Design And Analysis Of Algorithm notes part 1

Design And Analysis Of Algorithm (Islamic University of Science and Technology)

Module - 1

Dt - 10/01/18

ALGORITHM:

Algorithm is a step by step method to solve a problem.

characteristics of algorithm:

1. Input : Algorithm must contain '0' or more input.
2. Output : Algorithm must contain '1' or more output.
3. Finiteness : Algorithm must complete after finite no. of steps.
4. Definiteness : The step of the algorithm must be defined precisely or clearly
5. Effectiveness : The step of an algorithm must be unambiguous effective (can be done successfully)

Analysis of Algorithm:

Analysis of algorithm depend upon the time & memory taken by the algorithm.

Time Complexity:

The amount of time required by the algorithm is called time complexity.

Space Complexity:

The amount of space / memory required by the algorithm is called as Space complexity.

Growth functions:

	2^n	$n!$
$n=1$	2	1
$n=2$	4	2
$n=3$	8	6

$n! > 2^n \; \forall \; n \geq 4$

$n!$ has higher growth rate than 2^n

Q: Arrange the following according to the increasing order of growth rate.

$n, 2^n, n\log n, n!, n^3, n^5, n^2, 1$

Ans: $1, n, n\log n, n^2, n^3, n^5, 2^n, n!$

constant ≤ Logarithm ≤ polynomial ≤ ≤ exponential ≤ factorial

Date - 13/1/18

⇒ There are 3 notations for time complexity

1. Big oh (O) notation :

Let f(n) & g(n) are two functions

$f(n) = O(g(n))$ [read as f(n) is big oh of g(n)]

when $f(n) \leq c \cdot g(n)$. Here c = constant

Example:

We say that it requires Max^m 1hr from Cuttack to Bhubaneswar.

2. Omega (Ω) notation

Let f(n) & g(n) are two functions

$f(n) = \Omega(g(n))$ [read as f(n) is Omega of g(n)]

when $f(n) \geq c \cdot g(n)$

where, c is constant.

Example:

We say that it require Min^m 30 mins from Cuttack to Bhubaneswar.

3. Theta (Θ) notation:

Let f(n) & g(n) are two functions.

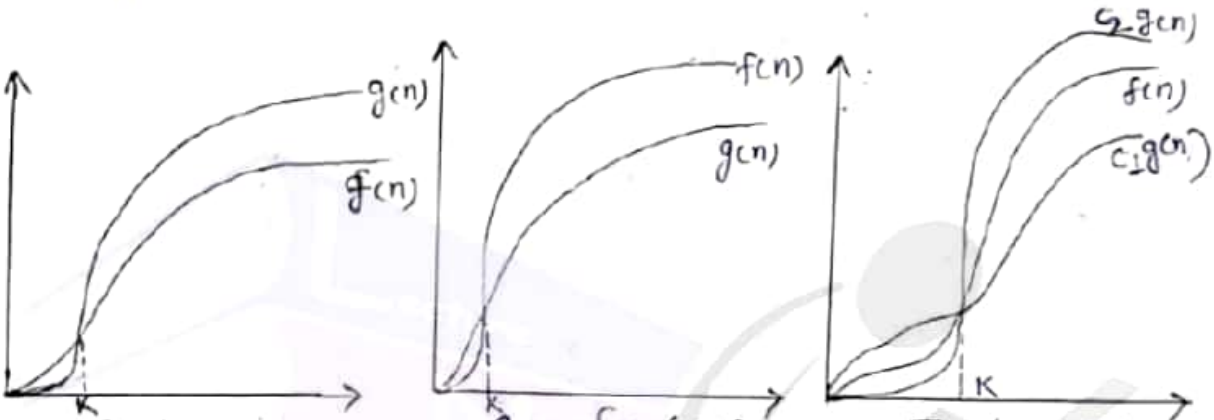
$f(n) = \Theta g(n)$ [read as f(n) is theta of g(n)]

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Here c_1, c_2 are constants.

Examples:

We say that it require approximately 40 mins from cuttack to Bhubaneswar.



Big oh (O) — $f(n) \leq g(n)$

Omega (Ω) — $f(n) \geq g(n)$

Theta (Θ)

Example:

$f(n) = n+5$

$f(n) = 2n+3$

$f(n) = n^2+6$

$f(n) = n^3+2n^2+3$

$f(n) = n+6n+3$

Big oh → Low upper bound / Max^m

Omega → High Lower bound / Min^m

Ex:

$2n+3 = O(n)$

$= O(n^2)$

$= O(n^3)$

Q Prove that $5n+3 = O(n)$

Ans: $f(n) = 5n+3$

$5n+3 \leq 6 \cdot n$

$= O(n)$

$\because f(n) \leq c \cdot g(n)$ (c: constant)

Q Prove that $6n^2+2n+3 = O(n^2)$

Ans: $6n^2+2n+3 \leq 7 \cdot n^2$

$= O(n^2)$

Q Prove that $5n+3 = \Omega(n)$

Ans: $5n+3 \geq 4 \cdot n$ here 'c' = 4

$= \Omega(n)$

Q Prove that $6n^2+2n+3 = \Omega(n^2)$

Ans: $6n^2+2n+3 \geq 5 \cdot n^2$ here c = 5

$= \Omega(n^2)$

'c' is decided by user. Means user have to put a value for 'c'.

