International Journal of Recent Technology and Engineering (IJRTE)
ISSN: 2277-3878, Volume-8 Issue-5, January 2020

Performance Anatomization of Computer Algorithms Based on Iterative and Recursive Methodologies

Raji Ramakrishnan Nair

Abstract*— An algorithm is a clear specification of a sequence of instructions which when followed, provides a solution to a given problem. Writing an algorithm, depends upon various parameters, which leads to strong algorithmic performance, in terms of its computational efficiency and solution quality. This research paper presents the different methodologies of writing algorithms of data structure and also provides their performance analysis with respect to time complexity and space complexity. As we know that, for the same problem, we will have different algorithms, written using different approaches. All approaches of algorithms are important and have been an area of focus for a long time but still the question remains the same "which to use when?", which is the main reason to perform this research. This research provides a detailed study of how algorithms being written using different approaches work and then compares them on the basis of various parameters such as time complexity and space complexity, to reach the conclusion.

Keywords: Algorithm, Performance Analysis, Time Complexity, Space Complexity, Iteration, Recursion.

I. INTRODUCTION

In the field of Computer Science, algorithm plays a vital role and it is said to be the core of every applications or technologies that exists today. Algorithm is an unambiguous, step – by – step procedure for solving a problem, which is guaranteed to terminate after a finite number of steps [1]. We can also say that an algorithm is a tool to solve computational problems. Designing an algorithm involves various techniques and methodologies. The speed of any algorithm depends upon the number of operations it performs. Every algorithm falls under certain class. From increasing order of growth, they are classified as constant time algorithm, logarithmic algorithm, linear time algorithm, polynomial time algorithm and exponential time algorithm [2]. Each algorithm falls under any of these following classes: Brute Force, Divide and Conquer, Dynamic Programming, Greedy algorithm, Backtracking algorithm, Transfer and Conquer, Decrease and Conquer [2].

Manuscript published on January 30, 2020.
* Correspondence Author
Ms. Raji Ramakrishnan Nair*, Assistant Professor in P. G. Department of Computer Applications, Marian College Kuttikkanam (Autonomous), India

Based on the way of writing, each algorithm can be classified into two areas: Iterative algorithm and Recursive algorithm. A recursive algorithm calls itself which usually passes the return value as a parameter to the algorithm again. This parameter is the input while the return value is the output [3]. Recursion usually divides the problem into smaller pieces of the same nature. The output of one recursion is the input for the next recursion. Examples of recursive algorithms are generation of factorial, Fibonacci number series, merge sort, quick sort, etc. An iterative algorithm is one which does not call itself and may contain 'for' loop or 'while' loop. When we focus on these two concepts, we find that any algorithm could be written in an iterative way or recursive way. That means, the algorithm which is written in an iterative way could be written in recursive way and vice-versa. So, we can say that power-wise both iterative and recursive algorithms are same. When we have to analyze these algorithms, the methods are different. To solve a given problem, P, different algorithms could be written by different programmers. Based on some parameters, we could analyze all those algorithms for the same problem P. In this research paper, I would like to focus on the performance analysis of any algorithm, which is written either using an iterative approach or a recursive approach.

II. COMPARATIVE STUDY ON ITERATIVE ALGORITHM AND RECURSIVE ALGORITHM

Performance analysis of an iterative algorithm and a recursive algorithm uses different approaches. The performance analysis of any algorithm is based on two parameters: Time Complexity and Space Complexity. Time Complexity is defined as the total amount of time needed by any algorithm, to complete its task. The time complexity of any algorithm, P, depends on the number of times, a loop structure in an iterative program get executed or a recursive function being called, etc. The analysis of an iterative algorithm, depends mainly on the part of the algorithm, where loop structure is used, loops such as 'for' loop, 'while' loop, etc. Taking some examples to explain how time complexity of an iterative algorithm is been calculated. In the following examples, I am not following any specific formats for writing an algorithm. All algorithms are written in simple English, to make it easy to understand.

(i) Consider the following algorithm written iteratively -

```
A()
{
```

```
int i, j;
for (i: =1 to n)
for (j: =1 to n)
   printf ("Sample");
}
```

In this algorithm, time complexity depends on the number of times the *printf* statement get executed in the given nested loop. For each value of 'i', 'j' loop executes for n number of times. So, the total number of times, the *printf* statement gets executed is n^2 times. Hence, the time complexity for this algorithm is $O(n^2)$.

