Study of Recursive and Iterative Approach on Factorial and Fibonacci Algorithm

Vatsal Shah¹, Jayna Donga²

¹Assist. Prof., IT Dept., BVM Engineering College, V.V.Nagar, vatsal.shah@bvmengineering.ac.in
²Assist. Prof., Computer Dept. MBICT, V.V.Nagar, jaynadonga@yahoo.com

Abstract: Algorithm is wide area for research. In part of algorithm is solved various real time problem like job scheduling, shortest path and Eight Queen etc. For finding a solution multiple approaches work on single problem. In this paper we work on that direction. There are two main basic algorithm can be implemented in different nature. In this paper we study factorial and Fibonacci algorithm solved by simple iterative approach and recursive approach. At last we compare number of parameter like their number of operation, memory utilization and Time complexity.

Key word: Iterative, Recursive, Time complexity

I. INTRODUCTION

Algorithm is collection of finite set of unambiguous instruction that occur in sequence. lots of approaches available in algorithm for solving a problem. We discuss basically two approaches are iterative and recursive. Recursion is an important problem solving and programming technique and there is no doubt that it should be covered in the first year introductory computer science courses, in the second year data structure course, and in the third year design and analysis of algorithms course. While the advantages of using recursion are well taught and discussed in textbooks, we discovered that its potential pitfalls are often neglected and never fully discussed in literature. For the purpose of our discussion, we shall divide recursive functions into linear and branched ones[1]. Linear recursive functions make only one recursive call to itself. Note that a function’s making only one recursive call to itself is not at all the same as having the recursive call made one place in the function, since this place might be inside a loop. It is also possible to have two places that issue a recursive call (such as both the then and else clauses of an if statement) where only one call can actually occur. The recursion tree of a linear recursive function has a very simple form of a chain, where each vertex has only one child. This child corresponds to the single recursive call that occurs. Such a simple tree is easy to comprehend, such as in the well known factorial function[3-4]. By reading the recursion tree from bottom to top, we immediately obtain the iterative program for the recursive one. Thus the transformation from linear recursion to iteration is easy, and will likely save both space and time. However, these savings are only in the constant of linear time complexity for both recursive and iterative solutions, and can be easily disregarded.

II. BACKGROUND THEORY

Iterative functions – are loop based imperative repetitions of a process (in contrast to recursion which has a more declarative approach). Iterative is part of Pseudo code and it is a language similar to a programming language used to represent algorithms. The main difference respect to
actual programming languages is that pseudo code is not required to follow strict syntactic rules, since it is intended to be just read by humans, not actually executed by a machine\textsuperscript{[5]}.

**Recursive function** – is a function that is partially defined by itself and consists of some simple case with a known answer. A *recursive procedure* is a procedure that invokes itself. Also a set of procedures is called *recursive* if they invoke themselves in a circle, e.g., procedure \( p1 \) invokes procedure \( p2 \), procedure \( p2 \) invokes procedure \( p3 \) and procedure \( p3 \) invokes procedure \( p1 \). A *recursive algorithm* is an algorithm that contains recursive procedures or recursive sets of procedures\textsuperscript{[7]}. Recursive algorithms have the advantage that often they are easy to design and are closer to natural mathematical definitions Example: Fibonacci number sequence, factorial function, quick sort and more. Some of the algorithms/functions can be represented in an iterative way and some may not.

The particular recursive algorithm for calculation Fibonacci series is less efficient. Consider the following situation of finding fib(4) through the recursive algorithm

\[
\text{int fib(n) :}
\]
\[
\text{if( n==0 || n==1 )}
\]
\[
\text{return n;}
\]
\[
\text{else}
\]
\[
\text{return fib(n-1) + fib(n-2)}
\]

Now when the above algorithm executes for \( n=4 \)

\[
\begin{align*}
fib(4) & \\
fib(3) & \quad fib(2) \\
fib(2) & \quad fib(1) \quad fib(1) \quad fib(0) \\
fib(1) & \quad fib(0)
\end{align*}
\]

It's a tree. It says that for calculating fib(4) you need to calculate fib(3) and fib(2) and so on.

