**Analysis of Algorithms (Background)**

**What is meant by Algorithm Analysis?**

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

**Why Analysis of Algorithms is important?**

* To predict the behavior of an algorithm without implementing it on a specific computer.
* It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
* It is impossible to predict the exact behaviour of an algorithm. There are too many influencing factors.
* The analysis is thus only an approximation; it is not perfect.
* More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

**Types of Algorithm Analysis:**

1. Best case
2. Worst case
3. Average case

**Asymptotic Analysis**

*In Asymptotic Analysis, we****evaluate the performance of an algorithm in terms of input size****(we don’t measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size.*

**Why performance analysis?**

There are many important things that should be taken care of, like user-friendliness, modularity, security, maintainability, etc. Why worry about performance?  The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun! To summarize, performance == scale. Imagine a text editor that can load 1000 pages, but can spell check 1 page per minute OR an image editor that takes 1 hour to rotate your image 90 degrees left OR … you get it. If a software feature can not cope with the scale of tasks users need to perform – it is as good as dead.

**Given two algorithms for a task, how do we find out which one is better?**

One naive way of doing this is – to implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for the analysis of algorithms.

* It might be possible that for some inputs, the first algorithm performs better than the second. And for some inputs second performs better.
* It might also be possible that for some inputs, the first algorithm performs better on one machine, and the second works better on another machine for some other inputs.

[*Asymptotic Analysis*](http://en.wikipedia.org/wiki/Asymptotic_analysis)*is the big idea that handles the above issues in analyzing algorithms. In Asymptotic Analysis, we****evaluate the performance of an algorithm in terms of input size****(we don’t measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size.*

**For example**, let us consider the search problem (searching a given item) in a sorted array.

The solution to above search problem includes:

* **Linear Search** (order of growth is linear)
* **Binary Search** (order of growth is logarithmic).

To understand how Asymptotic Analysis solves the problems mentioned above in analyzing algorithms,

* let us say:
  + we run the Linear Search on a fast computer A and
  + Binary Search on a slow computer B and
  + pick the constant values for the two computers so that it tells us exactly how long it takes for the given machine to perform the search in seconds.
* Let’s say the constant for A is 0.2 and the constant for B is 1000 which means that A is 5000 times more powerful than B.
* For small values of input array size n, the fast computer may take less time.
* **But, after a certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine**.

| **Input Size** | **Running time on A** | **Running time on B** |
| --- | --- | --- |
| **10** | 2 sec | ~ 1 h |
| **100** | 20 sec | ~ 1.8 h |
| **10^6** | ~ 55.5 h | ~ 5.5 h |
| **10^9** | ~ 6.3 years | ~ 8.3 h |

* The reason is the order of growth of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear.
* **So the machine-dependent constants can always be ignored after a certain value of input size.**

Running times for this example:

* Linear Search running time in seconds on A: 0.2 \* n
* Binary Search running time in seconds on B: 1000\*log(n)

**Does Asymptotic Analysis always work?**

Asymptotic Analysis is not perfect, but that’s the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine. Both of these algorithms are asymptotically the same (order of growth is nLogn). So, With Asymptotic Analysis, we can’t judge which one is better as we ignore constants in Asymptotic Analysis.

Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an asymptotically slower algorithm always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above

**Worst, Average and Best Case Time Complexities**

It is important to analyze an algorithm after writing it to find it's efficiency in terms of time and space in order to improve it if possible.

When it comes to analyzing algorithms, the asymptotic analysis seems to be the best way possible to do so. This is because asymptotic analysis analyzes algorithms in terms of the input size. It checks how are the time and space growing in terms of the input size.

In this post, we will take an example of Linear Search and analyze it using Asymptotic analysis.

We can have three cases to analyze an algorithm:

1. Worst Case
2. Average Case
3. Best Case

Below is the algorithm for performing linear search:

// Linearly search x in arr[].

// If x is present then return the index,

// otherwise return -1

int search(int arr[], int n, int x)

{

int i;

for (i=0;i<n;i++){

if(arr[i]==x){

return i;

}

}

return -1;

}

//Driver program to test above functions

int main(){

int arr[]={2,8,12,9};

int x=12;

int n=sizeof(arr)/sizeof(arr[0]);

printf("%d is present in %d index",x,search(arr,n,x));

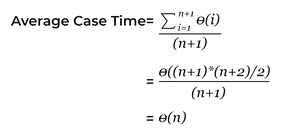
getchar();

return 0;

}

**Worst Case Analysis** **(Usually Done):** In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes the maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be  O(N), where N is the number of elements in the array.

