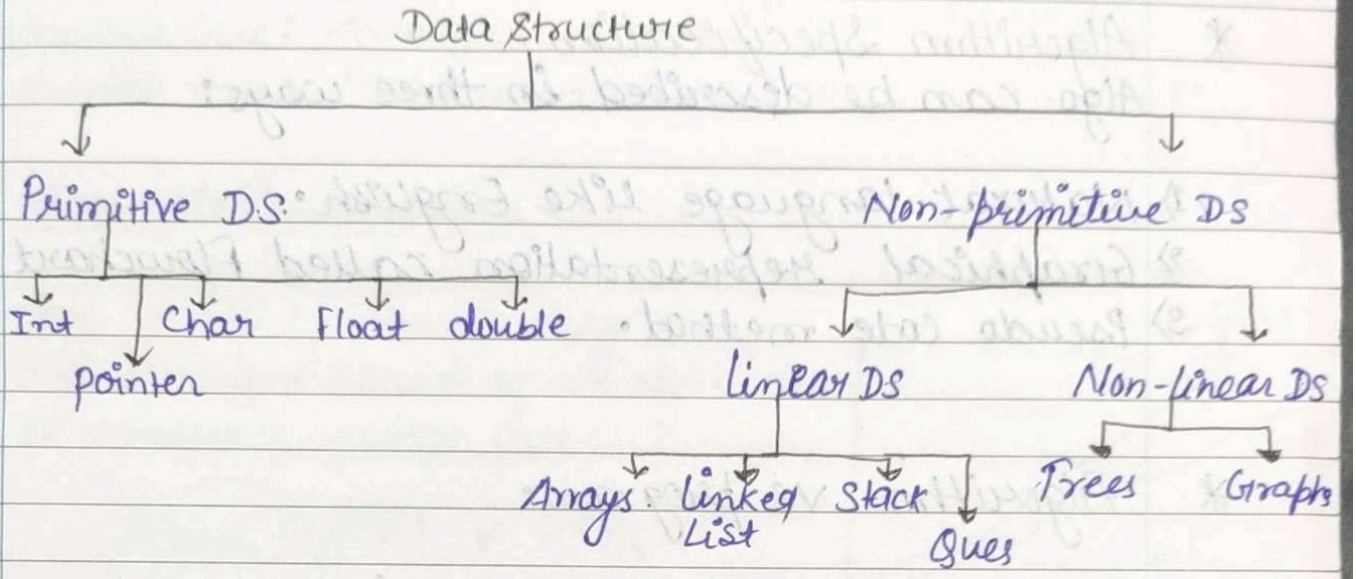
UNIT-1

Dashrath Nandan Date :

Data Structure:

Data Structure is a way to store and organise data so that it can be used efficiently.



☆ Algorithm:

A finite set of instruction that specifies a sequence of operation is to be carried out in order to solve a specific problem.

* Characteristics of an algorithm:

(1) **Input** : An algorithm should have 0 or more well defined Inp.
(2) Output : An algo. Should have 1 or more well defined outputs.
(3) Unambigous / Definiteness : An algo should be clear.
(4) Finiteness : An algo must terminate in finite time.
(5) Feasibility :
(6) Effectiveness or

* Issues or Study of Algorithm:

How to design an algorithm : Creating an algorithm.
How to express an algo. : definiteness.

Date :

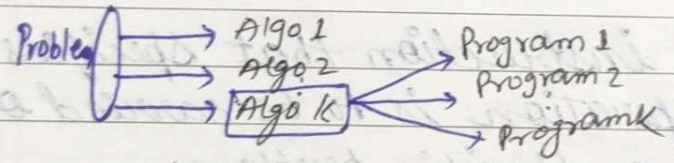
How to analysis an algo. : time and space complexity.
How to validate an algo. : finiteness
Testing the algo : checking for error.

* Algorithm Specification:
Algo. can be described in three ways:

1) Natural language like English.
2) Graphical representation called Flowchart.
3) Pseudo code method.

* Algorithm vs program:

Program refers to the code (written by programmers).



* Time Complexity : $T(n)$
Time complexity of an algorithm represents the amount of time required by an algo. to run to completion.

* Space Complexity:
Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle.

S.C. = Auxillary Space + Input Size

Date :

mathematical notations used to describe the running time of an algorithm for a large input

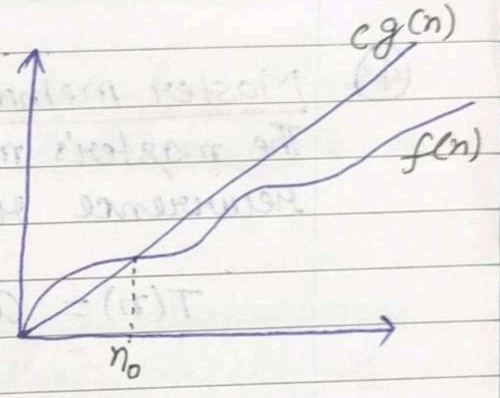
☆ Asymptotic analysis : Time required by an algo. comes under three types :

(i) Best Case : Minimum time required for program execution.
(ii) Average Case : Average " " "
(iii) Worst Case : Maximum time " " "

☆ Asymptotic notations :

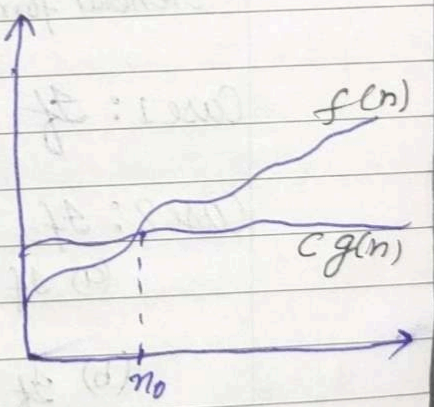
(I) **Big Oh (O)** : Formal way to express upper bound of an algo. It measures worst case.

$$f(n) \leq c \cdot g(n) \quad \forall \; n > n_0$$



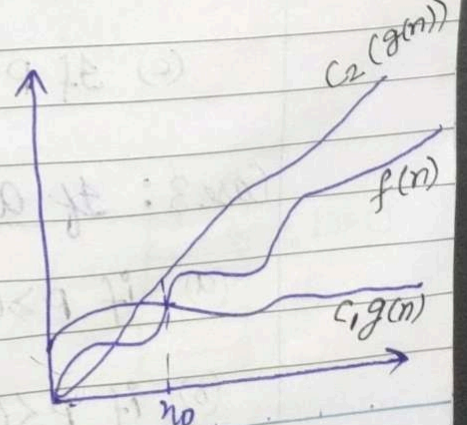
(II) **Omega (Ω)** : formal way to express lower bound of an algo. It measures best case time complexity.

$$f(n) \geq c \cdot g(n) \quad \forall \; n > n_0$$



(iii) **Theta (θ)** : Average case time complexity of an algorithm.

$$c_1 \cdot g(n) \leq f(n) \leq c_2 g(n) \quad \forall \; n > n_0.$$



A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs

Date :

☆ Recurrence Relation:

A recurrence relation is an equation which represents a sequence based on some rule.

