Performance Comparison of Iterative and Recursive Functions

Andrew McLain
Engineering and Computer Science Dept.
Seattle Pacific University
Seattle, WA, U.S.A.
mclaina1@spu.edu

I. Introduction

In computer programming, there often exists a choice to code the solution to a problem with either a recursive solution or an iterative solution. It can be important to understand why one choice may be more optimal than the other in these cases. In this experiment we will be comparing the run times of the recursive and iterative implementations of the same calculation: raising a constant base expression to a variable exponent. We will observe the results and try to form conclusions about what this means regarding the fundamental differences in performance between the two methods.

II. Background

First, we will establish definitions for several important terms along with some important concepts. An expression that represents repeated multiplication of the same number is called a *power*. The *base* refers to the number being multiplied, and the *exponent* refers to the number of times the base is used as a factor. *Iteration* in computer science refers to the process of repeating a sequence of instructions.

Programming languages often allow programmers to implement iteration through for-loops and while-loops. *Recursion* in computer science is a method of problem-solving which reaches a solution by solving smaller version of the same problem. Programmers can do this by calling a function within itself. Iteration and recursion are both ways for programmers to implement repetition in programs. Additionally, all iterative functions can be converted into recursive functions and vice versa by using *tail recursion*. Tail recursion refers to when the function makes the recursive call at the end of the function. This is why we are able to perform the same algorithm with the same time complexity using either recursive or iterative methods.

A *call stack* is a data structure which stores the active functions in a program. A main use of the call stack is to return control to the previously called function when the active function finishes running. The call stack does this by assigning a return address to each function as it is called. The call stack is often divided into *stack frames* which represent each function as it is called. These stack frames can contain information such as the return address, local variables, and parameters which were passed to the function. A *stack pointer* points to the top of the stack and increments or decrements appropriately to store information such as local variables. The *frame pointer* points to where the stack pointer pointed at the beginning of the function call so that control can be returned to the previous function when the time comes. The call stack can be visualized in a similar way to the stack abstract data type in that it operates according to the Last in First Out (LIFO) principle, i.e., the most recently called functions are at the "top" of the stack where the stack pointer is pointing.

There are also some important concepts and definitions we will be applying which relate to time complexity analysis. T(n) represents the run time of a program as a function of n. O(f(n)) represents the asymptotic *upper bounds* of T(n). We can define O(f(n)) as follows: T(n) is O(f(n)) if there exists a constant $c > 0$ and a constant $n_0 \geq 0$ such that for any $n \geq n_0$, $T(n) \leq c \cdot f(n)$. [4] Ω(f(n)) represents the asymptotic *lower bounds*. We can define Ω(f(n)) as follows: T(n) is Ω(f(n)) if there exists a constant $\varepsilon > 0$ and a constant $n_0 \geq 0$ such that for any $n \geq n_0$, $T(n) \geq \varepsilon \cdot f(n)$. [4] Θ(f(n)) represents asymptotically *tight bounds*. A function T(n) is Θ(f(n)) when it is both O(f(n)) and Ω(f(n)).

III. Methodology

The programming languages I used to implement the programs are C++ and R. I compiled the C++ program using the GNU g++ compiler and edited the code in CLion. The R program was run in RStudio. To time the C++ functions, I used the following from the C++ Standard Template Library: the chrono library's high_resolution_clock class now() function and chrono::nanoseconds, which is an instantiation of the std::chrono::duration class template. The machine on which I ran all of the programs is an Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz 2.19 GHz.

Fig. 1 is the C++ code I used to call and time the recursive and iterative functions (images are included at the end of this document due to their size). This program opens a .csv file and writes each of the n values and corresponding recursive and iterative function outputs to the file. I have denoted a highly abstracted time complexity analysis in the comments of the code. Fig. 2 is an example of how I compiled and ran the program from the command line using the g++ compiler. Fig. 3 is the code I wrote in R to process the .csv file and produce a graph. It is a simple program using the ggplot2 library which accepts the n values as the independent variable and the run times as the dependent variables. In order to remove outliers, I utilized the geom_smooth() function from the ggplot2 library. This function generates a linear regression of the data and uses a blur around the line to represent the 95% confidence interval.

