UNIT - I

⇒ **Algorithm:-** The Step-by-Step procedure written in Simple statements to Solve a particular problem is known as an algorithm.

* An Algorithm of a problem makes the Coder to understand the logic behind the problem in a detailed manner, so that the Coder will be able to write the Code (Executable program) from it.

Characteristics of an Algorithm:-

① **Finiteness:-** An Algorithm must Contain finite no. of Steps.
i.e., the program must terminate at some point.

② **Definiteness:-** Each step of an algorithm must be precisely stated and must be in an order.
i.e., Each Step of an algorithm must be unambiguous / Certain.

③ **Effectiveness:-** As a matter of time Complexity, an algorithm must not Contain the unnecessary /

redundant steps. So that the prog
Can run in less amount of time.

④ Generality :- An Algorithm must work for any type of input data provided by the user. So that the program may, be general for any type of input data.

⑤ Input (&) output :-
* The data given to a program to run, is known as input.
* The result of a program is known as output.
∴ An Algorithm must contain inputs and outputs.

→ The Algorithms are usually written in pseudo code for easeir, better, and quckly understand the logic behind the algorithm and the steps involved in it.

→ A pseudocode is a description of the Algorithm written using some simple conventions from programming languages (such as C and Pascal)

Ex:- Pseudo Code for an Algorithm of Calculating the Avg of three numbers, is:-

1. Input three numbers: num1, num2, num3
2. Calculating the sum of three numbers:
   sum = num1 + num2 + num3.
3. Calculate the avg: average = sum/3
4. Output the avg. //

⇒ **Performance Analysis:-**

* An Algorithm is built to solve a particular problem.
* There can be many ways to solve a problem.
* Hence, there can be many algorithms to solve a particular problem.
* But, our goal is to choose the most efficient algorithm [which takes less time and less space].
* Hence, Analysing the Performance of an Algorithm ultimately means that, analysing its efficiency.
* The Performance (efficiency) of an algorithm can be determined based on the two factors, viz:-
  1. time complexity
  2. Space complexity

① **Time Complexity:-**

It is a function which gives the

relation b/w Execution time and input size
i.e., the function gives the Execution time of an algorithm/Program for a given input size.

* which means that the Execution time depends on the provided input size.

- $\boxed{f(n) = \text{time Complexity}}$

where,

n = input size

* There are three case scenarios of it:-

(i) **Worst Case**:- This is when Execution time is maximum, for a given input size (n)

i.e., no matter how large the input size(n) is, the algorithm will not Exceed the time Complexity of this case.

i.e, $[f(n) = \text{Max. time Complexity}]$

(ii) **Best Case**:- This is when Execution time is minimum, for a given input size(n)

i.e., no matter how small the input size (n) is, the algorithm will not have lesser (better) time Complexity than this case.

i.e, $[f(n) = \text{Min time complexity}]$

(iii) Average Case:-

~~[illegible]~~

This is when the Execution time is the Average of Execution times of all possible cases, of the input size(n).

i.e., $\boxed{f(n) = \dfrac{\text{Execution times for all possible inputs}}{\text{no. of inputs (cases)}}}$

② Space Complexity:-
The total space taken by an Algorithm, which includes Auxiliary space and space taken by input size (n), is known as space complexity.

* An Auxiliary space is an extra / temporary space used by an algorithm.

i.e. total space taken by an algorithm with respect to the input

* The worst Space Complexity is Considered always, for Comparision and Selection. i.e, Big-oh Complexity.
* Auxiliary Space includes memory Consumed by Variables, other Arrays, etc. //

⇒ <u>Asymptotic Notations :-</u>

* An Asymptote is a st. line that Continually approaches a given curve but does not meet it, at any finite distance.

* Asymptotic Notation is a Mathematical notation used to describe the time/Space Complexity of an algorithm, as the input size apprаches infinity.

* It helps to compare the runtimes of different algorithms without actually Calculating their runtimes manually.
  i.e, runtime is Calcutated based on the input size of the algorithm.

* It is used to Analyze the Efficiency and performance of an algorithm.

