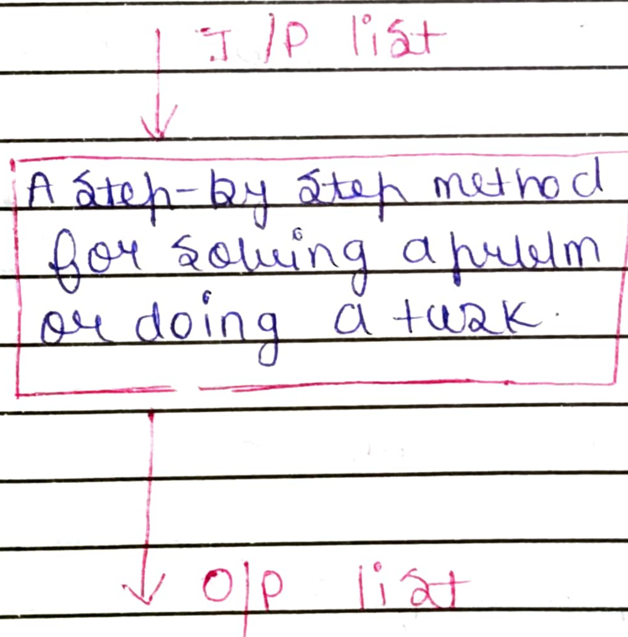
Analysis & design of algorithm

Algorithm :→:

It is step by step process performing some operations (or actions).

It takes I/P list & process step by step method for solving a problem & gives the O/P as a list.

Algorithm Diagram :→



There are two criteria for judging Algorithm

→ Correctness

→ Efficiency

Algorithm is independent of any programming language.

Available at : JSA, Palas Complex, Khajuri Bazar, Indore

Common terms used in algorithms:⇒:

(i). Variable:⇒

They are used in writing the algorithm.

variable are the name of memory location where we store data.

Ex ⇒ a, e, i, o, u.

(ii). Data-types:⇒

"set of variables takes it's values".

3. Statement:⇒:

"lines of codes".

Properties of Algorithm:⇒

(i). Input:⇒ Input refers to

"Algo. uses values from a specific set".

(ii). **Output:** ⇒ It specifies for each I/P or at least one quantity is produced.

(iii). **Precision:** ⇒ It's steps or precisely defined.

(iv). **Correctness:** ⇒ whatever the I/P is defined further that O/P is correct.

(v). **Fitness:** ⇒ Algo. produces O/P after finite no. of steps for each I/P.

(vi). **Determinasion:** ⇒ Result should be guaranteed.

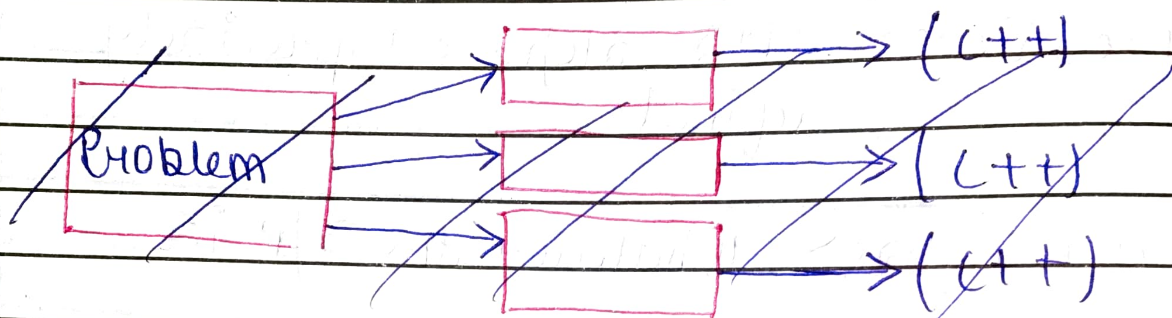
7. **Generality:** ⇒ Procedure apply to all problems not a special subset.

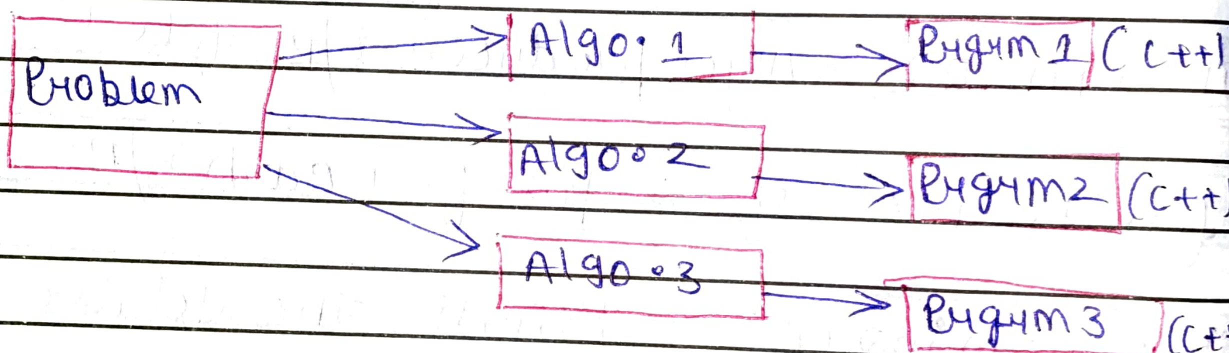
8. **Definiteness** → Steps of algo must be clear & unambigious.

9. **Feasiblity** → Algo must be terminate after finite no. of steps

Analyzing algorithms:→

Why we Analyze the Algorithm:→





→ Firstly suppose we have to find the top student in the class so we have problem & it can be divided into 3 sub-sections to solve problem in easier way. And we have also find that which algorithm is best so we ~~can find with the analyzing the algorithm.~~ can analysis the algorithm.

Algorithm Analysis helps us to determine which Algorithms is most efficient in terms of time & space consumed.

→ For a given problem, there are many ways to design algorithm for it.

→ Analysis of algorithm helps to determine which algorithm should be chosen to solve the problem.

Two efficiency / complexity

→ Performance analysis of algorithm :⇒

Time complexity ⇒ : T.C. of an algorithm is the total time required by the program to run till its completion.

Space complexity ⇒ S.C. is the total space required by an algo. to run till its completion. S.C. is required in situations when limited memory is available for multi users.

→ Time & space complexity depends on lot of things like hardware, OS, processor, etc.

→ we don't consider any of these factors while analyzing the algo. we will only consider the execution time of an algo.

Ex:=> swapping of 2 numbers =>:

Using 3rd variable:

a=2;
b=3;

t=a;
a=b;
b=t;

} more space

Without using 3rd variable:

a=2;
b=3;

a=a+b;
b=a-b;
a=a-b;

} less space

one more storage bach jayega

Asymptotic Notation:=>

→ Asymptotic notations are used to represent the complexity of an algorithm. [Time complexity]

→ With the help of asymptotic notations we can analyze run-time performance of an algorithm.

→ Asymptotic notations are used to describe the asymptotic (approx.) running time of an algo. by defining a time func. f(n) where n is the input size.

→ Usually, the time required by an algorithm falls under 3 types →:

Best case:⇒ Minimum time required for program Execution

Average case:⇒ Average time required for program Execution.

Worst case:⇒ Maximum time required for program Execution

For Ex ⇒ linear search

10 , 50 , 90

If we search element 10 so minimum time → B. case (at 1st posn).

