Recursion vs Iterative: Same Big-O Different Performance

Naveen Janarthanan

Seattle Pacific University

Professor Carlos Arias

January 10th, 2019

Introduction

An exponential iterative and recursive function both have the same time complexity. However, the performance can vary between these two functions based on various variables. From the type of complier, you use, to the language the time measurement can vary even with the same time complexity. Both the functions have a big-O notation of O(n).

In this report we will start with the background, which explains some concepts, like iterative and recursive function, factorials, and a call stack. Next, we will go on to the methodology. Here we will explain how we implemented the functions and how we captured the time. We also talk about machine specifications. After that we go to the results, which show a scatterplot of the recorded data. Then we go to the discussion where we talk about various things such as why there is difference in performance and why there is a bound for n amongst other conclusions. Finally, we come to the appendix, where we have our references and the code for our implementation.

Background

To find a factorial of 4, (4!) we multiply 1*2*3*4. One way to calculate this would be to use an iterative function. To do this you would loop from 1 to 4, multiplying as we go. Another way to do this would be to use a recursive function. Since 2! = 2*1! and 3! = 3*2! we use n * factorial(n-1), calling the function inside the function. Basically we see that every "...factorial is a product of itself and the preceding one."[d] This is a more readable solution. However, once we get to bigger factorials we start to run into problems when using a recursion for larger factorials. To understand why, we need to know about a call stack. "A call stack is a mechanism for an interpreter (like the JavaScript interpreter in a web browser) to keep track of its place in a script that calls multiple functions — what function is currently being run and what functions are called from within that function, etc." [e] Since we are calling a the function inside the function this causes problems when you are using a larger factorial. We will explore this more later on. We also need to know about CPU optimization. A CPU has a stack buffer which tries to predict the correct return. Again, we will discuss this later on. Finding and exponent using iterative and recursion is similar concept, except we just need take the power of a value. For example, 2^2 is 4. Again, with iterative you implement it using a loop and with recursion you call back the function.

Methodology

To test the performance of an iterative and recursive power function, C++ was used. Both functions were placed in a for loop, that had a maximum value of 4309. This was the value of the exponent. This value was used because, anything above this value, threw an exception, which means the computer that is being used could not hand a higher value. In the for loop, the recursive and iterative function was called with the base of 3.141569265359. Before the iterative function a timer is started and after the iterative function the timer is stopped and same with the recursive function. The time was measured in nanoseconds and calculated by subtracting the stop time from the start time and was cast

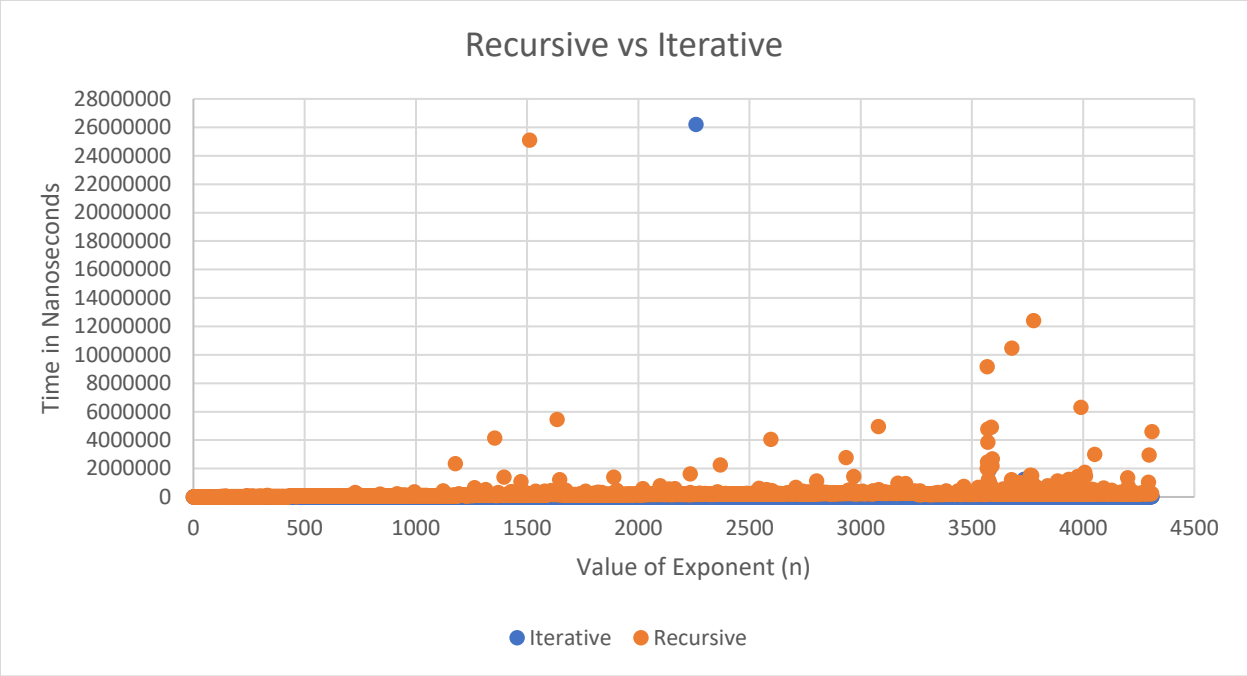
to a unit using duration_cast. The value of the duration was found by using duration.count(). The time measurement functions that were used were from the library called chrono. The code for this implementation is show is appendix [a]. To run this code, I used Visual Studio. This was done on a Lenovo Yoga 2 pro. It has 8gb of ram and an Intel Core i7-4500U CPU with a speed of 1.80Ghz and a max boost speed of 2.40Ghz. There are 2 cores on this CPU and it supports hyperthreading.