**Average Case Analysis** **(Sometimes done):**  In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (N+1). Following is the value of average case time complexity.



**Best Case Analysis (Bogus):** In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on N). So time complexity in the best case would be O(1).

**Example**-

 #include <bits/stdc++.h>  
using namespace std;  
  
// Linearly search x in arr[].  
// If x is present then return the index,  
// otherwise return -1  
int search(int arr[], int n, int x)  
{  
   int i;  
   for (i = 0; i < n; i++) {  
       if (arr[i] == x)  
           return i;  
   }  
   return -1;  
}  
  
// Driver's Code  
int main()  
{  
   int arr[] = { 1, 10, 30, 15 };  
   int x = 30;  
   int n = sizeof(arr) / sizeof(arr[0]);  
  
   // Function call  
   cout << x << " is present at index "  
        << search(arr, n, x);  
  
   return 0;  
}

**Time Complexity Analysis: (In Big-O notation)**

* **Best Case:**O(1),This will take place if the element to be searched is on the first index of the given list. So, the number of comparisons, in this case, is 1.
* **Average Case:**O(n), This will take place if the element to be searched is on the middle index of the given list.
* **Worst Case:**O(n), This will take place if:
  + The element to be searched is on the last index
  + The element to be searched is not present on the list

**Important Points:**

* Most of the times, we do the worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is a good piece of information.
* The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
* The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

**Asymptotic Notations**

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms:  
  
thetanotation

**1) Θ Notation:** The theta notation bounds a function from above and below, so it defines exact asymptotic behavior.   
A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants. For example, consider the following expression.   
3n3 + 6n2 + 6000 = Θ(n3)   
Dropping lower order terms is always fine because there will always be a number(n) after which Θ(n3) has higher values than Θ(n2) irrespective of the constants involved.   
For a given function g(n), we denote Θ(g(n)) is following set of functions. 

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

that 0 <= c1\*g(n) <= f(n) <= c2\*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1\*g(n) and c2\*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0. 

BigO

**2) Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.  
If we use O notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:  
1. The worst case time complexity of Insertion Sort is O(n^2).  
2. The best case time complexity of Insertion Sort is O(n).  
  
The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

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

n0 such that 0 <= f(n) <= c\*g(n) for

all n >= n0}

BigOmega

**3) Ω Notation:** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.  
  
Ω Notation can be useful when we have lower bound on time complexity of an algorithm. The Omega notation is the least used notation among all three.  
  
For a given function g(n), we denote by Ω(g(n)) the set of functions.

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

n0 such that 0 <= c\*g(n) <= f(n) for

all n >= n0}.

**Big O Notation**

 In this article, we will discuss the analysis of the algorithm using Big - O asymptotic notation in complete detail.   
 **Big-O Analysis of Algorithms**

We can express algorithmic complexity using the big-O notation. For a problem of size N:

* A constant-time function/method is "order 1" : O(1)
* A linear-time function/method is "order N" : O(N)
* A quadratic-time function/method is "order N squared" : O(N 2 )

Definition: Let g and f be functions from the set of natural numbers to itself. The function f is said to be O(g) (read big-oh of g), if there is a constant **c > 0** and a natural number **n0** such that f (n) ≤ cg(n) for all n >= n0 .

Note: O(g) is a set!

Abuse of notation: f = O(g) does not mean f ∈ O(g).  
The Big-O Asymptotic Notation gives us the Upper Bound Idea, mathematically described below:

f(n) = O(g(n)) if there exists a positive integer n0 and a positive constant c, such that f(n)≤c.g(n) ∀ n≥n0

The general step wise procedure for Big-O runtime analysis is as follows:

1. Figure out what the input is and what n represents.
2. Express the maximum number of operations, the algorithm performs in terms of n.
3. Eliminate all excluding the highest order terms.
4. Remove all the constant factors.