If we search element 90 so max. time ⇒ W. case

If we search element 50 so avg. time = av. case

↳ we have to compare with all element for n comparision n time will required

a = 2, b = 3
a = a + b
= 5
b = a − b
b = 5 − 3
b = 2
a = a − b
= 5 − 2
= 3

→ With the help of asympto notations we can compare the performance of various algorithms & such an analysis is independent of machine time, Programming style etc.

Types of asymptotic Notations :→

Commonly used asymptotic notations to calculate the running time complexity of an algo. are as follows -:

1. O-Notation (Big-oh Notation) (W.B.)

2. Ω - Notation (Big-Omega Notation (L.B

3. Θ Notation (Theta-Notation) (T.B)

Ex→ in linear Search
10 50 90

10 element → 1st posn → so represented by Ω[1]. (B.C.)

90 element → compare all → O(n) (W.C)

50 element → Average → Θ(n) (A.C.)

1. O-Notation (Big-oh Notation) ⇒ :

→ This Notation is used to express the upper bound of an algorithm running time.

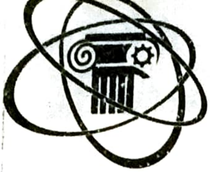
→ It represent the worst case of an algorithm's time complexity i.e. the largest amount of time an algorithm can possibly take to complete.

Defn →
→ A function $f(n) = O(g(n))$, if there exists a value of (+) integer n & n_0 & positive constant 'c'

such that

$$f(n) \le c \cdot g(n) \text{ for all } n \ge n_0$$

Hence function $g(n)$ is an upper bound for function $f(n)$, as $g(n)$ faster than $f(n)$.



$c > 0$
$n \geq 1$

Ex→ :

$f(n) = 2n^2 + 5n + 1$

$g(n) = n^2, \quad c = 8$

leading Term $= n^2$

$f(n) \leq c \cdot g(n)$

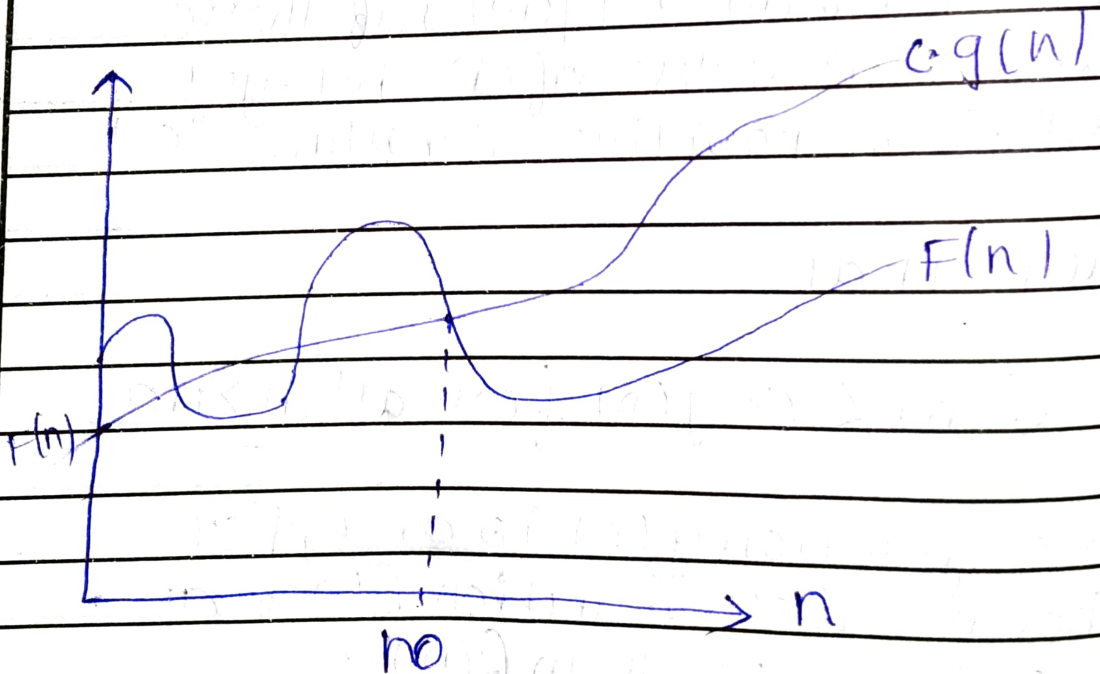
$\therefore f(n) = O(n^2) \quad \forall n \geq 1$

$2n^2 + 5n + 1 \leq 8n^2$

$n=1 \quad 2\times1 + 5\times1 + 1 \leq 8n^2$

$8 \leq 8$ → True

↳ less than



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

$f(n) = O(n^2) \ \forall \ n \geq 3$

Putting values $n = 1, 2, 3$
$c = 4, \ g = n^2$

(ii) Big omega notation (Ω) ->

-> It represents the lower bound of an algorithm running time.

-> It represents the best case of an algorithm time complexity i.e. the minimum time required by an algorithm to complete.

Defn -> A function $f(n) = \Omega(g(n))$, If there exist a function $f(n)$, positive integer n & n_0, positive constant c such that

$f(n) \geq c\,g(n) \ \forall \ n \geq n_0, \ n_0 \geq 1$ & $c > 0$

Hence func. $g(n)$ is an asymptotic lower bound of $f(n)$.

3. Theta-notation (Θ) →

→ It represent the average case of an algo. time complexity.

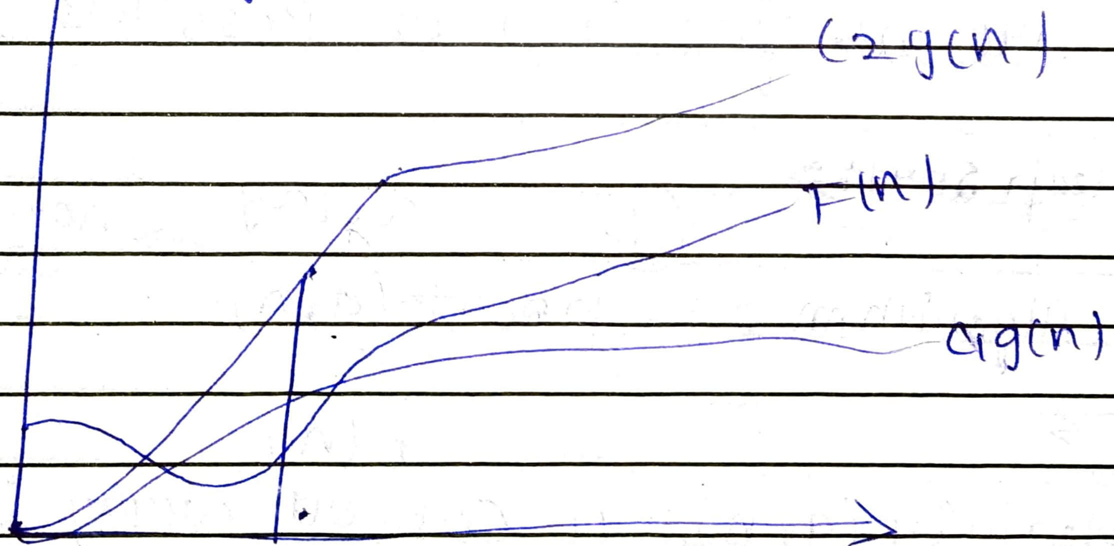
→ It denote the asymptotically tight bound (lower bound) & the (upper bound) of an algo. running time.

