Lab 3: Recursion vs Iteration

Objective

To understand and compare the concepts of iteration and recursion in C programming through the computation of the Fibonacci sequence.

Introduction

Recursion can be difficult to grasp, but it emphasizes many very important aspects of programming, including execution speed and complexity. For this lab, you will program two well-known problems in their iterative and recursive forms: "finding the nth Fibonacci number" and "finding the sum of array elements".

This lab has 2 parts, it is recommended that you complete part A before your assigned lab time. Part B will be assigned during lab by your lab TA.

Tasks To Do for part A:

1. Write a C function to compute the Fibonacci sequence up to a given term using iteration.

   Prototype: `int iterativeFibonacci(int n);`

   Write a C function to compute the Fibonacci sequence up to a given term using recursion.

   Prototype: `int recursiveFibonacci(int n);`

   Note that Fibonacci number (fib) is defined as the sum of the previous 2 numbers in its sequence, given that fib (1) = 1 and fib (2) = 1.

   fib (n) = fib (n-1) + fib (n-2)

   For example, fib (3) = fib (2) + fib (1) = 1 + 1 = 2. Similarly, fib (4) = fib (3) + fib (2) = 2 + 1 = 3.

2. Write a C function using iteration to compute and return the maximum value in an array of size n.

   Prototype: `int iterativeFindMax (int * arr, int size);`

   Write a C function using recursion to compute and return the maximum value in an array of size n.

   Prototype: `int recursiveFindMax (int * arr, int size);`

3. Compare the execution time of both programs for Fibonacci sequence computation. Use both small and large term values (e.g., 5, 10, 25, 30, 35, 40, etc.).

Each of the 4 functions described above should be contained in its own function file, named to match the name of the function inside. (Ex. the iterative Fibonacci function will be in *iterativeFibonacci.c*). These functions must be called by main in a program file called lab3aMain.c with the following specification:

a) The main program will prompt the user to enter a value of n, and SIZE number of values for array arr. You may use a static array or dynamic. And you may use the following definition for SIZE, also given in lab3.h.

```
#define SIZE 10
```

b) The main program will then run each of the four functions one after another, keeping track of the time that each function takes to run to completion. **On the last page of this document,** you will find a sample code using the **time functions** you may find useful for this lab. In the given program, `clock()` function is used to record the start and end times of the execution. The difference between these times gives the CPU time consumed by the program. We convert this difference into seconds by dividing by `CLOCKS_PER_SEC`. By running these programs with large term values, you can compare the time taken by the iterative and recursive approaches to compute the Fibonacci sequence and Sum of all array elements.

c) The main program will then output the results and runtime for each function. A sample input/output scenario is shown on the next page.

Output

Output will be formatted as 2 lines per function. A sample input/output scenario is shown below:

<u>Sample Input / Output:</u>

```
Enter an input value: 5

Iterative Fibonacci (5) = 5

Time elapsed for iterative Fibonacci is 0.000027 seconds

Recursive Fibonacci (5) = 5

Time elapsed for recursive Fibonacci is 0.000036 seconds


Array elements are: 6 29 15 32 11 51 47 38 82 51

Iterative find max = 82

Time elapsed for iterative find max is 0.000006 seconds

Recursive find max = 82

Time elapsed for recursive find max is 0.000023 seconds
```

Also note that the displayed time to complete may be different at different times and different on different machines. That happens because the time that a program takes to execute depends on the load that the machine has in that specific moment – so it is normal for it to vary. But it meets the purpose of this lab as it allows us to compare iterative and recursive functions – note that we are not interested in computing the exact time for these functions.

