Time & Space complexity

Introduction

Algorithms are an essential part of data structures, which are used to implement data structures. Some algorithms outperform others in terms of efficiency. Therefore, we want the most efficient algorithms. We need some parameters through which we can judge which algorithm is efficient and which is not. There are many criteria through which we can evaluate and compare algorithms.

- **CPU(time)/Execution time**
- **Memory usage**
- **Disk usage**
- **Network usage**

All of them are important, but we are most concerned about the first two, which are execution time and memory usage.

To analyze execution time & memory usage of an algorithm, we have two complexities: **Time-complexity & Space-complexity**. Before moving on, let us first understand the term complexity. The complexity of an algorithm is a function that describes the algorithm's efficiency in terms of the given input data. This might sound similar to the performance at first, but there is a considerable difference between these two terms.

Performance:- It is the time, memory, disk, etc., needed or used to run the program. It is determined by the machine, compiler, etc., and also by the algorithm we write.

Complexity:- It is a function that describes how the amount of time, memory, disk space, etc., varies with the given input.

Note:- Performance depends upon complexity, but complexity does not depend on the performance.

Now comes the time-complexity and space-complexity. They depend on many factors like the **machine's hardware, compiler, operating system, and so on**. However, none of the parameters will be considered when we examine the complexity of the program. We'll solely look at the algorithm's **execution time & space**.

Time-Complexity

Time complexity is a function that describes how long an algorithm takes in terms of the quantity of data it receives. The term "time" can refer to the number of memory accesses, integer comparisons, the number of times an inner loop executes, or any other natural unit relating to the amount of time the algorithm will take in real-time. In simpler words, the time complexity of an algorithm is the amount of time it takes to run as a function of the length of the input.

Irrespective of the length of the code, the complexity of an algorithm depends on the functions and operations executed while running the program. Therefore, we can say that the complexity of an algorithm is proportional to the functions and the operations. Moreover, the complexity depends upon the number of basic operations. Some of them are

- **Arithmetic operation(+,-)**
- **Bitwise operations(and, or , xor)**
- **Assignment operation(=)**
- **Comparison operation(==, >,<)**
- **Input/output operations. (cin,cout)**

These are the operations now coming to the functions part. So, some functions perform the same number of operations every time they are called. Like, consider a function that returns the sum of the first two numbers of an array (if the size is greater than 2). Now irrespective of the size of the input array, it will always

perform a single operation of addition. Hence, the number of operations is independent of the size of the array.

Other functions may perform different numbers of operations depending on the value of arguments/parameters. Like, consider a function that returns the sum of all the elements of the array. Therefore, in this case, the number of operations will increase if we increase the size of the array.

When we consider the complexity, we don't care about the exact number of operations that are performed. Instead, we care about the relation between the number of operations and the size of the input. If we double the input size, does the number of operations stay the same, get doubled, or increase by some other factor?

Notations

Like to express or analyze any mathematical functions we need notations. Similarly, to express the complexity (which is also a function) we also need some notations.

For this purpose, we use asymptotic notations.

Generally, there are 3 cases:-

- **Worst-case:-** Maximum number of operations required for program execution.
- **Average-case:-** Average number of operations required for program execution.
- **Best-case:-** Minimum number of operations required for program execution.

To encounter each of these cases we have different notations.

- **O-Notation.**
- **Ω-Notation.**
- **θ-Notation.**

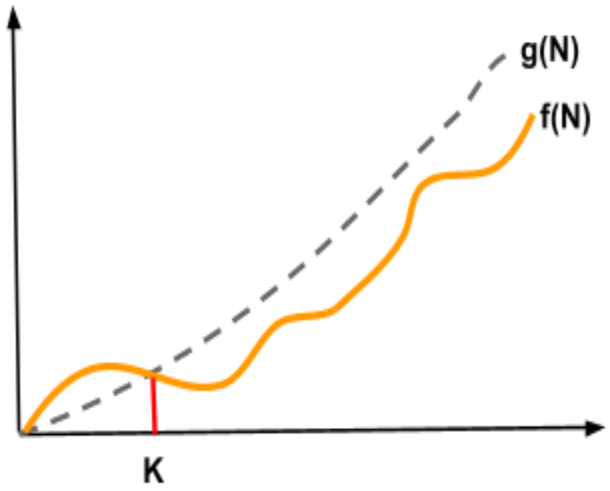
Big oh notation, O

The formal way to express the upper bound of an algorithm's running time is with

O(n). It calculates the worst-case time complexity, or the longest time an algorithm can take to complete.