* There are (5) types of it:-
  * (1) Big-oh (O) (worst)
  * (2) Big-omega (Ω) (best)
  * (3) Theta (Θ) (Avg)

  (4) Little-oh (o)

  (5) Little-omega (ω)

(1) Big-oh Notation:-

This notation is used to describe the worst case time/space Complexity.

i.e., • It gives the upper bound Complexity of an algorithm.

* which means that the Execution time/total space, cannot Exceed this upper bound.

* It is given by:-

$$f(n) = O(g(n))$$

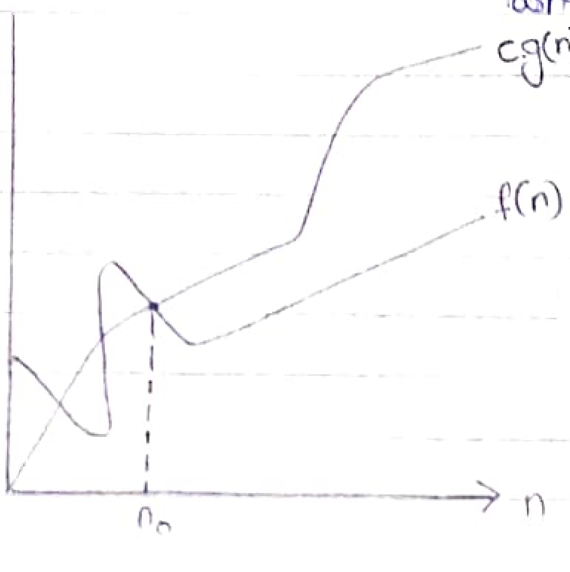
Such that, $0 \le f(n) \le c.g(n)$

where,

c => constant

n => input size.

$n \le n_0$



② Big-omega notation:-

This notation is used to describe the best case time/space complexity of an algorithm.

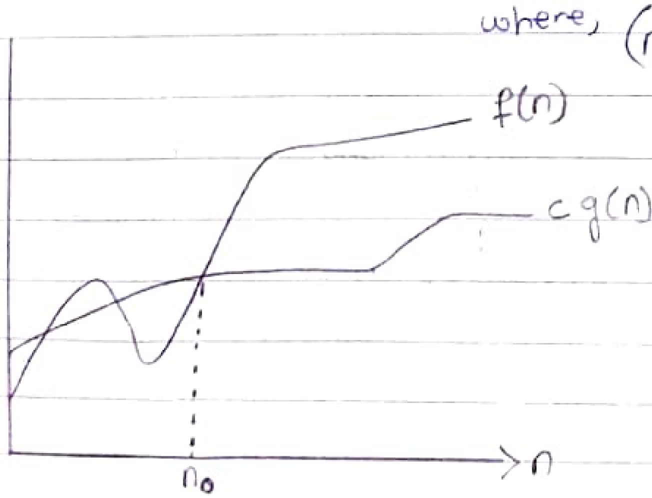
i.e, It gives the lower bound complexity of an algorithm.

* which means that the execution time/total space, cannot be lower than this lower bound.
* It is given by:-

$$f(n) = \Omega(g(n))$$

Such that, $0 \leq c.g(n) \leq f(n)$

where, $(n \leq n_0)$



③ Theta Notation:-

This Notation is used to describe the Avg case time/space complexity of an Algorithm.

i.e., It gives the Avg Curve of lower bound and upperbound of an algorithm.

