter
ter
er
r
```

## 4 Handin/Grader Instructions

1. Hand in your completed files:
    - `palindrome.c`

- `jacobsthal.c`
- `suffix.c`
- `worksheet.md`

through the webhandin (`https://cse-apps.unl.edu/handin`) using your cse login and password.

2. Even if you worked with a partner, you *both* should turn in all files.
3. Verify your program by grading yourself through the webgrader (`https://cse.unl.edu/~cse155e/grade/`) using the same credentials.
4. Recall that both expected output and your program's output will be displayed. The formatting may differ slightly which is fine. As long as your program successfully compiles, runs and outputs the *same values*, it is considered correct.

## 5 Advanced Activity (Optional)

1. Memoization is a technique that is used to avoid repeated recursive calls on the same value. The basic idea is that intermediate values are stored in a tableau. Upon a recursive call, the tableau is checked: if it contains a valid value, that value is used, otherwise the recursion is performed. When a value has been computed, it is added to the table so that subsequent recursive calls can use it and avoid repeating work. Redesign your Jacobsthal function to use memoization in its recursion.
2. Modify your recursive suffix function as follows. Instead of printing each suffix, store it in an array of strings. Think about who should be responsible for setting up the array–the caller or the actual function?