Example, For a function ***f(n)*** :-

```
O(f(n)) = {g(n) : there exists c > 0 and n0 such that f(n) ≤ c.g(n)
for all n > n0.}
```

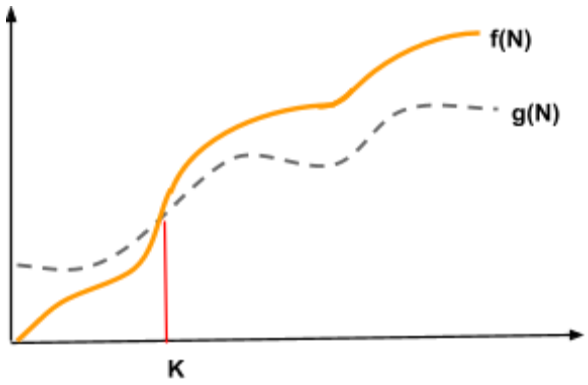


Omega-notation, Ω

The formal way to express the lower bound of an algorithm's running time is by notation **Ω(n)**. It calculates the best-case time complexity, or the shortest amount of time an algorithm can take to complete.

Example, For a function ***f(n):-***

```
Ω(f(n)) ≥ {g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n)
for all n > n0.}
```

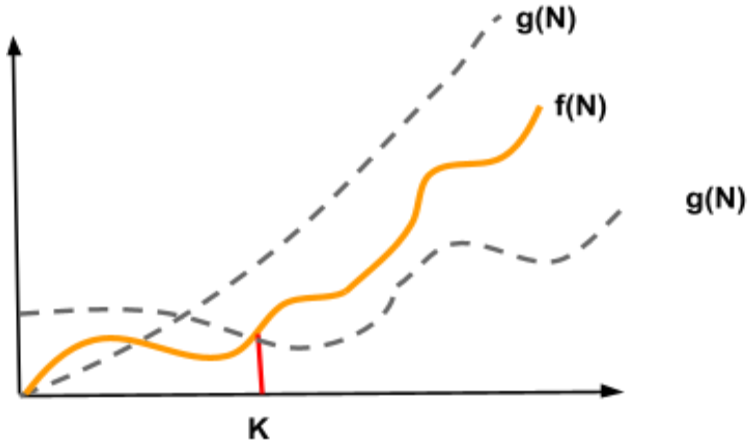


Theta-notation, θ

The notation ***θ*(n)** is used to express both the lower and upper bounds of an algorithm's running time. It calculates the average time complexity or the average amount of time an algorithm can take to complete.

Example,For a function ***f(n):-***

```
θ(f(n)) = { g(n) if and only if g(n) =  O(f(n)) and g(n) = Ω(f(n)) for all
n > n0. }
```



Moreover, we are interested in the worst case. What is the most number of operations that might be performed for a given problem size? For example:- A function that removes an element from the array. Basically, these functions take an index as an argument and remove the element present on that index and then shift

the elements towards the left by one unit present on the right side of the index. So, the worst case for this example is when we have to remove the first number because when we remove the first number, the function has to shift all the numbers present in the array (except which we remove). In contrast, the best case is when we remove the last element of the array because, in that case, the function doesn't shift any of the numbers. It will just erase the last number. Therefore, in the worst case, the time for removal is proportional to the number of items in the list, and we say that the worst-case time for removal is linear to the number of items in the array. For a linear-time function/method, the number of operations also doubles if the problem size doubles. Hence, we will consider the Big-Oh notation for further analysis.