(ii) Let us take another example written iteratively –

```
A()
{
 i = 1, S = 1;
 while (S < = n)
 {
    i = i + 1;
       S = S + i;
       printf ("Sample");
 }
 }
```

This is an example where *while* loop is used. In the *while* loop, the value of 'i' is incremented by one and the value of 'S' gets incremented by 'i' value. The total number of times the *printf* statement gets executed depends upon the number of times *while* loop condition becomes true.

S 1 3 6 10 15 21 …..

i 1 2 3 4 5 6 …..

From the above series we understood that as the value of 'i' gets incremented by 1, the value of 'S' is the sum of first 'i' natural numbers. The *while* loop condition becomes false only when the value of S becomes greater than the value of n, i.e., S > n. Suppose after *k* iterations, the value of S becomes greater than n. Since S is the sum of first 'i' natural numbers, we can write in the following way also –

$k (k + 1)/2 > n$

$(k^2 + k)/2 > n$

After continuing the derivation, we can conclude like this –

$k > \sqrt{n}$

Therefore, the time complexity of this algorithm is $O(\sqrt{n})$.

For analyzing a recursive algorithm, the method used for calculating the time complexity is different from that used for iterative algorithms. The reason is there is nothing to count here in the recursive algorithm. Suppose we have the following recursive algorithm –

```
A(n)
{
  if (n > 1)
  {
     return (A(n/2) + A(n/2))
  }
}
```

Let us assume T(n) as the time taken for algorithm A(n) then,

$T(n) = c + 2T(n/2)$

Where 'c' is the constant time needed for executing the 'if' statement and for invoking function A(n/2), the time taken will be T(n/2).

There are various methods for calculating the time complexity of recursive algorithms. Some of the methods used for the calculation of time complexity are: Back substitution method, Recursive tree method and Master theorem.

Suppose we have the following algorithm –

```
A(n)
{
  if (n > 1)
  {
     return (A (n – 1));
  }
}
```

Suppose time taken for executing the *if* statement is 1, then the recurrence relation for the algorithm is written as –

$T(n) = 1 + T(n-1)$ ----- equation (1).

Now solving the equation (1) using Back substitution method, we get the following steps –

$T(n) = 1 + T(n- 1)$

$= 1 + 1 + T(n – 2)$

$= 2 + T(n – 2)$

$= 3 + T(n – 3)$

$= 4 + T(n – 4)$

.

.

.

= k + T (n – k) ----- equation 2.

.

.

.

This algorithm is going to stop when the value of n becomes equal to 1. So, we can rewrite the recurrence relation as –

T(n) = 1 + T (n – 1); n >1

= 1; n = 1

Now we have to determine at what point (n – k) becomes equal to 1.

That means, n – k = 1

k = n – 1

We can rewrite our equation 2 in the following way –

T (n) = k + T (n – k)

= n – 1 + T (n – (n – 1))

= n – 1 + T (1)

= n – 1 + 1 (since T (1) = 1)

= n

Therefore, we can also say that the time complexity of this algorithm is O(n).

The space complexity of any algorithm is defined as the total amount of memory spaces (program space and data space) needed by that algorithm to complete its task. The space complexity of any algorithm depends upon various parameters, such as, memory space needed for variables, referenced variables, pointers, recursion stack space, etc.

One of the important factors of performance analysis of any algorithm is space complexity. It is calculated in different ways for an iterative algorithm and for a recursive algorithm. When considering an algorithm, the whole program can be visualized into two sections: Fixed component and Variable component.

Let us take one example to calculate the space complexity.

Consider the following iterative algorithm,

```
Algorithm array_add (a, n)
{
   Sum: = 0. 0;
   for i: = 1 to n do
       Sum: = Sum + a [i];
    return Sum;
}
```

Here some of the variables come under fixed component and some comes under variable component part. Suppose one word of memory is needed to store the value of a given variable. In the above-mentioned example, we can see that we have 4 variables, a, i, n and Sum. To store the values of n, i, and Sum, a total of 3 words are needed and for holding the values of array a, the total amount of memory space is dependent upon the size of the given array. So, in this case, the total amount of memory space needed for array a is n words. Hence, we can say that the total amount of memory space needed for the algorithm array_add is at least 3 + n words.