There are four methods for solving Recurrence:

1. Substitution Method.
2. Iteration Method.
3. Recursion Tree Method.
4. Master method.

(4) Master method:

The master's method is a formula for solving recurrence relation of the form —

$$T(n) = aT(n/b) + f(n) \quad ; a \geq 1, b > 1$$

$$f(n) = \theta(n^k \log^p n)$$

General form: $aT(\frac{n}{b}) + \theta(n^k \log^p n)$

Case 1: If $a > b^k \Rightarrow T(n) = \theta(n^{\log_b a})$

Case 2: If $a = b^k \Rightarrow$

(a) If $p > -1 \Rightarrow T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

(b) If $p = -1 \Rightarrow T(n) = \theta(n^{\log_b a} \log \log n)$

(c) If $p < -1 \Rightarrow T(n) = \theta(n^{\log_b a})$

Case 3: If $a < b^k \Rightarrow$

(a) if $p \geq 0 \Rightarrow T(n) = \theta(n^k \log^p n)$

(b) if $p < 0 \Rightarrow T(n) = \theta(n^k)$

Recurrence Relation Questions:

① Substitution Method:

Eg $T(n) = \begin{cases} T(n/2) + c & \text{if } n > 1 \\ 1 & n = 1 \end{cases}$

Sol:

$T(n) = T(n/2) + c \quad \cdots (i)$

$T(n/2) = T(n/4) + c \quad \cdots (ii)$

$T(n/4) = T(n/8) + c \quad \cdots (iii)$

substitute eq (ii) in (i)

$T(n) = T(n/4) + c + c$

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

$= T\left(\frac{n}{8}\right) + 2c + c$

$\vdots$ k times

$T\left(\frac{n}{2^k}\right) + kc$

Now, Put $\frac{n}{2^k} = 1$

$\Rightarrow n = 2^k \Rightarrow \log_2 n = k$

$\therefore T(n) = 1 + \log_2 n * c$

$\therefore \boxed{T(n) = \log_2 n}$

Eg: $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{, otherwise} \end{cases}$

$T(n) = 2T(n/2) + n \quad \cdots (i)$; $T(n/2) = 2T\left(\frac{n}{4}\right) + \frac{n}{2} \quad \cdots (ii)$

$T(n/4) = 2T(n/8) + n/4 \quad \cdots (iii)$

Substitute eq (ii) in (i) :-

$T(n) = 2\left[2T(n/4) + n/2\right] + n$

$T(n) = 2^2 T(n/2^2) + 2n \quad \cdots (iv)$

(ii) in (iv) $= 2^3 T(n/2^3) + n + 2n = 2^3 T\left(\frac{n}{2^3}\right) + 3n$

$\vdots$ k times

$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn \quad \text{——} \quad (5)$

Date :

Let, $\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow \log_2 n = k$

Put value of k in eq (5)

$$\boxed{T(n) = n + n\log n}, \quad \because n\log n > n.$$

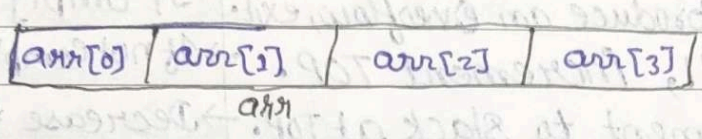
$$\therefore \boxed{O(n\log n)}$$

Date :

☆ **Array**: Array is a linear data structure which store similar type of data item at contiguous memory locations.

Element : Each item stored in an array.

Index : location of an element.



Types of array:

(1) One-dimensional array: It is also called as Single dimension array. By using an index number we can access an element.

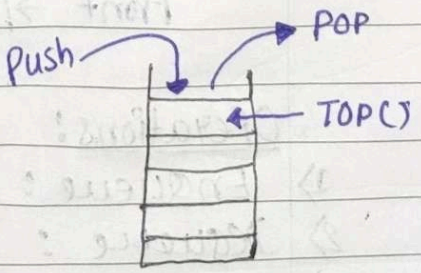
(2) Two-dimensional array: The 2D array is organized as matrices which can be represented as collection of row and columns.

Eg. arr [m] [n]

Basic operations: Traverse, Insertion, Deletion, Search, update.

☆ **Stack**: Stack is a linear data structure which follows LIFO (Last In First Out).

Stack operation:



- Push() : Inserting an element
- POP () : Deleting an element.
- Peek() : Get the topmost item.
- isfull() : chek if stack is full.

isEmpty():

Date :

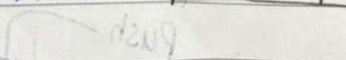
* Implementation:
  (a) Using Array
  (b) Using Linked List.

	Push Operation	POP Operation.
Step 1.	Check if stack is Full	→ Check if stack is empty.
Step 2.	If Full, produce an Overflow, exit.	→ If empty, print Underflow.
Step 3.	If not full, increment TOP.	→ If not empty, access the data at TOP.
Step 4.	Add element to stack at TOP.	→ Decrease the value of TOP by 1
Step 5.	Return Success.	→ Return Success.

* Linked List Implementation of Stack:

	PUSH	POP
Step 1:	Create a NewNode	→ Check Stack is Empty.
Step 2:	Check whether Stack is Empty (TOP == Null)	→ If Empty, then print deletion Not Possible.
Step 3:	If Empty, then set newNode → next = Null.	→ If NotEmpty, then define a Node Pointer 'temp', set it top.
Step 4:	If NotEmpty, then set newNode → next = top.	→ Then set top = top → Next.
Step 5:	Set, top = newNode.	→ delete temp. (free(temp))

☆ Queues: Queues is a linear data structure which follows FIFO (First-In-First-Out).



Operations:
1) Enqueue: Add an element
2) Dequeue: deletion
3) Front & Rear.

Date :

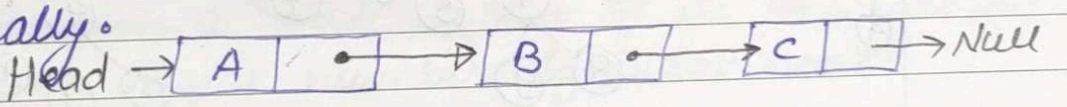
* Types of Queue :

1) Circular Queue : A Queue whose last position is connected to the first position to make a circle.

2) Dequeue : Double ended queue allow insertion and deletion from both the ends.

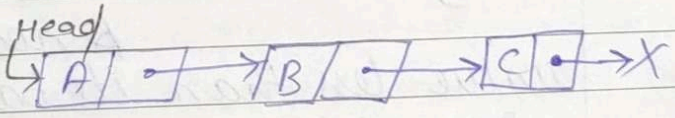
3) Priority Queue : A priority queue is a type of queue that arranges elements based on their priority value. Element with higher priority values are typically retrived before element with lower priority.

☆ Linked list : A linked list is a linear data struct. that stores a collection of data elements dynamically.