How to determine Complexities

Sequential Statement:

If we have statements that perform basic operations(mentioned above) such as assignments, comparisons, arithmetic, printing, and variable reading, we can assume they each take the constant-time ***O(1)*.**

```
statement1;
statement2;
...
statementN;
```

The total time complexity of these statements is:-

```
Total= time(statement1)+time(statement2)+...time(statementN)
```

Let's consider ***T(n)*** to be a function denoting total time for an input size of ***n*** and **t** to be the time taken by a statement. Therefore, we can write the above conclusion as follows.

```
T(n) = t(statement1)+ t(statement2)+...t(statementN)
```

If each statement performs a basic operation, we can say it takes a constant time ***O(1)***. Even if we have 1 or 100 of these statements, it will be a constant time as long as you have a fixed number of operations.

Conditional statement

Mostly in every program which we will write we are going to use conditional statements. So this becomes very important to understand the complexity of these statements. By keeping in mind that we care about the **worst-case** scenario with **Big-Oh** notation.

```
if (true) {
 statement1;
 statement2;
} else {
 statement3;
}
```

As we only care about the worst-case, that's why we will only take whichever is more time consuming.

```
 T(n) = max(t(statement1)+ t(statement2),t(statement3))
```

Loop statements

While calculating the time-complexity of a loop we first find out the runtime of statements inside the loop and then multiply it by the number of time, loop will run.

```
for(int i = 0; i < n; i++) {
 statement1;
 statement2;
}
```

In the above example the loop is running ***n*** times and executing the two statements inside it. Therefore the time complexity is:-

```
T(n) = n*[t(statement1)+ t(statement2)]
```

Loops that grow proportionally to the input size have a linear time complexity (***O(n)).***

Note:- If we halve the input **(n --> n/2)** (or divided by some other constant) or if we run the loop **twice** (or any constant time) in a ladder manner the complexity of the algorithm will remain linear and can be expressed as ***O(n).***

Conclusion:- **O(c*n) can be written as O(n),** where **n** is the size of the input and **c** can be any constant (can also be a fraction).

Logarithmic time loops

There are some algorithms that run in logarithmic time. One of the most common examples is **binary search**. We encounter these types of complexities whenever, after every iteration of the loop, the effective size of the array becomes smaller by a constant factor.

For example:-

```
int l=0, r=n;
while(l+1<r){
int mid=(l+r)/2;
if(arr[mid]<=arr[l])l=mid;
else r=mid;
}
```

In the above example after every iteration we are dividing the array into two halves using a variable, ***mid.*** Therefore, the while loop will run as many times as we can divide the length of the array (which is equal to ***n***) in half. This can be represented as a $\boldsymbol{log_2(n).}$ For example:- **n=8** the loop will run n times **(n/2,n/4,n/8)**.which is equal to $\boldsymbol{log_2(8)=3.}$ Therefore, the time complexity of these types of loops is **O(log n)**(base can be constant 2, 3, 4 depending on the constant by which we are reducing the size).

Nested loop

Sometimes, we have to run a loop several times and for doing it we use an outer loop which runs the inner loop multiple times. This formation of loops is known as a nested loop. Let us take an example for a better understanding.

```
for(int i=0;i < n; i++) {
 statement1;
 for(int j=0; j < m; j++){
  statement2;
```

```
  }
}
```

The time-complexity of the above nested loop can be represented as:-

```
T(n) = n*[t(statement1) + m*t(statement2)]
```

Let's assume that the statements are running in constant-time ***O(1).***
Then we will have a complexity of ***O(n*m).*** If instead of running the inner loop ***m*** times, we run it ***n*** then the complexity will become ***O(n^2).***