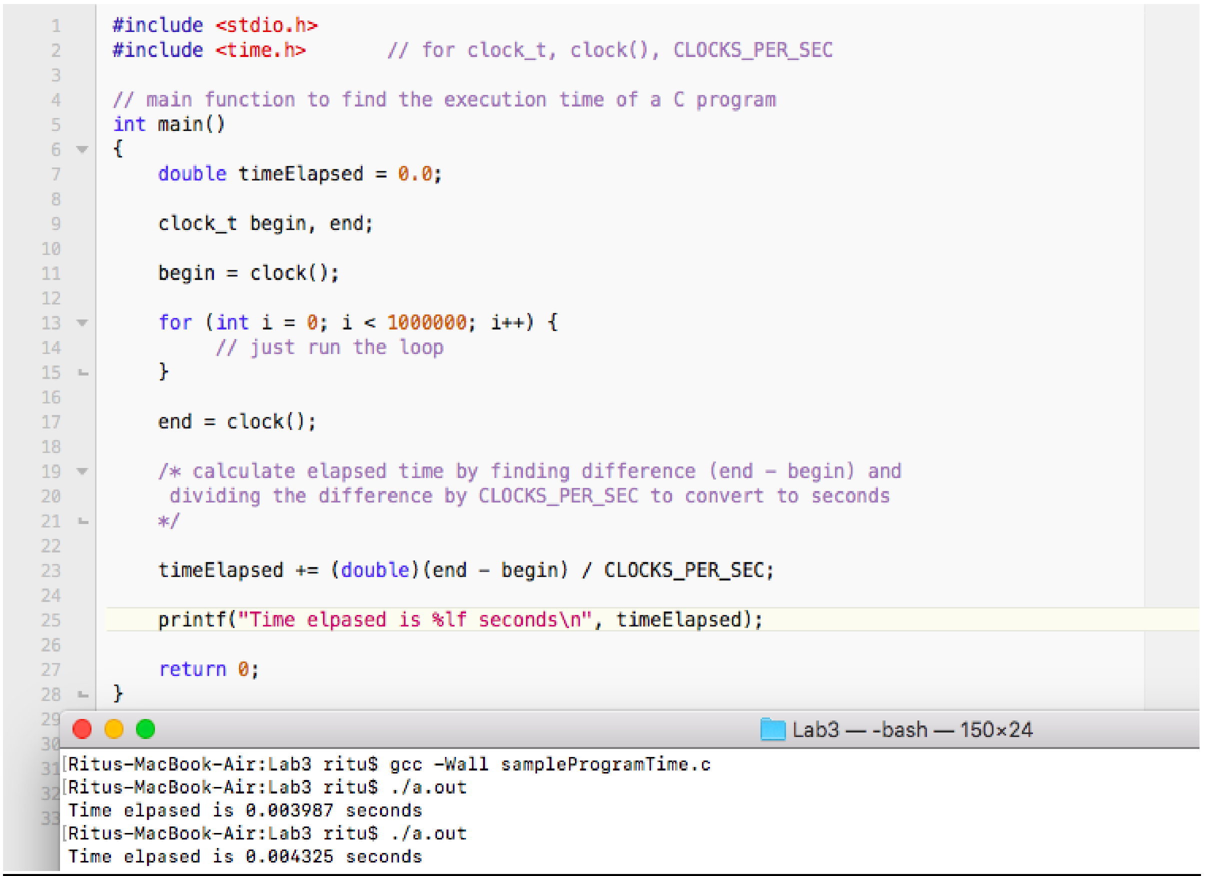
Tips

- Start with the iterative functions, as they are lengthier and more laborious to complete. More importantly, they'll provide expert knowledge about the algorithm that you'll need to understand the recursive versions.
- The recursive functions, by design are short and succinct. If your recursive functions start to go beyond 50 lines to complete, reconsider your algorithm: there might be a much simpler program that you can build.
- The main function is simple, but will require you to learn about how to time things in seconds and milliseconds. Standard library #include time.h has the functions you're looking for. **Page 3 has a sample code that you may use.**
- You may not see a major difference in timings between the functions for smaller values of input - try scaling up your program to use larger values of *n*. Some good numbers to try when scaling up might be 5, 10, 25, 30 and

so on . If you get up to these values and still don't see a difference, there might be something wrong with your implementation.

Sample code using clock () and time.h

Below is given a code that finds the clock time of a code that runs a for loop 1000000 times. As can be seen in the sample run, values of elapsed time are different for each run. This happens because the time that this program took to execute depends on the load that the machine has in that specific moment – so it is normal for it to vary. This code is given only to demonstrate clock() and CLOCK_PER_SEC – use them before and after each function call in your program.



```c
#include <stdio.h>
#include <time.h>       // for clock_t, clock(), CLOCKS_PER_SEC

// main function to find the execution time of a C program
int main()
{
    double timeElapsed = 0.0;

    clock_t begin, end;

    begin = clock();

    for (int i = 0; i < 1000000; i++) {
         // just run the loop
    }

    end = clock();

    /* calculate elapsed time by finding difference (end - begin) and
     dividing the difference by CLOCKS_PER_SEC to convert to seconds
    */

    timeElapsed += (double)(end - begin) / CLOCKS_PER_SEC;

    printf("Time elpased is %lf seconds\n", timeElapsed);

    return 0;
}
```

```
[Ritus-MacBook-Air:Lab3 ritu$ gcc -Wall sampleProgramTime.c
[Ritus-MacBook-Air:Lab3 ritu$ ./a.out
Time elpased is 0.003987 seconds
[Ritus-MacBook-Air:Lab3 ritu$ ./a.out
Time elpased is 0.004325 seconds
```

Submission

Submit your finished code to the lab 3 repository on GitLab. Include a **makefile** to link the four function files to the main and output an executable named "lab3a". The makefile must also include a 'clean' option, which removes all executables and (if present) intermediate .o files. All compilation should be done with the -Wall and -std=c99 flags.