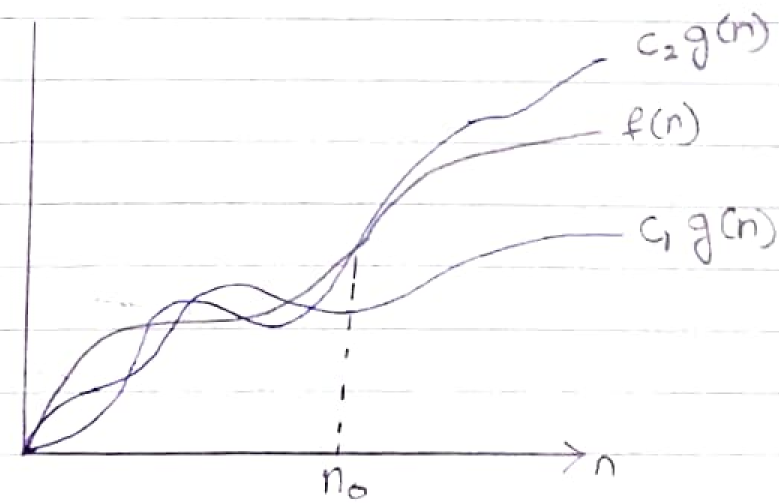
* It is given by:-

$$f(n) = \Theta(g(n))$$

Such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

where,

$n \geq n_0$
$c_1, c_2 \Rightarrow$ constants.
$n \Rightarrow$ input size.



(4) Little-oh Notation:-

It is similar to Big-oh notation but the time/space complexity cannot be equal to the upper bound.

* It is given by :-

$$f(n) = o(g(n))$$

Such that, $0 < f(n) < g(n).c$

(5) Little -Omega Notation :-
It is similar to Big-omega Notation, but the time/space Complexity Cannot be Equal to the lower bound of the algorithm.

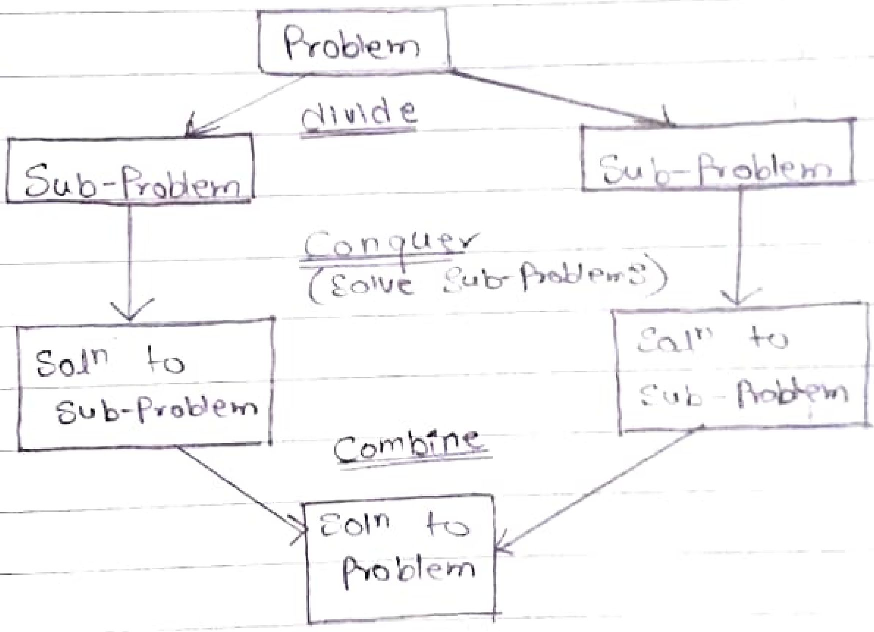
* It is given by :-

$$f(n) = \omega(g(n))$$

Such that,
$$0 < c.g(n) < f(n)$$

⇒ Divide and Conquer :-

* As the name Sugests, It is an approach of Solving a big/complex problem by dividing it into several Sub-Problems to solve them individually and then Combine them to get the Solution to the main problem.



* let f(n) be the no.of operations required to find the Soln of the original Problem (and) (b) be the no. of divisions, then the recursive relation of divide and Conquer algorithms will be of the form:-

[b>1]

$$\boxed{f(n) = a f\left(\frac{n}{b}\right) + g(n)}$$

where,

$f\left(\frac{n}{b}\right)$ ⇒ operations required to find the Soln of a Sub Problem

a ⇒ constant $(a \geq 1)$

$g(n)$ ⇒ operations required to combine the sub-Problem solutions

⇒ Applications of D & C [D&C Algorithms]

1. Binary Search
2. Merge Sort
3. Quick Sort
4. Strassen's matrix multiplication.

Advantages of D & C

* Efficiency :- A large problem can be solved quickly, which reduces the overall time complexity
* Parallelism :- The independent sub-problems can be solved concurrently.
* Scalability :- no matter how large the problem is, this concept can be applied always.

