Why should you learn about the time complexity of algorithms?

In computer programming, **an algorithm is a finite set of well-defined instructions** that are usually performed in a computer to solve a class of problems or perform a common operation.

According to the description, a sequence of specified instructions must be provided to the machine for it to execute an algorithm or perform a specific task. A particular series of instructions may be interpreted in any number of ways to perform the same purpose.

We have mentioned that the algorithm is to be run on a computer, which leads to the next step of varying the operating system, processor, hardware, and other factors that can affect how an algorithm is run.

Now that you know that a variety of variables will affect the outcome of an algorithm, it's important to know how effective those algorithms are at completing tasks. **To determine this, you must assess an algorithm's Space and Time complexity.**

An algorithm's space complexity quantifies how much space or memory it takes to run as a function of the length of the input while an algorithm's time complexity measures how long it takes an algorithm to run as a function of the length of the input.

That's why it's so important to learn about the time and space complexity analysis of various algorithms.

Let's explore each time complexity type with an example.

1. O(1)

Where an algorithm's execution time is not based on the input size n, it is said to have constant time complexity with order O (1).

Whatever be the input size **n**, the runtime doesn’t change. Here's an example:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main(){
    cout<<"Hello World!!";
}
```

As you can see, the message **"Hello World!!"** is printed only once. So, regardless of the operating system or computer configuration you are using, the time complexity is constant: O(1), i.e. any time a constant amount of time is required to execute code.

2. O(n)

When the running time of an algorithm increases linearly with the length of the input, it is assumed to have linear time complexity, i.e. when a function checks all of the values in an input data set (or needs to iterate once through every value in the input), it is said to have a Time complexity of order O (n). Consider the following scenario:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n;
    cin>>n;
    for(int i=0;i<n;i++){
        cout<<"Hello fellow Developer!!\n";
    }
}
```

Output:

```
5
Hello fellow Dev!!
Hello fellow Dev!!
Hello fellow Dev!!
Hello fellow Dev!!
Hello fellow Dev!!
```

Based on the above code, you can see that the number of times the statement **"Hello fellow Developer!!"** is displayed on the console is directly dependent on the size of the input variable 'n'.

If one unit of time is used to represent run time, the above program can be run in n times the amount of time. As a result, the program scales linearly with the size of the input, and it has an order of O(n).

3. O(n^2)

When the running time of an algorithm increases non-linearly O(n^2) with the length of the input, it is said to have a non-linear time complexity.