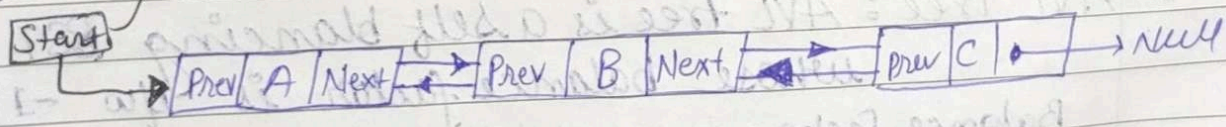


* Types of linked list :

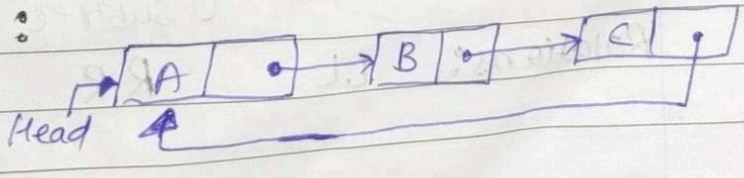
1) Singly linked list :



2) doubly linked list :



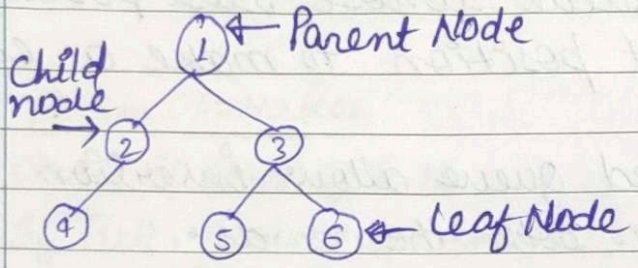
3) Circular Linked list :



* Operations : Insertion, deletion,

Date :

☆ **Trees:** A tree is a non-linear data struct- with a hierachy bast structure.

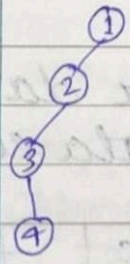


Tree Traversal:
1. Preorder. (NLR)
2. Postorder. (LRN)
3. Inorder (LNR)

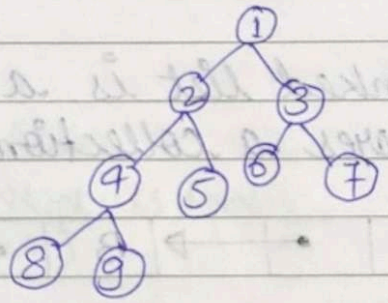
* **Binary Tree:** A binary tree has special condition that each node can have maximum of two children.

Types:

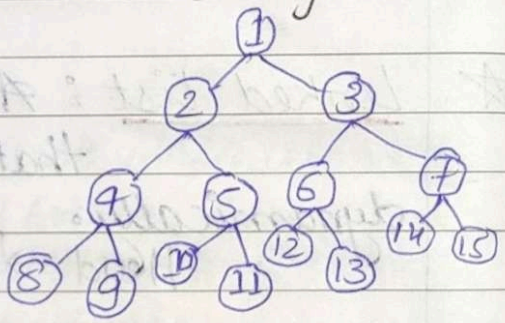
Skewed B.T.



Complete B.T.



Full Binary Tree



* **Binary Search Tree:** BST have condition that A node's left child have a value less than its parent's value and node's right child have value greater than parent's value

* **AVL Tree:** AVL tree is a self blancing BST whose balance factor is b/w -1 to 1.

Balance factor = Height of left subtree − Height of right subtree.

Rotations: LL Clockwise, RR Anti-clockwise, LR = LL+RR, RL = RR+LL

Date :

* **B-Tree**: A B-Tree of order m can have at most m-1 keys and m children.

properties:

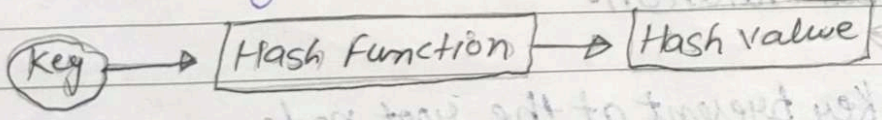
- Every node in a B-Tree have at most m children.
- Every node except root node and leaf node have at least $m/2$ children.
- The root node have at least 2 nodes.
- All leaf node must be at same level.

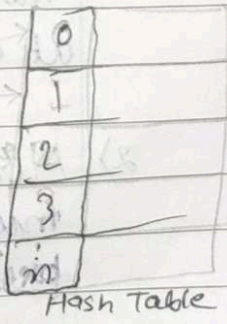
* **Red-Black Tree**: Red B.T. is a self balanced BST also called "Symmetric B-Tree".

Properties:

- Every node has a colour either Red or Black.
- Root of tree is always Black.
- There are no two adjacent Red nodes (A red node can't have red child or parent)
- Every path from a Node to any of its descendant Null node has same number of black node.

☆ **Hashing**: Hashing is a technique of mapping keys and values into the hash table by using a hash function.





Hash Table

Hash Table: Hash table is a type of data structure which is used for storing and accessing data very quickly.

Hash Value:- Serves as an index for a data item.

Hash function: It is a function that maps any big number to a small integer value.

Date :

* Types of hashing:

① Division Method: Hash function is dependent upon the remainder of a division.
h(key) = record(data) % table size.

② Mid Square Method: Firsty key is squared and then mid part of result is taken as the index.

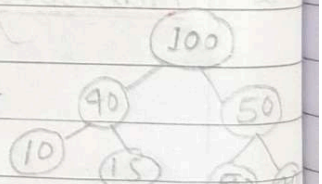
③ Digit Folding method: Key is divided separate parts and by using some operation these parts are combined to produce a hash key.

* Properties:

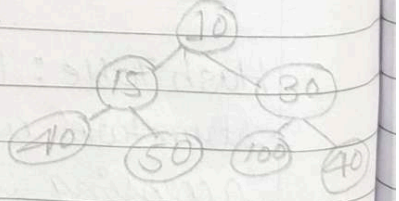
- It is efficiently computable.
- It minimizes the number of collision.

☆ HEAP: A Heap is a special Tree-based data structure in which the tree is a complete Binary Tree.

Types: 1) Max-Heap: Key present at the root node must be greatest among the keys present at all of its children.



2) Min Heap: Key present at the root node must be minimum among the keys present at all of its children.



* Operations: find, insert, delete, extract, replace size, is empty, merge, meld.

* Heap Sort, Heapify method, [Youtube]

In Incidence Matrix of Graph :
1 : represent row edge which is outgoing.
0 : not connected.
-1 : Represent row edge which is incoming.

☆ **Graphs** : A graph is a non-linear data structure consisting of nodes and edges (vertices and edges).

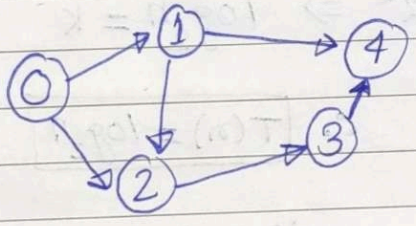
* DFS : Depth First Search . It uses Stack data struct.

