# Recursive and Iterative Solutions

## A Performance Analysis of Two Functions

Bridget Mae Erb
Seattle Pacific University
Seattle, Washington
erbb@spu.edu

***Abstract*—This paper serves as an example as to why it is important to examine which algorithm serves a situation best. We conduct an experiment to see solution to implementing a power function—a recursive or an iterative—is faster.**

***Index Terms*—algorithms, computer science, iteration, mathematics, performance, programming, recursion.**

### I. Introduction

When it comes to programming, taking care to use the right techniques can drastically save run time, programmer energy, and/or computer memory. However, it can be difficult to know what kind of implementation will serve a situation better. Sometimes algorithms with the same theoretical time complexity will empirically perform very differently in a specific use-case. The purpose of this research paper is to look at two implementations of the power function in order to observe this.

### II. Background

Before continuing, there are some terms that need to be defined:

- ***Subroutine***: refers to a set of programming instructions that will perform specific tasks that then can be called repeatedly within a program. [1]
- ***Call Stack***: sometimes referred to just as "the stack," the call stack is a term relating to how information on subroutines is stored within computers. The call stack is a stack data structure, meaning that information on the bottom is only processed once the subroutines on the top are done processing (Last In, First Out). Recursion is dependent on this structure, but iteration tends to not be. [2]
- ***Recursion***: Recursive functions call themselves. Once a recursive function is called, the only way for it to end is by reaching its base case.
- ***Iteration***: When a function is iterative, it is implemented using loops. The amount of loops an iterative function has been determined by a control variable.
- ***Power***: A power function is a function of the form f(x) = x^n, where n is any real number. [3]

### III. Methodology

To determine which solution empirically runs faster, we first build two power functions, one that utilizes iteration and one that utilizes recursion. They each should yield identical results, but this is not relevant to what we will be measuring.

Instead, once these functions are built, we create a for loop ranging from 0 to n, with n defined based on the programing language or limit of the call stack. [4] We time each function in this loop and then transfer those results into a csv file containing n, the iterative run times, and the recursive run times. These data are then graphed into a scatterplot. Since the purpose of this experiment is to understand and visualize the performance of these two functions—not to predict performance—outliers within the scatterplot will not be omitted.

For this experiment, the programming language Python3 was used to create the algorithms. Visual Studio Code editor was used for writing, debugging, and syntax highlighting. R and RStudio were used to create a scatterplot of the results.

The function `time.perf_counter_ns()` from the `time` module was utilized to time how long each function took to run, as using regular seconds would not be accurate enough to measure the variety of times recorded for the recursive and iterative functions. [5]

Although there are no bounds for integers in Python 3, the range for n was set from 0 to 998. The reason for this will be discussed at length in section V. [6]

This experiment was conducted on a Windows 10 Pro with a 64-bit operating system, x64-based processor, and 8.00 GB of RAM. The processor is an Intel(R) Core(TM) i7-7660U CPU @ 2.50GHz.

## IV. Results

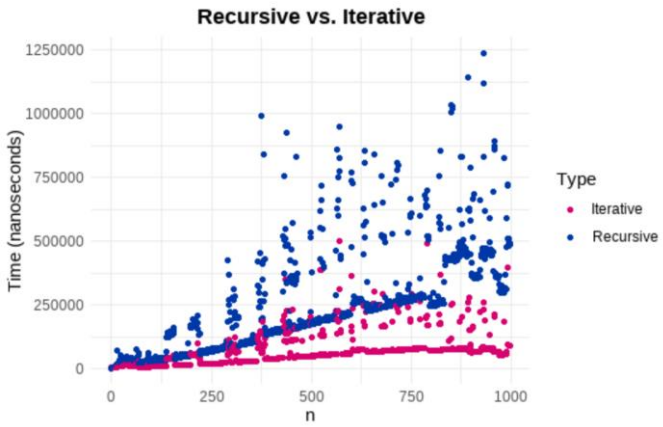


Fig. 1. Outcome of the Experiment

## V. Discussion

Looking at the results of each functions performance in this experiment, it is evident that iterative solutions are generally faster than recursive solutions.

This analysis can also be backed up once it is understood how each of these functions run. While any function relies on the call stack to execute (unless dynamically programmed), recursion has a unique dependency on the call stack because of the way recursion works. When a function is called, it immediately goes to the top of the stack—thus making that function the first thing needing to be executed before anything else is. [7] With recursion, the function is continuously calling itself and adding more and more calls to the stack. Therefore, this implementation takes longer, and interestingly enough—also why n is bound to 998.

While Python 2 had a maximum bound for an integer (n could not exceed 9223372036854775807), Python 3 does not. Instead, there are no bounds for integers, meaning there is no bound for n other than the available memory a machine has.[ 8] Python 3 does, however, have a limitation on how many times a recursive function can be called. Exceeding the limit of n = 998 results in this error:

```
RecursionError: maximum recursion depth exceeded
                in comparison
```

There is a perfectly logical explanation as to why Python 3 has this limit for recursion: stack overflow. When the allocated memory of the stack structure is reached, it causes stack overflow. If a base case for a recursive function is not defined properly, this can lead to an infinite recursion cycle and eventually crash the program because of this. [9] So, as a fail-safe, Python sets a limit to how many times recursion can happen. This can be avoided with tail recursion, but it will not stop an infinite recursion cycle. Additionally, if you change the recursion limit you are in effect changing the call stack - but you may get out of memory exceptions rather than stack overflow ones. [10] This means that an iterative solution is not bound to any n in Python 3, but recursive solutions are.