Q Prove that $5n+3 = \Theta(n)$

Ans: $4 \cdot n \leq 5n+3 \leq 6 \cdot n$

c_1 c_2

$= \Theta(n)$

Recurrence:

→ The word recurrence is derived from "recursion" or "repeatation"

→ Recursion is a technique which call the same function repeatedly.

Example: Factorial problem can be solved using recursion.

because factorial(n) = n * factorial(n-1)

$5! = 5.4!$
$= 4.3!$
$= 3.2!$
$= 2.1!$
$1! = 1$

→ Algorithm which use recursion is called recursive algorithm.

→ Time Complexity of recursive algorithm is given by a formula or equation called 'recurrence'.

→ Here the time complexity is denoted by T(n)

$T(n) = T(\frac{n}{2}) + O(n)$

D1 :- 17/01/18

There are 3 method to solve the recurrence,

1. Master Method
2. Substitution Method
3. Recursion Tree Method

1) Master Method :

$$T(n) = a\,T(n/b) + \Theta(n^k [\log n]^p)$$

Case-1 : If $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$

Case-2 : If $a = b^k$ then,

(a) $P > -1 \Rightarrow T(n) = \Theta(n^{\log_b a} \log n^{P+1})$

(b) $P = -1 \Rightarrow T(n) = \Theta(n^{\log_b a} \log(\log n))$

(c) $P < -1 \Rightarrow T(n) = \Theta(n^{\log_b a})$

Case-3 : If $a < b^k$ Then,

(a) $P \geq 0 \Rightarrow T(n) = \Theta(n^k \log n^P)$

(b) $P < 0 \Rightarrow T(n) = \Theta(n^k)$

Questions Bank on Master Methods

Example-1:

$T(n) = 4T(n/2) + \log n$

Solve the above recurrence using master method.

Ans: We know that, $T(n) = aT(n/b) + \Theta(n^k \log^p n)$

Here, $a = 4$
$b = 2$
$k = 0$
$p = 1$

$\because n^{k=0} \log n^{p=1} = \log n$

Find a, b^k,

$a = 4$, $b^k = 2^0 = 1$

$a > b^k \Rightarrow$ It is case 1

$T(n) = \Theta(n^{\log_b a})$
$= \Theta(n^{\log_2 4})$
$= \Theta(n^{\log_2 2^2})$
$= \Theta(n^2 \log_2 2) = \Theta(n^2)$ Ans

Example-2:

$T(n) = 3T\frac{n}{3} + \frac{n}{\log n^2}$

Solve the above recurrence using Master Method.

Ans: We know that

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

Here, $a = 3$
$b = 3$
$k = 1$
$p = -2$

$a = 3$, $b^k = 3^1 = 3$

$a = b^k \Rightarrow$ It is case 2

$p = -2 \Rightarrow$ It is case 2 (c)

Here $P < -1$ so case 2(c),

$$T(n) = \Theta(n^{\log_b a})$$
$$= \Theta(n^{\log_3 3})$$
$$= \Theta(n) \quad [\because \log_3 3 = 1]$$ Ans

Example-3 : Dt : 18/01/18

$$T(n) = 2T(n/2) + n$$

Solve the above recurrence by Master Method.

Ans: We know that,

$$T(n) = aT(n/b) + \Theta(n^k \log n^p)$$

Here,

$a = 2$
$b = 2$ $\quad \because n^1 \log n^0 = n$
$K = 1$
$P = 0$

$a = 2$, $b^K = 2^1 = 2$

$a = b^K \Rightarrow$ It is case 2

$P = 0$, Here $P > -1 \Rightarrow$ case 2(a),

$$T(n) = \Theta(n^{\log_b a} \log n^{P+1})$$
$$= \Theta(n^{\log_2 2} \log n^{0+1})$$
$$= \Theta(n \cdot \log n)$$ Ans

2. Substitution Method:

Substitution method has two steps,

1. Assume that solution is correct.
2. Prove this assumption by substitution method.

Example-1

$$T(n) = \begin{cases} 2T(n/2) & \text{for } n>0 \\ 0 & \text{for } n=0 \end{cases}$$

Prove that $T(n) = O(n\log n)$ by substitution method.

Ans: Assume that $T(n) = O(n\log n)$

$\Rightarrow T(n) \leq c \cdot n\log n$ [By defin $f(n) = O(g(n))$ $\Rightarrow f(n) \leq c \cdot g(n)$]

$\Rightarrow T(n/2) \leq c\frac{n}{2}\log\frac{n}{2}$ — (1)

Given that,

$$T(n) = 2T(n/2)$$

$\Rightarrow T(n) \leq 2 \cdot c\frac{n}{2}\log\frac{n}{2}$ [Putting the value of $T(n/2)$ from eqn (1)]

$\Rightarrow T(n) \leq cn\log n/2$

$\Rightarrow T(n) \leq cn(\log n - \log 2^2)$ [$\log\frac{a}{b} = \log a - \log b$]

$\Rightarrow T(n) \leq cn(\log n - 1)$

$\Rightarrow T(n) \leq cn\log n - cn$.

$\Rightarrow T(n) \leq cn\log n$ [cn is neglected for small value]

$\Rightarrow T(n) = O(n\log n)$

(Proved)

Example-2 :

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

Prove that $T(n) = O(2^n)$ by substitution method.