* BFS : Breadth First Search . It uses Queue data struct.

* Graph Representation :
(i) Adjacency Matrix .
(ii) Incidence Matrix .
(iii) Adjacency List .

* Applications :
- Used to represent network.
- Used in social network like LinkedIn, Facebook, etc.

Eg



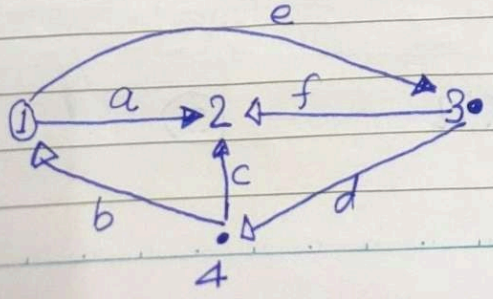
Adjacency Matrix :

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	0	1
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0

⇒ Directed Graph and Undirected Graph.
Weighted graphs, Connected graphs.

* basic operations :
Add vertex, Add edge, Display Vertex.

* Incidence Matrix :



brandes / Node	a	b	c	d	e	f
1	1	-1	0	0	1	0
2	-1	0	-1	0	0	-1
3	0	0	0	1	-1	1
4	0	1	1	-1	0	0

Unit- 1.3 Dashrath Nandan Date :

☆ Counting Sort :

- Time Complexity = $O(n+r)$
- Space Complexity = $O(n+r)$

- Non- Comparision based sorting techniques.
- Counting Sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values.

* Algorithm :

2	9	7	4	1	8	4

Step 1: Find the maximum element from the given array.

9	2	7	4	1	8	4

max

Step 2: Initialize, array of length max+1 (or range's length) having all 0's, that store the count of the element in array.

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Count array.

Step 3: Now, Store the Count of each array elements at their corresponding index in the count array

Given array =

2	9	7	4	1	8	4

Count array =

0	1	2	3	4	5	6	7	8	9
0	1	1	0	2	0	0	1	1	1

Count of each stored element

Date : _____

Store the cumulative sum of count array elements.

0	1	2	3	4	5	6	7	8	9
0	1	1	0	2	0	0	1	1	1

0+1=1 1+1=2 2+0=2 2+2=4

0	1	2	3	4	5	6	7	8	9
0	1	2	2	4	4	4	5	6	7

Step 4: Find the index of each element of the original array.

Original array: | 2 | 9 | 7 | 4 | 1 | 8 | 4 |

Count array:

0	1	2	3	4	5	6	7	8	9
0	1	2	2	4	4	4	5	6	7

7-1=6

Output (Sorted array):

0	1	2	3	4	5	6
1	2	4	4	7	8	9

↓ Size same as original array

☆ **Radix Sort:** It is the linear sorting algorithm that is used for integers. In this, there is digit by digit sorting is performed that is started from least significant digit to the most significant digit.

Algorithm / Procedure:

181	289	390	121	145	736	514	212

Step 1: Find the largest element (736), we have to go from least significant digit to most significant digit of largest element for sorting (i.e. to the hundred place).

Date :

Pass 1: The list is sorted on the basis of digit at 0's place.

181	181
289	121
390	212
121	514
145	145
736	736
514	289
212	

→

3	9	0
1	8	1
1	2	1
2	1	2
5	1	4
1	4	5
7	3	6
2	8	9

Pass 2: Pass 1 output will be sorted on the basis of 10th place.

390	390
181	181
121	121
212	212
514	514
145	145
736	736
289	289

→

2	1	2
5	1	4
1	2	1
7	3	6
1	4	5
1	8	1
2	8	9
3	9	0

Pass 3: Pass 2 output will be sorted on the basis of (100th place) next significant digit.

212	121
514	145
121	181
736	212
145	289
181	390
289	514
390	736

→

1	21
1	45
1	81
2	12
2	89
3	90
5	14
7	36

∴ Sorted array is

[0]	[1]	[2]	[3]	[4]
121	145	181	212	289

[5]	[6]	[7]
390	514	736

* Time Complexity: Best : $\Omega(n+k)$
  Average : $\theta(nk)$
  Worse : $O(nk)$

* Space Complexity: $O(n+k)$

Date :

☆ Bucket Sort :

- Best Case : $O(n+k)$
- Worst Case : $O(n^2)$

⇒ Bucket Sort is a sorting algorithm that separate the element into multiple groups (buckets), and then they are sorted by any other sorting algorithm.

Algorithm :

```
Bucket Sort (A[])
1. Let B[0... n-1] be a new array
2.  n = Length [A]
3. for i = 0 to n-1
4.      make B[i] an empty list
5. for i = 1 to n
6.      insert A[i] into list B[n*a[i]]
7. for i = 0 to n-1
8.   Sort list B[i] with insertion-sort.

9. Concatenate list B[0], B[1], .... B[n-1] together in order.
```

* Advantages :

i) It Reduces the no. of Comparisons.
ii) It is asymptically fast because of the uniform distribution of elements.

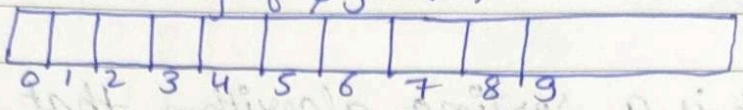
* Disadvantages :

i) It may or may not be a stable algorithm.
ii) Not useful for large array.

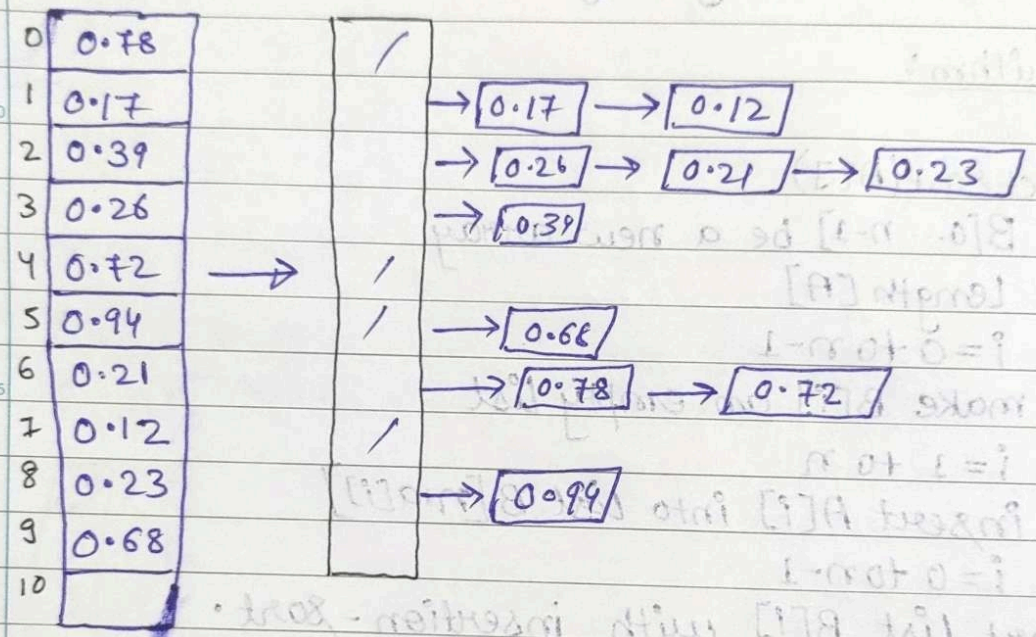
Date :