→ A function $f(n) = \Theta(g(n))$, If there exists c_1, c_2, positive constant & (+) integer n & no such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \ \forall \ n \geq n_0$$

$$n_0 \geq 1 \ \& \ c > 0$$

Hence fun. g(n) is asymptotic tight bound for f(n).



Ex = $F(n) = 3n + 2$ $g(n) = n$

$c_1 = 1$, $c_2 = 4$

2. → Algo properties.
3. → why we analyze Algo.

Topics →:

4. Performance analysis of algorithm.

Date:→ 22/06/20

→ Each Algorithm must have:→

→ Specification → means Description of compt. pro

→ Pre-condition → The conditions(s) on I/P.

→ Body of the Algorithm → A sequence of clea & unambigious i

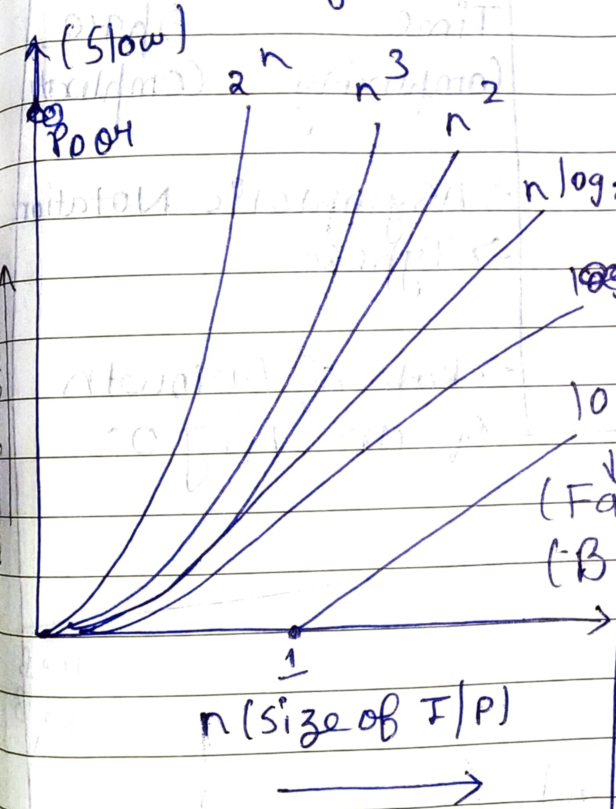
→ Post-condition → The condn on o/P

Date:⇒ 23/06/20

Asymptotic Notations ⇒:

Rate of growth ⇒:

→ The growth rate of an algorithm is the rate at which the running time (cost) of an algorithm grows as the size of the input grows.



→ If the Rate of growth is high n Algorithm slow.

→ we select those algorithm there Rate of growth is lesser.

$n = 64$

2^n	n^3	n^2	$n\log_2 n$
2^{64}	64^3	64^2	$64\log_2 64$

n	$\log_2 n$
64	$\log_2 64$

$\log_2 8 = 1$

[8 steps] → steps.

*How to find log values.

Slow = n!, 2^n, n^3 ---

Fast = $\log_2 n$

$= \log_2 64 = 8$

= 1 → Fast

n! → Slow

23-06/20

Types of asymptotic Notations :-

(i). O – Notation (Big-oh Notation)

→ Worst case (Upper Bound).

(ii). Ω – Notation (Omega – Notation)

→ Best case (Lower bound).

3. Θ – Notation (Theta Notation)

→ Average case (Tight bound).

Ex → Linear Search

10, 50 & 90

Best case → (Omega Notation)
element Search → 10
→ Ω(1). (L.B.)

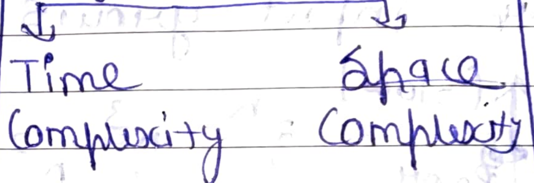
Average case → (Θ Notation)
element Search → 50
→ Θ(n). (T.B)

Worst case → (O-Notation)
element Search → 90
→ O(n). (U.B).

U.B
L.B
T.B

Topics

→ Algorithm
→ Common terms of Algo
→ Properties of Algo
→ Analyzing Algorithm
→ Performance Analysis of Algo.



→ Asymptotic Notations
→ Types

→ Rate of Growth of an Algo.

T.B → A.C

037

Time complexity	Name	Example
1	Constant	
log n	Logarithmic	
n	Linear	
n log n	Linear logarithmic	→ on
n^2	qudratic	P.NO
n^3	Cubic	23
2^n	Exponential	BOOK
n!	Factorial	

Date → 24/06/20

Topics
→ Types of Asymptotic Notations.

CPU Time → Time complexity
→ Main memory space → Space complexity

→ Algorithm Analysis →: Time complexity of an algo. can be calculated in 2 ways →.

→ A Priori Analysis → it's theoretical analysis of an algorithm. Efficiency of an algo. is measured by assuming that all other factors.

Ex → processor speed are constant & have no effect on implementation

→ Posterior Analysis → This is emphrical analysis of an algo. The selected algo. is implemented using any programming language. In this analysis, actual statistics like running time & space required are collected.

↑ Designing technique.

Divide & Conquer techniques ⇒

In divide & conquer method, a given problem is ⇒:

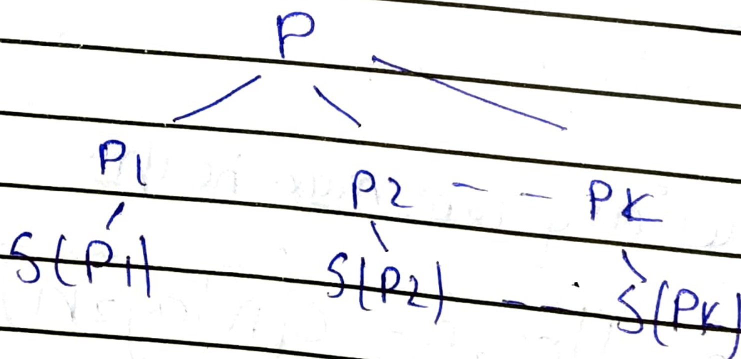
(i) Divided into smaller subproblems.

2. These subproblems are solved independently.

3. If necessary the solutions of the subproblems are combined to get a soln to the orignal problem.

If the subproblems are large enough then D & C is reapplied

For Ex:⇒ Let P will problems



→ The generated subproblem are usually of same type as the original problem. Hence recursive algorithm are used in D & C strategy.

→ Ex of algo that use divide & conquer technique.

→ Recursive Binary search
→ quick sort
→ ~~Heapsort~~ Merge sort
→ Strassen Matrix multip.

... the optimal merge pattern, Huffman coding, minimum spanning tree, knapsack problems, job sequencing with deadlines, single source shortest path algorithm.

× × × ×

Greedy strategy:⇒ Greedy method is a technique.

→ Simplest & straight forward approach.
→ Used to solve optimization problem

⇒ In Greedy method the decision is taken on the basis of current available information without worrying about the effect of current decision in future.

⇒ This technique is to determine feasible solⁿ that may or may not optimal.

Feasible solⁿ⇒: is any subset that satisfy the given condition. (that may be multiple)

Ex⇒ If we take any subset that satisfy the condition → Fea. Sol

Optimal sol → It takes best or most favourable solⁿ.