Consider the following recursive algorithm,

```
Algorithm Rec_Sum(a, n)
{
if (n≤ 0) then return 0.0;
Else
return Rec_Sum(a, n-1) + a[n];
}
```

The invocation of the recursive function Rec_Sum depends upon the value of n. If n > 0, then n number of times recursive function will be called, otherwise the given algorithm will return 0. Let us assume one word of memory space is needed to store the return address. Each invocation of the function requires 3 memory words, one memory word each for holding the value of n, holding the return address and holding the value for the pointer to a [ ]. Here the height of the recursion stack is n+1 (n for the n number of times recursive call occurs, when n > 0 and 1 for the first invocation of the main algorithm by the operating system).

So, in general, we can say that at least 3(n+1) memory words are needed for any recursive algorithm.

Table - I: Performance Analysis

Type of Algorithm	*Time Complexity*	*Space Complexity*
Iterative Algorithm	more	less
Recursive Algorithm	less	more

III. RELATED WORKS

All the sorting algorithms are usually falling under any one of the following categories: Internal Sorting Algorithm and External Sorting Algorithm. Researches have already come up with the differences between these two categories. All these sorting algorithms can be implemented by a non-recursive program or a recursive program. When sorting algorithms are analyzed for their empirical performance, it has been found that performance of any algorithm changes as the size of the data set varies.

Same algorithm shows average performance for a particular data set and the same algorithm shows its worst performance when provided with another data set.

It was a quest from centuries, to develop the most efficient and the fastest algorithm to solve real world issues. Today, the amount of data is too large and we require such an efficient algorithm to solve the major problems that occur as and when in the world [3].

Algorithm is defined as a group of unambiguous instructions, which are arranged in some meaningful sequence. There are lots of approaches available in algorithm, any one of them can be used for solving a problem, depending upon the present constraints of a problem. Out of the available approaches, recursion is said to be an important problem solving and programming technique.

Here, considering algorithms for Fibonacci series and Factorial, implemented using both the available approaches: Iterative and Recursive. While implementing these algorithms, we can use any programming language. Since C is one of the simplest programming languages, I am using C for implementing Fibonacci series and Factorial.

Recursive functions may be divided into linear and branched ones [4].

Consider the situation of calculating Fibonacci through the recursive algorithm [4].

```
int fibo (n)
   if (n = = 0 || n = = 1)
      return n;
   else
      return fibo (n – 1) + fibo (n – 2);
```

Now, when the above algorithm executes for n = 5, then we will have the following tree:

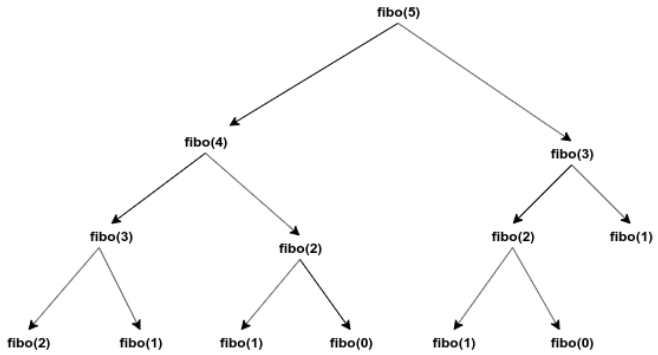

Fig. 1. Function Calls for fibo(5).

When we analyze the above tree, we get the following information,

Sl. No.	Function Name	Total number of times function invoked
1.	fibo (1)	4
2.	fibo (2)	3
3.	fibo (3)	2
4.	fibo (4)	1
5.	fibo (5)	1
6.	fibo (0)	2

So, here we noticed that for a small input n = 5, fibo (1) is calculated for 4 times, fibo (2) is calculated for 3 times, and so on. This number of additions even grows for larger number, if provided as input. Hence, we can say that duplication of function is the main cause for the reduced performance of this algorithm.

The iterative algorithm for Fibonacci series is considerably faster since it doesn't involve calculating the redundant things [4].

Let us take an algorithm, Factorial, which is written to find the factorial of a given number. As analyzed in [4], the iterative approach of Factorial is written as:

Iterative factorial (int num)

```
sum <- 0
for 1 to num do
    sum <- sum + i;
return sum;
end;
```

And the recursive approach of factorial is written as,

Recursive factorial (int num)

```
if num <= 0
   return 1;
else
   return n * factorial (num – 1);
end;
```

After implementation of factorial using both approaches for input size starting from 10 to 100000 [4], is shown below:

Table- II: Comparison of iterative and recursive approaches for factorial algorithm.