Ex: [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]

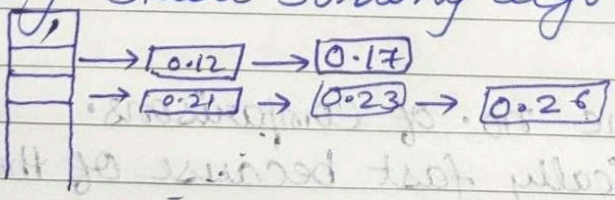
Step ① Create an array of size 10;



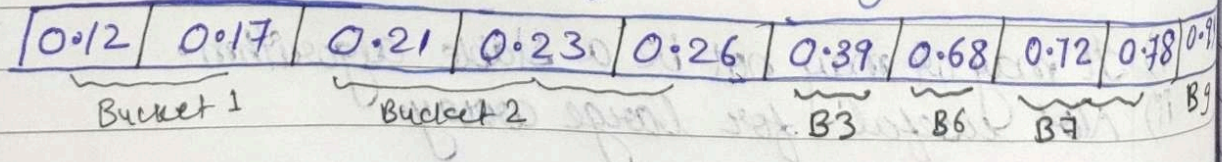
Step ② Insert element into bucket with multiplying 'n' Then convert it into integer. Eg. 0.78 X 10 = 7.8



Step3 Sort the element in each bucket by applying Stable Sorting algo.



Step 4: Gather the element from each bucket and put them into the original array.



Step 5: Return the sorted array.

UNIT: 2.1 Dashrath Nandan Date :

Divide and Conquer :

In this, a problem is divided into smaller problems, then smaller problems are solved independently, and smaller problems are combined into a solution.

Examples: Binary Search, Merge Sort, Strassen's Matrix Multiplication.

* Binary Search :

Binary Search performed on a sorted array.

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(\frac{n}{2})+1 & \text{otherwise,} \end{cases}$$ Binary Search uses $O(\log n)$ time.

* Max-Min Problem :

problem statement: Find the Max^m. and Min^m value in a array.

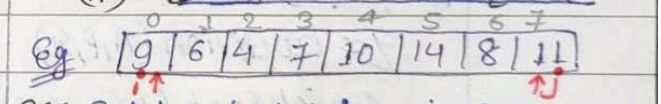
Method-

(i) Naive Method

```
Max := Numbers[1]
Min := Numbers[1]
for i = 2 to n do
    if numbers[i] > max then
        max := number[i]
    if numbers[i] < min then
        min := number[i]
return (max, min)
```

(ii) Divide and Conquer

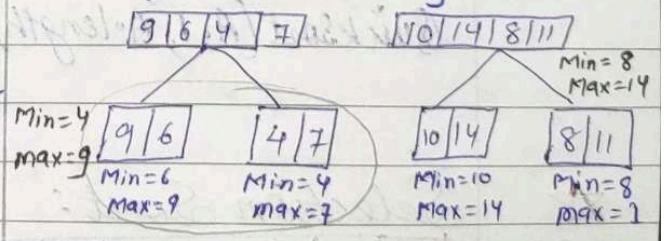
Eg



S1: Set two point i=0, j = n-1

S2: Find mid; $\frac{0+7}{2} = [3.5] = 3$

S3: Split the array —



S4: Find Min & max at each level and update till start.

Min = 4 ; Max = 14

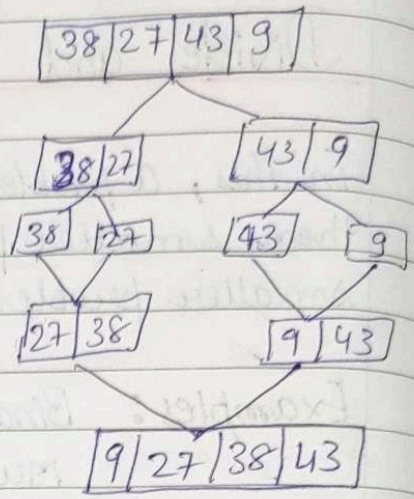
$$T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + T(\frac{n}{2}) + 2 & n > 2 \\ 1 & n = 1 \\ 0 & n < 0. \end{cases}$$

Analysis :

No. of Comparision in naive method = $2n-2$ and in Divide & Conquer = $\frac{3n}{2} - 2$

Date:

★ Merge Sort: Time complex. = $\theta(n \log n)$



MergeSort (arr[], l, r)
if r > l
Step 1: find middle point and divide
mid: m = (l+r)/2
Step 2: Call merge for first half
Call mergeSort (arr, l, m)
Step 3: Call mergeSort (arr, m+1, r) step2 step3
Step 4: Merge the two halves sorted (S2, S3)
Call merge (arr, l, m, r)

★ Quick Sort: T.C. = $\theta(n \log n)$, W.C. = $O(N^2)$

This algorithm is based on Divide and Conquer algo. that picks an element as a pivot and partitions the given array around the pivot by placing in Correct ord

Algo:
QuickSort (A, P, r)
if p < r
q = partition (A, P, r)
QuickSort (A, P, q-1)
QuickSort (A, q+1, r)
For entire array A,
QuickSort (A, 1, A.length)

Partition (A, P, r)
1. x = A[r]
2. i = P-1
3. for j = P to r-1
if A[j] ≤ x
i = i+1
exchange A[i] with A[j]
4. Exchange A[i+1] with A[r]
5. return i+1.

★ Selection Sort:

Ex: 19 | 33 | 27 | 30 | 35 | 19 | 42 | 44

min Select first element and replace it with smallest one.

10 | 33 | 27 | 14 | 35 | 19 | 42 | 44

10 | 14 | 27 | 33 | 35 | 19 | 42 | 44

Continue this process until the array is sorted.

