Recursive vs Iterative Efficiency

Duncan Hook
Engineering and Computer Science Department
Seattle Pacific University
Seattle, Washington
Hookd@spu.edu

I. Introduction

When solving a problem in programming knowing how to approach and solve the problem is only half of the battle. Many problems have a vast array of possible solutions, none of them being necessarily the right way to approach the problem. That does not mean that some of these solutions are not better than others, as some of these solutions are undoubtedly more efficient than other solutions, figuring out the most optimal/efficient way to solve the problem is the other half of the battle. In this paper I will illustrate an example of this exact problem, by testing two different implementations of the same function. Seeking to determine which is more efficient between an iterative implementation and a recursive implementation of a function to determine the value of 2 raised to a certain power.

II. Background

This section will be used to explain any terms that are critical to understand the experiment. The experiment itself shall be detailed further in section III.

A. Power

A number being raised to a power, is to say how many times you multiply a number by itself. For example: a number raised to the power of three, means that the number will be multiplied by itself three times (1). The number is referred to as the base, and the power it is raised to is known as the exponent.

$$2^3 = 2 * 2 * 2 = 8 \tag{1}$$

B. Iterative Solutions

Iteration is an implementation that means a piece of code will be repeated either a fixed number of times, or until a set condition is met, or for as long as a condition is true. There are several common structures used for iteration including: while loops, for loops, and repeat loops.

C. Recursive Solutions

Recursion is also an implementation that will repeat, but as opposed to iteration recursion is repeated by making calls to itself. When a function calls itself, it requires something called a call stack, or just a stack in order to keep track of the calls that have yet to complete. If a recursive function makes too many calls without finishing it has the possibility of "overflowing" the stack, which is where the computer can not keep track of anymore calls without removing some of the already made calls. This overflowing will normally result in the program crashing.

D. Call Stack

The call stack is the method of which a computer keeps track of different consecutive calls to functions that have yet to resolve or finish in other words. The most recent thing to join the stack is the first thing that must be removed as well. Picture a stack of plates, you will take the top plate before any of the plates underneath, this is how the stack works. Once the most recent call has been resolved the stack will unwind, meaning that it will go through resolving as many of the earlier unresolved calls that it can.

E. Tick Period

The tick period of a clock represents the amount of time between recorded periods of a clock. The smaller the tick period the more precise a clock is, and thus can record smaller amounts of time accurately than a clock with a higher tick period.

III. Methodology

This section will detail my approach to the problem, as well as the specifications of the computer I used to run my experiments. Additionally, there will be a brief discussion on how I analyzed severe outliers in my results.

A. Solution

Since the implementations for both the iterative and recursive functions were provided by Dr. Arias, I had to figure out how to call the functions for certain exponents, record these results, and then analyze the data. The language I selected to do this experiment in is C++ due to my familiarity with the language. Calling the functions was easily solved using a for loop with the variable i, then passing the i as the desired exponent to the functions. Then continuously increasing i until it reached the maximum exponent that I set. 4310 times was the final value settled upon, this specific number shall be discussed in section V.

Timing the execution time of each function was done using C++'s chrono library and using the high-resolution clock class. This clock has the shortest tick period, and thus has the highest precision of the clocks available in the chrono library. The highest precision was necessary, especially on the smaller exponent values because the function calls only take a handful of nanoseconds, so in order to accurately record these miniscule times the tick period had to be incredibly quick.

I use the high-resolution clock to record the time of my system before I call one of the functions, and then use the clock to record the time of my system after the call has returned. Then I take the difference between these two times and record it in a variable associated with either the iterative or recursive function.

After both functions have been called and their execution times for a particular exponent are recorded, I export the current exponent to comma-separated value file. I do this for every value until the designated exponent has been reached. Finally, I took the generated file into R and I generated a graph based off the generated data points.

B. *Machine Description*

The computer I used for this experiment was:

Surface Pro 6.

Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz   2.11 GHz

16.0 GB RAM

C. *Outliers*

In the following section you will see the resultant graph has some extreme outliers, especially in the recursive results. I left these outliers in the graph because they occurred so regularly it leads me to believe it was not an implementation problem but more of a problem with my machine. I came to this conclusion after talking with some colleagues about how many calls they could do before experiencing a stack overflow error. My computer could only do 4310 recursive calls before experiencing this error and crashing, whereas my colleagues were able to perform upwards of 10,000 calls before crashing.

This leads me to believe that these spikes in execution time is due to my computer itself having problems allocating memory for the act of recursion. Leaving me with access to a smaller stack than my compatriots. But not so much smaller as to be unable to answer the questions I sought out to solve.

