**Complexity Analysis**

1. **Time Complexity** :- The time complexity is the simple complexity that describes the amount of time it takes to run an algorithm.

As the amount of resources required to run an algorithm generally varies with the size of the input so it is recommended to check complexity with higher input size, the complexity is typically expressed as a function *n* → *f*(*n*), where *n* is the size of the input and *f*(*n*) is the time taken by the resource.

Here *f*(*n*) can be the **worst-case complexity** or **average-case complexity** .

**There are two kinds of complexities: time and space**

1. **Asymptotic Notations** :- Asymptotic notations are mathematical tools to represent the time complexity of algorithms The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms.

(i) **Θ Notation(theta)**

(ii) **Big O Notation**

(iii) **Ω Notation(omega)**

**Big O Notation :-** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.

**Ex:**-

BigO

Where 0 <= f(n) <= c\*g(n) for all n >= n0

You can see there is a point n0 from where g(n) is constantly in upper side of f(n). hence if we pass n>=n0 we will always get g(n) as upper bound.

**Θ Notation(Theta) :-** The theta notation bounds a functions from above and below it means bound to the middle.

For example, consider the following expression.  
g(n)= 3n3 + 6n2 + 6000 = Θ(n3) (higher order of degree)

thetanotation

Dropping lower order terms is always fine because there will always be a n0 after which Θ(n3) has higher values than Θn2) 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

**Ω Notation(omega) :-** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Omega notation is the least used notation among all three.

BigOmega

f(n)= Ω(g(n)): n0 such that 0 <= c\*g(n) <= f(n) for all n >= n0.

1. **Auxiliary Space Complexity :-** Auxiliary Space is the extra space or temporary space used by an algorithm.

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Ex :- if we want to reverse the array and created an new array to store elements in reverse order then we are using O(size of array) auxiliary space.

1. **why should we do analysis of our function/Algorithm ?**

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.

we will take an example of Linear Search and analyze it using Asymptotic analysis.

Linear search ?

We can have three cases to analyze an algorithm:

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

**Worst Case Analysis (Usually Done) :-** In the worst case analysis, we calculate upper bound on running time of an algorithm.

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.

Ex :- code in Intellij

**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).

**Average Case Analysis (Sometimes done) :-** 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.

Some examples :-

**O(1) :- A swap function or any mathematical operation or variable declaration .**

**Fun( int[ ] arr ,int i, int j){**

**int temp =arr[i];**

**arr[i]=arr[j];**

**arr[j]=temp;**

**}**

**O(n) :- any loop .**

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

**// some O(1) expressions**

**// swap();**

**}**

**O(n^2) :- any nested loop**

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

**for (int j = 1; j <=n; j==) {**

**// some O(1) expressions**

**//swap();**

**}**

**}**

Ex :- what will be the complexity of below algorithm ?

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

// some O(1) expressions // constant

}

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

// some O(1) expressions // constant

}

There is 2 loops not nested do we have to calculate the complexity individually

For first loop it will be O(m)

For second it will be O(n)

So the final complexity will be O(m+n).

If m and n both are same the O(2n) and will ignore the constant and the complexity will be O(n) or O(m).

For complexity O(log n) we will understand the binary search first.

Binary search always works in a sorted array or objects.

**Binary Search** is a searching algorithm for searching an element in a sorted list or array. Binary Search is efficient than Linear Search algorithm and performs the search operation in logarithmic time complexity for sorted arrays or lists.

Binary Search performs the search operation by repeatedly dividing the search interval in half. The idea is to begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



int binarySearch(int arr[], int l, int r, int x)

{

while (l <= r) { // l= lower index of an arr , r = length-1 index of element

int m= (l+r)/2; // m = l + (r - l) / 2; to remove overflow condition.

// Check if x is present at mid

if (arr[m] == x)

return m;

// If x greater, ignore left half

if (arr[m] < x)

l = m + 1;

// If x is smaller, ignore right half

else

r = m - 1;

}

// if we reach here, then element was

// not present

return -1;

}

**Time Complexity**: O(Log N), where N is the number of elements in the array.

Let’s talk something more why analysis of function/Algorithm is necessary or needed look at below problem.