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

Date :

☆ Strassen's Matrix Multiplication:

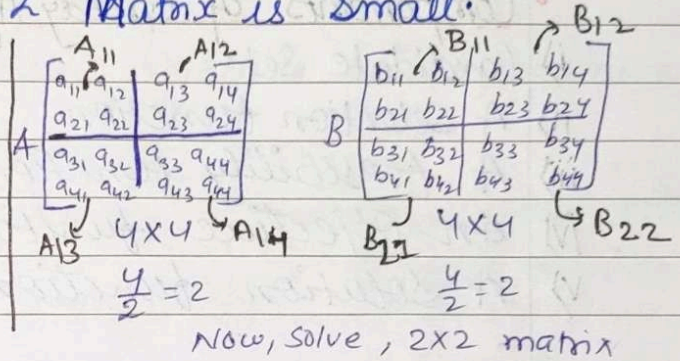
- Naive Method

$$A=\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * B=\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = C=\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} ; C_{ij}=\sum_{K=1}^{n} A_{ik} * B_{kj}$$

T.C: $O(n^3)$

- Divide & Conquer Method: 2X2 Matrix is small.

$$c_{11} = a_{11}\cdot b_{11} + a_{12}\cdot b_{21}$$
$$c_{12} = a_{11}\cdot b_{12} + a_{12}\cdot b_{22}$$
$$c_{21} = a_{21}\cdot b_{11} + a_{22}\cdot b_{21}$$
$$c_{22} = a_{21}\cdot b_{12} + a_{22}\cdot b_{22}$$



T.C. ⇒ $8T\left(\frac{n}{2}\right)+n^2$

$T(n) = \theta(n^3)$

✳ Strassen's Method formula:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = C\begin{bmatrix} P+S-T+V & R+T \\ Q+S & P+R-Q+U \end{bmatrix}$$

A

$P = (A_{11}+A_{22})(b_{11}+b_{22})$ → Σdiagonal (A) − Σdiagonal (B)

$Q = B_{11}(A_{21}+A_{22})$ B ①

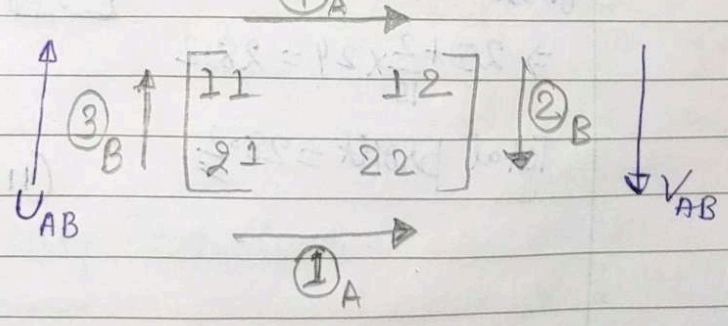
$R = A_{11}(B_{12}-B_{22})$ A ②

$S = A_{12}(B_{21}-B_{11})$ A ③

$T = B_{12}(A_{11}+A_{12})$ B ④

$U = (B_{11}+B_{12})(A_{21}-A_{11})$

$V = (B_{21}+B_{22})(A_{12}-A_{22})$



✳ T.C. ⇒ $T(n) = 7T\left(\frac{N}{2}\right)+O(N^2)$, $T(n) = \theta(N^{2.8074})$

Date :

☆ Greedy Method

In this approach, the decision is taken on the basis of current available information without worrying about the effect of current decision in future.

General algo:

```
getOptimal(item, arr[], n)
1) initialize empty result
   : result = {}
2) while (! item empty)
   i = SelectAnItem();
   if (feasible (i))
   result = result Ui
3) return result.
```

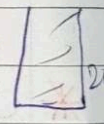
- Components of Greedy Algorithm
  i) Candidate Set
  ii) A Selection function
  iii) A feasibility function
  iv) An Objective function
  v) A Solution function

☆ Fractional Knapsack Problem

Eg

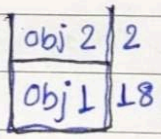
Objects	Obj 1	Obj 2	Obj 3
Profit	25	24	15
Weight	18	15	10
P/W	1.3	1.6	1.5

Knapsack Capacity (M) = 20



I) Greedy about Profit

Select the max. profit first.



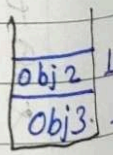
$\Rightarrow 25 + \frac{2}{15} \times 24 = 28.2$

Total profit = 28.2

II) Greedy about weight

Select the minimum weight first.

Profit = $15 + \frac{10}{15} \times 24$ (obj 3, obj 2)



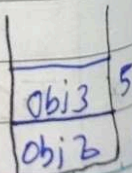
= 31

(III) Greedy about P/W.

First Select object with Maximum P/W ratio.

Profit = $24 + \frac{5}{10} \times 15$

Profit = 31.5 max among all



Date :

☆ Minimum Spanning Tree (MST):

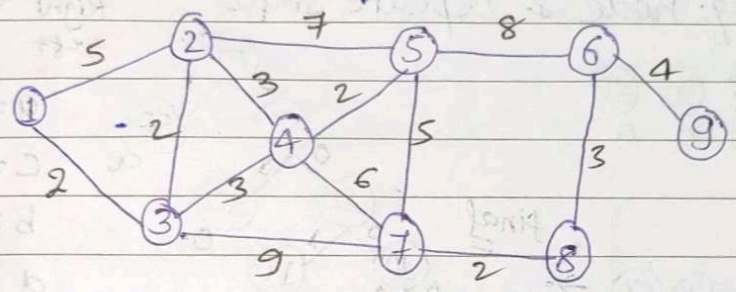
A spanning tree is a subset of an undirected graph that has all the vertices connected by minimum no. of edges.

properties:

i) A spanning tree does not have any cycle.

ii) Any vertex can be reached from any vertex.

* MST connects all the vertices together with minimum possible total edge weights.



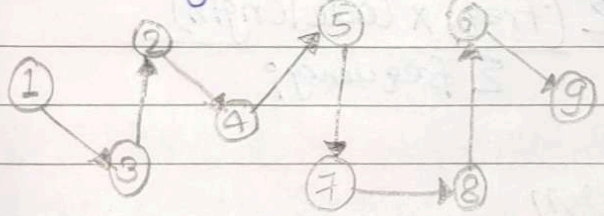
↳ FIFO

↳ जो node पहले आयेगा वो पहले traverse होगा।

(i) Prim's Algorithm

⇒ Start from any node

↳ Select edge with min. cost.



(ii) Kruskal's Algorithm

↳ Sort all edges in non-decreasing order.

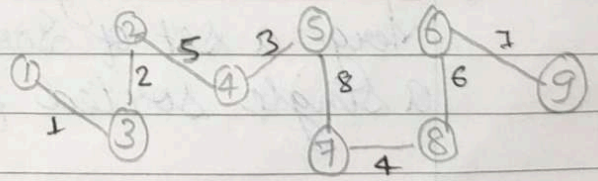
↳ pick smallest, no cycle.

↳ repeat.

1(1-3), 2(3-2), 2(4-5), 2(6-8), 3(3-4)

3(2-4), 3(6-8), 4(6-9), 5(1-2), 5(5-7)

6(4-7), 7(2-5), 8(5-6), 9(3-7)



☆ Huffman Coding: [Greedy Technique]

It is a lossless data compression algorithm.

Prefix Rule: The code that is allocated to a character shall not be another code's prefix.