Available at :JSA. Palash Complex, Kajuri Bazar, Indore

characteristics:→

(i). To construct the solu^n in an optimal way. Algo maintains 2 sets→:

→ one contains chosen items &
→ ~~other~~ other contain rejected items.

(ii). Greedy algo. make good local choice

→ an optimal solu^n
→ an fasible solu^n.

Applications→

→ Shortest Path
→ MST
→ Job sequencing with deadline
→ Knapsack problem.

Algorithm→

```
Algorithm Greedy (a,n)
{
  solution = ∅;
for(i = 1 to n do
  {
    x = select (a);
```

ACROPOLIS TECHNICAL CAMPUS

optimal (minimum or maximum) 03 solun
Source → Destination
Indore 1hr Dewas
Route ← [R1 (R2) (R3) R4] → objective ↑ min cost
1hr ↳ Feasible soln

If feasible (soln, x) then

solution := Union (solution, x);

}

return solution;

}

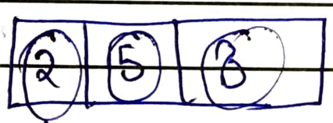
Time complexity = $O(n^2)$
↳ G.A.

(i) optimal-merge Pattern ⇒

→ It relates to the merging of two or more sorted file in a single sorted file.

→ This type of merging can be done by the 2-way merging method.

(n) 3 records

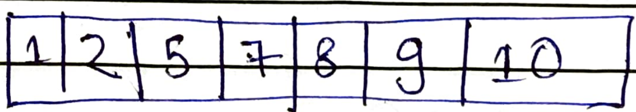


2	5	8

(m) 4 records

1	7	9	10

Firstly inn dono ko compare karna jo chota hae vo aage aayega.



1	2	5	7	8	9	10

max. comparision = 3+4 = 7

(n+m) comparison

CLASS WORK

SESSIONAL WORK

ASSIGNMENT No

EXPERIMENT

SUBMITTED ON .... MARKS OR GRADE OBTAINED ....

NAME .... ROLL NO ....

CLASS .... DEPARTMENT ....

SUBJECT .... CODE NO ....

Signature of Student

Signature of Professor :

Ada Unit ⇒ # 03

Date ⇒ 16/04/19.

Overview ⇒:

→ Concept of dynammic programming

→ Problems based on this approach

→ Such as 0/1 knapsack

→ Multistage graph

→ Reliability design

→ Floyd warshall algorithm.

Dynam pr ⇒ Dynammic programming is a method for solving a problem by breaking it down into collection of simpler subproblems, solving each of these subproblems just once & store their solun in memory like array. So the next time the same subproblem occurs instead of recomputing its solutions, simply looks up the previously computed solun,

there by saving computation time.

Ex → 1+2+3 = 6

Next time 1+2+3+1
→ = 6+1 = 7

Overlapping sub problems [repeated].

→ Memorization → It's a technique of storing solutions to subproblems instead of recomputing them.

→ Dynammic programming is an algorithm design technique for optimization problems: minimizing or maximizing

→ like divide & conquer technique, dyn prgr solves problems by combining soln to subproblem

→ In D.P. there is overlapping in subproblems while in D&C, there are non overlapped subproblem.

CLASS WORK
SESSIONAL WORK

ASSIGNMENT
EXPERIMENT
No

SUBMITTED ON .... MARKS OR GRADE OBTAINED ....
NAME .... ROLL NO ....
CLASS .... DEPARTMENT ....
SUBJECT .... CODE NO ....

Signature of Student Signature of Professor

→ In D.P, save answers of overlapping smaller subproblem to avoid recomputation.

Principle of optimality:→

A problem is said to satisfy the principle of opt. if the subsolution of an optimal soln of the problem are themselves optimal soln for their subproblems.

Ex:→ Shortest Path problem satisfy the ~~follow~~ principle of optimality - If a $x_1, x_2 --- x_n$, b is a shortest Path from node a to node b in a graph, then the portion of x_i to x_j on that Path is a Shortest path from x_i to x_j

↓ optimal sol^y

Ex ⇒ a —— b

|

Sub solution $x_i - x_j$] → Sol^n are optimal

S.P

|

$y_i - y_j$

S.P

Diff^n B/w Greedy Technique & Dy. Prog:

Greedy Techn. method	Dynammic Prog.
Both are optimization Technquies	
(i). There is no principle of optimality	(i). It uses princip of optimality
(ii). In Greedy methody the decision is taken on the basis of current information.	3. D. P. is an algo. designing method that can be used when the solun to a problem be viewed as the res of sequence of decision.

CLASS WORK

SESSIONAL WORK

SUBMITTED ON ....

NAME ....

CLASS ....

SUBJECT ....

Signature of Student

3. It is a d approac get sol a pr

4. Greedy method looking or re previo choices

04.

CLASS WORK
SESSIONAL WORK

ASSIGNMENT / EXPERIMENT	No

SUBMITTED ON .... MARKS OR GRADE OBTAINED ....

NAME .... ROLL NO ....

CLASS .... DEPARTMENT ....

SUBJECT .... CODE NO ....

Signature of Student — Signature of Professor

3. It's a deterministic approach to get soln for a problem	3. It's a not deterministic approach to get a soln of problem. Here we consider all possible soln & select best or optimal soln.
4. Greedy method never looking back or revising previous choices.	4. D. P. looking back or revising previous choices.

Date:→ 26/07/20

→ Searching is a process of finding an item with specified properties from collection of items.

→ The items may be stored as records in database, simple data elements in array, nodes in trees, vertices & edges in a graph.

Why we need Searching:→

→ We know that today's computer store a lot of information.

→ To retrieve the info. proficiently we need very efficient searching algorithms.

→ That means, if we keep the data in proper order, it's easy to search the required element.

→ Sorting is one of the techniques for making element orderd

Types of Searching :→

→ Linear Search (unordered)
→ Sorted / ordered L.S.
→ Binary Search
→ Interpolation Search.
→ Binary Search Trees
→ Symbol Tables & Hashing
→ String Search Algorithms: Tries, Ternary Search & Suffix Trees.

→ Unordered Linear Search

Ex.1

→ Let us assume we are given an array where the order of elements is not known. That means the elements of the array are not sorted. In this case to search an element, we have to scan the complete array & see if the element is present or not.

Time ⟶ O(n) → worst case

→ It has T.C. of O(n) which means the time is linearly dependent on no. of elements, which is not bad, but not good too.

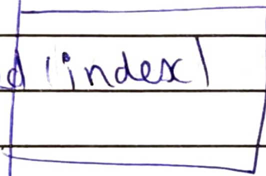
02.

Linear Search:→

→ Linear Search is the simplest search algorithm & often called Sequential search.

→ In this type of searching, we simply traverse the list completely & match each element of list with the item whose location is to be found.

→ If the match found then location of the item is returned (index) otherwise Algo. return -1 or NULL.



T.C → O(n).
S.C → O(1)

→ L.S. is applied on unsorted or unordered lists, when there are fewer element in a list.

How Linear Search works ?

