*Iteration vs. Recursion*

Christopher Moroney  
*Seattle Pacific University*  
*Department of Computer Science*Seattle, Washington, United States  
moroneyc1@spu.edu

*Abstract* — In computer programming, there are many different strategies and techniques to creating a solution. In the case of creating a power function, which is a simple mathematical function that takes base (b) to the power of an exponent (n), there are two common and simple methods: iteration and recursion. Both techniques can provide the correct solution easily and quickly. However, while both methods may be able to carry out the power function easily, is the implementation of either technique necessarily its fastest version? The purpose of this project is to determine if the iterative power function or the recursive power function has a faster run time.

Keywords —

Power – mathematical operation that takes a base number (b) and multiples itself a given number of times (n), known as the exponent.

Iteration – a function where provided number of instructions is completed for several times, known as looping, until completing the loop, halted by a stop command, or a solution is found

Recursion – a function where a function is calls on itself in order to find a solution

Function – a set of instructions that is used to organize programming in general variables so that it can be called at different times

Call stack – a stack data structure that stores run time operations and instructions when the program runs, in which the deepest operations and instructions are the first ones executed

Run time – the duration for the program to complete the set of instructions coded into the main function

# Introduction

The purpose of computer programming is to create instructions for a computer than can assist us in finding solutions for any productive purpose. Primarily, the objective is to create a program that works, and can provide the correct solutions as quickly and cleanly as possible. Correct code and solutions that are fast is very useful in today’s world, especially with the quick development of programs and services in the technology industry. However, with the evolution of computer science and programming, programmers have become very skilled at finding solutions. The next important factor in programming beyond solutions is the allowing a program to run quickly. Functions that can run efficiently and quickly while still outputting correct solutions have paved the way for newer types of programs to carry out larger and more complicated functions, such as Artificial Intelligence. In order to run fast code, however, we need to see what types of techniques and functions can reduce our program’s runtime. Two common techniques of calling a function is through iteration and recursion, and the objective is to determine which of these two techniques for a function has a faster run time in nanoseconds.

# Background

## Iteration

Iterative functions have typically been known as “the easier” functions to first learn [1]. The typical iterative function looks like a set of instructions that have been set to continue executing repeatedly for a given number of loops. Once the loops have been closed, the function is pretty much complete unless needing to return a value or call on another function. While sometimes iterative functions can be chunky and is just the “simplest” of code, iterative functions have been able to complete many tasks and have been easier to comprehend by students [2].

Iteration functions have always been much easier to write out and visualize simply due to its ability to list the number of times we want to loop the instructions. This leads to one drawback, however. While the loop number can make sense for our code at times, the program may only be running because of a variable that determines the number of loops the program takes [2]. This differs from recursion in that a recursive function will operate until it no longer can run through itself (explained below). The iterative method is running mostly on user input of number of loops, which can limit the capability of a program slightly. For instance, if we use an iterator or iteration technique to analyze or parse through a string, the number of loops we conduct is potentially dependent on the length of the string, and the programmer has to manually insert the length of the string as number of times to loop through the function. This type of iteration differs from recursion, in that the number of times the instructions are executed depends on what the programmer determines opposed to when the function is told to stop calling on itself (explained in B).

## Recursion

Recursion was an idea that had been derived from iteration in that if a computation was able to be executed on the Turing machine and logically was correct, that functions could be referenced within an instruction [2]. Instead of repeating set instructions repeatedly in a loop for a set number of times, the recursive method would allow for a function to be called repeatedly until the function itself establishes to completing the same instructions. This prevents the possibility of illogical statements that can sometimes occur in iteration and to compress the code a little more instead of written in chunks like iteration usually is [2].

The downsides of recursion however are that recursive logic has been conceptually difficult for people to understand. Recursive functions involve reaching down all the way down to where the function is told to stop, and then execute instructions almost in a “reverse” sense. Thus, the logic within the function must be perfect, or else there will be mathematical or logical errors in the algorithm [3]. Because of this, recursion is more likely to cause a program to crash opposed to iteration due to incorrect logic. Iteration can cause incorrect solutions with a logical error, but recursion is more likely to crash if written incorrectly. Recursion is also more difficult to debug because the function will complete instructions in its depth first [3]. This means that it is harder to track a bug, say in the middle of the recursion, opposed to in iteration where the bug can be just in one of the loops. The run time is supposedly slower in recursion because of the depth of performing operations and working back up to the original call [3].