Notice that even for a small value of 4, fib(2) is calculated twice and fib(1) is calculated thrice. This number of additions grows for large numbers.

There is a conjecture that the number of additions required for calculating fib\( (n) \) is

\[
\text{fib(n+1) -1}
\]

So this duplication is the one which is the cause of reduced performance in this particular algorithm.

The iterative algorithm for Fibonacci series is considerably faster since it does not involve calculating the redundant things.

It may not be the same case for all the algorithms though.

**III. IMPLEMENTATION AND RESULT ANALYSIS**
In implementation point of view, we can use any of language for implementation. C programming is simple and we use for implementation of Fibonacci series and factorial using both approaches.

**Factorial algorithm**

Iterative approach of factorial

```c
Iterative_factorial (int num)
sum <-0
for 1 to num do
    sum<- sum+ i;
return sum;
End;
```

Recursive approach of factorial

```c
Recursive_factorial (int num)
if num=0
    return 1;
else
    return n*factorial(num-1);
end;
```

After implementation of factorial using both approach for taking input size starting from 10 to 100000.

<table>
<thead>
<tr>
<th>N</th>
<th>Recursive</th>
<th>Iterative</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>334 ticks</td>
<td>11 ticks</td>
</tr>
<tr>
<td>100</td>
<td>846 ticks</td>
<td>23 ticks</td>
</tr>
<tr>
<td>1000</td>
<td>3368 ticks</td>
<td>110 ticks</td>
</tr>
<tr>
<td>10000</td>
<td>9990 ticks</td>
<td>975 ticks</td>
</tr>
<tr>
<td>100000</td>
<td>stack overflow</td>
<td>9767 ticks</td>
</tr>
</tbody>
</table>

*Table 2: comparison of Iterative and recursive for factorial*

The reason for the poor performance is heavy push-pop of the registers in the ill level of each recursive call.

**Fibonacci Algorithm:**

Iterative approach of Fibonacci
Recursive approach of Fibonacci

After implementation of Fibonacci using both approach for taking input size starting from 5 to 100000.

<table>
<thead>
<tr>
<th>N</th>
<th>Recursive</th>
<th>Recursive opt.</th>
<th>Iterative</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5 ticks</td>
<td>22 ticks</td>
<td>9 ticks</td>
</tr>
<tr>
<td>10</td>
<td>36 ticks</td>
<td>49 ticks</td>
<td>10 ticks</td>
</tr>
<tr>
<td>20</td>
<td>2315 ticks</td>
<td>61 ticks</td>
<td>10 ticks</td>
</tr>
<tr>
<td>30</td>
<td>180254 ticks</td>
<td>65 ticks</td>
<td>10 ticks</td>
</tr>
<tr>
<td>100</td>
<td>too long/stack overflow</td>
<td>158 ticks</td>
<td>11 ticks</td>
</tr>
<tr>
<td>1000</td>
<td>too long/stack overflow</td>
<td>1470 ticks</td>
<td>27 ticks</td>
</tr>
<tr>
<td>10000</td>
<td>too long/stack overflow</td>
<td>13873 ticks</td>
<td>190 ticks</td>
</tr>
<tr>
<td>100000</td>
<td>too long/stack overflow</td>
<td>too long/stack overflow</td>
<td>3952 ticks</td>
</tr>
</tbody>
</table>

Table 2: comparison of Iterative and recursive for fibonacci

As before, the recursive approach is worse than iterative however, we could apply memorization pattern (saving previous results in dictionary for quick key based access), although this pattern isn't a match for the iterative approach (but definitely an improvement over the simple recursion).

IV. CONCLUSION

In this paper we discuss both approaches on different algorithm so we conclude that the matrix method of generating Fibonacci numbers is more efficient than the simple iterative algorithm, though in order to see its benefits, you will probably have to work with numbers consisting of hundreds of bits or more. We can directly use previous find value in to next steps. For memory purpose iterative is more appropriate compare to recursive. For smaller numbers, the simplicity of the iterative algorithm is preferable.
REFERENCE