Search for an element k = 1

0	1	2	3	4
2	4	0	1	9

(1). Start from the first element, compare k with each element x

k = 1

2	4	0	1	9
↑ x	↑ x	↑ x	↑ x	↑ x

k = 1 ≠ k = 2

(2). k = 1, k = 4 = ≠

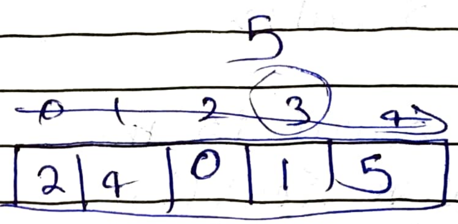
(3). k = 1, k = 0 = ≠

(4). k = 1, k = 1 = ✓

(If x == k) then return index.
else:
return not found

5 0 1

Linear Search implementation in Python →




```
def li.s(array, n, x):
    for i in range(0, n):
        if(array[i]==x):
            return i
    return -1

array=[2, 4, 0, 1, 9]
x = 1
n=len(array)
result= li.s.(array, n, x)
if(result==-1):
    print(" Element not found")
else:
    print
      ("Element found at index:", result)
```

[2, 4, 0, 1, 5]

x = 1

```
def Search(array, x):
    for i in range(len(arr)):
        if (array[i]==x):
            return i
    return -1

array=[2, 4, 0, 1, 5]
result = Search(array, x)
if (result==-1):
    print(" Element not found")
else:
    print(" Element found at index", result)
```

time	O(1)	O(n) avg	O(1) O(n)	Space

Advantages →:

1. Implementation is easy.
2. Linear Search can be applied on both sorted & unsorted list of data.

Disadv →:

→ If the list have large no. of data then it's insufficient for searching data.

→ If there is 200 elements in the list & you want to search element at Posn 199 then you have to search entire list, that's consume time.

Binary Search table.

Worst case	O(log n)
Best case	O(1)
Average case	O(log n)
W.C.	O(1)

Binary Search →:

→ Binary Search is the search technique which works efficiently on the sorted lists.

→ Binary Search follows divide & conquer approach, in which the list is divided into two halves & the item is compared with middle element of list.

→ If the match is found the location of middle element is returned, otherwise we search into either of the halves depending upon the result produced through the match.

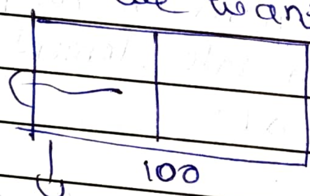
Real life Ex → of Binary Search

→ ~~If we search~~

→ Suppose we want to open ~~page~~ Page no 200 a book of 200 Pages. In L.S. one by one comparsion this is time consuming.

→ So we open middle Page bcoz $\frac{200}{2} = 100$ (centre).

For Ex → 200 Page book & we want to open 20 no page so it's middle is 100



So we Search only in this Part bcoz no's are in ↑ order.

→ So Search time reduce.

→ Rule for binary Search

→ list should be Sorted in ↑↑↑ order.

Ex →

midvalue ↑

10	20	30	40	50
a[0]	a[1]	a[2]	a[3]	a[4]

len(arr) = 5 (size).

~~max~~ (index) = 4

high →

low index = 0

max = -1

Formula =

$$mid = \frac{low + high}{2}$$

2.5 → So we take floor value → 2

$$= \frac{0+4}{2} = \frac{4}{2} = (2)$$

Binary Search Implementation

Process →

1. Compare x with middle element.

2. If x matches with middle element we return the mid index

3. Else if x is greater than mid element, then x can only lie in the right (greater) half subarray after the mid element. Then we apply again for right half.

4. Else if x is smaller, the target x must lie in the left half. So we apply for the left half.

is extremely efficient Algorithm.

Binary Search ⇒:

→ To do binary search, first we had to sort the array elements. The logic behind this technique is: →

→ First find the middle element of the array or list.

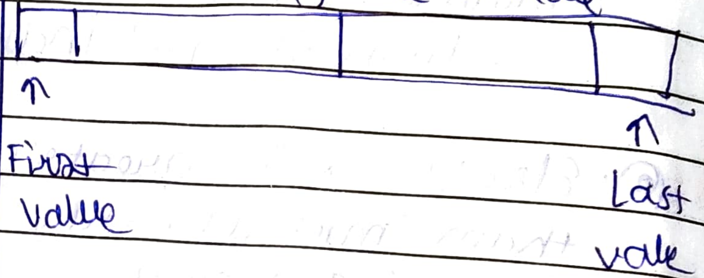
→ Compare the middle element with an item.

→ There are 3 cases:

(a). If it's a desired element then search is successful.

(b). If it's a less than desired element then search only the first half of the array.

(c). If it's a greater than desired element search in the second half of the array.



→ Repeat the same steps until an element is found or exhausts in the search area.

→ In this Algorithm we are reducing the search area.

→ so no. of comparisons keep on decreasing.

max = 9   high = max − 1 = 9 − 1 = 8

Binary Search Implementation with ex. →

$$\begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{matrix}$$

ex → arr = {1, 5, 7, 8, 13, 19, 20, 23, 29}.

→ Find location of item 23 in array.

1st step →

beg = 0 & end = 8

$mid = \frac{int((beg + end))}{2}$

$= int\left(\frac{(0+8)}{2}\right) = \frac{8}{2} = 4$

mid = 4

a[mid] i.e a[4] = 13

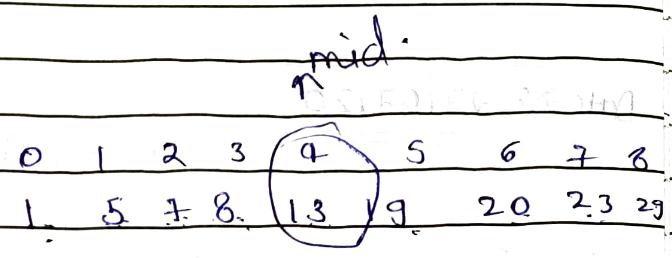
St → II

compare mid element with item.

13 < 23 then

beg → mid + 1

beg → 4 + 1 = 5

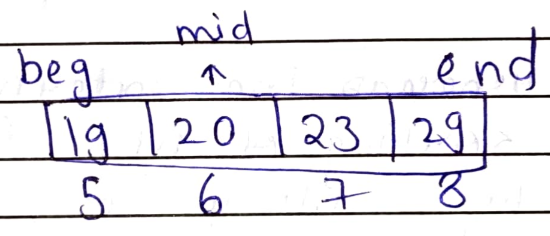


step III

$mid = \frac{int((beg + last))}{2}$

$= \frac{5+8}{2} = \frac{13}{2} = 6$

mid = 6 = 20



20 < 23

beg = mid + 1

= 6 + 1 = 7

End = 8

$\frac{beg + end}{2} = \frac{7+8}{2} = \frac{15}{2} = 7$ ↓ mid

⇒

a[mid] == 23

23 == 23

→ (7)

item < a [illegible]

13 < 23

↳ beg = mid + 1

one condition

else

end = mid - 1

Date :- 27/07/20

Sorting

→ Sorting refers to the operation of arranging data in some given sequence i.e. ↑ order or ↓ order.

→ Sorting is categorised as internal sorting & External sorting.

→ By I.S. we means we arranging the no.'s within the array only which is in computer primary memory.

→ Whereas the external sorting is the sorting of no. from external file by reading it from secondary memory.

Sort

Internal (in Primary memory). (Apply on small amount of data).
- → Selection
- → Bubble
- → Insertion
- → Quick
- → Merge
- → Radix
- → Shell
- → Heap

External (Apply on large amount of data).
- → Natural
- → Balanced
- → PolyPhase
- → K-way merging

Internal Sorting	External Sorting
→ Apply on small amount of data	→ on large amount of data
→ The entire sorting can be done in main memory	→ The entire sorting can't be done in M.M.
→ Requires less time for sorting	→ Requires more time for sorting.
→ There is no need to swap record.	→ we need to swap records b/w M.M & secondary storage.