# Methodology

The following methodology is variable and can be completed in different languages and techniques. The results produced from this project are my own and from my own techniques. The exact code from my Java is found at in the appendix of this paper.

## Solution / Code

\*\*\* See appendix \*\*\*

* The project used TimeUnit nanoseconds using System.nanoTime()

## Language and IDE

* The language used for this project was Java 7
* The IDE used for coding was NetBeans 8.2

## Machine and programs

* The machine used for this program was an Apple MacBook Pro, APPLE SSD AP0128J, 2.3 GHz Intel Core i5, 8 GB 2133 MHz LPDDR3

## Description

In order to test whether iteration or recursion has a faster run time, the function that will be used to compare these techniques will be a simple power function. The power function takes two values (base and exponent) and will output the mathematical function base to the power of exponent. Power in mathematics means that the base will be multiplied by itself for the exponent number of times. The base will be set as a double data type and will be defaulted as 3.14159265359. The exponent will be an int data type and increasing by 1, starting at 0. The output will be printed onto a CSV file.

All instructions that involve returning a value are being executed by the program are placed in a type of structure known as the call stack. A stack is a structure that contains values where the values that are inputted first are outputted last. In the case of the call stack, the active instructions that are inputted first are outputted last. This can be clearly seen in recursion, but this also will happen in iteration as well. In both cases, the printing to a file will be the last instruction.

For the iteration solution, which means to complete the instructions repeatedly until the number of loops are completed, the call stack is going to consist of instructions that are very similar in just multiplying the initial value (which is one, explained in section III) for a number of times. The instructions that are inserted are depending on the next instructions and results. This is how the call stack is implemented. While the call stack may call for the last multiplication instruction first, that instruction requires the resulting value from the next instruction. Thus, the higher exponents will require a larger runtime.

The recursive solution involves the returning value calling on the function itself continually until reaching a certain value. The call stack contains instructions that reaching down as far until the exponent equals 0 (explained in section III), and then perform the operations back up to the original instruction. The first instructions involve calling the function with the largest exponent needed for that value, and then calling the function repeatedly with the exponent decreasing by 1. Since higher exponents require calling the function recursively more and involve more instructions in the call stack, the runtime will be much larger for larger exponents.

## Time Measurement

* In order to take time measurements of the functions, I take two data types, known as longs, which are the data type compatible with System.nanoTime(). I place one of these longs before calling the iterative or recursive function, and one after.
* The System.nanoTime() will take the time measurement from the beginning of the program until that point. These numbers will be quite large because the time it takes to get to that point of the solution is quite a while. However, I took the difference between the second long and the first long. This difference was the time it took for the iterative or recursive function to be executed.

## Definitions

Double – a numerical data type that contains decimal points and values

Int – a numerical data type that consists only of integers

Long – a numerical data type that is compatible with System.nanoTime()

Base – the value that is being multiplied to itself mathematically

Exponent – the value that determines how many times the base multiplies itself

Call stack – a stack data structure that stores run time operations and instructions when the program runs, in which the deepest operations and instructions are the first ones executed

Instructions – an operation written in the programming language that the computer processes

* Difference – mathematical operation that subtracts one value from another

# Results and data

The following data was collected by me on the machine above. The program to process by CSV file was Microsoft Excel. All the data collected is my own and was not recreated from any outside source.

## Table

## The column value n represents the power that the Power function was raised to, which was **39681 iterations and recursive calls.** Anymore calls to the functions causethe program crashed. Iterator-time represents the time in nanoseconds that the iterative power function ran for each power n. Recursive-time represents the time in nanoseconds that the recursive power function ran in nanoseconds for each power of n.

|  |  |  |
| --- | --- | --- |
| n | iterator-time | recursive-time |
| 0 | 2075 | 1012 |
| 1 | 555 | 384 |
| 2 | 332 | 329 |
| 3 | 312 | 414 |
| 4 | 261 | 500 |
| 5 | 281 | 516 |
| 6 | 289 | 684 |
| 7 | 310 | 723 |

…

|  |  |  |
| --- | --- | --- |
| 39679 | 44269 | 117516 |
| 39680 | 44783 | 172912 |
| 39681 | 70773 | 150707 |

## Graph

\*\* note that the different colors for the trendline are used to

allow for the trendlines to be more visible. Red is for the

recursive trendline, and green is used for the iterative trendline

\*\* sample size of 500 exponents were used. Normally my

sample size is 1000, but 1000 points were not able to show up on

## the graph visibly with the trendline. Thus, I cut my sample size down to 500. I also used the =RAND() function in Microsoft Excel to randomize my points and plot them onto my graph, which means sample size points were randomized for my graph.