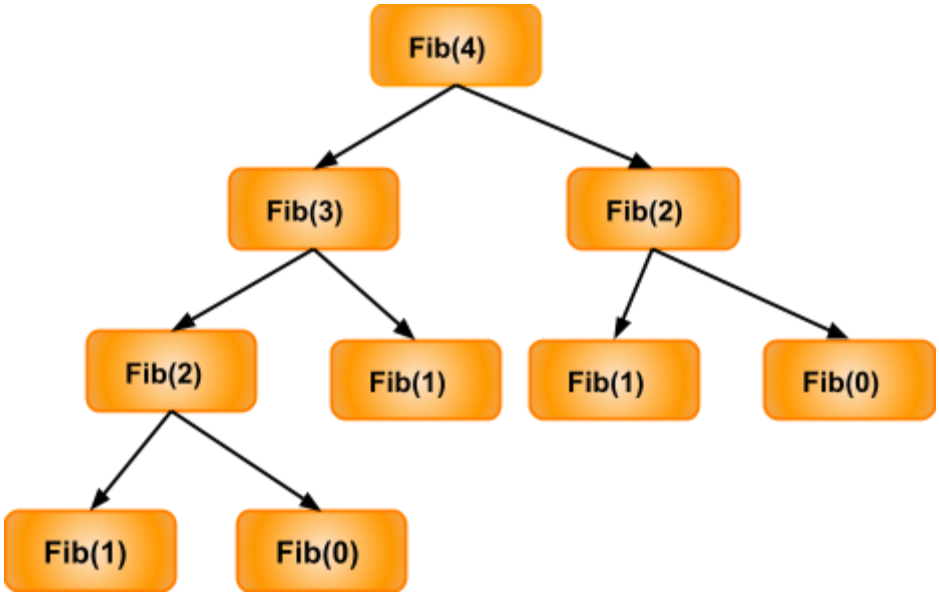
Recursive functions statements

Finding out the complexity of the recursive function is a bit tougher than others. It is a bit tricky too. There are several ways to do it. One of them is by exploring a recursive tree which is the most intuitive and interesting way.

Example:-

```
function rec(n) {
 if (n < 0) return 0;
 if (n < 2) return n;
 return rec(n - 1) + rec(n - 2);
```

Let's take some examples to understand the recursive tree.
Recursive tree for **n=4**.

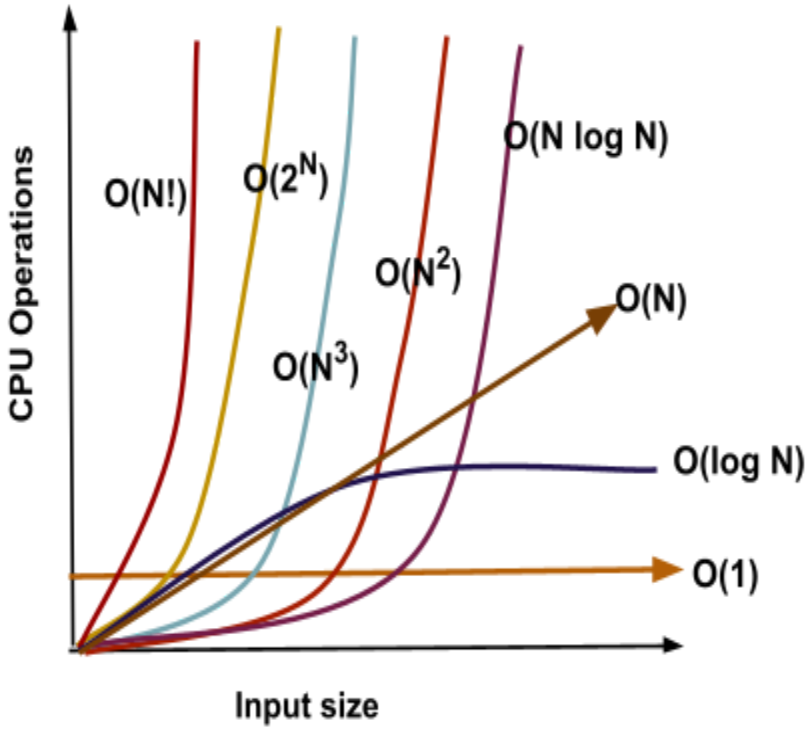


As we can see that for **n=4,** the number of calls is equal to **8.** Similarly, we can see that for **n=3** the number of calls is equal to **4** and for **n=2** is **2.** Since it's a binary tree, we can sense that every time ***n*** increases by one, we would have to perform at most the double of operations.
The total number of calls in a complete binary tree is exponential, hence the time complexity of these types of recursive functions is in the order of **$O(2^n)$**.

Table most common Big Oh notations.