Regardless of time, both recursive and iterative solutions are useful algorithms to keep in the mind of any programmer. Fig. 1 shows how in this instance, even though the functions both have a time complexity of O(n), the iterative version is empirically faster than the recursive. This occurrence remains generally true for most algorithmic implementations of iterative and recursive solutions. [11]

However, one cannot ignore the limitations of iterative solutions either. Although iterative solutions do tend to be faster, more complicated problems will result in making iterative code messier and longer. Having to change iterative code may be a long, tedious, and confusing process. In contrast, recursive solutions are often quick to both write and comprehend and may be suited for simple tasks requiring few computing resources. [12]

## VI. References


[1] Beecher, K. (2018). Bad programming practices 101 : Become a better coder by learning how (not) to program. Berkeley, CA: Apress, pp. 87.

[2] Liu, Yanhong Annie. Systematic Program Design : From Clarity to Efficiency. New York: Cambridge University Press, 2013. Accessed January 20, 2022. ProQuest Ebook Central.

[3] Röss, D. (2011). Learning and teaching mathematics using simulations plus 2000 examples from physics (De Gruyter textbook). Berlin ; Boston: De Gruyter.

[4] "Built-in Types" - *Python 3.10.2 documentation*. [Online]. Available: docs.python.org.

[5] "Time" - *Time access and conversions - Python 3.10.2 documentation*. [Online]. Available: docs.python.org

[6] "Python: Handling recursion limit," GeeksforGeeks, 26-May-2021. [Online]. Available: geeksforgeeks.org

[7] Mongan, J., Kindler, Noah, & Giguère, Eric. (2018). Programming interviews exposed : Coding your way through the interview (Fourth ed., Wrox professional guides). Indianapolis, IN: John Wiley & Sons, pp. 127-128.

[8] Bagheri, *Reza. Python Stack Frames and Tail-Call Optimization.* Apr. 24th, 2020. Accessed on Jan 17th, 2022. Available: towardsdatascience.com

[9] Mongan, J., Kindler, Noah, & Giguère, Eric. (2018). Programming interviews exposed : Coding your way through the interview (Fourth ed., Wrox professional guides). Indianapolis, IN: John Wiley & Sons, pp. 126

[10] Liu, Yanhong Annie. Systematic Program Design : From Clarity to Efficiency. New York: Cambridge University Press, 2013. Accessed January 20, 2022. ProQuest Ebook Central, pp. 86.

[11] Mongan, J., Kindler, Noah, & Giguère, Eric. (2018). Programming interviews exposed : Coding your way through the interview (Fourth ed., Wrox professional guides). Indianapolis, IN: John Wiley & Sons, pp. 128

[12] Felleisen, M. (2001). How to design programs : An introduction to programming and computing. Cambridge, Mass.: MIT Press.


## VII. Appendix

Python 3 Source Code:

```
import time
```

```
   import csv

   def iterativePower(base, exponent): # Iterative
Implementation
       retVal = 1.0
       if (exponent < 0):
           return  1.0  /  iterativePower(base,  -
exponent)
       else:
           for x in range(1, exponent):
               retVal *= base
       return retVal


   def recursivePower(base, exponent): # Recursive
Implementation
       if(exponent < 0):
           return 1/recursivePower(base, -exponent)
       elif (exponent == 0):
           return 1
       else:
           return   base   *   recursivePower(base,
exponent - 1)


   baseNum = 3.14159265359

   with open('test.csv', mode='w') as csv_file:
       fieldnames = ['n', 'Iterative', 'Recursive']
       writer      =      csv.DictWriter(csv_file,
fieldnames=fieldnames)
       writer.writeheader()
       for x in range (0,998):
           iStart = time.perf_counter_ns()
           iterativePower(baseNum, x)
           iEnd = time.perf_counter_ns()

           rStart = time.perf_counter_ns()
           recursivePower(baseNum,x)
           rEnd = time.perf_counter_ns()

           writer.writerow({'n':   x,   'Iterative':
iEnd-iStart, 'Recursive': rEnd - rStart})
```

R Source Code for Scatterplot:

```
   library(ggplot2)
   library (tidyverse)

   theme_set(theme_minimal())
   theme_update(text    =    element_text(family    =
"sans", color = black))
   theme_update(plot.title  =  element_text(hjust  =
0.5, face = "bold"))

   test <- read_csv("test.csv")
   scatter <- test %>%
     pivot_longer(!n, names_to = "Type", values_to
= "Time")

   ggplot(scatter, aes(x=n,y=Time, color=Type))+
     geom_point() +
     scale_color_manual(values                      =
c('#D70270','#0038A8')) +
     labs(title="Recursive vs. Iterative",
          x="n", y = "Time (nanoseconds)") +
     theme(
       plot.title = element_text(hjust = 0.5))
```