IV. Results

Fig. 4 is a sample of the C++ program's output to the .csv file. Column A contains the n values, column B contains the iterative function run time in nanoseconds, and column C contains the recursive function run time in nanoseconds. Fig. 5 shows the first graph produced by the R program, which displays a scatter plot with every value of n plotted. The second graph, shown in Fig. 6, displays a smoothed linear regression graph which removes outliers. The results of the experiment show that the recursive and iterative functions both display linear time complexity, with the recursive function clocking in at 8.23 nanoseconds / n and the iterative function clocking in at 3.36 nanoseconds / n. So, the recursive function took approximately 2.5 times as long. When the value of n reached around 50,000, the computer started taking a considerably long time to run the program and the graphs began displaying unpredictable behavior. The highest value for which I could get a meaningful output was n = 646, which evaluated to 1.66085e+308. After this point, the values would output as "inf" since I exceeded the maximum value a double could hold in my program. When running the iterative function by itself, I could reach input sizes of 150,000 before the program started taking a very long time to run.

V. Discussion

First, we will conduct a time complexity analysis of the recursive and iterative functions based on the code itself. After completing this analysis, we will compare it to the results of the experiment. Every constant-time operation is simply denoted with a non-specified constant because it is not practical to provide an exact number. Trying to do so would be meaningless because each of these steps will become a fixed number of more primitive steps as the program is compiled and is ultimately dependent on the particular architecture used to do the computing [4]. Equations (1) through (4) include the time complexity analysis for the iterative function, which corresponds to the comments in the source code.

$$T(n) = c1 + c2 + c3 + c4 \cdot n + c5 \quad (1)$$

The iterative function is O(n) or higher since the degree of the polynomial T(n) is 1. This can also be demonstrated according to the definition of O(f(n)) provided in the *background* section:

$$T(n) = c1 + c2 + c3 + c4 \cdot n + c5 \leq c1 \cdot n + c2 \cdot n + c3 \cdot n + c4 \cdot n + c5 \cdot n = (c1 + c2 + c3 + c4 + c5) \cdot n \quad (2)$$

For all $n \geq 1$. Therefore,

$$T(n) \leq k \cdot n \text{ for all } n \geq 1, \text{ where } k \geq (c1 + c2 + c3 + c4) \quad (3)$$

which fits the requirements of the definition of O(f(n)).

The iterative function is also Ω(n) or lower since the degree of the polynomial T(n) is 1. This can be demonstrated according to the definition of Ω(f(n)) provided in the *background* section:

$$T(n) = c1 + c2 + c3 + c4 \cdot n + c5 \geq c4 \cdot n \quad (4)$$

which meets what is required by the definition of Ω(f(n)) where $\varepsilon = c4 > 0$. Since T(n) is both O(n) and Ω(n), it is Θ(n).

Equations (5) through (7) include the time complexity analysis of the recursive function:

$$T(n) = (a1 + a2 + a3 + a4 + a5) \cdot n \quad (5)$$

This is because the recursive function has a total of 5 possible constant-time operations and is called a total of n times.
The recursive function is O(n) or higher since the degree of the polynomial T(n) is 1. This can also be demonstrated according to the definition of O(f(n)) provided in the *background* section:

$$T(n) = (a1 + a2 + a3 + a4 + a5) \cdot n \leq k \cdot n, \text{ for all } n \geq 1 \text{ when } k \geq (a1 + a2 + a3 + a4 + a5) \quad (6)$$

The recursive function is also Ω(n) or lower since the degree of the polynomial T(n) is 1. This can be demonstrated according to the definition of Ω(f(n)) provided earlier.

$$T(n) = (a1 + a2 + a3 + a4 + a5) \cdot n \geq k \cdot n, \text{ for all } n \geq 1$$

$$\text{when } k \leq (a1 + a2 + a3 + a4 + a5) \quad (7)$$

Since T(n) is both O(n) and Ω(n), it is Θ(n).