Some of the useful properties of Big-O notation analysis are as follow: 

▪ Constant Multiplication:   
If f(n) = c.g(n), then O(f(n)) = O(g(n)) ; where c is a nonzero constant.   
▪ Polynomial Function:   
If f(n) = a0 + a1.n + a2.n2 + ---- + am.nm, then O(f(n)) = O(nm).   
▪ Summation Function:   
If f(n) = f1(n) + f2(n) + ---- + fm(n) and fi(n)≤fi+1(n) ∀ i=1, 2, ----, m,   
then O(f(n)) = O(max(f1(n), f2(n), ----, fm(n))).   
▪ Logarithmic Function:   
If f(n) = logan and g(n)=logbn, then O(f(n))=O(g(n))   
; all log functions grow in the same manner in terms of Big-O.

 Basically, this asymptotic notation is used to measure and compare the worst-case scenarios of algorithms theoretically. For any algorithm, the Big-O analysis should be straightforward as long as we correctly identify the operations that are dependent on n, the input size.

**Runtime Analysis of Algorithms**

In general cases, we mainly used to measure and compare the worst-case theoretical running time complexities of algorithms for the performance analysis.   
The fastest possible running time for any algorithm is O(1), commonly referred to as *Constant Running Time*. In this case, the algorithm always takes the same amount of time to execute, regardless of the input size. This is the ideal runtime for an algorithm, but it's rarely achievable.   
In actual cases, the performance (Runtime) of an algorithm depends on n, that is the size of the input or the number of operations is required for each input item.   
The algorithms can be classified as follows from the best-to-worst performance (Running Time Complexity):

▪ A logarithmic algorithm - O(logn)   
Runtime grows logarithmically in proportion to n.   
▪ A linear algorithm - O(n)   
Runtime grows directly in proportion to n.   
▪ A superlinear algorithm - O(nlogn)   
Runtime grows in proportion to n.   
▪ A polynomial algorithm - O(nc)   
Runtime grows quicker than previous all based on n.   
▪ A exponential algorithm - O(cn)   
Runtime grows even faster than polynomial algorithm based on n.   
▪ A factorial algorithm - O(n!)   
Runtime grows the fastest and becomes quickly unusable for even   
small values of n.

Where, n is the input size and c is a positive constant. 



**Algorithmic Examples of Runtime Analysis**:   
Some of the examples of all those types of algorithms (in worst-case scenarios) are mentioned below: 

▪ Logarithmic algorithm - O(logn) - Binary Search.   
▪ Linear algorithm - O(n) - Linear Search.   
▪ Superlinear algorithm - O(nlogn) - Heap Sort, Merge Sort.   
▪ Polynomial algorithm - O(n^c) - Strassen’s Matrix Multiplication, Bubble Sort, Selection Sort, Insertion Sort, Bucket Sort.   
▪ Exponential algorithm - O(c^n) - Tower of Hanoi.   
▪ Factorial algorithm - O(n!) - Determinant Expansion by Minors, Brute force Search algorithm for Traveling Salesman Problem.

**Mathematical Examples of Runtime Analysis**:   
The performances (Runtimes) of different orders of algorithms separate rapidly as n (the input size) gets larger. Let's consider the mathematical example:

If n = 10, If n=20,

log(10) = 1; log(20) = 2.996;

10 = 10; 20 = 20;

10log(10)=10; 20log(20)=59.9;

102=100; 202=400;

210=1024; 220=1048576;

10!=3628800; 20!=2.432902e+1818;

**Memory Footprint Analysis of Algorithms**

For performance analysis of an algorithm, runtime measurement is not only relevant metric but also we need to consider the memory usage amount of the program. This is referred to as the Memory Footprint of the algorithm, shortly known as Space Complexity.   
Here also, we need to measure and compare the worst case theoretical space complexities of algorithms for the performance analysis.   
It basically depends on two major aspects described below:

* Firstly, the implementation of the program is responsible for memory usage. For example, we can assume that recursive implementation always reserves more memory than the corresponding iterative implementation of a particular problem.
* And the other one is n, the input size or the amount of storage required for each item. For example, a simple algorithm with a high amount of input size can consume more memory than a complex algorithm with less amount of input size.

Algorithmic Examples of Memory Footprint Analysis: The algorithms with examples are classified from the best-to-worst performance (Space Complexity) based on the worst-case scenarios are mentioned below:

▪ Ideal algorithm - O(1) - Linear Search, Binary Search,

Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Shell Sort.

▪ Logarithmic algorithm - O(log n) - Merge Sort.

▪ Linear algorithm - O(n) - Quick Sort.

▪ Sub-linear algorithm - O(n+k) - Radix Sort.

