Recursive vs. Iterative Algorithm Performance in Calculating Exponentials

Andrew Macpherson   
*Computer Science Department*  
*Seattle Pacific University*Seattle, Washington  
macphersona@spu.edu

*Abstract*—This paper explores two approaches to a problem of exponentials – recursive and iterative – and compares the runtimes of the algorithms.

Keywords—recursion, iteration, algorithms, exponentials

# Introduction (*Heading 1*)

In Computer Science, there are often multiple ways to approach a problem. Algorithms can accomplish the same things in totally different ways or even in different languages. However, not all algorithms are created equal. While multiple algorithms may solve a given problem, some will do it more efficiently than others, sometimes only for some inputs and sometimes consistently regardless of input. Given a problem of raising a base number to a power n, the two primary solutions are iterative and recursive. Here I compare runtimes of these two algorithms.

# Background

The calculation required to raise a base number to a power n is fairly simple for a computer to perform when n is relatively small. However, as the exponent value grows larger so does the runtime. Each of the two algorithms performs the same calculations but approaches the problem differently. The iterative solution initializes a return value variable of 1.0 and then begins iterating through a for loop that runs n times. For each iteration of the loop, the return value is multiplied by the base. Thus, an n-value of 3 would multiple the base by itself 3 times, thus producing the solution of basen. This solution does not require the function to “remember” anything from previous iterations since they are all stored in the return variable. Each iteration then only requires simple multiplication. The recursive solution takes a slightly cleverer approach. Starting with the highest value of n (given n > 1), it calls itself with decrementing values of n until n is 0. At n = 0, the solution to basen is 1 since any variable raised to the 0th power is 1. It then progresses through the calls to the function going back up the ladder (aka the call stack where they have been stored). The call stack operates with a LIFO methodology – Last In, First Out – that is, calls are added to the top of the stack and then removed from the top [1]. As the algorithm traverses back up the call stack, each call returns the base (now a value starting with 1) multiplied by the next value of n. The recursive solution can thus be described as an algorithm that lays out the problem by specifying to the computer exactly which operations it plans on doing and in what order before performing them. (Both of these algorithms are also capable of calculating exponentials for n < 0, but n > 0 for all calls in this analysis).

# Methodology

To test which of these algorithms operated faster, I repeatedly called both of them for progressively higher values of n and measured how long they took to calculate each exponential using the <chrono> library [2]. This allows the program to measure a clock before and after the call to each function and take their difference to find the total runtime. Specifically, this uses the function high\_resolution\_clock::now() and calculates the difference as duration = duration\_cast<microseconds>(stop - start) where start and stop are the respective values of the clock taken immediately before and after each function call. The base was kept the same at 3.14159265359. To find the maximum value of n, I tested the program increasing values until it would throw a stack overflow error. After that, I would try a lower number that was still higher than my previous maximum value of n that worked and continued this process of refining upper and lower bounds until I arrived at n = 165,000, which is at or near the upper bound of what the computer can handle. The machine I used was a 2019 MacBook Air with a 1.6 GHz Dual-Core Intel Core i5 processor and 8GB 2113 MHz of memory running macOS Ventura 13.0.1.

# Results

Chart, scatter chart

Description automatically generated

1. Scatterplot of n-value vs. milliseconds required to run calculation for recursive and iterative functions

# Discussion

The runtime for the iterative function was on average much faster than that of the recursive function to accomplish the same task. It should be noted that there are individual pink dots above other blue dots (signifying a faster recursive runtime), but these are seemingly random outliers that do not appear to occur with any predictable frequency or have any significant impact on the runtimes of the two algorithms. At relatively low values of n (n < 25,000) the two algorithms have similarly low runtimes but as n increases, so does the discrepancy between them.

My understanding of why the recursive algorithm takes longer revolves around the aforementioned concept of the call stack. In a sense, the pushing of each recursive call to the stack can be considered an iterative process. As Old Dominion University notes, “all function calls (including the recursive ones) are implemented via a runtime stack” [3]. Recursion also involves high overhead, which can slow it down. Since recursive calls use up more space in memory (in the form of the stack) than iteration, they lose corresponding efficiency of time [5]. With its multiple function calls that the iterative solution does not have, the recursive algorithm gives the CPU more to do and it slows down the algorithm [5]. The effects of this can be particularly noticeable when a computer has a small memory (such as 8 GB in this analysis). While some problems are best solved with recursive algorithms, it is worth remembering how stack buildup can slow down runtimes, especially when there is a large number of calls involved. However, there is no theoretical bound to the number of calls to a recursive function, and thus to n in this particular analysis. However, these algorithms are limited by hardware to a much greater extent than iterative algorithms. Any value of n can be used but on this particular machine, the stack reached its maximum capacity around a value of n= 165,000.

##### References

1. S. Mitra. “Recursion.” The University of Texas at Austin. Web https://www.cs.utexas.edu/users/mitra/csSpring2017/cs303/lectures/recursion.html (retrieved Jan. 16, 2023).
2. S. Mahapatra. “Meaure execution time of a function in C++.” Geeks for Geeks. https://www.geeksforgeeks.org/measure-execution-time-function-cpp/ (retrieved Jan. 16, 2023).
3. S. J. Zeil. “Converting Recursion to Iteration.” Department of Computer Science – Old Dominion University. https://www.cs.odu.edu/~zeil/cs361/latest/Public/recursionConversion/index.html (retrieved Jan. 16, 2023).
4. K. Academy. “Improving efficiency of recursive functions.” Khan Academy. https://www.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/improving-efficiency-of-recursive-functions (retrieved Jan. 16, 2023).
5. D. Pathak. “Difference between Recursion and Iteration.” Geeks for Geeks. https://www.geeksforgeeks.org/difference-between-recursion-and-iteration/ (retrieved Jan. 16, 2023).

# Appendix – C++ Implementation of Algorithms

/\*  
 \* Original function code sourced from Dr. Carlos Arias - Recursive vs. Iterative HW Assignment  
 \* Use of chrono library derived from https://www.geeksforgeeks.org/measure-execution-time-function-cpp/  
 \*  
 \* By: Andrew Macpherson  
 \* 1/16/2023  
 \* \*/  
  
#include <iostream>  
#include <chrono> //Used to measure runtimes  
#include <fstream> //Used to output file  
using namespace std::chrono;  
  
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);  
 }  
}  
  
  
int main() {  
 double base = 3.14159265359;  
 //Maximum value of n  
 int max = 165000;  
  
 std::ofstream data;  
 data.open("data.csv");  
  
 //CSV Headers  
 data << "n," << "Iterative Runtime (ms)," << "Recursive Runtime (ms)\n";  
  
  
 //Measures n as many times as possible starting at 1  
 for (int n = 1; n < max+1; n++) {  
 //Measures runtime of iterativePower function  
 auto start = high\_resolution\_clock::now();  
 iterativePower(base, n);  
 auto stop = high\_resolution\_clock::now();  
 auto duration = duration\_cast<microseconds>(stop - start);  
  
 //Saves data to CSV  
 data << n << "," << duration.count();  
  
 //Measures runtime of recursivePower function  
 start = high\_resolution\_clock::now();  
 recursivePower(base, n);  
 stop = high\_resolution\_clock::now();  
 duration = duration\_cast<microseconds>(stop - start);  
  
 //Saves data to CSV  
 data << "," << duration.count() << std::endl;  
 }  
  
 data.close();  
  
 return 0;  
}