Discussion

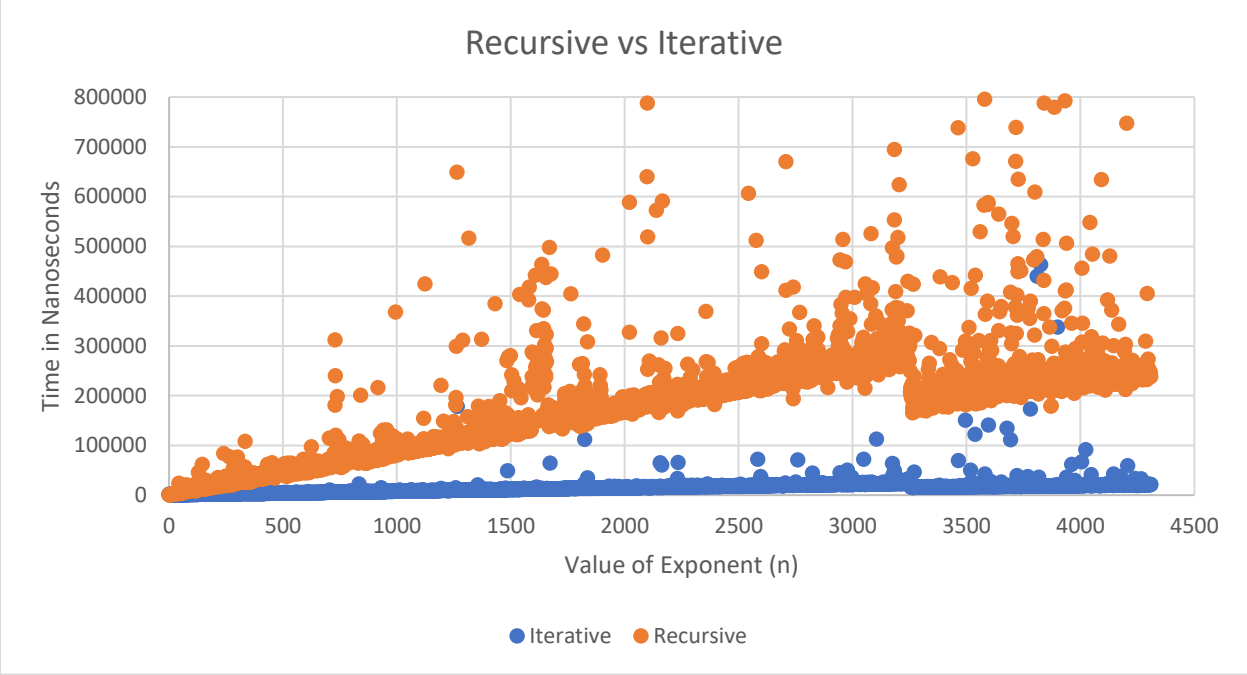
There could be a difference performance of these two functions, because although they have the same big-O notation, the real-world performance can still differ. This is because the big-O notation is a mathematical model. So just because we think the performance would be the same, doesn't mean that the big-O notation is wrong mathematically. But it does mean that it wrong for modeling real world performance. [b] So the question still stands, why is the real world performance different. One reason for this is, because when you look at recursion in assembly there are far more steps that take place, over just an iterative function. Another reason for the performance difference is the way recursion effects the processor. CPUs optimize performance by trying to predict what's going to happen by storing a cache. With recursion it can be hard for the CPU to predict returns and returns can be predicted wrong, which can cause long delays. [c] You can see this in data below, with recursion points straying further from the average than the iterative data points. Knowing this information is useful, because as we enter professions related to computer science we are better able to understand what kind of functions perform better. Also, we now know, that just because a function may have the same big-O notation, doesn't mean that it will have the same real-world performance. Deciding on when to use an iterative function or a recursive function, we have to consider various factors. If you are solving a large problem, using recursion could cause stack overflow problems, so it would be better to use an iterative function. Recursion functions have more readability and work better with smaller problems. The values of n are bounded by how much the CPU can handle. With recursion the CPU can handle far less than it can with iterative. The bound also depends on the language and the complier. We can't use any value of n for each of the implementations, because of a lack of memory in the CPU and higher that 4309 causes and overflow error. We can use any value for n up till the bound.

Results

Scatterplot with outliers included:



Scatterplot without outliers:



The bound of n is 4309.

Appendix

[a].

```cpp
#include <iostream>
#include <fstream>
#include <chrono>
#include <algorithm>

using namespace std;
using namespace std::chrono;

double iterativePower(double base, int exponent);
double recursivePower(double base, int exponent);

int main(int argc, char* argv[]) {


    std::ofstream myfile;
    myfile.open("example.csv");
    myfile << "n,Iterative,Recursive" << endl;
    for (int i = 0; i < 4310; i++) {
        auto start = high_resolution_clock::now();
        iterativePower(3.14159265359, i);
        auto stop = high_resolution_clock::now();

        auto duration = duration_cast<nanoseconds>(stop - start);

        auto start1 = high_resolution_clock::now();
        recursivePower(3.14159265359, i);
        auto stop1 = high_resolution_clock::now();

        auto duration1 = duration_cast<nanoseconds>(stop1 - start1);

        myfile << i << "," << duration.count() <<","<< duration1.count() << endl;


    }









    return 0;
```

```
}



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;
}

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);
    }
}
```


[b]. Lemire, D. (n.d.). Big-O notation and real-world performance. Retrieved from https://lemire.me/blog/2013/07/11/big-o-notation-and-real-world-performance/

[c]. https://cs.stackexchange.com/questions/56867/why-are-loops-faster-than-recursion

[d]. https://www.advanced-ict.info/programming/recursion.html

[e]. https://developer.mozilla.org/en-US/docs/Glossary/Call_stack