**Space-Time Tradeoff and Efficiency**

There is usually a trade-off between optimal memory use and runtime performance.   
In general for an algorithm, space efficiency and time efficiency reach at two opposite ends and each point in between them has a certain time and space efficiency. So, the more time efficiency you have, the less space efficiency you have and vice versa.   
For example, Mergesort algorithm is exceedingly fast but requires a lot of space to do the operations. On the other side, Bubble Sort is exceedingly slow but requires the minimum space.   
At the end of this topic, we can conclude that finding an algorithm that works in less running time and also having less requirement of memory space, can make a huge difference in how well an algorithm performs.

Example of Big-oh noatation:

C++

// C++ program to findtime complexity for single for loop

#include <bits/stdc++.h>

using namespace std;

// main Code

int main()

{

//declare variable

int a = 0, b = 0;

//declare size

int N = 5, M = 5;

// This loop runs for N time

for (int i = 0; i < N; i++) {

a = a + 5;

}

// This loop runs for M time

for (int i = 0; i < M; i++) {

b = b + 10;

}

//print value of a and b

cout << a << ' ' << b;

return 0;

}

**Output**

25 50

**Explanation :**  
First Loop runs *N* Time whereas Second Loop runs *M* Time. The calculation takes *O(1)times*.  
So by adding them the time complexity will be O ( N + M + 1) = O( N + M).

**Time Complexity** : O( N + M)

**Omega Notation**

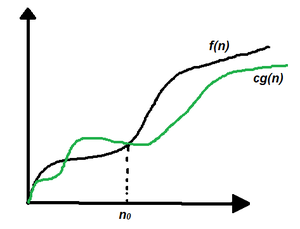
This article will discuss Big - Omega Notation represented by a Greek letter (Ω).

**Definition:** Let g and f be the function from the set of natural numbers to itself. The function f is said to be Ω(g), if there is a constant c > 0 and a natural number n0 such that c\*g(n) ≤ f(n) for all n ≥ n0

**Mathematical Representation:**

Ω(g) = {f(n): there exist positive constants c and n0 such that 0 ≤ c\*g(n) ≤ f(n) for all n ≥ n0}   
Note: Ω (g) is a set

**Graphical Representation:**

Graphical Representation

In simple language, Big - Omega(Ω) notation specifies the asymptotic (at the extreme) lower bound for a function f(n).

Follow the steps below to calculate Big - Omega(Ω) for any program:

1. Break the program into smaller segments.
2. Find the number of operations performed for each segment(in terms of the input size) assuming the given input is such that the program takes the least amount of time.
3. Add up all the operations and simplify it, let's say it is f(n).
4. Remove all the constants and choose the term having the least order or any other function which is always less than f(n) when n tends to infinity, let say it is g(n) then, Big - Omega (Ω) of f(n) is Ω(g(n)).

**Example:**Consider an example to print all the possible pairs of an array. The idea is to run two nested loops to generate all the possible pairs of the given array.

The pseudo-code is as follows:

int print(int a[], int n)

{

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if(i != j)

cout << a[i] << " "

<< a[j] << "\n";

}

}

}

Below is the implementation of the above approach:

C++Java

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

// Function to print all possible pairs

int print(int a[], int n)

{

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

if (i != j)

cout << a[i] << " " << a[j] << "\n";

}

}

}

// Driver Code

int main()

{

// Given array

int a[] = { 1, 2, 3 };

// Store the size of the array

int n = sizeof(a) / sizeof(a[0]);

// Function Call

print(a, n);

return 0;

}

**Output**

1 2

1 3

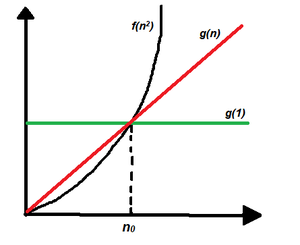
2 1

2 3

3 1

3 2

In this example, it is evident that the print statement gets executed n2 times therefore if the running time vs n graph is plotted a parabolic graph will be obtained, f(n2). Now linear functions g(n), logarithmic functions g(log n), constant functions g(1) all are less than a parabolic function when the input range tends to infinity therefore, the worst-case running time of this program can be Ω(log n), Ω(n), Ω(1), or any function g(n) which is less than n2when n tends to infinity. See the below graphical representation:



**When to use Big - Ω notation:**Big - Ω notation is the least used notation for the analysis of algorithms because it can make a correct but imprecise statement over the performance of an algorithm. Suppose a person takes 100 minutes to complete a task, using Ω notation, it can be stated that the person takes more than 10 minutes to do the task, this statement is correct but not precise as it doesn't mention the upper bound of the time taken. Similarly, using Ω notation we can say that the worst-case running time for the [binary search](https://www.geeksforgeeks.org/binary-search/) is Ω(1), which is true because we know that binary search would at least take constant time to execute.