These time complexity analyses are consistent with the findings included in the *results* section, which showed a linear growth rate for both functions. Beyond this, there are two possible reasons for the practical differences in performance between the recursive and iterative implementations. One reason is based on the time complexity analysis from this section. There are three conditional operations that the recursive implementation must execute (if, else-if, and else) for almost every value of n (*n - 1* times for a positive exponent and *n - 2* times for a negative exponent since the function is tail recursive) so that it can determine whether or not to recurse. These additional operations would not increase the growth rate of the recursive function, but could possibly cause it to take longer by a factor of a constant value relative to the iterative function. However, I believe a more likely reason is that the stack calls require more steps for the computer than the arithmetic operations. Every recursion call must save a register to the stack, save parameters to the stack call, save the next instruction pointer to the stack, and make a stack frame with the local variables. Every recursion return must compute a return value, empty the stack frame, save a return value to the stack, pop the return address from the stack, execute from that return address, pop the return value from the stack, and restore registers appropriately [5]. The unexplained behavior and eventually the inability to run the program when n reached values of 50,000 and beyond can also be explained by the use of the stack for the recursive function. The highest value of n that could run would likely be limited by the stack size allocated at the beginning of the program. For the iterative implementation, I do not see any obvious hard upper bounds for the value of n other than that it would eventually take a very long time to run. Understanding how two algorithms with the same time complexity on paper can have meaningfully different run times and performance in real-world situations is a good example of how we must connect our theoretical knowledge with practical applications. It is also a reminder of the essentialness of understanding hardware components if you want to deeply understand how software works.

VI. References

[1] Dennis C. Smolarski. "Notes A3: Recursion vs. Iteration" Math 60 Santa Clara University. http://math.scu.edu/~dsmolars/ma60/notesa3.html (Accessed January 21, 2021)

[2] "Tail Call" Wikipedia https://en.wikipedia.org/wiki/Tail_call (Accessed January 21, 2021)

[3] "Call Stack" Wikipedia https://en.wikipedia.org/wiki/Call_stack (Accessed January 21, 2021)

[4] Jon Kleinberg, Eva Tardos "Basics of Algorithm Analysis" in Algorithm Design, 1st ed. Pearson Education, Inc. 2006, ch. 2, sec. 2, pp. 36-38.

[5] Richard C. Demter "Procedures" in Assembly Language and Computer Architecture, 3rd ed. Jones & Bartlett Learning, Burlington, MA, 2015, Ch. 6, sec. 1, pp 111-113

Figure. 1. C++ Source Code.

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

//function to calculate value of base and exponent iteratively
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;
}

//function to calculate value of base and exponent recursively
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);
    }
}

int main() {
    //initialize constants
        const int PI = 3.14159265359;
    const int N = 50000;
    //initialize clock variables
    auto start = high_resolution_clock::now();
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<chrono::nanoseconds>(stop - start);
    double result;
    ofstream fout;
    //open csv file to output to
    fout.open("output.csv");
    //for each value of n, calculate run time for both functions
    for(int i=1; i<=N; ++i){
        //calculate iterative power run time and output to csv
        start = high_resolution_clock::now();
        result = iterativePower(PI, i);
        stop = high_resolution_clock::now();
        duration = duration_cast<chrono::nanoseconds>(stop - start);
        fout << i << "," << duration.count() << ",";
       //calculate recursive power run time and output to csv
        start = high_resolution_clock::now();
        result = recursivePower(PI, i);
        stop = high_resolution_clock::now();
        duration = duration_cast<chrono::nanoseconds>(stop - start);
        fout << duration.count() << endl;
    }

    return 0;
}
```

Figure 2. Command line compilation and execution.