Ans: Assume that $T(n) = O(2^n)$

$\Rightarrow T(n) \leq c \cdot 2^n$ [By defn]

$\Rightarrow T(n-1) \leq c \cdot 2^{n-1}$ — (1)

Given that,

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

$\Rightarrow T(n) \leq 2 \cdot c2^{n-1} + 1$

$\leq c \cdot 2^n + 1$

$\leq c \cdot 2^n$ [1 is neglected since it is a small value]

$\Rightarrow T(n) = O(2^n)$ (Proved)

Dt - 20/01/18

Recursion Tree method :

This method is used to guess the solution.

→ Recursion tree gives a graphical view of recurrence.

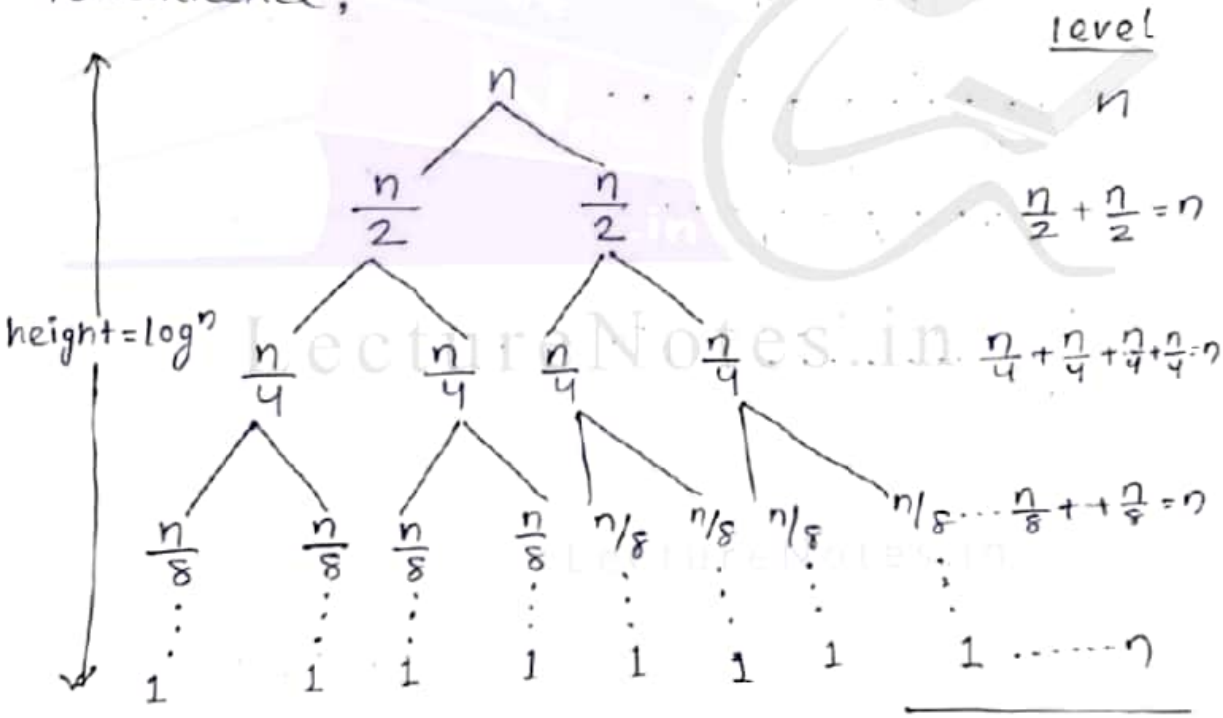
Dt - 25/01/18

Consider the following recurrence ;

$$T(n) = 2T(n/2) + O(n)$$

$$\Rightarrow T(n) = T(n/2) + T(n/2) + O(n)$$

The recurrence tree for the above recurrence,



Total time = $(n + n + \ldots\ldots)$ ↓ log n times

$= n \log n$

cost of level = addition of nodes present in the level.

total cost = addition of the all level.

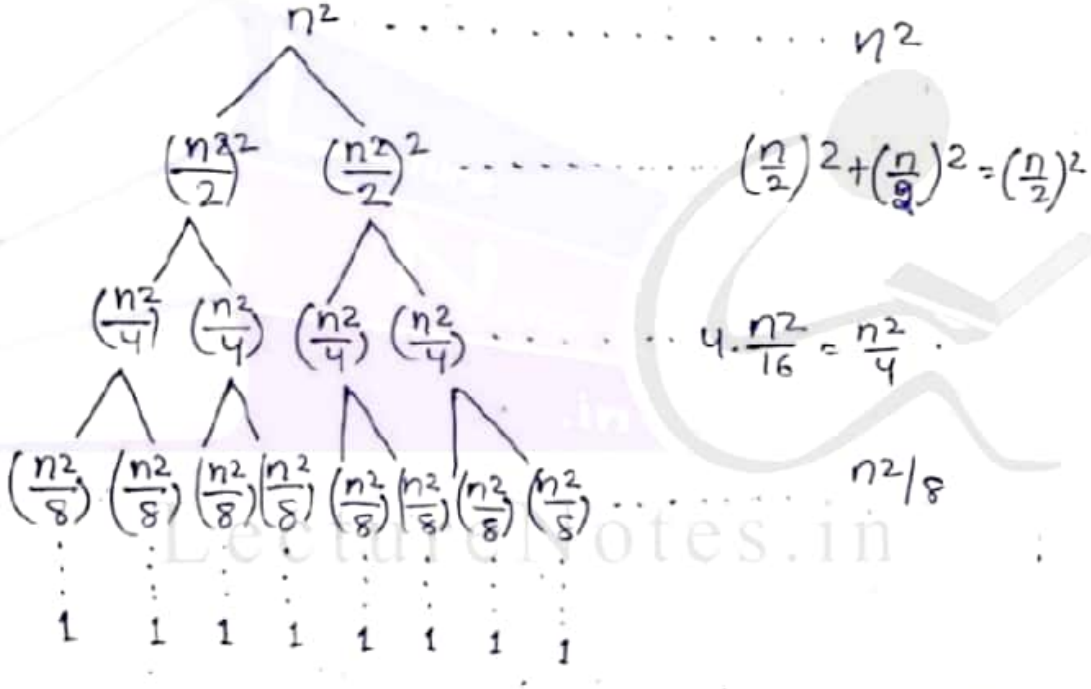
[∵ cost indicates time]

Example:

Consider the following recurrence

$T(n) = 2T(n/2) + O(n^2)$

Design the recursion tree. Find total time complexity.