IV. Results

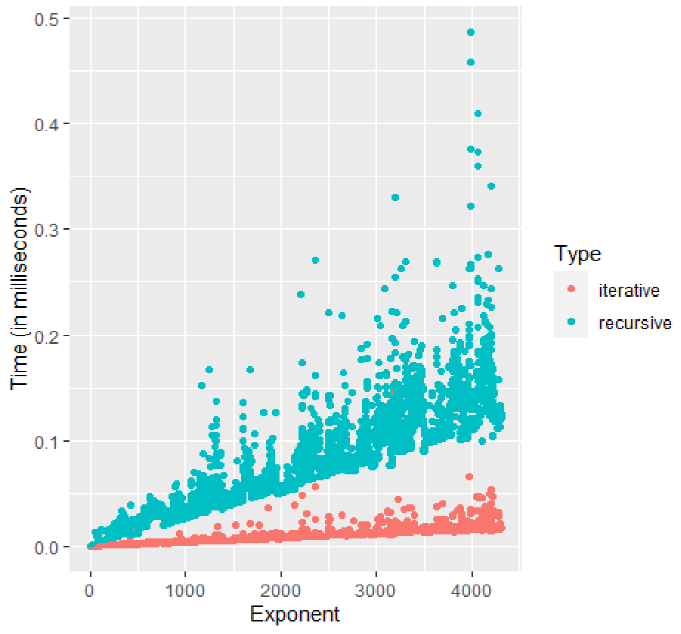

Fig. 1. Graph of experiments result, exponent (n) axis going to 4310.

V. Discussion

The difference between iterative and recursive implementations are quite apparent, growing ever more apparent the higher n becomes. I believe that this is due to how the recursive implementation must interact with the stack and how that can put a strain on your computer. Due to the stack piling higher and higher until calls start resolving [1]. Thus, because iteration does not have this piling operation cost it can perform incredibly more efficiently than recursive implementation can do.

This is also what results in n being limited, at least for recursions implementation. I do not believe this limits iteration, in my testing I could go up to 500,000+ as a value of n, with no issues (other than the total time it takes to finish that high of an n value). Whereas, as discussed earlier recursion was limited to 4310 calls (which was discovered through trial and error of adjusting the n value until I no longer experienced a stack overflow error). This value I believe may change depending on the power of your computer, as your computer could handle more unresolved items on the stack than a less powerful computer could. Thus, I believe with an enormously powerful supercomputer you could use any value of n for either implementation, but recursive will always be worse for incredibly large calls like this even if it theoretically *could* handle as large an n as you could throw at it.

I believe this knowledge will be incredibly useful in a professional setting. Because it gives me the knowledge that even though recursive implementations may be shorter than iterative implementations, they are not equivalent in efficiency as the size of n rises. For smaller amounts of n its possible using recursion implementation becomes more viable.

[1] Scott, M., 2016. Programming Language Pragmatics. Waltham: Morgan Kaufmann, pp.120-121 & 223.

VI. References

[1] Scott, M., 2016. Programming Language Pragmatics. Waltham: Morgan Kaufmann, pp.120-121 & 223.

[2]Enseignement.polytechnique.fr.2021. Std::Chrono::Duration - Cppreference.Com. [online] Available at:<http://www.enseignement.polytechnique.fr/informatique/INF478/docs/Cpp/en/cpp/chrono/duration.html#:~:text=Class%20template%20std%3A%3Achrono,tick%20count%20of%20type%20Rep%20.> [Accessed 23 January 2021].

[3] Cplusplus.com. 2021. High_Resolution_Clock - C++ Reference. [online] Available at: <https://www.cplusplus.com/reference/chrono/high_resolution_clock/> [Accessed 23 January 2021].

[4] "Input/output with files," cplusplus.com. [Online]. Available: https://www.cplusplus.com/doc/tutorial/files/. [Accessed: 23-Jan-2021].

[1] Scott, M., 2016. Programming Language Pragmatics. Waltham: Morgan Kaufmann, pp.120-121 & 223.

VII. Appendix

```
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
using namespace std;

//Iterative implementation
double iterativePower(double base, int exponent) {
	double retVal = 1.0;
	if (exponent < 0) {
		return 1.0 / iterativePower(base, -exponent);
	}
	else {
		for (int i = 0; i < exponent; i++)
			retVal *= base; }
	return retVal; }

//Recursive implementation
double recursivePower(double base, int exponent) {
	if (exponent < 0) {
		return 1.0 / recursivePower(base, -exponent);
	}
	else if (exponent == 0) {
		return 1.0;
	}
	else { return base * recursivePower(base, exponent - 1);
	}
}
void main() {

	//create and open output file
	std::ofstream myFile("output.csv");

	//naming columns of file
	myFile << "n , iterative, recursive \n";

	//name a variable to adjust the for loop easily
	long runCount = 4310;

	//make variables to handle the execution time of each function
	auto start = chrono::high_resolution_clock::now();
	auto end = chrono::high_resolution_clock::now();
	double iterativeTime, recursiveTime;

	for (long i = 0; i < runCount; i++) {
		//start clock
		start = chrono::high_resolution_clock::now();

		//iterative call with i exponent
		iterativePower(2, i);

		//end clock
		end = chrono::high_resolution_clock::now();

		//save iterative time
		iterativeTime = chrono::duration_cast<chrono::nanoseconds>(end - start).count();
```

[1] Scott, M., 2016. Programming Language Pragmatics. Waltham: Morgan Kaufmann, pp.120-121 & 223.

```
            //start clock
            start = chrono::high_resolution_clock::now();

            //recursive call with i exponent
            recursivePower(2, i);

            //end clock
            end = chrono::high_resolution_clock::now();

            //save recursive time
            recursiveTime = chrono::duration_cast<chrono::nanoseconds>(end - start).count();

            myFile << i << ", " << iterativeTime << ", " << recursiveTime << ", " << "\n";
        }//end for


        myFile.close();
}//end main
```

[1] Scott, M., 2016. Programming Language Pragmatics. Waltham: Morgan Kaufmann, pp.120-121 & 223.