Disadvantages of D&C :-

* Memory usage :- These type of algorithms are recursive, which consumes large amount of memory in stack.

① Binary Search :-

This Algorithm is used to search the specified data in the large . sorted array

* This algorithm is inspired from the "Binary search tree" Data structure.

Working:-

* The specified element to be searched is compared with the middle element of the given sorted list.
* If the given element is lesser than the middle element, then the left half part is selected for further search.
* If the given element is greater than the middle element, then the right half part is selected for further search.
* This process continues until the middle element and specified element becomes equal [i.e, element is found].
* Hence, if $B(n)$ represents the total no. of comparisions required to search for an element in sorted set of (n) elements, then the recursive relation for Binary search is:-

$$\boxed{B(n) = B\left(\frac{n}{2}\right) + 2}$$

$$[B(1) = 2]$$

where,

$n \Rightarrow$ even ; $n \geq 1$

Algorithm:-

Step ①:- point (i) to first element (i.e., i=1) and (j) to last element (i.e., j=n)

Step② Run a loop from the current value of (i) to current value of (j) [i.e., $i \leq j$]

Step③ Compute the position of middle element by using the formula

$$\boxed{m = \frac{i+j}{2}}$$

Step④ :- If $(x < B_m)$, move the (j) pointer to $(m-1)$ position [i.e., select left sublist]
If $(x > B_m)$, move the (i) pointer to $(m+1)$ position [i.e, select right sublist]
Else, return (m) [i.e., $x == B_m$]

Step⑤ If the loop runs completely and $(x \neq B_m)$, then return (0) [indicating that element is not found in the list]

~~Advantages~~

Time Complexities:-

→ Best case :- O(1)

[Element is found at the middle of the original array]

→ Average case :- O(log n)

→ worst case :- O(log n)

[Element is found after all possible no. of comparisions.]

Space Complexity:-

Advantages:-

* Efficient searching Algorithm in a sorted list.

Disadvantages:-

* The list of data must be sorted first.
  i.e, this Algorithm only works for sorted data sets.
* Stack memory usage is more.

~~Examples~~

Pseudo Code:-

[returns the position (m) of the Element to be searched (x) in the list. If not Present, returns '0']

```
i: = 1
j: = n
x: = Element to be searched in the list.
P: = List of Elements.
while i ≤ j
      m: = (i+j)/2

      if x < P_m
            j := m-1
      else if x > P_m
            i := m+1
      else return m
return 0
```

Example:-

Search (15) in the list {9, 15, 17, 25, 47, 50, 48, 70}

left sublist / right sublist

9	15	17	25	47	50	48	70
1	2	3	4	5	6	7	8
i			m				j

$\Rightarrow (x = 15)$

$$m = \frac{1+8}{2} = 4$$

$$P_4 = 25$$

$$\Rightarrow x < P_m$$

$$j = m - 1 = 3$$

ignored sub-list

left, right

9	15	17	25	47	50	48	70
1	2	3	4	5	6	7	8

$\uparrow$ i $\quad \uparrow$ m $\quad \uparrow$ j

$$m = \frac{1+3}{2} = 2$$

$P_2 = 15$

$\Rightarrow x = P_2$

$\Rightarrow$ return $(m=2)$

---

② **Merge Sort:-** [Relocating the maximum and minimum elements in the sequence]

A sorting algorithm in which all the elements of an array are first divided and then sorted individually by merging the divided elements in an order, is known as Merge Sort.

★ The Recurrence Relation for Merge Sort is given by:-

$$\boxed{M(n) = 2M\left(\frac{n}{2}\right) + 2} \Leftrightarrow (n \text{ is even})$$

$$(n \geq 1)$$

here, $M(n)$ represents the total no. of comparisions required to sort the whole given set of (n) elements.