Constant	**O(1)**
Logarithmic	**$O(\log_x n)$ (x :-2 ,3 4...)**
Linear	**O(n)**
Quadratic	**$O(n^2)$**
Cubic	**$O(n^3)$**
Polynomial	**$O(n^x)$ (x = 2, 3, 4...)**
Exponential	**$O(2^n)$**
Factorial	**O(n!)**

Following is the graph of variation of various most common Big Oh notation examples.



Constraints Analysis

While solving problems, we don't know about the complexity of the algorithm which we have to implement to solve the problem. Because the order of complexity won't be mentioned in the problem statement we have to figure it out on our own. The things which will be included in the problem statement are **Constraints** (range of the input data) and the **time limit** (mostly in seconds). Using these two things we have to figure out the order of time complexity which will be accepted by the online judge. Otherwise, we can get the **Time limit exceeded (TLE).**

Things to remember:-

- In one second the program can run upto 10^8 basic operations/statements.(this may vary 10^7-10^9 depending on the online judge)
- While writing algorithms we should always consider the **worst-case** scenario for calculation time-complexity.
- Always check the constraints of the problem before starting writing the algorithm and then design the algorithm accordingly.

For example, if the time limit is **2 seconds.** And the constraints of the input data ***n*** is upto 10^6**.** Then we have to design an algorithm such that in the **worst-case** the total number of operations performed by the algorithm is less than $2*10^8$ (Considering online judges can run 10^8 operations per second). Therefore the worst acceptable complexity is somewhere between **linear O(n)** and **quadratic O(n^2)**. To be more precise **O(2*10^2 *n).**

The table below will help you understand the growth of several common time complexities, allowing you to determine whether your algorithm is fast enough to receive an Accepted.

Length of the input (N)	Complexity of worst accepted algorithm
<= [10...11]	O(N!)
<= [29...31]	O(2^N)
<= 100	O(N^4)
<= 400	O(N^3)
<= 2x10^3	O(N^2 x $\log_2$(N))
<= 10^4	O(N^2)
<= 10^6	O(N x $\log_2$(N))
<=10^8	O(N)

Space Complexity

Space complexity is a function that describes how much memory (space) an algorithm requires in relation to the amount of input to the algorithm.

We frequently discuss the need for additional memory, which does not include the memory required to store the input itself. To measure this, we once again use natural (but fixed-length) units.

We can use bytes, but it's easier to use other metrics, such as the number of integers used, the number of fixed-sized structures, and so on.

Finally, the function we devise will be independent of the number of bytes required to represent the unit.

Space complexity is sometimes overlooked because the space used is minimal and/or noticeable, but it can be just as crucial as time complexity.

Notations of the space complexity are the same as the time complexity, and here also, we only care about the worst-case (Big-oh) as in time complexity.

How to determine the space-complexity

The methods of finding the space complexity of an algorithm are pretty similar to the time complexity. Let’s understand a few of them by taking examples.

Variables

Variables are things that we are going to use in every single problem. So it is essential to understand their complexity. There are many types of variables and the size of them depends on their data type, some of them are listed below:

- Character type **(char) = 1 byte**
- Integer type **(int) = 4 bytes**
- Float type **(float) = 4 bytes**

- Large integer **(long long int) = 8 bytes**

It means if we define a variable of an integer type, it will take 4 bytes of memory. But while considering the complexity, we don't have to care about it much; what we care about is the proportionality of the space with respect to the input data. Similar to time complexity.

```
int a,b;
char c;
```

Let's consider ***S(n)*** to be a function denoting total space for an input size of ***n*** and **s** to be the space taken by a data type variable. Therefore, we can write the above conclusion as follows.

```
S(n) =  s(a) + s(b) + s(c)
```

The total space required in the above example is **9** (4*2+1*1)Bytes. As we can see, this space won't be affected by the input data. Therefore, we can say that it is a constant space. Hence, we can represent it as **O(1).**

Arrays

In a problem where we need many variables of the same data type, we often declare an array or a vector instead of declaring multiple variables. So, let's understand the space complexity of the arrays.

```
int arr1[n],arr2[m];
char arr3[p];
```