Total time $= n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} \cdots)$

$= n^2(1 + 1/2 + 1/4 + 1/8 + \cdots)$

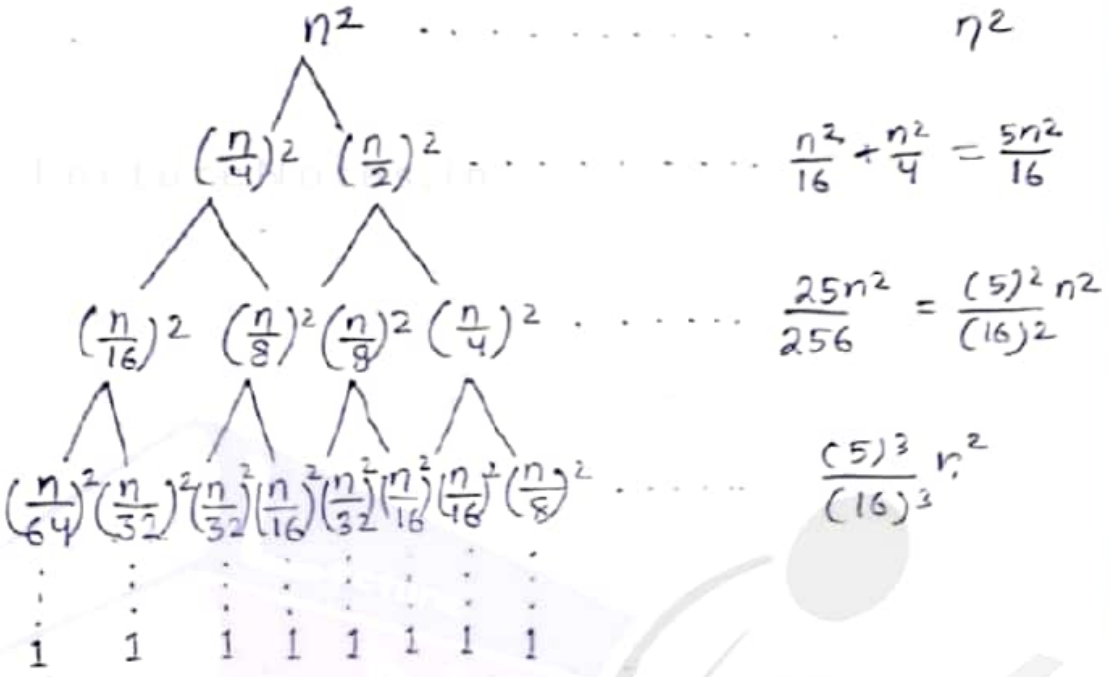
$= n^2$ (constant)

$= O(n^2)$

Example:

Draw recursion tree for following

$T(n) = T(n/4) + T(n/2) + n^2$



Total time $= n^2 (1 + 5/16 + 5^2/16^2 \ldots$

$= n^2$ (constant)

$= O(n^2)$

Dt - 29/01/18

Little oh (o) Notation:

Let f(n) & g(n) are two functions

$f(n) = o\,g(n)$ [read as f(n) is Little oh of g(n)]

when $f(n) < c \cdot g(n)$ Here c = constant

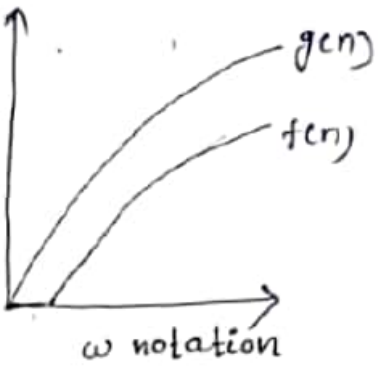
Little omega (ω) Notation:

Let f(n) & g(n) are two functions

$f(n) = \omega\, g(n)$ [read as f(n) is omega of g(n)]

when $f(n) > c \cdot g(n)$

where, c is constant.



$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = o(g(n))$

$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty \Rightarrow f(n) = \omega(g(n))$

Divide and conquer method:

→ Divide and conquer is also called DANDC

→ It is one of the algorithm design method

→ DANDC has 3 parts

1. Divide the problem into number of subproblems.
2. Conquer means solve the subproblems recursively
3. Combine the solution of subproblems

Example:

Binary search, quick sort, merge sort.

Note:

DANDC is normally recursive

Binary Search :

It searches for an element from given state set of elements.