★ Merge Sort can be implemented very

quickly and efficiently with lesser space complexity, by using the recursive implementation.

Algorithm:-

Part ①:- Division

↳ Step ①:- Divide the list into halves by using mid index.

↳ Step ②:- Repeat the step ① for sub lists until all of the elements are divided into (n) parts. [n ⇒ size of list]

[this can be done using the condition (left < right).

[left ⇒ left most index
right ⇒ right most index]

i.e., if (left < right), then divide else, stop dividing.]

Part ②:- Merging [Sorting]:-

↳ Step ①:- Merge back the divided element into sub-lists by applying the reverse procedure of division, which is done by sorting the individual elements in an order (Ascending / Descending)

i.e, if, M [left] > M [right], then swap and merge.

else, just merge [for according order]

Time Complexity:-

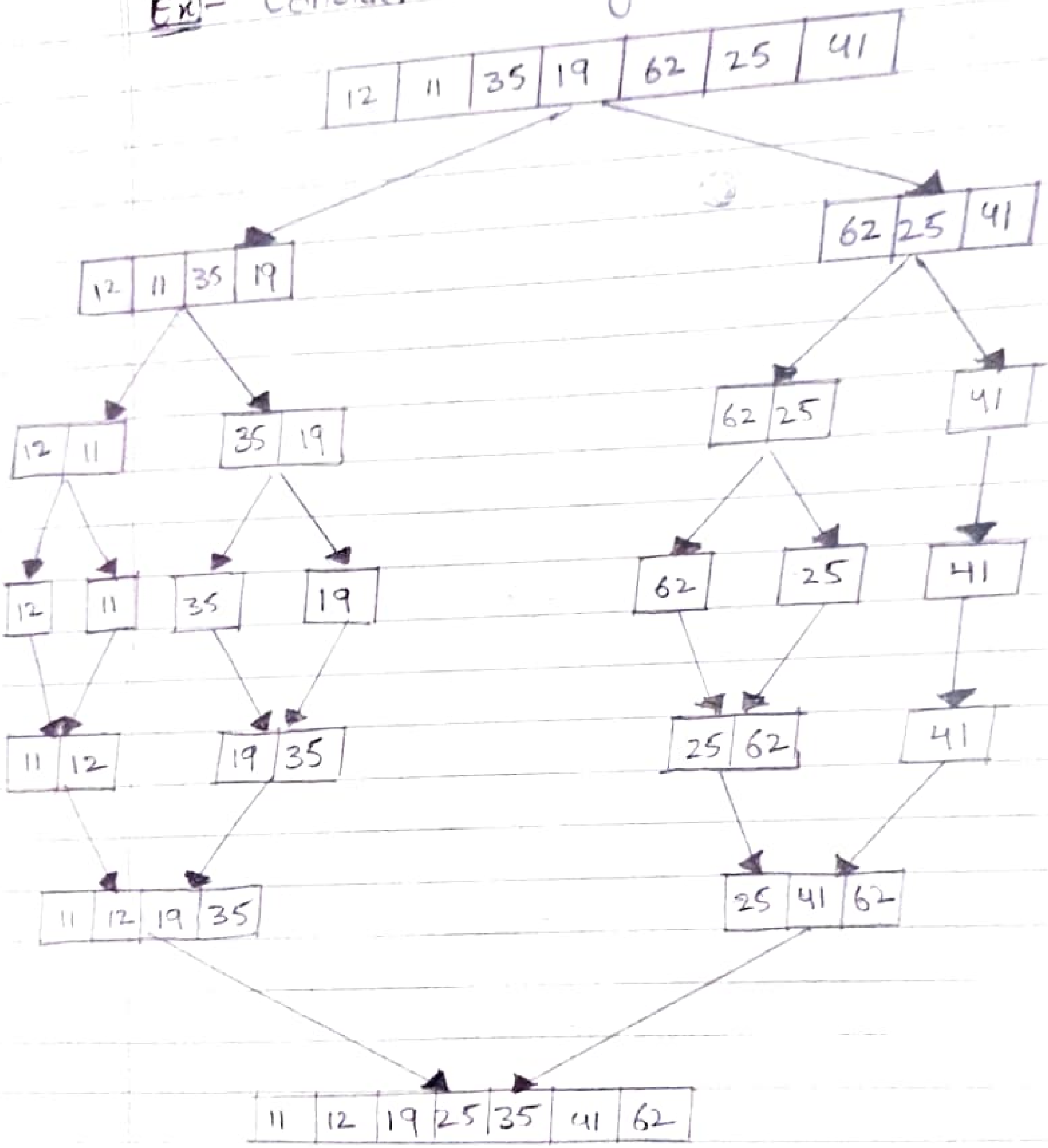
All cases :- O(n log(n))