# Conclusions and discussion

## Runtime

Based on the data, it appears that the iterator power function has a lower runtime rate than the recursive power function. While initially the recursive function had a lower runtime than the iterative, the entire trend of data shows that the iterator time has a faster rate of completion than the recursive time. There are also exponents where the recursive time is faster than the iterative time. However, these points are either outliers, or simply just pieces of data that don’t show the overall trend in the data. Again, the data shows in the table that the iterative function has a faster runtime than the recursive run time, which is seemingly more than double when at the largest exponent points.

## Outliers

There are several outliers in the data, as shown on the graph. It appears to be that many of the outlier points are found as a result of the recursive function opposed to the iterative function. A study [4] showing that analyzed outliers and created a “fuzzy” algorithm to make sense of the outliers showed that outliers tend to come more frequently as a result of recursive programs. Thus, it seems to make sense that there are a lot of recursive outliers.

Thinking about the logic as well, it seems like there would be naturally a lot of outliers in the recursive program because of the instructions that are performed and the order they are being performed in. In iterative functions, the set of instructions are the same, but repeated over a provided number of loops, whereas recursive functions depend on the number of inputs. Thus, if the program must make a difficult mathematical operation that involves a lot of decimal points, the program may take abnormally longer before recalling back to the original instruction.

## Discussion

Based on the trendlines in the data, we can see that the iterative power function has a faster runtime than recursive runtime. Why is this the case? Well, one possible explanation as to why this may be the case may be because of the function itself. We are specifically only looking at a power function, which may just be a reason why a recursive method doesn’t work as well as an iterative method. According to this study [5], “transforming recursion into iteration eliminates the use of stack frames during program execution.” This means that there is less pieces of data and instructions (given by the frames) that are being put into the call stack. Clearly, if less instructions are being put into the call stack, then less instructions will naturally cause the program time to lower.

This same study [5] also raises an interesting point, in that there are cases where recursion is a better solution than iteration. Such a case arises when there are multiple operations and instructions that can arise, and when using specific data structures such as stacks. This could make sense if we changed the type of function that we were using to a generic mathematical function instead of just a single operation, such as the power function. This also raises questions though, such as what in what cases is neither recursion nor iteration is a good method to use? Knowing which method between iteration and recursion is important because for enormous data sets, being able to process the data sets quickly is important. In the professional field, large data bases are quite common and being able to quickly process the data is important. If we think about companies and programs that contain huge amounts of data, such as Microsoft, Google, Apple etc., are their programs going to be able to function if it runs slow? While recursion vs. iteration may not be the entire answer, any improvement to speed while yielding the same results without error is a huge benefit. While for this project, the number of values we computed into the power function was 39681, the difference in iteration and recursion was at most about 15000 to 20000 nanoseconds. This is not a big difference of time. However, in the large companies with lots of data, how long could the program take to not only compile but continually run with world-wide data? This detail on running the program with the best possible method is important for the professional industry, and recursion vs. iteration is one of those small details that matter.

I understand that the maximum number of exponents of “39681” seems totally random, but this number was calibrated specifically for the machine I ran this program on. I noticed when I was testing this program that when I did even just one more exponent, my program would crash, and I received a StackOverflowError. Clearly, my MacBook Pro is not one of the most powerful computers in the world. However, many other computers are very powerful and could process the amount of data that I had in a fraction of the time that my computer was able to. The boundaries for n can really be anything that the experimenter wants. However, in this specific project, a greater number of iterations or recursive calls would seem more logical so that we could not only compare times between iteration and recursion but see how much of an increase in time there would be between n-values. If the experimenter sees enough values and increase to identify a trend, then supposedly that amount of n values is good enough. I would suggest just by seeing the number of outliers in the little bit of data I graphed, more data is better so that there can be more to analyze on a graph, such as the different outliers of recursive values as mentioned above. More outliers can potentially show whether iteration or recursion is a better method for the power function. People that have larger computer processors probably would want to take advantage of using more data so that more testing can be done on the program. However, selectin just any huge number of n is not necessarily going to provide more useful data. N should be tested for and maximized to see not only how fast the program can run with that n, but also to see how much we can maximize our data findings graphically.