Condition :

The set of elements must be sorted
↓
[increasing or decreasing order]

Steps :

step-1 : Enter the no. of elements in increasing or decreasing order.

step-2 : Enter the searching element.

step-3 : Find the middle element.

step-4 : Compare searching element with middle element according to following case,

i) Searching element is less then middle element. (search in left side of middle element.

ii) Searching element is greater then middle element (search in right side of middle ele)

iii) Searching element is equal to middle element (element found)

step-5 : Repeat step-3 & step-4 until Searching is successful

Dt- 31/01/18

Example :

0	1	2	3	4	5	6
12	14	17	19	25	28	35

mid

Let searching element = 25

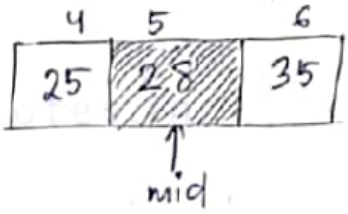
Searching element is in the right side of middle element.

Ans: low = 0
high = 6

$$mid = \frac{low + high}{2} = \frac{0+6}{2} = 3 \ (19)$$

Let Searching element = 25

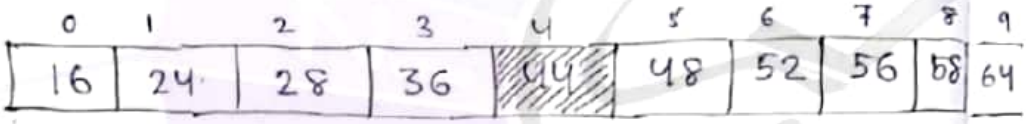
Searching element is in the right side of midle element.



$$mid = \frac{4+6}{2} = 5$$

Searching element is in the leftside of the middle element

Q Explain Binary Search for following elements

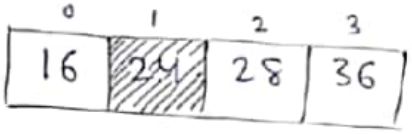


Searching element is 24.

Ans: low = 0
high = 9

$$mid = \frac{low + high}{2} = \frac{0+9}{2} = 4.5 \cong 4$$

Searching element is in the left side of middle element.

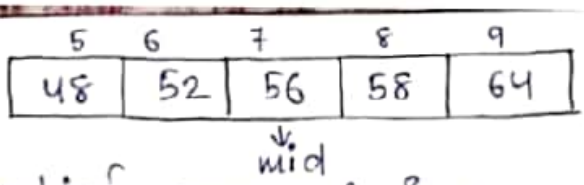


$$mid = \frac{0+3}{2} = 1.5 \cong 1$$

→ Searching element is 59.

$$mid = \frac{0+9}{2} = 4.5 = 4 \ (44)$$

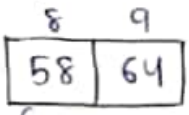
Searching element is in the right side of middle element.



$mid = \frac{5+9}{2}$

$= 14/2 = 7$

Searching element is in the right side of the middle element.



$mid = \frac{8+9}{2} = \frac{17}{2} = 8.5 \approx 8$

So searching element present in right side of middle element.

$mid = \frac{9+9}{2} = \frac{18}{2} = 9$

Searching element is in leftside of middle element.
But no element available ⇒ element not found.

Q: Write an algorithm for binary search.

Ans:

```
Algorithm Binary Search (A, ele)
{
    // A is an Arrray of 'n' elements
    // ele is the searching element
    low = 0      // low is the index of 1st element
    high = n-1   // high is the index of last element
    while (low <= high)
    {
        if ele < A[mid]      // ele: searching element; A[mid]: middle element
            high = mid - 1
        else if ele > A[mid]
            low = mid + 1
        else
            Print "element found"
    }
    Print "element not found"
}
```

Q: Write the recurrence eqn of binary search?

Ans:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

$T(n)$: Time to Search n elements; $T\left(\frac{n}{2}\right)$: time to search $\frac{n}{2}$ element; $O(1)$: 1 comparison with middle

Time complexity of algorithm is analyzed under 3 cases,

1. Best case : This case take minimum time
2. Worst case : This case take maximum time
3. Average case : This case take average time.

Analysis of Binary Search :

1. Best case : In this case searching element is in the middle position. So, time complexity = O(1) (1 comparison)
2. Worst case : In this case searching element is in the first or last position. So, time complexity = O(log n)
3. Average case : In this case searching element is not in first or last position.
   So time complexity = O(log n)

Q what is the time complexity of Binary Search
Ans: O(log n)

Q what is the time complexity of Linear Search
Ans: O(n)

Q Which is better : Linear Search or Binary Search.
Ans: Binary Search is better then Linear Search. Because it takes less time then Linear Search.

$O(\log n) < O(n)$

↑ Time for binary Search ↑ Time for Linear Search

Merge Sort :

→ Merge means combine

→ Sorting means arranging elements in increasing or decreasing order.