Problem :- Find the index of first 1 in a sorted array of 0’s and 1’s

Input : arr[] = [0, 0, 0, 0, 0, 0, 1, 1, 1, 1]

Output : 6 (The index of first 1 in the array is 6)

Solution 1 :- FindFirstIndexOfOneNavieSolution.java O(n)

Solution 2 :- FindFirstIndexOfOneBetterComplexity.java O(log n)

**Calculating Time complexity:**

* + Let say the iteration in Binary Search terminates after **k** iterations. In the above example, it terminates after 3 iterations, so **here k = 3**
  + At each iteration, the array is divided by half. So let’s say the length of array at any iteration is **n**
  + At **Iteration 1**,

Length of array = **n**

* + At **Iteration 2**,

Length of array = **n⁄2**

* + At **Iteration 3**,

Length of array = **(n⁄2)⁄2** = **n⁄22**

* + Therefore, after **Iteration k**,

Length of array = **n⁄2k**

* + Also, we know that after

After k divisions, the **length of array becomes 1**

* + Therefore
  + Length of array = **n⁄2k = 1**
  + => **n = 2k**
  + Applying log function on both sides:
  + => **log2 (n) = log2 (2k)**
  + => **log2 (n) = k log2 (2)**
  + As **(loga (a) = 1)**  
    Therefore,
  + => **k = log2 (n)**

**Hence, the time complexity of Binary Search is**

***log2 (n)***

This is all about binary search.

O(n log n) :- for example Sorting algorithm

1. Merge sort (all case O(nlogn)
2. Quick sort (best and average case O(nlogn) and worst case O(n^2).

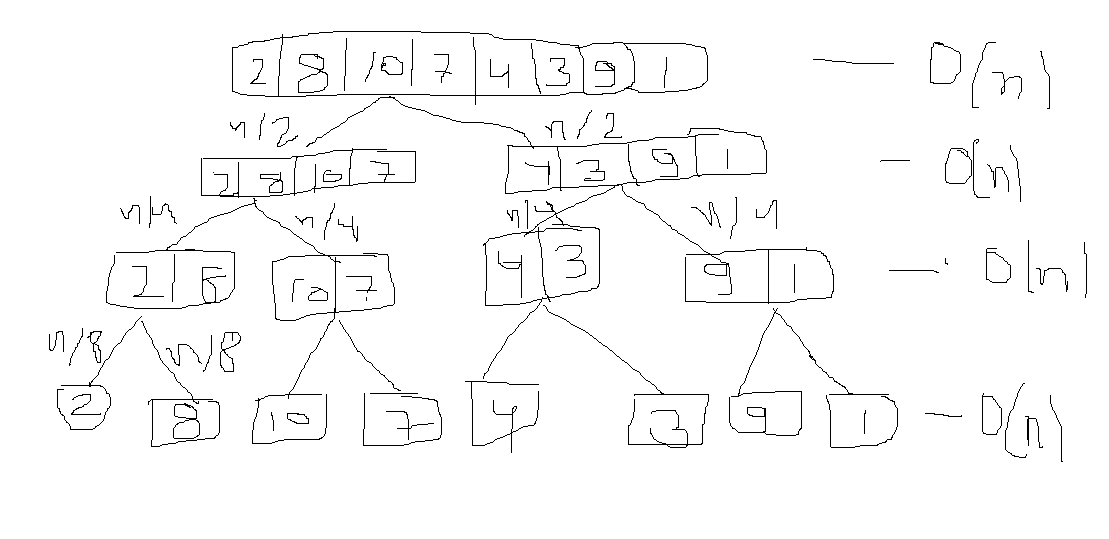
**Merge Sort** is a Divide and Conquer algorithm.

**MergeSort(arr[], l, r)**  
If r > l  
 **1.** Find the middle point to divide the array into two halves:   
 middle m = (l+r)/2  
  **2.** Call mergeSort for first half:   
 Call mergeSort(arr, l, m)  
 **3.** Call mergeSort for second half:  
 Call mergeSort(arr, m+1, r)  
 **4.** Merge the two halves sorted in step 2 and 3:  
 Call merge(arr, l, m, r)



**Time Complexity:** Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

T(n) = 2T(n/2) + Θ(n)



in above image I tried to explain the complexity of above recursion .

as you can see we are dividing our input by 2 in very case so we will calculate the total work done at each level . so we are doing O(n) work at each level. Now we have to calculate the total no of time we are doing O(n) work. So clearly as per the binary search we are doing (log n ) time every work

so the final complexity :- n \* log n

Time complexity of Merge Sort is **Θ(nLogn)** in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.  
  
**Auxiliary Space:** O(n)

Increasing Order of time complexity.

**O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(2^n)**

Refer the image below.