**Theta Notation**

This article will discuss Big - Theta notations represented by a Greek letter (Θ).

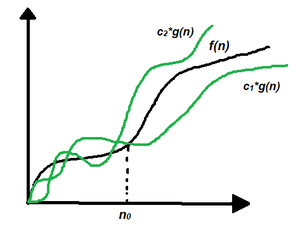
**Definition:** Let g and f be the function from the set of natural numbers to itself. The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that c1\* g(n) ≤ f(n) ≤ c2\* g(n) for all n ≥ n0

**Mathematical Representation:**

Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1\* g(n) ≤ f(n) ≤ c2\* g(n) for all n ≥ n0}  
Note: Θ(g) is a set

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1 \* g(n) and c2 \* g(n) for large values of n (n ≥ n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

**Graphical Representation:**

Graphical Representation

In simple language, Big - Theta(Θ) notation specifies asymptotic bounds (both upper and lower) for a function f(n) and provides the average time complexity of an algorithm.

Follow the steps below to find the average time complexity of any program:

1. Break the program into smaller segments.
2. Find all types and number of inputs and calculate the number of operations they take to be executed. Make sure that the input cases are equally distributed.
3. Find the sum of all the calculated values and divide the sum by the total number of inputs let say the function of n obtained is g(n) after removing all the constants, then in Θ notation its represented as Θ(g(n))

**Example 1:**

 var doLinearSearch = function(array, targetValue)

    {  for (var guess = 0; guess < array.length; guess++)

        {    if (array[guess] === targetValue)

             {        return guess;  // found it!

              }

         }

      return -1;  // didn't find it

   };

Each time the for-loop iterates, it has to do several things-

* compare guess with array.length
* compare array[guess] with targetValue
* possibly return the value of guess
* increment guess.

So the complexity will be theta n.

**Example 2:**Consider an example to find whether a key exists in an array or not using linear search. The idea is to traverse the array and check every element if it is equal to the key or not.

The pseudo-code is as follows:

bool linearSearch(int a[], int n, int key)

{

for (int i = 0; i < n; i++) {

if (a[i] == key)

return true;

}

return false;

}

Below is the implementation of the above approach:

C++Java

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

// Function to find whether a key exists in an

// array or not using linear search

bool linearSearch(int a[], int n, int key)

{

// Traverse the given array, a[]

for (int i = 0; i < n; i++) {

// Check if a[i] is equal to key

if (a[i] == key)

return true;

}

return false;

}

// Driver Code

int main()

{

// Given Input

int arr[] = { 2, 3, 4, 10, 40 };

int x = 10;

int n = sizeof(arr) / sizeof(arr[0]);

// Function Call

if (linearSearch(arr, n, x))

cout << "Element is present in array";

else

cout << "Element is not present in array";

return 0;

}

**Output**

Element is present in array

In a linear search problem, let's assume that all the cases are [uniformly distributed](https://www.geeksforgeeks.org/python-uniform-discrete-distribution-in-statistics/) (including the case when the key is absent in the array). So, sum all the cases (when the key is present at position 1, 2, 3, ......, n and not present, and divide the sum by n + 1.

Average case time complexity = ∑�=1�+1�(�)�+1

⇒ �((�+1)∗(�+2)/2)�+1

⇒ �(1+�/2)

⇒ �(�) (constants are removed)

**When to use Big - Θ notation:**Big - Θ analyzes an algorithm with most precise accuracy since while calculating Big - Θ, a uniform distribution of different type and length of inputs are considered, it provides the average time complexity of an algorithm, which is most precise while analyzing, however in practice sometimes it becomes difficult to find the uniformly distributed set of inputs for an algorithm, in that case, Big-O notation is used which represent the asymptotic upper bound of a function f.

**Analysis of Common loops**

In this post, an analysis of iterative programs with simple examples is discussed.   
  
**1) O(1):**Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion, and call to any other non-constant time function. 

// set of non-recursive and non-loop statements

For example, swap() function has O(1) time complexity.   
A loop or recursion that runs a constant number of times is also considered as O(1). For example, the following loop is O(1). 

