Time Performance Between

Iterative and Recursive Implementations

Chandler Stevens   
Department of Computer Science  
Seattle Pacific UniversitySeattle, WA

*Abstract*— An implementation of a solution to a problem can vary in performance dependent on a few factors. One factor involves the selection of the most appropriate programming technique. Although two or more techniques may have a similar time complexity, there may exist a substantial variation in empirical time performance. Therefore, a “constant” factor influences each implementation. The purpose of this report is to observe and report this phenomenon.

Keywords—performance, iterative, recursive, implementation

# Introduction

In developing high performance software development solutions, selecting proper techniques of implementation ensures a fast and responsive product. Since there are various techniques to implementing solutions to each problem, an analysis of the empirical time performance of a candidate algorithm is necessary. Measuring the time an algorithm takes to execute exposes the constant factor that is not present in the time complexity of a Big-O Notation figure [1]. For example, it is possible to develop an iterative algorithm for a problem with a time complexity on the order of O(n). But it is likewise possible to create a recursive algorithm for the same problem that also has a time complexity of O(n).

# Background

## Definitions

Power: A number known as an exponent that raises another number known as the base; a function that multiplies the base by itself the same number of times as the value of the exponent.

Iterative Solution: An algorithm that produces a result utilizing a process of sequentially looping through a range of values.

Recursive Solution: An algorithm that produces a result utilizing a process of calling itself until resolving to a base case.

Call Stack: An ordered sequence of the currently active calls of a function.

Big-O Notation: An approximate mathematical expression of the upper bound of the time complexity of an algorithm represented without any constant factors.

## Theory

Using Big-O Notation can provide an approximation for the temporal performance of an algorithm but obscures the true performance due to a constant factor dependent on various system-specific factors [2]. Recursive algorithms increase the size of the call stack with each recursion. If the capacity of the call stack is exceeded, then a stack overflow occurs. Although recursive algorithms tend to have high space complexities due to the amount of memory required for many recursions, recursive algorithms have the potential to have faster time complexity than a counterpart iterative algorithm. Therefore, only an empirical time measurement can determine which implementation possesses a smaller constant factor and, thereby, delivers faster performance for the end-user.

# Methodology

Utilizing tools to precisely measure the execution time of an algorithm provides a more accurate figure for the effective time performance on a given system. Comparing the time needed to execute the power function in both iterative and recursive implementations will provide a benchmark for the distinct performance between these two techniques. The technique that has an overall shorter execution time is deemed faster and, therefore, possesses a smaller value constant factor. The selected base to test was pi approximated to eleven decimal places. The average times for each execution were compiled into a single CSV file and then mapped to a scatter plot in Microsoft Excel.

## Programming Language

The programming language chosen to implement the iterative power function, recursive power function, and time comparison function was Python version 3.7.1. In Python, any absolute value greater than 1.8e308 evaluates to "inf" [3]. Since pi to the power of 621 is greater than 1.8e308, positive 620 was designated the maximum exponent and negative 620 was designated the minimum exponent.

## Test Machine

The system chosen to run the program was a Dell Inspiron 5759 with an Intel® Core™ Processor i7-6500U CPU @ 2.50 GHz with 16.0 GB of installed DDR3 RAM installed with Windows 10 Home version 1909 64-bit.

## Time Measurement

The time measurement function chosen to determine execution time was "repeat" from the "timeit" package available for Python. This function executes a segment of code a given number of iterations and repeats this process a given number of repetitions. The "timeit.repeat" function returns a list of integer values with each value representing the total time taken to run all iterations of a code segment within the corresponding repetition. For the number of repetitions, the default value of three was chosen as to provide three attempts at determining the total execution time since external processes could greatly inflate the total time of a given repetition [4]. For the number of iterations to run within each repetition, a value of ten thousand was selected as to allow the program to complete within a reasonable time frame (about 90 minutes) while still providing decent resolution. As per the recommended procedure, selecting the minimum integer value from the list returned from "timeit.repeat"provides the most likely total time of execution, since there are external processes also running on the system [4]. Dividing that integer value by the number of iterations provides the average time in seconds for the code segment to execute one time. This result can then by multiplied by a factor of one billion to convert it into nanoseconds; thus, revealing how many nanoseconds on average each implementation takes to execute for each possible exponent.