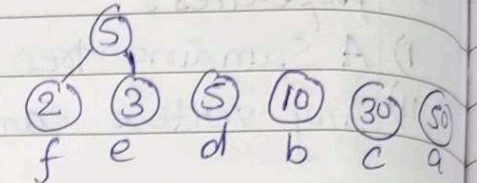
Date :

Ex: a = 50, b = 10, c = 30
d = 5, e = 3, f = 2

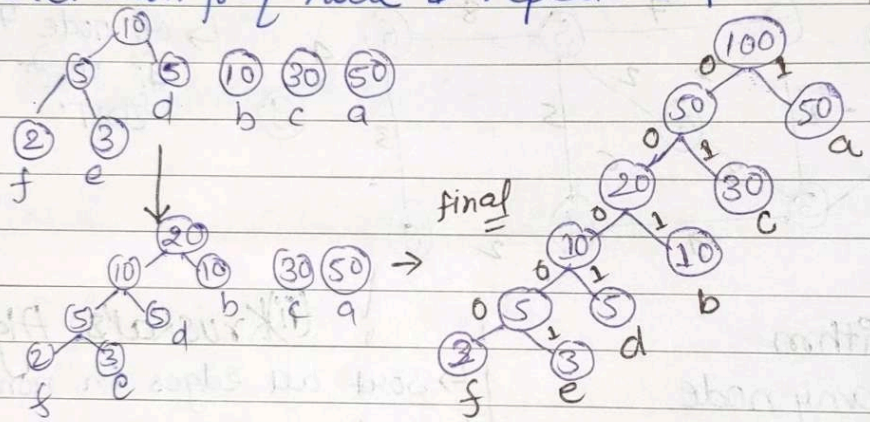
Step1: Build a min heap in which each node represent the root of a tree with a single node.

(2) (3) (5) (10) (30) (50)
f e d b c a

Step2: Obtain two min frequency node from the min heap, Add a third internal node 2+3=5 which is created by joing the extracted nodes.



Step3: Get min freq. node & repeate step2.



Right edge weigh = 1
Left = 0

	bits
a = 1	= 50x1
c = 01	= 50x2
b = 001	= 10x3
d = 0001	= 5x4
e = 00001	= 3x5
f = 00000	= 2x5
	185 bits

Avg. Code length = $\frac{\sum (\text{freq}_i \times \text{code length})}{\sum \text{frequency}_i}$

(n log n)

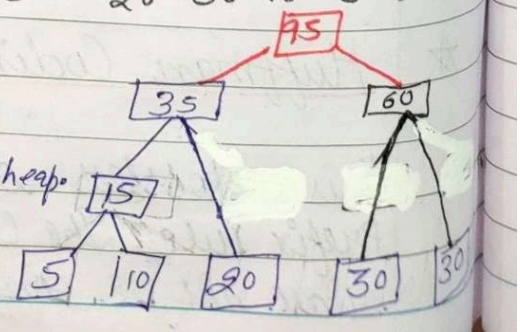
☆ Optimal Merge Pattern

Merge a set of sorted files of different length into a single sorted file with minimum computation.

Ex: files: f_1 f_2 f_3 f_4 f_5
sizes: 20 30 10 5 30

Algorithm:

Step1: Create a min Heap with n files.
Step2: At each level remove two min 'a' & 'b' from min heap and put ab again into heap.



Total cost of merging ⇒ 95+35+60+15 = 205
or
5x3 + 10x3 + 20x2 + 30x2 + 30x2 = 205

UNIT: 2.2

Dashrath Nandan

Date :

☆ Dynamic Programming

It is also used in optimization problems.
It solves problem by combining the problem of subproblems.

* Two main properties are:-

1) Overlapping Sub-problems: Like Divide & Conquer, it also combines solutions to Sub-problems. The computed solution are stored in a table, so these don't have to re-computed.

2) Optimal Sub-Structure: A given problem has Optimal Substructure properties, if the optimal sol? of the given problem can be obtained using optimal sol?. of its Sub-problems.

* Steps of DP approach

Step 1: Charterize the strct. of an optimal sol?
2: Recursively define the value of optimal sol?
3: Compute the value, in bottom-up fashion.
4: Construct optimal sol? from Computed Info.

Applications

Matrix Chain Multiplication.
Longest Common Subsequence
Travelling Salesman Problem

☆ 0-1 Knapsack : Dynamic-Programming Approach

↳ We cannot fraction of an object. ; $x_i = 0/1$.

Eg: $P = \{1, 2, 5, 6\}$, m (capacity) $= 8$
$W = \{2, 3, 4, 5\}$ $n = 4$

Step 1: Create a Table or Matrix of 2-D. Rows represents the individual weight in ascending order.
Step 2: fill 0^{th} col & 0^{th} row with zero.

Youtube

Date:

P_i	W_i
1	2
2	3
5	4
6	5

i ↓ V & W →	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1
2	0	0	1	2	3	3	3	3
3	0	0	1	2	5	5	6	7
4	0	0	1	2	5	6	6	8

General Formula ⇒ $V[i,W] = \max\{V[i-1,W], V[i-1, W-W[i]+P[i]]\}$

⇒ Selected item :→ x_1 x_2 x_3 x_4

0 1 0 1

item Selected are (x_2, x_4).

Profit Value = P

8-6=2

2-2=0

☆ Subset-Sum Problem: (Youtube)

Problem Statement ⇒ Let A be an array or set which contains 'n' non-negative integers. Find Subset 'x' of Set 'A' such that Sum of all elements of $x = W$.

eg. $A = \{2, 3, 5, 7, 10\}$ Sum (W) = 14

True = 1
false = 0

Sum - A[i]

Just above row ka pta index pr value.

n (i) \ Sum (j) →	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
2	1	0	1	1	0	1	0	0	0	0	0	0	0	0
3	1	0	1	1	0	1	0	1	1	0	1	0	0	0
5	1	0	1	1	0	1	0	1	0	1	1	0	1	1
7	1	0	1	1	0	1	0	1	1	1	0	1	1	1
10	1	0	1	1	0	1	0	1	1	1	0	1	1	1

Algo: Initally for i=0 to n-1 , A[i][0] = 1

Now, for i=0 to n-1

for j=1 to Sum

if (j < A[i]) {

Ans[i][j] = Ans[i-1][j] , i>0 }

else {

~~Ans[i][j] = Ans[i-1][j-A[i]] || Ans[i-1][j]~~ }

Date :

☆ Minimum Change making problem / Coin Change

Problem: We are having coins with different dimensions. Now, we have to make an amount by using these coins such that a min number of coins

Eg Coins = {1, 5, 6, 9} W = 10

i \ W(j)	0	1	2	3	4	5	6	7	8	9	10	
1	0	1	2	3	4	5	6	7	8	9	10	
5	0	1	2	3	4	1+0=1	1+1=2	1+2=3	1+3=4	5	2	← (5,5)
6	0	1	2	3	4	1	1	2	3	4	2	(5,5) ←
9	0	1	2	3	4	1	1	2	3	1	2	←

Coins Used ⇒ (5,5)

Algorithm :

Create a table of size (Sum+1) × Coin.Size().

```
if j< coin[i]
    ans[i][j] = 0 || ans[i-1][j]      if i>0
else
    ans[i][j] = min(ans[i-1][j], 1 + ans[i][j - coins[i]])
```