→ Merge sort has two operations,

1. Divide
2. Merge

Consider an array of n elements,

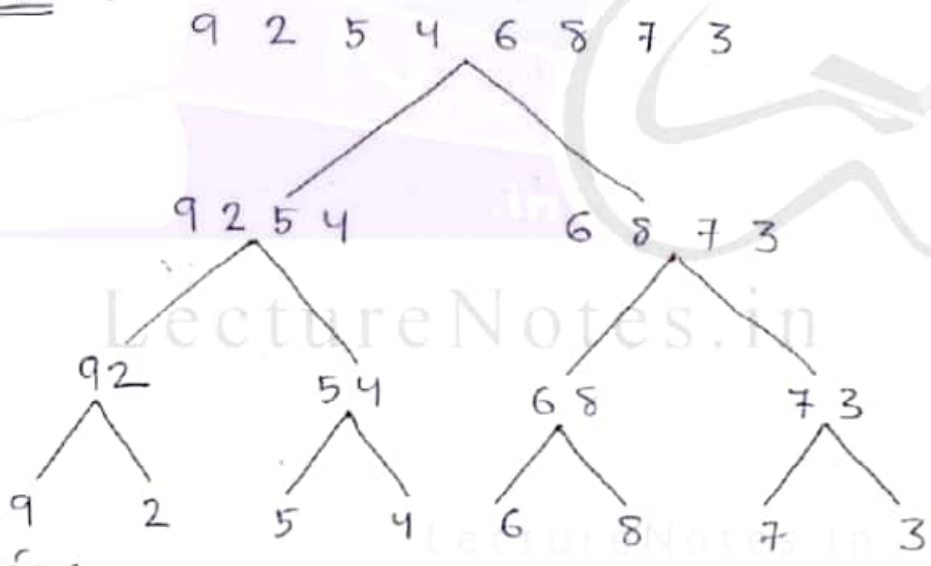
1. Divide the array into sub array recursively
2. Merge the subarray recursively.

Merging operation creates sorted elements. Hence, the name is merg sort

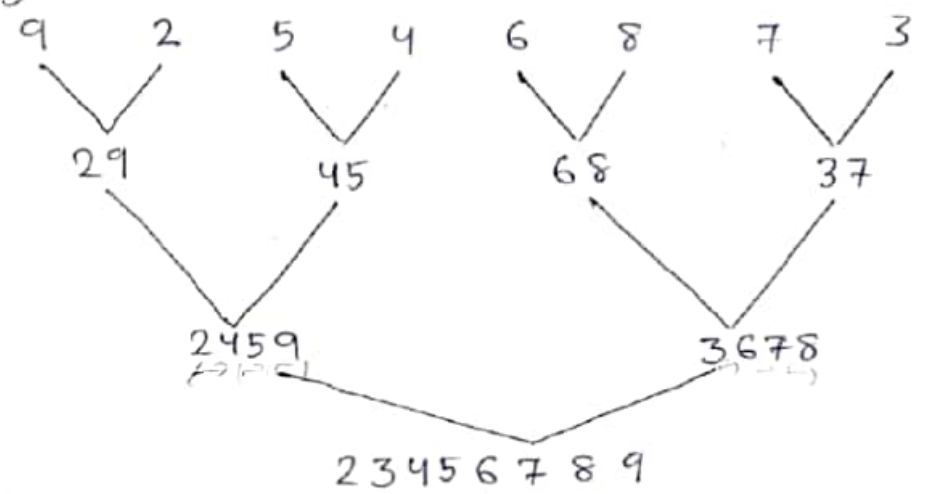
01-05/02/18

Example : 9 2 5 4 6 8 7 3

Divide :



Merge :



Divide Operation :

1. Divide the array into two sub array at mid position.

   Subarray 1 is from low to mid.
   Subarray 2 is from mid+1 to high.

2. Divide the subarray recursively until more than one element is present. (low < high)

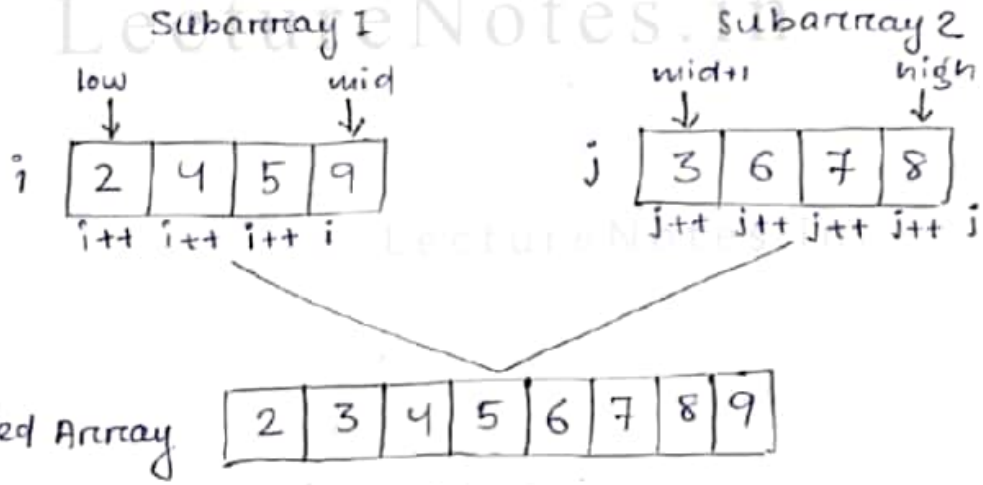
Merge Operation :

// A[i] = element of subarray 1
// A[j] = element of Subarray 2

Compare A[i] and A[j] as following

```
if A[i] < A[j]
        store A[i] in merged array
Else
        store A[j] in merged array
```

Example :



Compare two elements at position i and j.
The smaller element is stored at position k.
Increment the value of i, j, k accordingly.