Bubble
why bubble are on top bcoz weight is less from water

0000000000 → water

Bubble Sort:→

→ In bubble sort each element of the array is compared with its adjacent element.

→ If the first element is larger than second one the Posn of the elements are interchanged, otherwise its not changed.

→ Then next, element is compared with its adjacent element & the same process is repeated for all elements in the array.

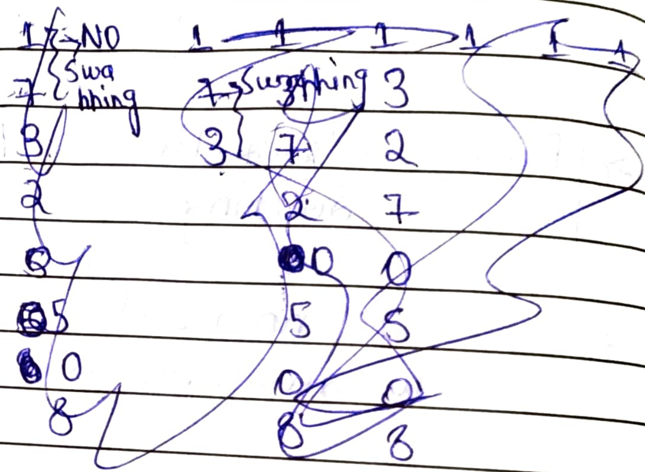
→ The same process is repeated until no more element are left for comparison. Finally the array is sorted one.

Ex →

a, 18, 2, 18, 6, 80

First Pass:→

1, 7, 3, 2, 5, 0, 8



(If 1 is greater than 7 so its Posn is changed bcoz bubble are on top.)

Comp. → 1→2 4–5
2→3 5–6 → Coding
12° 3–4

062

Ex → Pass → I

1	1	1	1	1	1	1	1
7	7	3	3	3	3	3	3
3	3	7	2	2	2	2	2
2	2	2	7	0	0	0	0
0	0	0	0	7	5	5	5
5	5	5	5	5	7	0	0
0	0	0	0	0	0	7	7
8	8	8	8	8	8	8	8

Pass → I

Case → I

1 > 7 → False (No Swapping)

Case II → 7 > 3 → True (Swap 3 on top)

7 ⟷ 3 = 3 ⟷ 7

Case III → 7 > 2 → True (Swap)

Case IV → 7 > 0 (Swap)

Case V → 7 > 5 (S)

Case VI → 7 > 0 (S)

Case VII → 7 > 6

Pass → II

Case → I 1 > 3 → NO

Case → II 3 > 2 → S

Pass II

1	1	1	1	1	1	1	1
3	3	2	2	2	2	2	2
2	2	3	0	0	0	0	0
0	0	0	3	3	3	3	3
5	5	5	5	5	0	0	0
0	0	0	0	0	5	5	5
	6	6	6	6	6	6	6
	7	7	7	7	7	7	7

Pass → II

1	1	1	1	1	1	1	1
3	3	2	2	2	2	2	2
2	2	3	0	0	0	0	0
0	0	0	3	3	3	3	3
5	5	5	5	5	0	0	0
0	0	0	0	0	5	5	5
7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8

on proper Posn

n = 7

Pass → III → Apply on 6 elements

1	1	1	1	1	1
2	2	0	0	0	0
0	0	2	2	2	2
3	3	3	3	0	0
0	0	0	0	3	3
5	5	5	5	5	5
7	7	7	7	7	7
8	8	8	8	8	8

→ Only 5 elements left

Pass → IV → n = 5

1	0	0	0	0
0	1	1	1	1
2	2	2	0	0
0	0	0	2	2
3	3	3	3	3
5	5	5	5	5
7	7	7	7	7
8	8	8	8	8

→ In Proper posn

Pass → V → n = 4

1	0	0	0
1	1	0	0
0	0	1	1
2	3	2	2
3	3	3	3
5	5	5	5
7	7	7	7
8	8	8	8

→ 3 elements left

Pass - VI

0	0	0
0	0	0
1	1	1
2	2	2
3	3	3
5	5	5
7	7	7
8	8	8

Sorted →

0, 0, 1, 2, 3, 5, 7, 8.

temp = 10
a = 10
b = 20
6.

temp = 9
a = b

a = b
temp = 10
a = 20

temp = b
063

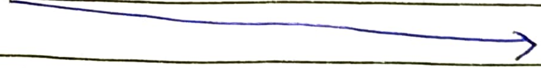
Bubble Sort implementation →:

```
def Sort (num):

    n=len(num)

    for i in range (n-1):

        for j in range (0, n-i-1):
            if num[j] > num[j+1]:
                temp = num[j]
                num[j] = num[j+1]
                num[j+1] = temp

num= [3, 5, 6, 7, 2]
Sort (num)
print (num)
```

O/P →

[2, 3, 5, 6, 7]

5, 0, -1



5 4 3 2 1 0

Program steps:

1. num value
2. call function

n= 5

1st loop → n-1
= 5-1 = 4
[2, 3, 5, 6, 7]
0 1 2 3 4

2nd loop →
0 5-5-1
0-1
-1
0, -1 4-5
4-5-1
-1-1
-2

Rough →

6-1 = 5,

A. C | O(n)
B. C | O(n^2).

Selection Sort :→

→ In Selection sort first, find the smallest element of the array & place it on first Position. Then, find the second smallest element of the array & place it on second Posn. The process continues until we got sorted array.

Ex→ Reverse of Bubble sort

5, 7, 3, 6, 2, 8, 4, 1

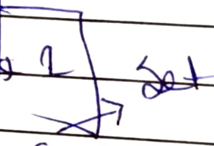
I Pass →:

(5)	(5)	(3)	(3)	(2)	(2)	(2)	1
(7)	7	7	7	7	7	7	7
3	(3)	5	5	5	5	5	5
6	6	(6)	6	6	6	6	6
2	2	2	(2)	3	3	3	3
8	8	8	8	(8)	8	8	8
4	4	4	4	4	(4)	4	4
1	1	1	1	1	1	(1)	2

II Pass →:

(1)	(1)	(1)	(1)	(1)	(1)	(1)	1
7	7	7	7	7	7	7	7
5	(5)	5	5	5	5	5	5
6	6	(6)	6	6	6	6	6
3	3	3	(3)	3	3	3	3
8	8	8	8	(8)	8	8	8
4	4	4	4	4	(4)	4	4
2	2	2	2	2	2	(2)	2

→ In II Pass 1 element set & we don't consider start from 2



set

(1)	1	1	1	1	1	1
(7)	(5)	(5)	(3)	(3)	(3)	2
(5)	7	7	7	7	7	7
6	(6)	6	6	6	6	6
3	3	(3)	5	5	5	5
8	8	8	(8)	8	8	8
4	4	4	4	(4)	4	4
2	2	2	2	2	(2)	3

→ S.S → $O(n^2)$

III Pass→:

→ set

1	1	1	1	1	1
2	2	2	2	2	2
7	6	5	5	4	3
6	7	7	7	7	7
5	5	6	6	6	6
8	8	8	8	8	8
4	4	4	4	4	4
3	3	3	3	3	4

1	1	1	1	1	1	1
2	2	2	2	2	2	2
7	6	5	5	4	4	3
6	7	7	7	7	7	7
5	5	6	6	6	6	6
8	8	8	8	8	8	8
4	4	4	4	5	5	-
3	3	3	3	3	3	

1	1	1	1	1	1
2	2	2	2	2	2
7	6	5	5	4	3
6	7	7	7	7	7
5	5	6	6	6	6
8	8	8	8	8	8
4	4	4	4	5	5
3	3	3	3	3	4

→ till sorted list.