One final point that I will add is that in the data, the first few pieces of data had recursion having a faster time than iteration. One question that could arise would be that does a reduced number of times a function is called recursively eventually result in a faster program time compared to iteration? In the data, we see that early on, recursion is running slightly faster than iteration, but is this a good enough measurement? If there is no benefit to recursion in terms of runtime, then what is the point of using recursion?

## Takeaways

* In general, with more inputs n in a power function, iteration have a faster runtime than recursion.
* Just because iteration has a faster runtime does not mean that iteration is the better method to use.
* While this project shows iteration runs faster than recursion in terms of nanoseconds (which is not extremely substantial), the potential for a program to run faster in a much larger data set can have much larger benefits with faster code
* Data is good, more data can sometimes be good, but not always. The reason is that if we keep increasing our values of n, but not seeing any difference or substantial outliers, then we are just taking up memory, and the program is more likely to crash.
* Outliers possibly come from mathematical operations and logic, where the runtime may increase if the math becomes complex or the system must call on many pending operations (recursion) [4]

##### References

1. S. Eilenberg and C. C. Elgot, “Iteration and Recursion,” *Columbia University, NY*, pg. 378, October 1968
2. C. Kessler and J. Anderson, “Learning Flow of Control: Recursive and Iterative Procedures,” *Human Controller Interaction (2)2,* vol 2, no.2, pg 136 – 140, June 1986
3. R. I, Soare, “Computability and Recursion,” *The Bulletin of Symbolic Logic*, vol. 2, no. 3, pg 1 – 10, 1996
4. Y. Bodyanskiy, “Outlier resistant cecursive fuzzy clusering algorithms,” *Computational Intelligence, Theory and Applications*, pg. 647 -- 652, 2006
5. L. A. Yanhong and S. D. Stoller, “From recursion to iteration: what are the optimizations?,” *ACM SIGPLAN Notices*, vol. 34, no. 11, pg. 1 – 10, November 2019
6. C. Arevalo, “2020 Winter Recursive vs Iterative Assignment (HW1)”, *Canvas*, pg 1-3, January 2020

\*\* NOTE: classes iterativePower and recursivePower came from source [6] (see references) \*\*

package csc3430p1;

import java.io.\*;

/\*\*

\* @param args the command line arguments

\* @throws java.io.IOException

\* @throws java.lang.InterruptedException

\*/

public class CSC3430P1 {

public static void main(String[] args) throws IOException, InterruptedException {

try (PrintWriter out = new PrintWriter("FileforCSC3430.csv")) {

double base = 3.14159265359;

int limit = 39681;

StringBuilder header = new StringBuilder("");

header.append("n");

header.append(", ");

header.append("iterator-time");

header.append(", ");

header.append("recursive-time");

header.append("\n");

out.write(header.toString());

header.delete(0, header.length());

StringBuilder data = new StringBuilder("");

for (int i = 0; i <= limit; i++){

long start = System.nanoTime();

double returnVal = iterativePower(base, i);

long end = System.nanoTime();

long firstTime = (end - start);

long start2 = System.nanoTime();

double returnVal2 = recursivePower(base, i);

long end2 = System.nanoTime();

long secondTime = (end2 - start2);

data.append(i);

data.append(",");

data.append(firstTime);

data.append(",");

data.append(secondTime);

data.append("\n");

String printVal = data.toString();

out.print(printVal);

data.delete(0, data.length());

}

}

}

public static double iterativePower(double base, int n){

double returnVal = 1.0;

if (n < 0){

return 1.0 / iterativePower(base, -n);

} else {

for (int i = 0; i < n; i++){

returnVal \*= base;

}  
}

return returnVal;

}

public static double recursivePower(double base, int n){

if (n < 0){

return 1.0/recursivePower(base, -n);

} else if (n == 0){

return 1.0;

} else {

return base \* recursivePower(base, n - 1);

}

}

}