N	Recursive	Iterative
10	334 ticks	11 ticks
100	846 ticks	23 ticks
1000	3368 ticks	110 ticks
10000	9990 ticks	975 ticks
100000	Stack Overflow	9767 ticks

As found in [4], the poor performance for the recursive approach compared to an iterative approach is that, recursive algorithm contains heavy push-pop of the registers in the worst level of each function call.

IV. RESULTS

Recursion is good, elegant and powerful structure compared to iteration. The good thing about recursion is the way it simplifies the source code. For solving the same problem, iteration and recursion have functional equivalences, while they have different run – time behaviors. In today's world of data, all computational tasks are done using any of the two basic mechanisms: Iteration and Recursion [5]. As we know, Iteration is "repetition of a specified set of computer instructions in sequence, for a specified number of times` or until a condition is met" [6]. This mechanism of iteration is used when a situation demands a set of actions to be performed a certain number of times [7].

This classical approach of iteration is included in today's programming language, using cycles or loops.

On the other hand, recursion is "an another programming technique that mainly involves procedures, functions, subroutines, or algorithms, that is supposed to call itself a specified number of times or until a condition is met and the process of calling itself follows the principle of stack, that is, it follows the principle of Last-In-First- Out (LIFO)" [6].

V. CONCLUSION AND FUTURE SCOPE

In this paper, I tried to do performance analysis of algorithms, written in an iterative way as well as in recursive way. What I did is that I have taken some problems and analyzed them, that means, calculated the time complexity and space complexity for the problem, written in an iterative way as well as in the recursive way. Here I would like to conclude that the requirement of memory resources for both iteration and recursion are different. Iteration needs less memory resource but more processing resource, whereas recursion needs more memory resource and less processing resource. The solution to the question "which to use when?" depends upon the problem we are trying to solve and the programming language that we have chosen for solving it. I would like to conclude also that a novice programmer always prefers to use iterative mechanism than a recursive one. Studies also shows that they start to prefer recursion over iteration only when there are clear indications [8]. This is an area where future scope of research exists tremendously.

REFERENCES

1. Pooja K. Chhatwani, Jayashree S. Somani, N. P. Hirani,"*Comparative Analysis & Performance of Different Sorting Algorithm in Data Structure*", International Journal of Advanced Research in Computer Science and Software Engineering, Vol.3, Issue.11, pp.500-507, 2013.
2. Raji Ramakrishnan Nair, Divya Joseph, Alen Joseph, "*A Quick Reference to Data Structures and Compueter Algorithms – an Insight on the Beauty of Blockchain*", BPB Publications, India, pp. 5-15, 2019.
3. Ayush Pathak, Abhijeet Vajpayee, Deepak Agarwal, "*A Comparative Study of sorting Algorithm Based On their Time Complexity*", International Journal of Engineering Sciences and Research Technology, ISSN: 2277-9655.
4. Vatsal Shah, Jayna Dovya, "*Study of Recursive and Iterative Approach on Factorial and Fibonacci Algorithm*", International Journal of Advance Engineering and Research Development (IJAERD), Volume 1 Issue 1, Febraury 2014, ISSN: 2348-4470.
5. Gabbrielli, M.S. Martini Programming Languages: Principles and Paradigms. Spinger Science and Business Media, 2010.
6. Kamthane, A. Programming and Data Structures. Pearson Education, India, 2003.
7. Meriam-Webstar Dictionary.
   http://www.merriam-webster.com.
8. Vladimir Sulow, "*Iteration vs Recursion in Introduction to Programming Classes: An Empirical Study*", Cybernetics and Information Technologies, Volume 16, No. 4, Print ISSN: 1311-9702; Online ISSN: 1314-4081.

AUTHORS PROFILE

Ms. Raji Ramakrishnan Nair pursued Bachelor of Computer Applications from Mahatma Gandhi University, Kottayam, India, in 2000 and Master of Computer Applications from Mahatma Gandhi University, Kottayam, India, in year 2004. She is currently pursuing Ph.D. from Lovely Professional University and also working as Assistant Professor in P. G. Department of Computer Applications, Marian College Kuttikkanam (Autonomous), India since 2007. She has published a book with BPB Publications, India. Her research work focuses on Algorithms, Data Structures, and Blockchain. She has 15 years of teaching experience.