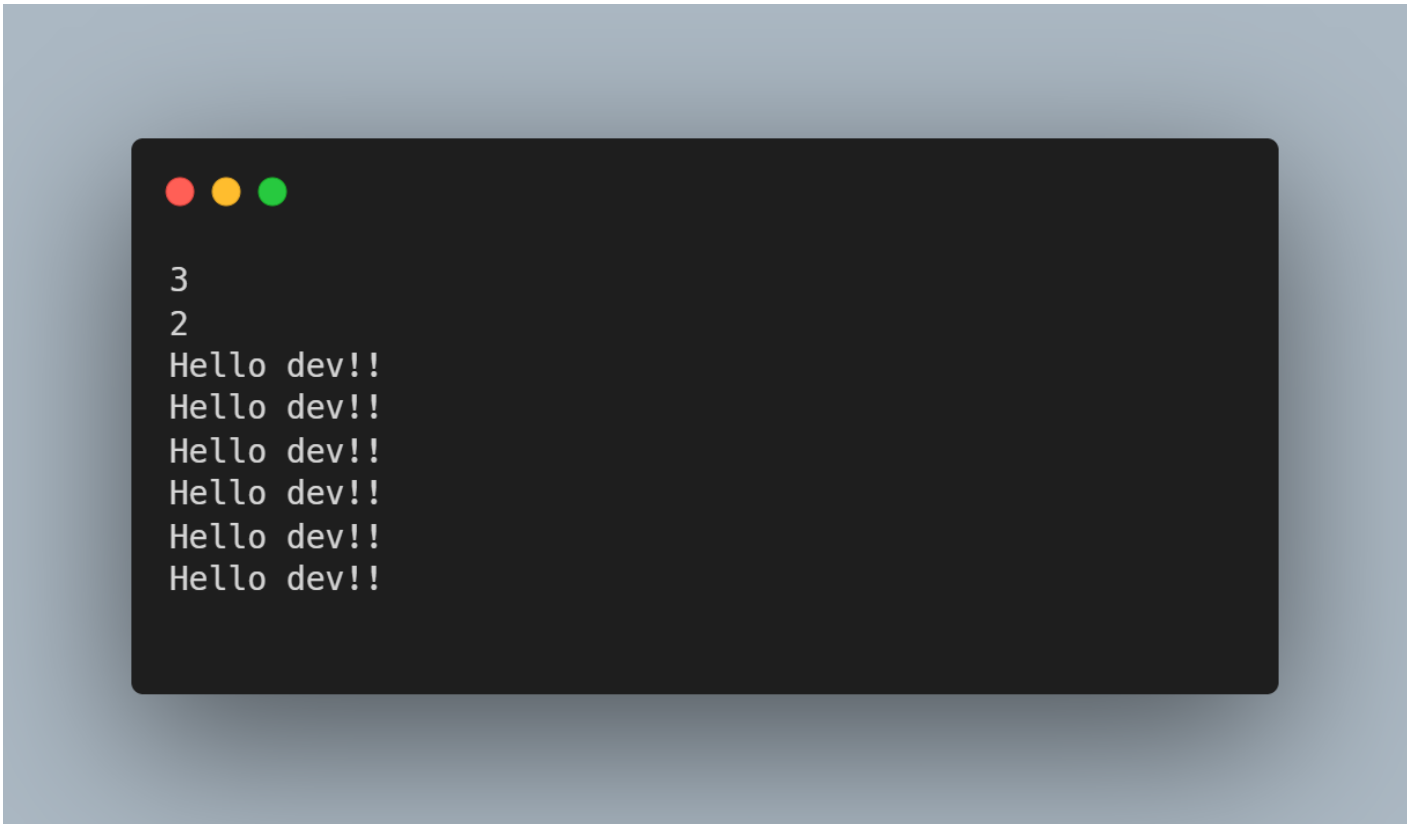
In general, nested loops fall into the O(n)*O(n) = O(n^2) time complexity order, where one loop takes O(n) and if the function includes loops inside loops, it takes O(n)*O(n) = O(n^2).

Similarly, if the function has ‘m' loops inside the O(n) loop, the order is given by O (n*m), which is referred to as polynomial time complexity function.

Consider the following program:

```
#include <bits/stdc++.h>
using namespace std;
int main( ){
    int n,m;
    cin>>n>>m;
    for(int i=0;i<n;i++){
          for(int j=0;j<m;j++) cout<<"Hello dev!!";
    }
}
```

Output:



As you can see, there are two nested for loops such that the inner loop's complete iteration repeats based on the value of the outer loop. This is the primary reason why you see **"Hello dev!!" printed 6 times (3*2)**.

This amounts to the cumulative time complexity of **O(m*n)** or **O(n^2)** if you assume that the value of m is equal to the value of n.

4. O(log2 n)

When an algorithm decreases the magnitude of the input data in each step, it is said to have a logarithmic time complexity. This means that the number of operations is not proportionate to the size of the input.

O (log2 n) basically implies that time increases linearly while the value of **'n'** increases exponentially. So, if computing **10 elements take 1 second**, computing **100 elements takes 2 seconds**, **1000 elements take 3 seconds**, and so on.