Space Complexity:- O(n)

Pseudo Code:-

```
MergeSort(M[], left, right){
    if(low < high){
        mid = (left + right)/2
        //divide left sub-Array
        MergeSort(M[], left, mid)
        //divide right sub-Array
        MergeSort(M[], mid+1, right)
        //Merge them back
        Merge(M[], left, mid, right)
    }
}
```

Ex:- Consider an Array[7] = {12, 11, 35, 19, 62, 25, 41}



∴ the final array is:-

array[7] = {11, 12, 19, 25, 35, 41, 62}

(3) Quick Sort :-

Quick Sort is a highly efficient Sorting algorithm that works by partitioning an array into smaller subarrays based on a pivot element, and then recursively Sorting those Subarrays.

* This algorithm picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.
* A pivot is an element of the given array around which the other elements Rotates (Swaps).

Procedure:-

Step(1):- Choose a Pivot

Select a pivot element from the array. The choice of pivot can significantly affect the performance of the algorithm. Commonly, the pivot is chosen as the last element, the first element, or a random element.

Step(2):- Partitioning

Rearrange the elements from the array so that all elements less than the pivot come before it, and all elements greater than the pivot come after it. After partitioning, the pivot element is in its final sorted position. This is called the partitioning operation.

③ QUICK SORT

<u>Step ③</u>:- Recursively Sort Subarrays
Recursively apply the above steps to the subarrays formed by the partition until the entire array is sorted.

<u>Step ④</u>:- Combine Sub arrays.
No additional combining step is needed, as the array is sorted in place during the partitioning step.

<u>Pseudo Code</u>:-

```
quickSort(arr, low, high){
      if(low < high){
            pivotIndex = partition(arr, low, high);
            // ↳ Partition the array
            // Recursively Sort the Subarrays,
            quickSort(arr, low, pivotIndex-1);
            quickSort(arr, pivotIndex+1, high);
      }
}

Partition(arr, low, high){
      Pivot = arr[high]; //choose last Element as Pivot
      i = low-1;
      for(j=low; j<high; j++){
            if(arr[j] < pivot){
                  i++;
                  Swap(arr[i], arr[j]);
            }
```

```
    Swap(arr[i+1], arr[high]);
    return (i+1);
}
```