Let's consider ***S(n)*** to be a function denoting total Space for an input size of ***n*** and **s** to be the size of a single block in an array of a data type. Therefore, we can write the above conclusion as follows.

```
S(n) = n*s(arr1) + m*s(arr2) + p*s(arr3)
```

The total space required for the above example is **(n*4 + m*4+ p*1) Bytes.** We can see that this space depends on the input data, and to be more precise, it depends on the data linearly. Therefore the space complexity of the above example can be represented as **O(n) (O(n+m+p)).**

2-D Array, Table, Matrix

In some of the problems, we must store the data in the form of a table or a 2-D array (D = dimension). We can determine the space complexity of the table as follows.

```
int arr1[n][m]
```

Let's consider ***S(n)*** to be a function denoting total space for an input size of ***n*** and **s** to be the size of a single block in an array of a data type. Therefore, we can write the above conclusion as follows.

```
S(n) = n*s(m*s(arr1))
```

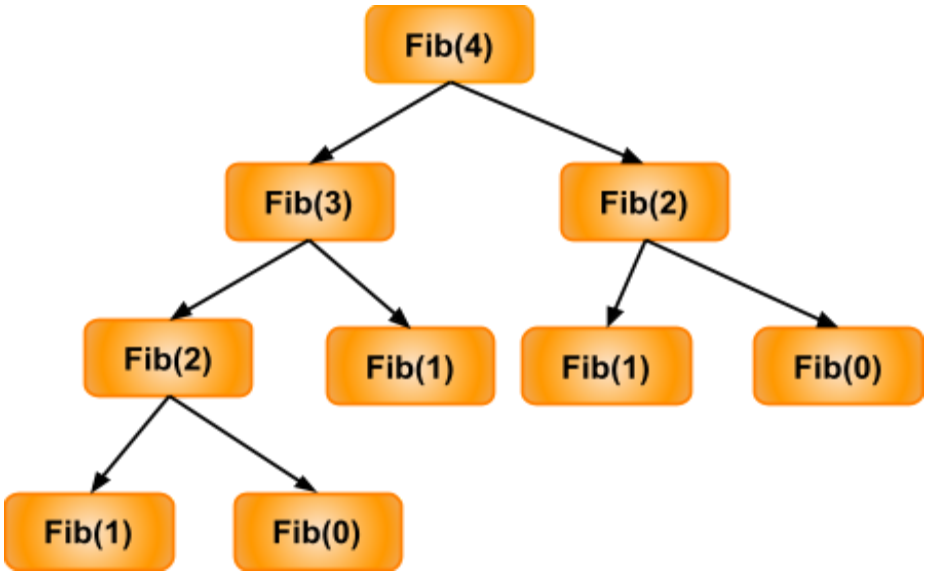
The total space required for the above example is **(n*(m*4)) Bytes.** We can see that this space depends on the input data, and to be more precise, it depends on the data quadratically; therefore, the space complexity of the above example can be represented as **O(n^2) (O(n*m)).**