Consequently, each of the two implementations was called for all 1,241 possible exponents (from -620 to +620) with each exponent executed 10,000 times with each iteration repeated 3 times, which equates to a combined total of 74.46 million executions of both function calls.

# Results

Fig. 1. Scatter plot of the empirical time performance measurement of the iterative power function compared to the recursive power function.

# Discussion

As evident from Figure 1 above, both implementations (from 0 to +620 or from 0 to -620) provide a linear time complexity of O(n). However, the recursive approach clearly takes more time than the iterative approach, especially for larger exponents. In fact, analyzing the raw data itself revealed that the recursive function was only faster for exponents from -3 to +3, which included the best factor at an exponent of zero with the recursive function performing merely 203 nanoseconds faster. Using the linear best-fit-line feature in Microsoft Excel revealed that (from 0 to +620) the algorithms had the following respective equations:

These equations provide the approximate value for the constant factor applied to each algorithm. The iterative implementation has a constant factor of only about 54; whereas, the recursive implementation has a constant factor almost exactly 4 times greater at about 215. Since the recursive algorithm is typically only 25% the speed of the iterative algorithm except for very low exponents, the iterative implementation is decidedly the most appropriate solution, namely as a power function solution.

## Explanantion of Performance Difference

The reason why the recursive approach is so much slower is due to the overhead present in managing the call stack frame [5]. Since there is back-end code that translates to substantial machine language at the hardware level, the system simply must take more time to keep track of the call stack when handling recursions. The iterative approach benefits from very little overhead since the machine code involves simple "jump" statements rather than having to manipulate memory on the stack. The iterative algorithm merely loops through a range of numbers, but the recursive algorithm must setup a new function call and roll back the call stack upon reaching the base case with each recursion [6].

## Exponent Bounds

In the iterative solution, the bounds of the exponent are determined by the largest number supported by the programming language or operating system, since increasingly larger numbers require more memory to store their value. In the recursive solution, the bounds of the exponent are limited by the maximum recursion depth supported by the programming language or possible due to the call stack capacity available in the system memory in addition to the same bounds limiting the iterative solution [7]. Although Python 3 only supports numbers up to 1.8e308 meaning that the exponents in this experiment were bounded to an absolute value of 620, the iterative solution would theoretically be able to support many more larger numbers than the recursive solution [3]. Therefore, we cannot provide literally any exponent value to these implementatations, but can provide a fairly large range of values in both implementations under the existing language constraints.

## Professional Formation

These results are applicable to the professional formation of entry-level associate software engineers since informs intuition and imparts experience towards best programming practices. Although recursion typically is slower in time performance than iteration, software engineers must utilize discernment to determine if recursion may actually be the more appropriate implementation for a specific solution to a problem, which may factor in programming language, data structures involved, and common input values. As software engineers, we must balance the priorities associated with performance and decide the proper techniques to implement the best solution possible.

##### References

1. Cake Labs, Inc., “Big O Notation,” InterviewCake.com, Janurary 2020.

<https://www.interviewcake.com/article/java/big-o-notation-time-and-space-complexity>.

1. “Big-O notation,” KhanAcademy.org, Janurary 2020.

<https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation>.

1. “Python Numbers,” LearnByExample.org, January 2020.

<https://www.learnbyexample.org/python-numbers/>.

1. Python Software Foundation, “timeit — Measure execution time of small code snippets,” docs.python.org, January 2020.

<https://docs.python.org/3/library/timeit.html>.

1. D. Pathak, G. Kumar, and M. Kumar, “Difference between Recursion and Iteration,” GeeksForGeeks.org, January 2020.

<https://www.geeksforgeeks.org/difference-between-recursion-and-iteration>/.