Why Selection Sort→:

→ In ~~multiple iterations~~ bubble sort in each iteration we have done multiple swapping but swapping consume more CPU Powers & memory is nothing wrong in traversing from start to end.

Ex→ 0 1 2 3 4
list1 = [10 , 3 , 4 , 6 , 7]

temp = 10
list1[0] = list1[min_ind]
list1[0] = 3
list1[min_ind] = 10

X

Insertion sort :→

→ It's a simple sorting Algorithm, that builds the final sorted list one item at a time.

→ In this we go through all the elements of list & find index & insert it on correct Position.

Ex→ deck of cards

→ Every no. compare with previous no

9	15	20	25	26	30	99

(Final)

Ex

0 1 2 3 4

list1 | 10 | 4 | 25 | 1 | 5 |

① 10 > 4

| 4 | 10 | 25 | 1 | 5 |

② 4 > 10 → False
4 > 25 → F
4 > 1 → T

| 4 | 10 | 25 | 4 | 5 |

| 1 | 4 | 10 | 25 | 5 |

Ex→
25, 15, 30, 9, 99, 20, 26

Pass I

25	15	30	9	99	20	26

15	25	30	9	99	20	26

15	25	9	30	99	20	26

9	15	25	30	99	20	26

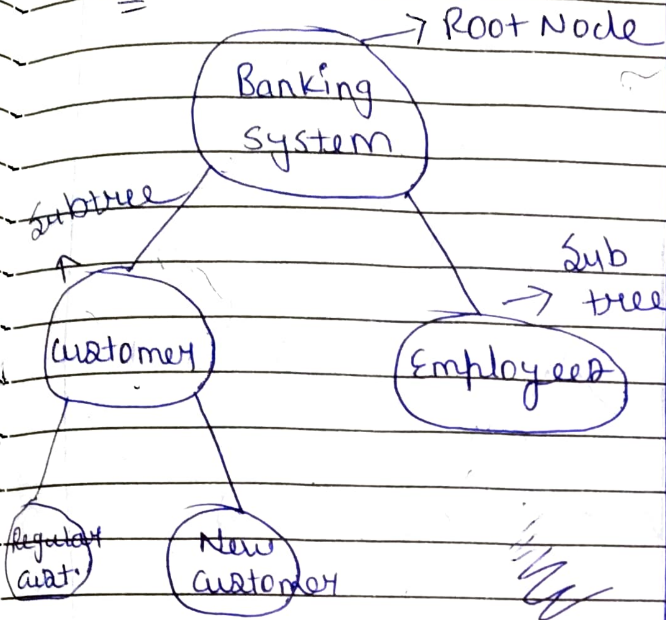
9	15	25	30	99	20	26

9	15	20	25	30	99	26

Tree

→ Trees are basically used to represent the data object in hierachical manner.

Ex→:



Definition:→

→ Tree is a finite set of one or more data items(nodes) such that.

1. There is special data item called the root of the tree.

2. And it's remaining data items are Partitioned into no. of mutually exclusive subsets, each of which is itself a tree. They are called subtree.

Tree Terminology:→

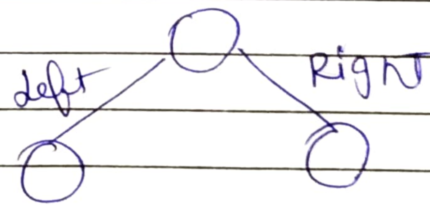
→ There are no. of terms associated with trees which are listed below:→

(i). Root

→ It's a unique node in the tree to which further subtrees are attached.

(ii). Node → Each data item in a tree is called node. It specifies the data information & links (branches) to other data items.

3. Parent Node →: The node having further subbranches.



4. child Node →: The node having no further subtrees.

5. Terminal / Leaf Node :->

-> A node with degree zero / no child.

6. Non-terminal Node :->

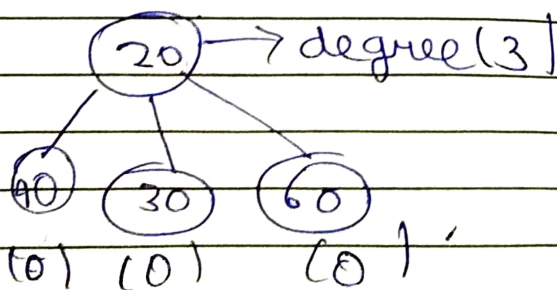
-> Any node (except the root node) whose degree is not zero / having childs.

-> Non-T.N. are the intermediate nodes in traversing the given tree from it's root node to the terminal node (leaves).

7. Siblings :-> (children of same parents) The children node of a given Parent node are called siblings or nodes with common Parents.

8. Degree of a node :->

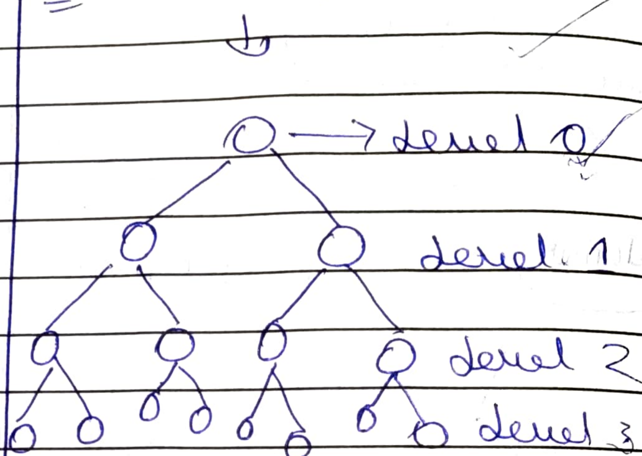
-> The total no. of subtrees attached to that node is called degree of a node



9. Degree of a tree :->

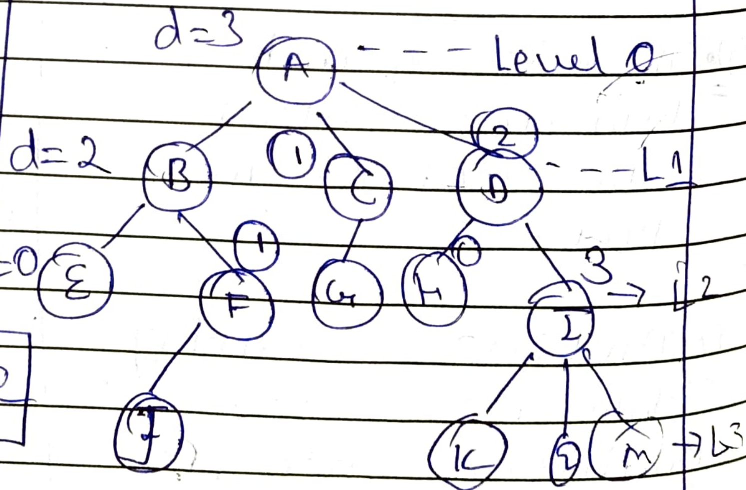
-> The maximum degree in the tree is the degree of tree

10. Level of a tree :->



11. Height / Depth of tree :->

-> The maximum level is the height of a tree.