```
claina1@Lando-Dell-Laptop-JKB5RJR:~/development/csc3430/recursive vs iterative$ g++ -o main main.cpp
claina1@Lando-Dell-Laptop-JKB5RJR:~/development/csc3430/recursive vs iterative$ ./main
```

Figure 3. R code to produce graphs.

```r
library(readr)
library(ggplot2)

# read csv file from directory
outputCSV <- read_csv("C:/Users/andre/Development/CSC3430/recursive vs iterative/output.csv",
                      col_names = FALSE)
View(outputCSV)

# Plot the scatter plot
ggplot(data = outputCSV, mapping = aes(x=X1)) +
   geom_point(aes(y=X2, color="iterative")) +
   geom_point(aes(y=X3, color="recursive")) +
   xlab("N") +
   ylab("Run time (nanoseconds)") +
   labs()

# Plot the smooth graph
ggplot(data = outputCSV, mapping = aes(x=X1)) +
      geom_smooth(aes(y=X2, color="iterative")) +
      geom_smooth(aes(y=X3, color="recursive")) +
      xlab("N") +
      ylab("Run time (nanoseconds)") +
      labs()
```

Figure 4. Csv file with small sample of output.

A1

	A	B	C
1	1	3500	1400
2	2	1000	1000
3	3	1000	1000
4	4	1000	900
5	5	1000	1000
6	6	1000	1000
7	7	1000	1000
8	8	1000	1000

Figure 5. Scatter plot produced by R.

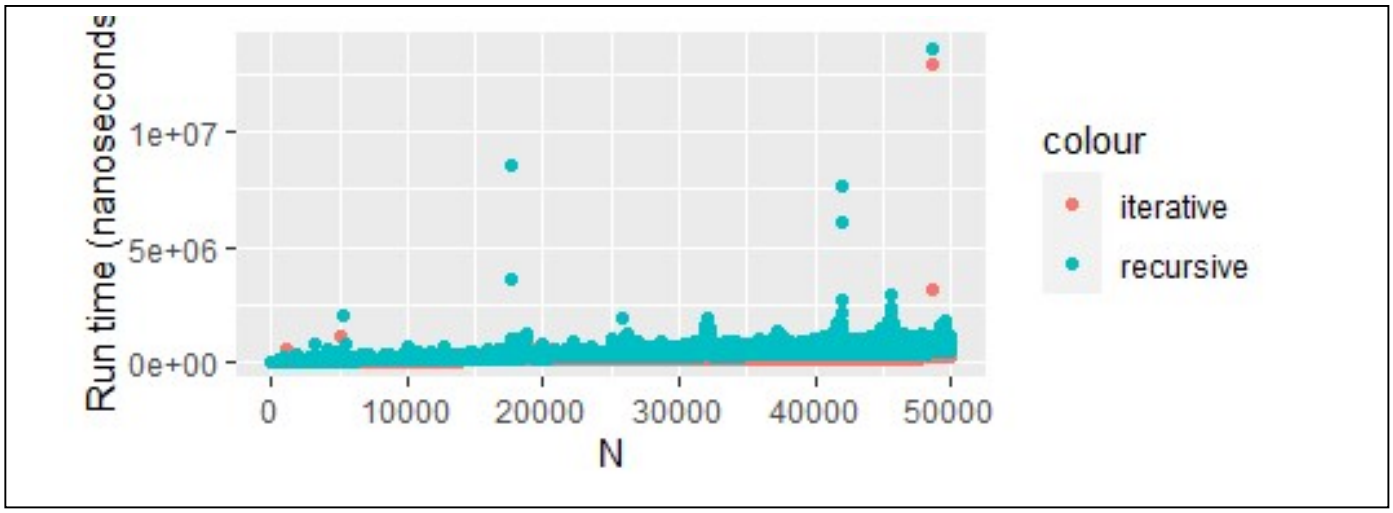

Figure 6. Linear regression graph produced by R.