Q. Write a procedure (Algorithm or pseudo code) for divide and merge operation.

```
Algorithm divide (low, high)
{
  // low = index of 1st element of array
  // high = index of last element of array

  low = 0
  high = n-1          ┌ divide until more than one element
  while (low < high)
  { // divide the array of mid position.

     divide (low, mid)
          // Subarray 1 is from low to mid
     divide (mid+1, high)
          // Subarray 2 is from mid+1 to high.
     merge (low, mid, high)
  }
}
```

01-07/02/18
01-08/02/18

```
Algorithm merge (low, mid, high)
{
  // A is an array
  // B is an array to store result (merged array)
  while (i<=mid and j<=high)
  {
                ┌ element at position i   ┌ element at position j
     if (A[i] < A[j])
     {
          B[k] = A[i]
          k++
          i++
     }
     else
     {
          B[k] = A[j]
          k++
          j++
     }
  }
}
```

Analysis of Merge sort:

Merge sort apply a common technique in all cases.

Hence, Merge sort has one case for all set of Problems.

=> Merge sort does not have best, worst or average case.

The recurrence, equation of amerge sort is given by.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

- $T(n)$: Time to sort n elements
- $2T\left(\frac{n}{2}\right)$: time to sort 2 subarray of size $n/2$
- $O(n)$: merge the 2 subarray

Solve the above recurrence by master method

$$T(n) = O(n \log n)$$

Q: what is the time complexity of merge sort.

Ans: The time complexity of merge sort is $O(n \log n)$.

Quick Sort:

Quick sort is an "Divide and Conquer" (solve) algorithm

Steps:

1. Select a pivot element. Pivot means target. Any element can be taken as pivot.
2. Partition Operation: Partition means divide.
   - → Partition operation devide the array into 2 subarray. [Left & right subarray]
   - → Partition operation places the point element at proper position.

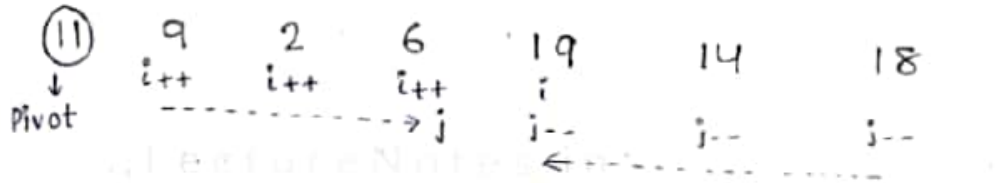
     that means, all element before pivot is smaller.

     and all element after pivot is grater.

3. Recursively, apply quicksort to subarrays

left subarray right subarray

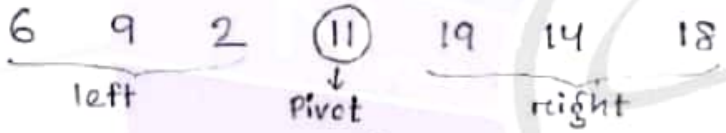
Example 1 — 01-10/02/18



9 < 11 => i++
2 < 11 => i++
6 < 11 => i++
19 ≮ 11 => stop i

18 > 11 => j--
14 > 11 => j--
19 > 11 => j--

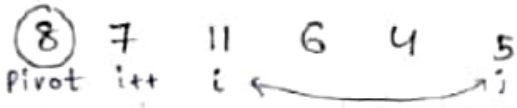
Now i and j cross each other => stop
Swap 11 and 6 [i.e swap pivot and element at j]
Elements after swapping are shown below.



Now, pivot is placed in proper position

Example-2

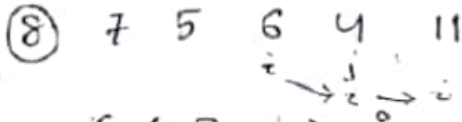
Arrange following elements by appling quickshot.



7 < 8 => i++
11 ≮ 8 => stop i

9 > 8 => j--
5 ≯ 8 => stop j

Swap the element at i and j. elements after swapping as shown below.



6 < 8 => i++
4 < 8 => i++

Now, i and j cross each other => stop.

Swap 8 and 4 [i.e swap pivot and element at j]
Elements after swapping are shown below,

Write an algorithm for quick sort.

Algm:

// A is an array of n elements
// Low = index of 1st element of array.
// high = index of last element of array

step 1 : Low = 0
high = n-1

step 2 : Consider 1st element is pivot

step 3 : i = 1
j = high

step 4 : Continue i++ while (A[i] < pivot) [element at i]
Continue j-- while (A[j] > Pivot) [element at j]
[i moves in forward direction
j moves in backward direction]

step 5 :
Swap A[i] and A[j]
make i++ and j--

step 6 :
Repeat step4 and step5 until i and j Cross each other.

step 7 :
Swap pivot and A[j]
Now, pivot is placed in proper position.

Step 8 :
Divide the array into 2 subarray.
Left subarray = Elements present in the left side of pivot
Right Subarray = Elements present in the right side of pivot.

Step 9: Apply Quickshot to both subarray recursively.

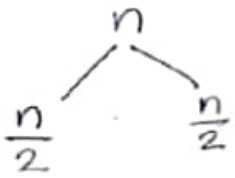
Analysis of Quick sort:

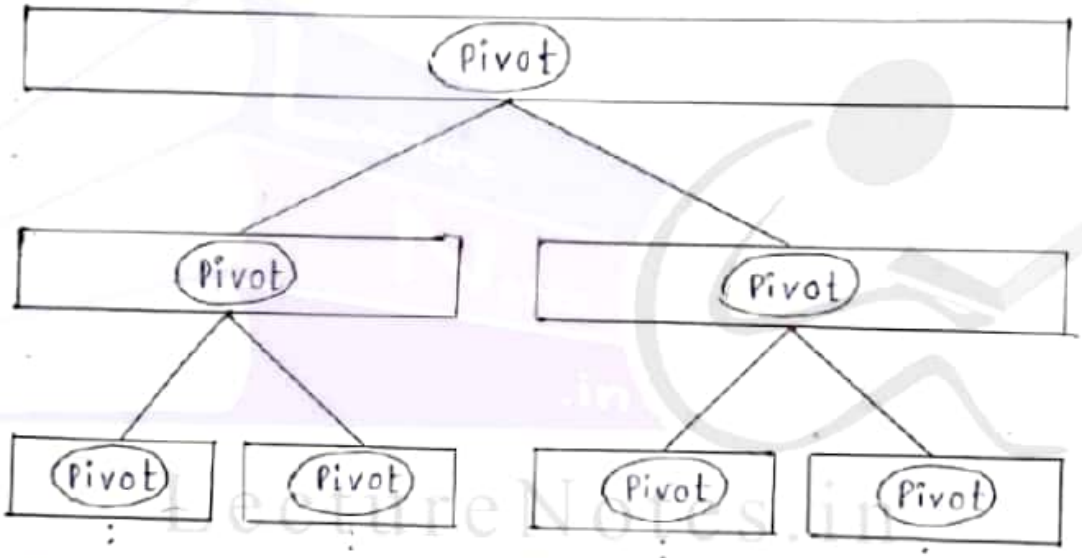
Time complexity depends on the position of pivot element.

1) Best Case:

→ Pivot is placed in the middle position.

→ 'n' elements are devided into $n/2$ and $n/2$





The recurrence eqn is

$$T(n) = T(n/2) + T(n/2) + O(n)$$

Solving this recurrence we get,

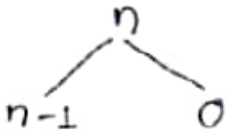
$$T(n) = O(n \log n)$$

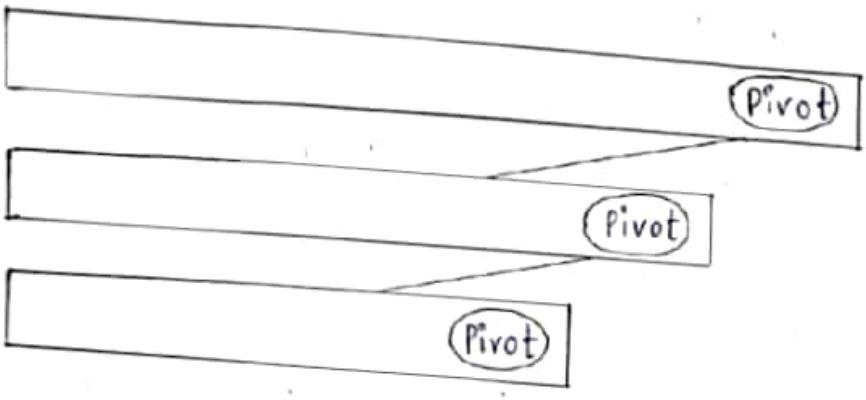
∴ Time complexity = $O(n \log n)$

2) Worst Case:

→ Pivot is placed in the first or last position

→ 'n' elements are devided into $(n-1)$ and 0





Recurrence eqn is $T(n) = T(n-1) + O(n)$

Solving this recurrence we get,

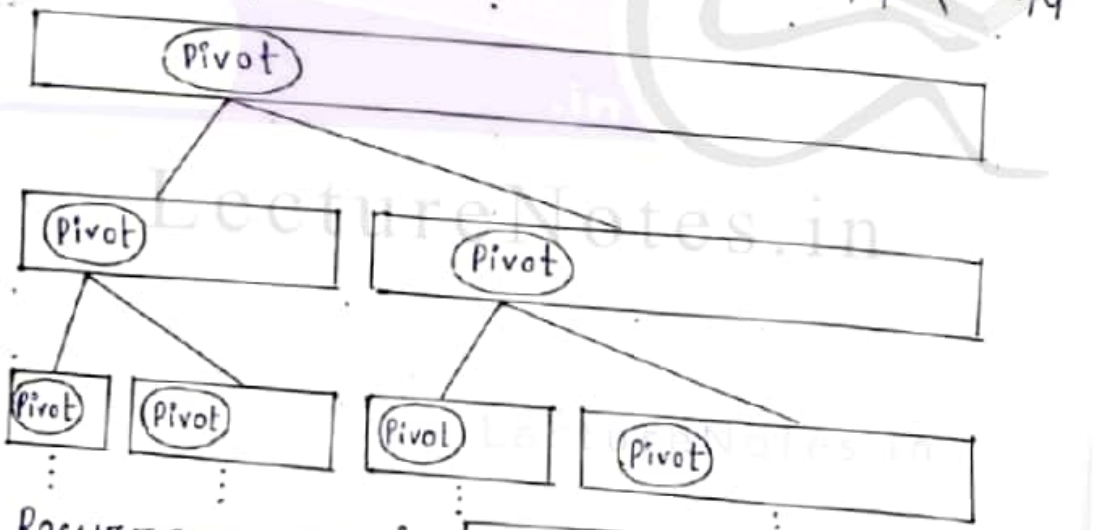
$T(n) = O(n^2)$

∴ Time complexity = $O(n^2)$

3. Average Case:

Pivot is not in middle, first or last position

Let, n elements are divided into n/4 & 3n/4



Recurrence eqn is $T(n) = T(n/4) + T(3n/4) + O(n)$

Solving this recurrence we get,

$T(n) = O(n \log n)$

∴ Time complexity = $O(n \log n)$