-> Degree of a tree = 3 (maximum degree)

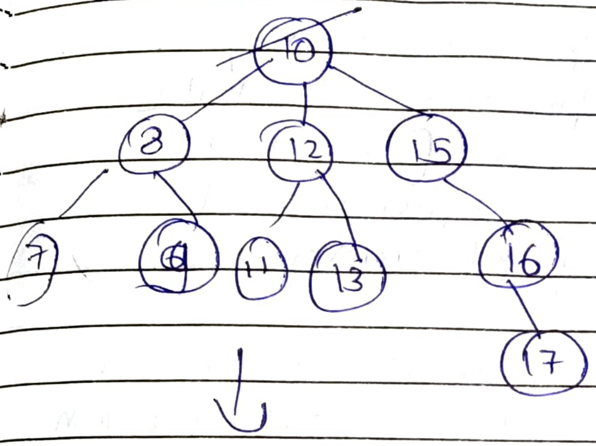
Height = max. level
3

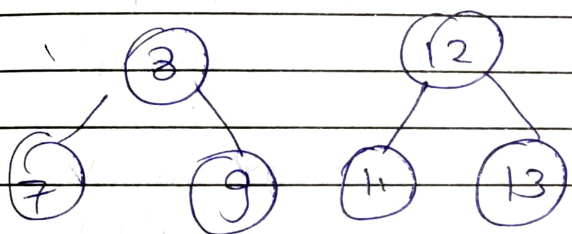
12. Forest:->

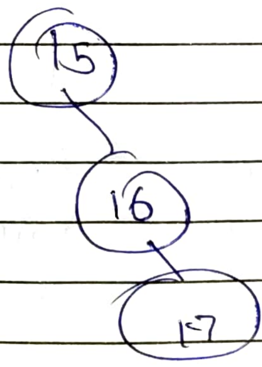
-> Forest is a collection of disjoint trees.

If we delete the root node of a tree then we get a forest

Ex->







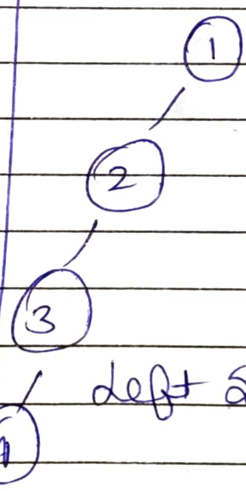
-> Forest with 3 Trees.

13. Skew Trees ->:

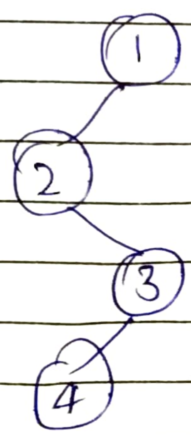
-> If every node in a tree has only one child (except leaf node) then we call skew trees.

-> If every node has only left child then we call them left skew trees.

-> If every node has only right child we called right skew trees



Left skew Tree



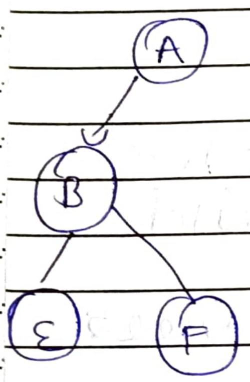
-> Skew Tree

level are called

14. Ancestors :-

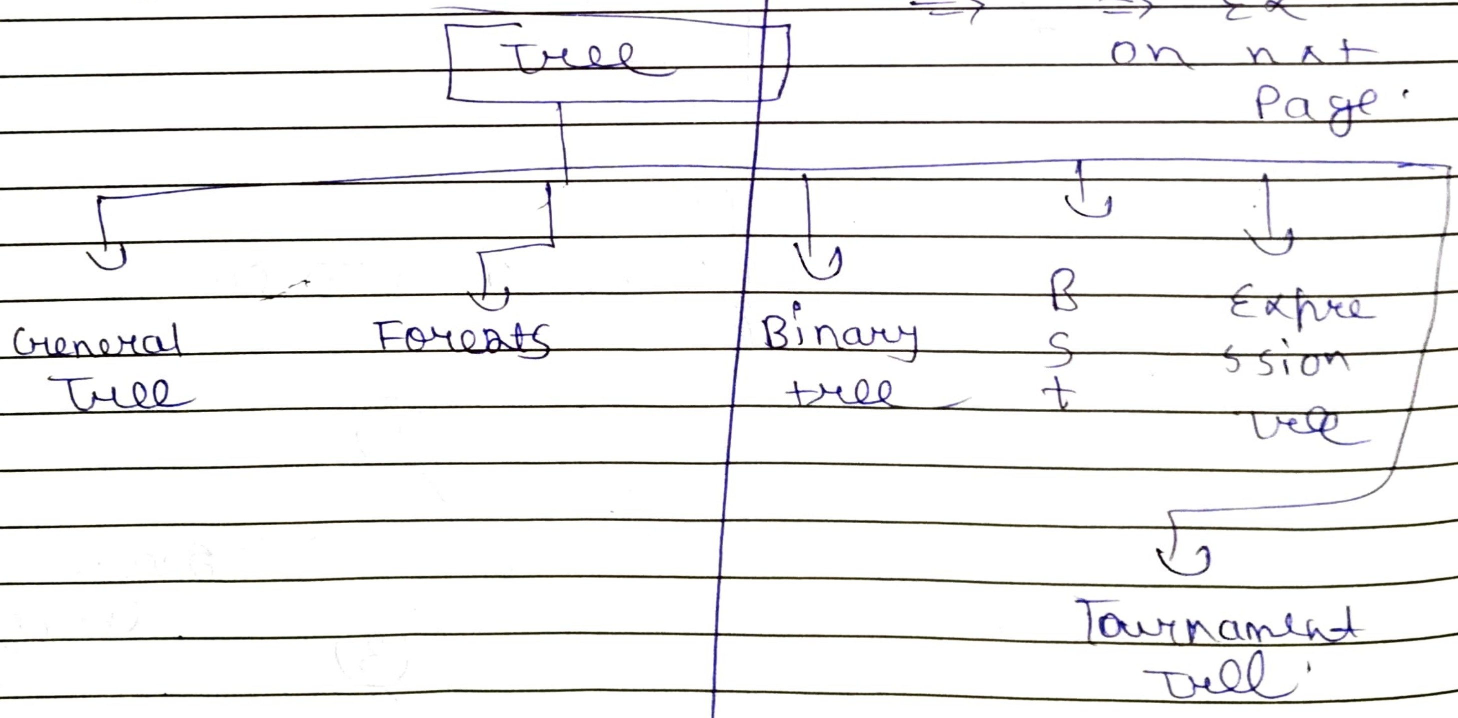
→ An ancestors of a node is any predecessor node on a path from root to that node.

→ The root node doesn't have any ancestors.



→ In this B & A are ancestors of F

Types of Tree :-



Expression Tree :-

→ are used to evaluate the simple arithmetic expressions.

→ Exp. Trees is basically a binary tree where internal nodes are represented by operators while the leaf nodes are represented by operands

→ widely use to solve algebraic expressions like $(a+b) * (a \div b)$

⇒ ⇒ Ex on next page.