**Time complexity**

The Time complexity can be defined as the amount of time taken by an algorithm to execute each statement of code of an algorithm till its completion with respect to the function of the length of the input.

**What is Time complexity?**

Diagram

Description automatically generated

Space and Time define any physical object in the Universe. Similarly, Space and Time complexity can define the effectiveness of an algorithm. While we know there is more than one way to solve the problem in programming, knowing how the algorithm works efficiently can add value to the way we do programming. To find the effectiveness of the program/algorithm, knowing how to evaluate them using Space and Time complexity can make the program behave in required optimal conditions, and by doing so, it makes us efficient programmers.

While we reserve the space to understand Space complexity for the future, let us focus on Time complexity in this post. Time is Money! In this post, you will discover a gentle introduction to the Time complexity of an algorithm, and how to evaluate a program based on Time complexity.

The Time Complexity can also be elaborated by estimating the counts of the number of elementary steps that are performed by any algorithm to finish the execution. As we saw in above example where we shared the two approaches to find a square of a number, then with the first approach the time taken will remain constant while for the second it varies with respect to the **'n'** that runs on the loop to give an output.

**Why Time complexity is required?**

It can clearly distinguish between two different algorithms based on their efficiency.

It’s independent of the machine on which the algorithm is run.

We can get a direct correlation with the length of the input.

It’s important to note here that time complexity doesn’t really measure the actual time taken by an algorithm to run (Since that kind of depends on the programming language, processing power etc.). It calculates the execution time of an algorithm in terms of the algorithms and the inputs.

**Procedure for analyzing complexity**

**1.Recurssive Method**

The [Recursion Tree Method](https://www.geeksforgeeks.org/analysis-algorithm-set-4-master-method-solving-recurrences/) is a way of solving[recurrence relations](https://www.geeksforgeeks.org/different-types-recurrence-relations-solutions/). In this method, a recurrence relation is converted into recursive trees. Each node represents the cost incurred at various levels of recursion. To find the total cost, costs of all levels are summed up.

Steps to solve recurrence relation using recursion tree method:

1. Draw a recursive [tree](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/) for given recurrence relation
2. Calculate the cost at each level and count the total no of levels in the recursion tree.
3. Count the total number of nodes in the last level and calculate the cost of the last level
4. Sum up the cost of all the levels in the recursive tree

**Asymptotic Notation**

The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don’t depend on machine-specific constants and don’t require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.

Asymptotic notations are mathematical notations which are used to describe the running time of an algorithm when input tends towards infinity.

It’s important to note here that O and are not functions. For example, O(n) represents the class of all functions that grow at most as quickly as the linear function f(n)=n.

Big-O notations give us a convenient way to talk about upper bounds. For example, we can say the time complexity of the algorithm is O(n^3) (i.e., T(n) O(n^3)), which means that the running time of the algorithm is at most cubic.

Another point to note here is that running time and time complexity are two different things, for example, if the running time of an algorithm is the following

T(n)= 3\*n^2 + 4\*n + 2, the time complexity would be O(n^2).

**There are mainly 3 types of Asymptotic notations**:

1. Big-O notation: The Big-O notation describes the worst-case running time of an algorithm. It is computed by counting the number of operations it will take in the worst-case scenario with the input ‘n’.

BigO

O(g(n)) = {f(n): there exist positive constants c and n0

such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0}

2. Big Omega () notation: The notation describes the best running time of an algorithm. It is computed by counting the number of operations it will take in the best-case scenario with the input ‘n’.

BigO

Ω(g(n)) = {f(n): there exist positive constants c and n0

such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0}

3. Big Theta () Notation: The theta notation encloses the function from above and below, therefore it defines the exact asymptotic behaviour. The notation is used for analysing the average runtime of an algorithm.

A picture containing diagram

Description automatically generated

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0

such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0}

**2.Master’s theorem**

The most widely used method to analyse or compare various algorithms is by computing their time complexities. As the algorithm gets complex, the time complexity function calculation also complexifies.

Recursive functions call themselves in their body. It might get complex if we start calculating its time complexity function by other commonly used simpler methods. Master's method is the most useful and easy method to compute the time complexity function of recurrence relations.

First, consider an algorithm with a recurrence of the form

**T(n) = aT(n/b)**

where *a* represents the number of children each node has, and the runtime of each of the three initial nodes is the runtime of T(n/b)

The tree has a depth of log*b*​*n* and depth icontains *ai* nodes. So, there are *a*log*b*​*n*=*n*log*b*​*a* leaves, and hence the runtime is  Θ(*n*log*b*​*a*).

We can apply Master’s Theorem for:

1.Dividing functions

2.Decreasing Functions

Master's Method for Dividing Functions

Used to directly calculate the time complexity function of 'dividing' recurrence relations of the form:

T(n)T(n) = aT(n/b) + f(n)aT(n/b)+f(n)

Where, f(n)f(n) = θ(n^{k} log^{p}n)θ(nklogpn)

Compare *logb*​*a* and kk to decide the final time complexity function.

Master's Theorem is the most useful and easy method to compute the time complexity function of recurrence relations. Master's Algorithm for dividing functions can only be applied on the recurrence relations of the form: T(n)*T*(*n*) = aT(n/b) + f(n)*aT*(*n*/*b*) +*f*(*n*),

where f(n) = θ(n^{k} log^pn)*f*(*n*)=*θ*(*nklogpn*)

for example:

* *T*(*n*) = 2T(n/2)2*T*(*n*/2) + n^{2}*n*2
* T(n)*T*(*n*) = T(n/4)*T*(*n*/4) + nlogn

where,

n = input size (or the size of the problem)  
a = count of subproblems in the dividing recursive function  
n/b = size of each subproblem (Assuming size of each subproblem is same)

Master's Theorem for Decreasing Functions

* Used to directly calculate the time complexity function of 'decreasing' recurrence relations of the form:

**T(n)T(n) = aT(n-b)aT(n−b) + f(n)f(n)**

**f(n)f(n) = θ(n^{k})θ(nk)**

* The value of 'a' will decide the time complexity function for the 'decreasing' recurrence relation.

For decreasing functions of the form T(n) = aT(n-b) + f(n)T(n)=aT(n−b)+f(n),  
where, f(n)f(n) = θ(n^{k})θ(nk)

for example:

T(n) = T(n-2) + 1T(n)=T(n−2)+1

T(n) = 2T(n-1) + n^2T(n)=2T(n−1)+n2

Where,  
n = input size (or the size of the problem)  
a = count of subproblems in the decreasing recursive function.