Recursive function

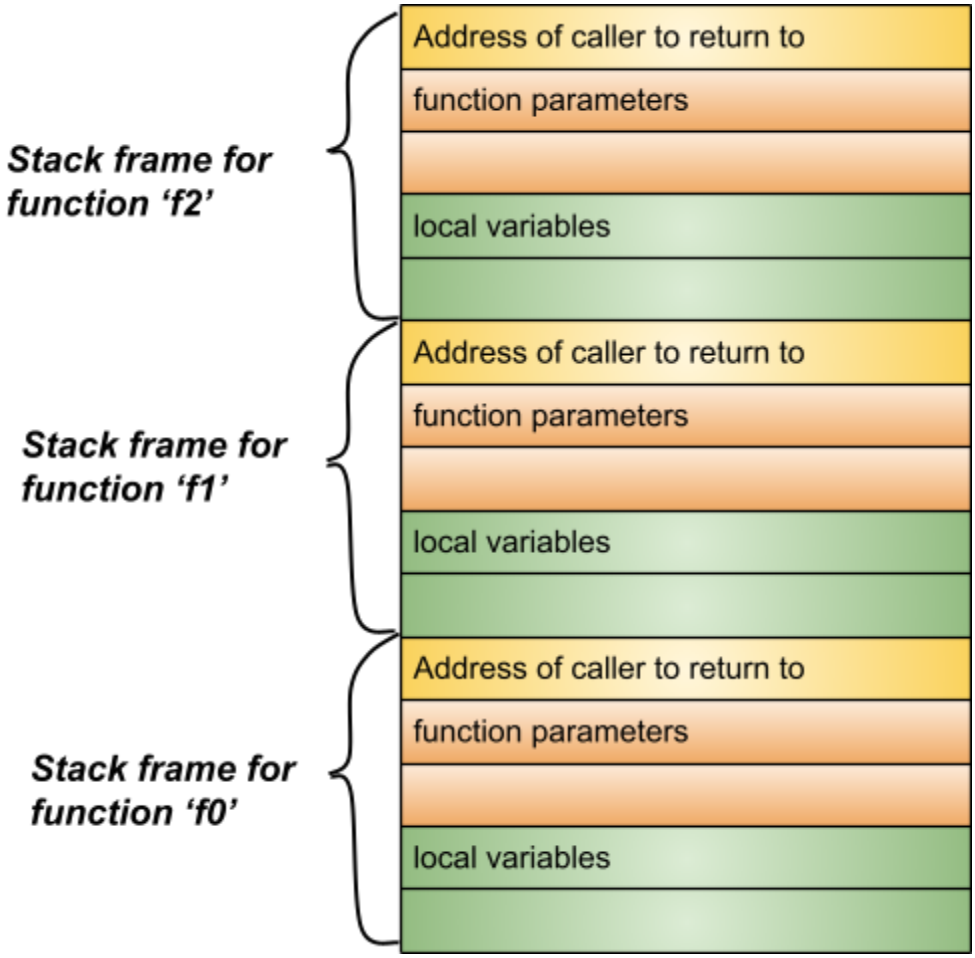
Understanding the space complexity of a recursive function is a bit harder than others. As we all know, whenever a function is called, it uses stack memory. Therefore, to understand the space complexity of recursive functions, We need to know how the stack frames are generated in memory for recursive call sequences. Let's take the same example which we have taken for the time complexity analysis.

```
function rec(n) {
 if (n < 0) return 0;
 if (n < 2) return n;
 return rec(n - 1) + rec(n - 2);
```

The recursive tree of the above code is:-



To calculate the time complexity of the following example, we must first understand when the stack frames are generated and for how long they are kept in the memory. When a function **'f1'** is called from function **'f0'**, the stack frame of function **'f1'** is created.The stack frame kept in the memory until the call corresponds to **'f1'** is not terminated. This stack frame is responsible for saving all the arguments, local variables or arrays present in the function **'f1'**. If the function **'f1'** calls a function **'f2'** further, then the stack frame of the function **'f2'** is also generated and kept in the memory until the call to **'f2'** is terminated. Following is the stack frame of the above-explained example:-



Call stack in memory for the call f0->f1->f2

Now, when the call to function **'f2'** returns, the stack frame corresponding to **'f2'** is deleted from the memory. Same will happen with the stack frame of function **'f1'** and **'f0'.**

Using the above analogy for recursive call sequence, we can say that the memory used by the algorithm depends upon the maximum stack frames present in the memory at any point of time equal to the maximum depth of the recursive tree. In the recursive tree, when the recursive calls hit the leaf node, they start returning the value, and hence the stack starts getting empty. In the above recursive tree when left and bottom most state **fibo(1)** is getting executed, the call sequence which led to this state would be **fibo(4)->fibo(3)->fibo(2)->fibo(1)** and all the stack would be present in the memory before **fibo(1)** returned its stack frame. Hence, the maximum stack frames present in

the memory at any point are equal to **4,** which is nothing but the depth of the recursive tree.

To conclude, the space complexity of the recursive algorithm is proportional to the maximum depth of the recursion tree. If each call of the recursive algorithm takes **O(m)** space and if the maximum depth of the tree is ***n*** then space complexity of the algorithm would be O(n*m).