// Here c is a constant

for (int i = 1; i <= c; i++) {

// some O(1) expressions

}

**2) O(n):** Time Complexity of a loop is considered as O(n) if the loop variables are incremented/decremented by a constant amount. For recursive call in recursive function, the time complexity is considered as O(n). For example following functions have O(n) time complexity. 

// Here c is a positive integer constant

for (int i = 1; i <= n; i += c) {

// some O(1) expressions

}

for (int i = n; i > 0; i -= c) {

// some O(1) expressions

}

//Recursive function

void recurse(n)

{

if(n==0)

return;

else{

// some O(1) expressions

}

recurse(n-1);

}

**3) O(nc)**: Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example, the following sample loops have O(n2) time complexity 

for (int i = 1; i <=n; i += c) {

for (int j = 1; j <=n; j += c) {

// some O(1) expressions

}

}

for (int i = n; i > 0; i -= c) {

for (int j = i+1; j <=n; j += c) {

// some O(1) expressions

}

For example, Selection sort and Insertion Sort have O(n2) time complexity.

**4) O(Logn)** Time Complexity of a loop is considered as O(Logn) if the loop variables are divided/multiplied by a constant amount. 

for (int i = 1; i <=n; i \*= c) {

// some O(1) expressions

}

for (int i = n; i > 0; i /= c) {

// some O(1) expressions

}

For example, Binary Search(refer iterative implementation) has O(Logn) time complexity. Let us see mathematically how it is O(Log n). The series that we get in the first loop is 1, c, c2, c3, ... ck. If we put k equals to Logcn, we get cLogcn which is n.   
**5) O(LogLogn)** Time Complexity of a loop is considered as O(LogLogn) if the loop variables are reduced/increased exponentially by a constant amount. 

// Here c is a constant greater than 1

for (int i = 2; i <=n; i = pow(i, c)) {

// some O(1) expressions

}

//Here fun is sqrt or cuberoot or any other constant root

for (int i = n; i > 1; i = fun(i)) {

// some O(1) expressions

}

See [this](https://www.cdn.geeksforgeeks.org/time-complexity-loop-loop-variable-expands-shrinks-exponentially/)for mathematical details.

**How to combine the time complexities of consecutive loops?**   
When there are consecutive loops, we calculate time complexity as a sum of time complexities of individual loops. 

for (int i = 1; i <=m; i += c) {

// some O(1) expressions

}

for (int i = 1; i <=n; i += c) {

// some O(1) expressions

}

Time complexity of above code is O(m) + O(n) which is O(m+n)

If m == n, the time complexity becomes O(2n) which is O(n).

**How to calculate time complexity when there are many if, else statements inside loops?**   
As discussed [here](https://www.cdn.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/), worst-case time complexity is the most useful among best, average and worst. Therefore we need to consider the worst case. We evaluate the situation when values in if-else conditions cause a maximum number of statements to be executed.   
For example, consider the linear search function where we consider the case when an element is present at the end or not present at all.   
When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if-else and other complex control statements.

**How to calculate the time complexity of recursive functions?**   
The time complexity of a recursive function can be written as a mathematical recurrence relation. To calculate time complexity, we must know how to solve recurrences. We will soon be discussing recurrence solving techniques as a separate post.

The following is a cheat sheet of the time complexities of various algorithms.

**Algorithms Cheat Sheet**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| Selection Sort | O(n^2) | O(n^2) | O(n^2) |
| Bubble Sort | O(n) | O(n^2) | O(n^2) |
| Insertion Sort | O(n) | O(n^2) | O(n^2) |
| Tree Sort | O(nlogn) | O(nlogn) | O(n^2) |
| Radix Sort | O(dn) | O(dn) | O(dn) |
| Merge Sort | O(nlogn) | O(nlogn) | O(nlogn) |
| Heap Sort | O(nlogn) | O(nlogn) | O(nlogn) |
| Quick Sort | O(nlogn) | O(nlogn) | O(n^2) |
| Bucket Sort | O(n+k) | O(n+k) | O(n^2) |
| Counting Sort | O(n+k) | O(n+k) | O(n+k) |

**Analysis of Recursion**

Many algorithms are recursive. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in Merge Sort, to sort a given array, we divide it into two halves and recursively repeat the process for the two halves. Finally, we merge the results. Time complexity of Merge Sort can be written as T(n) = 2T(n/2) + cn. There are many other algorithms like Binary Search, Tower of Hanoi, etc.   
  