* Best Case time Complexity :- $O(n \log(n))$

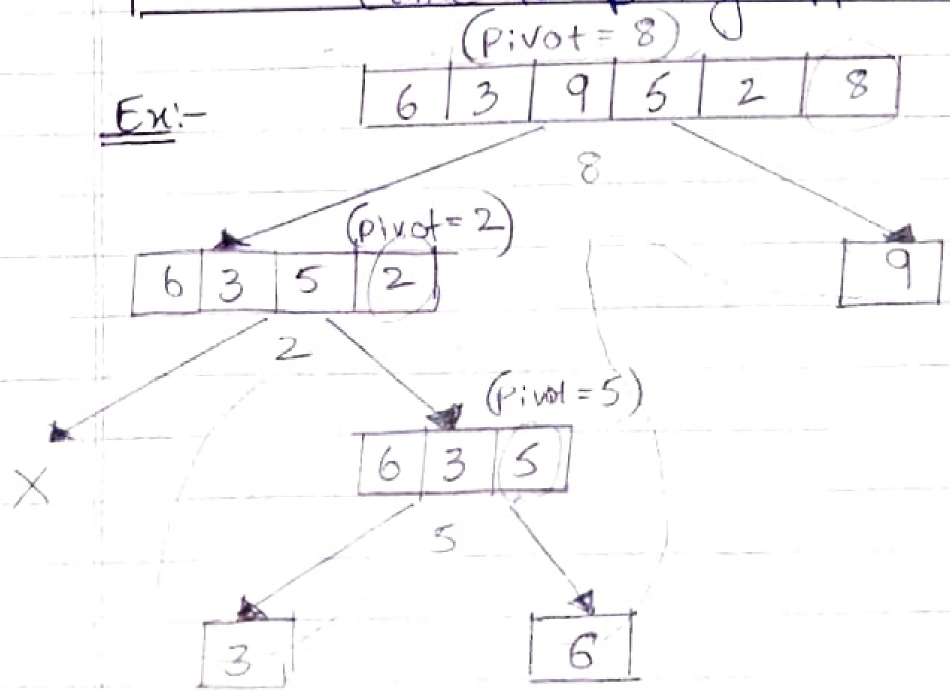
* Avg Case time Complexity :- $O(n \log(n))$

* Worst Case time Complexity :- $O(n^2)$.

∴ Time Complexity is :-

$$\boxed{\begin{aligned} T(n) = &(\text{time for Partitioning}) + \\ &(\text{time for Sorting lower sub array}) + \\ &(\text{time for sorting upper sub array}). \end{aligned}}$$

Ex:-



∴ Sorted Array => | 2 | 3 | 5 | 6 | 8 | 9 |

Strassens Matrix Multiplication

* ~~[illegible]~~

→ using Divide and Conquor technique, A given large matrix is divided/broken into Submatrices until the 2x2 Matrices are formed.

~~[illegible]~~

→ The Solution for this (2x2) matrices are found first, and then the solns to bigger ~~[illegible]~~ Sub Matrices and so on...

→ The moto is to reduce the no. of multiplications, even if the no. of additions had to be increased. Because Multiplications take more Computing time then that of for addition.

* Standard Matrix Multiplication:-

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

where,

$$\left[\begin{array}{l} C_{11} = A_{11} B_{11} + A_{12} B_{21} \\ C_{12} = A_{11} B_{12} + A_{12} B_{22} \\ C_{21} = A_{21} B_{11} + A_{22} B_{21} \\ C_{22} = A_{21} B_{12} + A_{22} B_{22} \end{array}\right.$$

∴ No. of Multiplications = 8

No. of Additions = 4

Algorithm [Pseudo Code]:-

```
for (i=0; i<n; i++){
        for (j=0; j<n; j++){
                C[i,j]=0;
                for (K=0; K<n; K++){
                        C[i,j] += A[i,K] * B[K,j];
                }
        }
}
```

∴ time complexity $= O(n^3)$
(n = order of matrix)

* **Strassen's Multiplication Method:-**

Strassen's Multiplication Method provides a set of formulae for $C_{11}, C_{12}, C_{21}, C_{22}$.

→ These formulae returns the same results as that for standard method, but the no. of multiplications are reduced. Hence, time complexity will be less. Hence, the algorithm is optimized.

→ The formulae are:-

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$

where,

$$\left[\begin{array}{l} P = (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q = (A_{21} + A_{22}) B_{11} \\ R = A_{11}(B_{12} - B_{22}) \\ S = A_{22}(B_{21} - B_{11}) \\ T = (A_{11} + A_{12}) B_{22} \\ U = (A_{21} - A_{11})(B_{11} + B_{12}) \\ V = (A_{12} - A_{22})(B_{21} + B_{22}) \end{array}\right.$$

$\therefore$ No. of Multiplications $= 7$

No. of Additions $= 18$ //

$\therefore$ Time Complexity $= O(n^{\log_2 7}) \approx O(n^{2.81})$ //