1. “Difference Between Recursion and Iteration,” TechDifferences.com, May 2016.

<https://techdifferences.com/difference-between-recursion-and-iteration-2.html>.

1. A. Bharadwaj, “Python | Handling recursion limit,” GeeksForGeeks.org, January 2020.

<https://www.geeksforgeeks.org/python-handling-recursion-limit/>.

##### Appendix

# Chandler Stevens  
# CSC 3430: Algorithm Design and Analysis  
# Dr. Arias  
# Python 3 program that compares the performance between iterative and  
# recursive implementations of the power function  
  
# Import repeat function from timeit package  
**from** timeit **import** repeat  
  
# Initialize constants  
# Set number of iterations to execute  
ITERATIONS **=** 10 **\*\*** 4  
# Set conversion factor from seconds to nanoseconds accounting for averaging  
NANO **=** 10 **\*\*** 9 **/** ITERATIONS  
# Set maximum value of n that does not return as plus or minus "inf"  
SYSTEM\_MAX\_EXPONENT **=** 620  
# Set an approximated value for pi as the base  
PI **=** 3.14159265359  
  
  
# Purpose: Iteratively calculate the power of a given number  
# Parameters: base which represents the base number (float)  
# exponent which represents the power to raise the base (integer)  
# Returns: Calculated power of the base raised by the exponent (float)  
**def iterativePower(**base, exponent**):** # Initialize return value to one  
 retVal **=** 1.0  
 # If exponent is negative  
 **if** exponent **<** 0**:** # Return reciprocal of recursively called function and positive exponent  
 **return** 1.0 **/** iterativePower**(**base, **-**exponent**)** # Otherwise  
 **else:** # Iterate from zero to the exponent  
 **for** i **in** range**(**exponent**):** # Multiply the base to the current calculated result  
 retVal **\*=** base  
 # Return the final calculated result  
 **return** retVal  
  
  
# Purpose: Recursively calculate the power of a given number  
# Parameters: base which represents the base number (float)  
# exponent which represents the power to raise the base (integer)  
# Returns: Calculated power of the base raised by the exponent (float)  
**def recursivePower(**base, exponent**):** # If exponent is negative  
 **if** exponent **<** 0**:** # Return reciprocal of recursively called function and positive exponent  
 **return** 1.0 **/** recursivePower**(**base, **-**exponent**)** # Otherwise, if exponent is zero  
 **elif** exponent **==** 0**:** # Return one (base case)  
 **return** 1.0  
 # Otherwise  
 **else:** # Return base multiplied by recursively called function  
 # with decremented exponent  
 **return** base **\*** recursivePower**(**base, exponent **-** 1**)**  
# Purpose: Function to time performance and compile CSV file  
# Parameters: None  
# Returns: Nothing  
**def compareImplementations():** # Initialize output to empty string  
 output **= ""** # Iterate from most negative value of n to most positive value of n  
 **for** n **in** range**(-**SYSTEM\_MAX\_EXPONENT, SYSTEM\_MAX\_EXPONENT **+** 1**):** # Append to output string:  
 # Value of n and comma  
 # Average time in ns of iterative power function for n and comma  
 # Average time in ns of recursive power function for n and newline  
 output **+= (**str**(**n**) + "," +** str**(**min**(**repeat**(lambda:** iterativePower**(**PI, n**)**,  
 number**=**ITERATIONS**)) \*** NANO**) + "," +** str**(**min**(**repeat**(lambda:** recursivePower**(**PI, n**)**,  
 number**=**ITERATIONS**)) \*** NANO**) + "\n")** # Display value of n to track progress  
 print**(**n**)** # Create/Overwrite CSV file and open file stream  
 fout **=** open**("Iterative-vs-Recursive-Power.csv"**, **"w")** # Write output string to CSV file  
 fout.write**(**output**)** # Close CSV file stream  
 fout.close**()**# Execute main function  
compareImplementations**()**