There are mainly three ways of solving recurrences:

## Substitution Method:

We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

T(n) = 2T(n/2) + n  
     <= 2c(n/2Log(n/2)) + n  
       =  cnLogn - cnLog2 + n  
       =  cnLogn - cn + n  
    <= cnLogn

## ****Recurrence Tree Method:****

In this method, we draw a recurrence tree and calculate the time taken by every level of the tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically arithmetic or geometric series. 

For example, consider the recurrence relation

T(n) = T(n/4) + T(n/2) + cn2

            cn2  
/      \  
  T(n/4)     T(n/2)

If we further break down the expression T(n/4) and T(n/2),   
we get the following recursion tree.

                    cn2  
              /             \        
    c(n2)/16          c(n2)/4  
   /         \            /         \  
T(n/16)  T(n/8)  T(n/8)    T(n/4)

Breaking down further gives us following

                       cn2   
                /                \       
       c(n2)/16              c(n2)/4  
    /          \                 /          \  
c(n2)/256  c(n2)/64  c(n2)/64   c(n2)/16  
 /    \            /    \      /    \        /    \

To know the value of T(n), we need to calculate the sum of tree   
nodes level by level. If we sum the above tree level by level,

we get the following series T(n)  = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ....  
The above series is a geometrical progression with a ratio of 5/16.

To get an upper bound, we can sum the infinite series. We get the sum as (n2)/(1 - 5/16) which is O(n2)

## ****Master Method:****

Master Method is a direct way to get the solution. The master method works only for the following type of recurrences or for recurrences that can be transformed into the following type. 

T(n) = aT(n/b) + f(n) where a >= 1 and b > 1

There are the following three cases:

* If f(n) = O(nc) where c < Logba then T(n) = Θ(nLogba)
* If f(n) = Θ(nc) where c = Logba then T(n) = Θ(ncLog n)
* If f(n) = Ω(nc) where c > Logba then T(n) = Θ(f(n))

### **How does this work?**

The master method is mainly derived from the recurrence tree method. If we draw the recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at the root is f(n), and work done at all leaves is Θ(nc) where c is Logba. And the height of the recurrence tree is Logbn 

Master Theorem

In the recurrence tree method, we calculate the total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically the same, then our result becomes height multiplied by work done at any level (Case 2). If work done at the root is asymptotically more, then our result becomes work done at the root (Case 3).   
  
**Examples of some standard algorithms whose time complexity can be evaluated using the Master Method**

* [Merge Sort](http://geeksquiz.com/merge-sort/): T(n) = 2T(n/2) + Θ(n). It falls in case 2 as c is 1 and Logba] is also 1. So the solution is Θ(n Logn)
* [Binary Search](http://geeksquiz.com/binary-search/): T(n) = T(n/2) + Θ(1). It also falls in case 2 as c is 0 and Logba is also 0. So the solution is Θ(Logn)

**Notes:**

* It is not necessary that a recurrence of the form T(n) = aT(n/b) + f(n) can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence T(n) = 2T(n/2) + n/Logn cannot be solved using master method.
* Case 2 can be extended for f(n) = Θ(ncLogkn)   
  If f(n) = Θ(ncLogkn) for some constant k >= 0 and c = Logba, then T(n) = Θ(ncLogk+1n)

Recursion Tree Method for Solving Recurrences

The Recursion Tree Method is a way of solving recurrence relations. 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:**

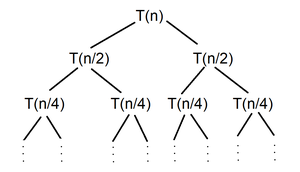
1. Draw a recursive tree for given recurrence relation
2. Calculatethe cost at each level and count the total no of levels in the recursion tree.
3. Count the total number of nodes in the last level and calculate the cost of the last level
4. Sum up the cost of all the levels in the recursive tree

**Let us see how to solve these recurrence relations with the help of some examples:**

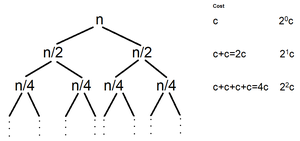
**Question 1:** T(n) = 2T(n/2) + c

**Solution:**

* **Step 1:**Draw a recursive tree

Recursion Tree

* **Step 2:** Calculate the work done or cost at each level and count total no of levels in recursion tree

Recursive Tree with each level cost

**Count the total number of levels -**

Choose the longest path from root node to leaf node

 n/20 -→ n/21 -→ n/22 -→ ......... -→ n/2k

Size of problem at last level = n/2k

At last level size of problem becomes 1

 n/2k = 1

 2k = n

**k = log2(n)**

**Total no of levels  in recursive tree = k +1 = log2(n) + 1**

* **Step 3:** Count total number of nodes in the last level and calculate cost of last level

 No. of nodes at level 0 = 20 = 1

No. of nodes at level 1 = 21 = 2

 ...............................................................