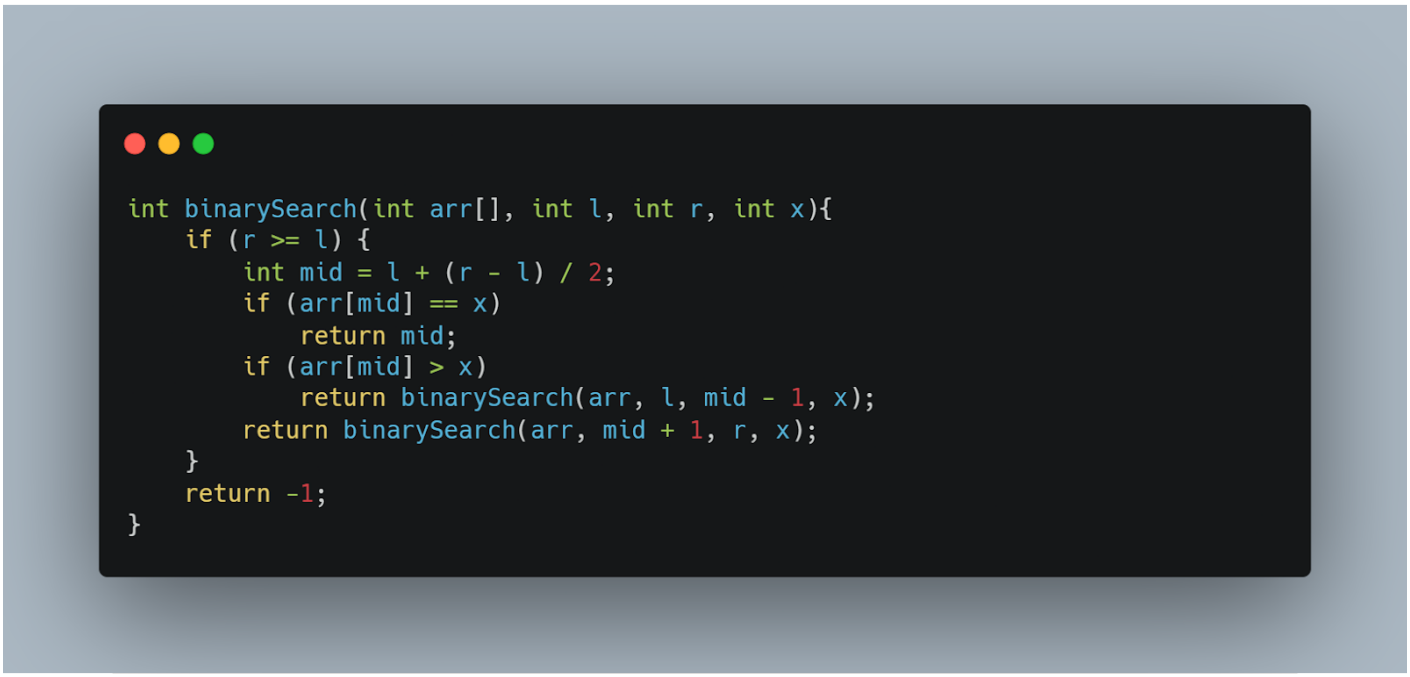
When using divide and conquer algorithms, such as binary search, the time complexity is O(log n).

Another example is quicksort, in which we partition the array into two sections and find a pivot element in O(n) time each time. As a result, it is O(log2 n)

Binary trees and binary search functions are examples of algorithms having logarithmic time complexity. Here, the search for a particular value in an array is done by separating

the array into two parts and starting the search in one of them. This guarantees that the action isn't performed on every data element.

Consider the following code snippet:



```
int binarySearch(int arr[], int l, int r, int x){
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}
```

The above program is a demonstration of the binary search technique, a famous divide-and-conquer approach to searching in logarithmic time complexity.

Time Complexity Analysis

Let us assume that we have an array of length **32**. We'll be applying **Binary Search** to search for a random element in it. At each iteration, the array is halved.

- Iteration 0:
  - Length of array = 32
- Iteration 1:
  - Length of array = 32/2 = 16
- Iteration 2:
  - Length of array = 32/2^2 = 8
- Iteration 3:
  - Length of array = 32/2^3 = 4
- Iteration 4:
  - Length of array = 32/2^4 = 2
- Iteration 5:
  - Length of array = 32/2^5 = 1

Another example would be that for an array of size **1024**, only **10** iterations are needed to approach unity. For an array size of **32768**, we'll need only 15 iterations. Thus we can see that the number of operations grows at a very small rate compared to the size of the input array while complexity is logarithmic.

To generalize, after k iterations, our array size approaches 1.

Hence, n/2^k = 1 => n = 2^k

Applying logarithmic function on both sides, we get
=> log2 (n) = log2 (2^k) => log2 (n) = k log2 (2)
or, => k = log2 (n)
Hence, the time complexity of Binary Search becomes ***log2(n), or O(log n)***

5. O (n log n)

This time complexity is popularly known as linearithmic time complexity. It performs slightly slower as compared to linear time complexity but is still significantly better than the quadratic algorithm.