 No. of nodes at level log2(n) = 2log2(n) = nlog2(2)= n

 Cost of sub problems at level log2(n) (last level) = nxT(1) = n x c = nc

* **Step 4:** Sum up the cost all the levels in recursive tree

 T(n) = c + 2c + 4c + ---- + (no. of levels-1) times + last level cost

 = c + 2c + 4c + ---- + log2(n) times + Θ(n)

 = c(1 + 2 + 4 + ---- + log2(n) times) + Θ(n)

 1 + 2 + 4 + ----- + log2(n) times --> 20 + 21 + 22 + ----- + log2(n) times --> Geometric Progression(G.P.)

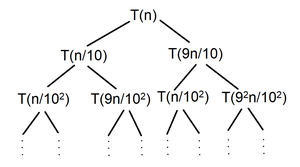
= c(n) + Θ(n)

Thus, ***T(n) = Θ(n)***

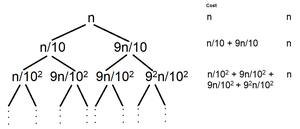
**Question 2: T(n) = T(n/10) + T(9n/10) + n**

**Solution:**

* **Step 1:** Draw a recursive tree

Recursive Tree

* **Step 2:** Calculate the work done or cost at each level and count total no of levels in recursion tree

Recursion Tree with each level cost

**Count the total number of levels -**

 Choose the longest path from root node to leaf node

(9/10)0n --> (9/10)1n --> (9/10)2n --> ......... --> (9/10)kn

 Size of problem at last level = (9/10)kn

 At last level size of problem becomes 1

(9/10)kn = 1

(9/10)k = 1/n

**k = log10/9(n)**

**Total no of levels in recursive tree = k +1 = log10/9(n) + 1**

* **Step 3:** Count total number of nodes in the last level and calculate cost of last level

No. of nodes at level 0 = 20 = 1

No. of nodes at level 1 = 21 = 2

...............................................................

No. of nodes at level log10/9(n) = 2log10/9(n) = nlog10/9(2)

Cost of sub problems at level log10/9(n) (last level) = nlog10/9(2) x T(1) = nlog10/9(2) x 1 = nlog10/9(2)

* **Step 4:** Sum up the cost all the levels in recursive tree

T(n) = n + n + n + ---- + (no. of levels - 1) times + last level cost

 = n + n + n + ---- + log10/9(n) times + Θ(nlog10/9(2))

= nlog10/9(n) + Θ(nlog10/9(2))

Thus, ***T(n) = Θ(nlog10/9(2))***

Space Complexity

**Space Complexity:**   
The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.   
  
*Auxiliary Space* is the extra space or temporary space used by an algorithm.   
  
*Space Complexity*of an algorithm is the total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.   
  
For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criterion than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort, and Heap Sort use O(1) auxiliary space. The space complexity of all these sorting algorithms is O(n) though.

Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will require O(n) space. If we create a two-dimensional array of size n\*n, this will require O(n2) space.

In recursive calls stack space also counts.

Example :

int add (int n){

if (n <= 0){

return 0;

}

return n + add (n-1);

}

Here each call add a level to the stack :

1. add(4)

2. -> add(3)

3. -> add(2)

4. -> add(1)

5. -> add(0)

Each of these calls is added to call stack and takes up actual memory.

So it takes O(n) space.

However, just because you have n calls total doesn't mean it takes O(n) space.

Look at the below function :

int addSequence (int n){

int sum = 0;

for (int i = 0; i < n; i++){

sum += pairSum(i, i+1);

}

return sum;

}

int pairSum(int x, int y){

return x + y;

}

There will be roughly O(n) calls to pairSum. However, those

calls do not exist simultaneously on the call stack,

so you only need O(1) space.

**Note:** It’s necessary to mention that space complexity depends on a variety of things such as the programming language, the compiler, or even the machine running the algorithm.