Consider the program snippet given below:

```
void merge(int arr[], int l, int m, int r){
    int n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    int i = 0, j=0, k=l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i]; i++;
        }
        else {
            arr[k] = R[j]; j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i]; i++; k++;
    }
    while (j < n2) {
        arr[k] = R[j]; j++; k++;
    }
}

void mergeSort(int arr[],int l,int r){
    if(l>=r) return;
    int m =l+(r-l)/2;
    mergeSort(arr,l,m);
    mergeSort(arr,m+1,r);
    merge(arr,l,m,r);
}
```

The program above represents the **merge sort algorithm**.

Sorting arrays on separate computers take a significant time. Merge Sort algorithm is recursive and has a recurrence relation for time complexity as follows:

```
T(n) = 2T(n/2) + θ(n)
```

The Recurrence Tree approach or the Master approach can be used to solve the aforementioned recurrence relation.

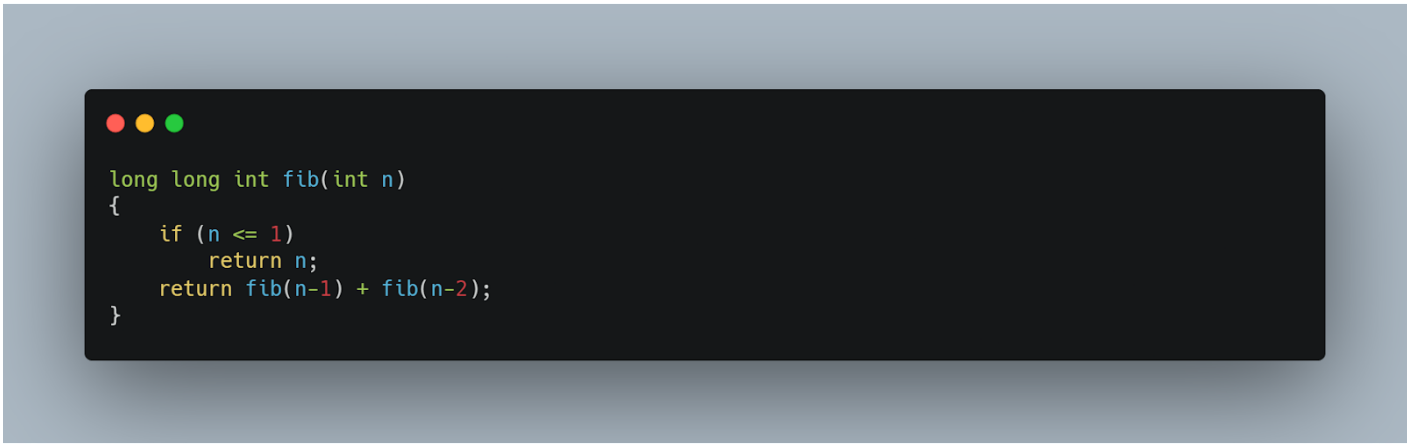
It belongs to Master Method Case II, and the recurrence answer is **O(n*logn)**. Since Merge Sort always partitions the array into two halves and merges the two halves in linear time, it has a time complexity of (n*logn) in all three circumstances (worst, average, and best).

Also, remember this, when it comes to sorting algorithms, **O(n*logn)** is probably the best time complexity we can achieve.

6. O(2^N)

An algorithm with exponential time complexity doubles in magnitude with each increment to the input data set. If you're familiar with other exponential growth patterns, this one works similarly. The time complexity begins with a modest level of difficulty and gradually increases till the end.

The **Fibonacci series** is a great way to demonstrate exponential time complexity. Given below is a code snippet that calculates and returns the nth Fibonacci number:



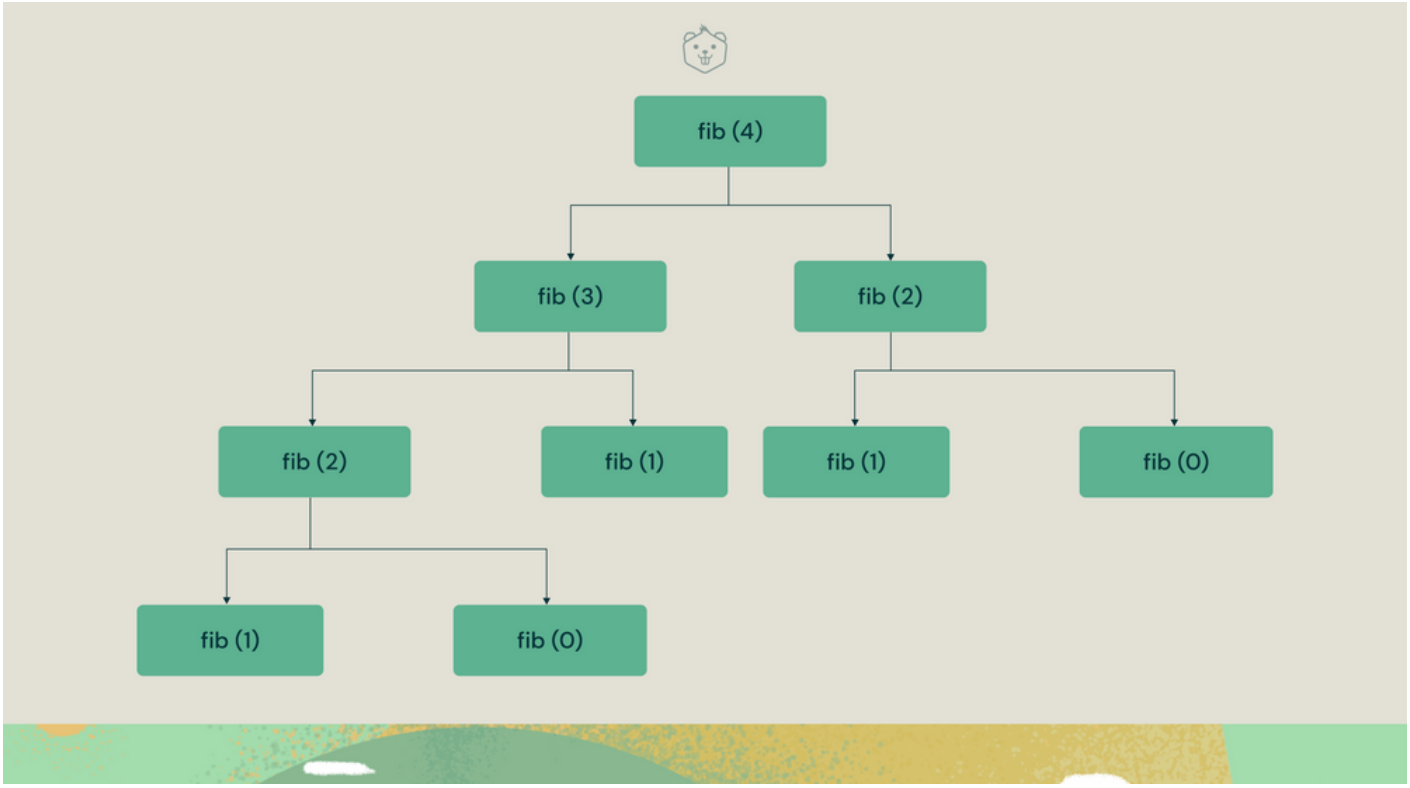
```
long long int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Time Complexity Analysis:

The recurrence relation for the above code snippet is:

```
T(n) = T(n-1) + T(n-2)
```

Using the recurrence tree method, you can easily deduce that this code does a lot of redundant calculations as shown below.



Thus the time complexity of the Fibonacci series becomes exponential owing to these repetitive calculations.

Let's take an example here to drive home the magnitude of this time complexity.

For n=3, it takes **8** operations to calculate the 8th Fibonacci number.

For n=4, it takes 16 operations

.

.

For n=10, it takes **1024** operations to reach the 10th Fibonacci number.

.

.

Now if we bump it up to **100 (n=100)**, we literally exceed **billions and billions of calculations** just to reach the 100th Fibonacci number.

That's scary right. That's why exponential time complexity is one of the worst time complexities out there.

Now that you've explored the major time complexities in algorithms and data structures, take a look at their graphical representation below: (Time vs Input Size)

Here's a simple comparison between different time complexities to help you understand the graph better:

```
O(1) < O(log2n) < O(n1/2) < O(n) < O(n logn) < O(n^2) < O(n^3) < . . . . < O(2^n) < O(3^
```

Of course, a visual representation clarifies a lot of concepts particularly related to the efficiency of different time complexities.

The noticeable thing to observe is how inefficient exponential and quadratic time complexities are with increasing input data size.

On the other hand, linearithmic and linear time complexities perform almost in a similar fashion with increasing input size.

The astounding part is the performance of logarithmic time complexity which clearly supersedes most algorithmic time complexities (except O(1)) in terms of execution time and efficiency.