☆ Optimal Binary Search Tree

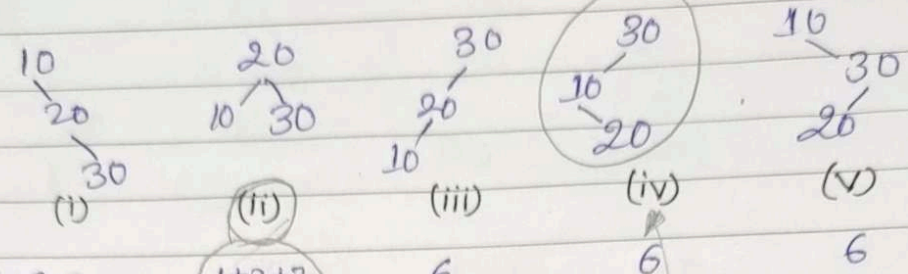
- An OBST, also known as weighted BST, is a BST that minimizes the expected search cost (the no. of comparison) required to search a key.
- The expected search cost of a node is the sum of the product of its depth and weight.
- For 'n' node → $T(n) = \frac{2nC_n}{n+1}$, BST possible.

Date :

Eg 10, 20, 30 , $T(n) = \frac{6C_3}{4} = \frac{20}{4} = 5$

Key = 5 2 6

OBST



No. of comparision → 1+2+3 = 6 | 1+2+2 = 5 Height balanced. | 6 | 6 | 6

Cost → 5+2x2+6x3 = 27 | 2+10+12 = 24 | 6+4+15 = 25 | 6+10+6 = 22 minimum cost | 5+12+6 = 23

★ Matrix Chain Multiplication:

Problem: Given the dimension of a sequence of matrix in an array arr[], where the dimension of i^{th} matrix is (arr[i-1] x arr[i]), the task is to find the most efficient way to multiply these matrices together such that the total no. of elements multiplication is minimum.

Eg. $A_1 \cdot A_2 \cdot A_3 \cdot A_4$

5x4 4x6 6x2 2x7

$d_0\ d_1\ d_1\ d_2\ d_2\ d_3\ d_3\ d_4$

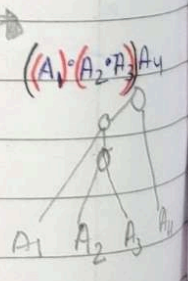
, No. of ways = $\frac{2nC_n}{n+1}$

M

	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

S

	1	2	3	4
1		1	1	3
2			2	3
3				3
4				



M[1,2] = $A_1 \cdot A_2$; M[2,3] = $A_2 \cdot A_3$

5x4 4x6 = 5x4x6 = 120 4x6 6x2 = 4x6x2 = 48

Date :

$M[3,4] = A_3 \cdot A_4$; $M[1,3] = (A_1 \cdot A_2) A_3$ or $A_1 \cdot (A_2 \cdot A_3)$

$6 \times 2 \quad 2 \times 7$ $\quad$ $5 \times 4 \quad 4 \times 6 \quad 6 \times 2$ $\quad$ $5 \times 4 \quad 4 \times 6 \quad 6 \times 2$

$= 6 \times 2 \times 7 = 84$

$= M[1,2] + M[3,3] + 5 \times 6 \times 2$ $\quad$ $M[1,1] + m[2,3] + 5 \times 4 \times 2$

$= 120 + 0 + 60 = 180$ $\quad$ $= 0 + 48 + 40 = 88$ min

General Formula $\Rightarrow$

$$M[i,j] = \min_{i \le k \le j} \{ m[i,k] + m[k+1][j] + d_{i-1} \times d_k * d_j \}$$

$T.C. = \Theta(n^3)$

☆ Longest Common Subsequence

Problem: Given two string S1 and S2, the task is to find the longest common subsequence i.e. longest subsequence present in both strings.

S_1 = Stone , S_2 = longest

S_1 \ S_2		L	O	n	g	e	s	t
	0	0	0	0	0	0	0	0
S	0	←0	←0	←0	←0	←0	1	←1
t	0	←0	←0	←0	←0	←0	↑1	2
o	0	←0	1	←1	←1	←1	←1	↑2
n	0	←0	↑1	2	←2	←2	←2	←2
e	0	←0	↑1	↑2	←2	3	←3	←3

one

→ Length of Common Sequence = 3

First create $(n+1) \times (m+1)$ matrix, fill 0^{th} row & col = 0;

```
if (S1[i] != S2[j])
    C[i][j] = max(C[i][j-1], C[i-1][j])
}
else
{   C[i][j] = 1 + C[i-1][j-1]
}
```

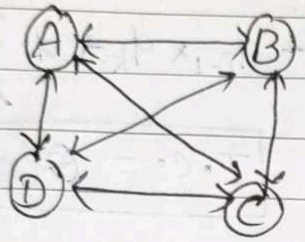
Date :

★ Travelling Salesman Problem:

↳ There should be complete graph.

↳ find minimum weight Hamiltonian Cycle (traverse each node once).

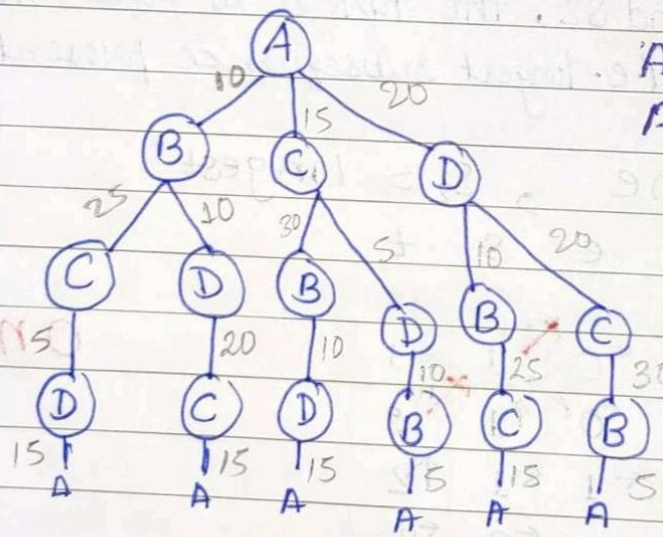
Eg



	A	B	C	D
A	0	10	15	20
B	5	0	25	10
C	15	30	0	5
D	15	10	20	0

Greedy: → A $\xrightarrow{10}$ B $\xrightarrow{10}$ D $\xrightarrow{20}$ C $\xrightarrow{15}$ A ⇒ Total cost = 55

Dynamic Programming:



Path	Cost
ABCDA	⇒ 55
ABDCA	⇒ 55
ACBDA	⇒ 70
ACDBA	⇒ 35
ADBCA	⇒ 70
ADCBA	⇒ 70

∴ Our Optimal path is A→C→D→B→A

Date :

★	Divide & Conquer	Dynamic Programming
1)	Subproblems are solved independently.	→ Consider large no. of decision sequences & all overlapping substance
2)	It is recursive.	→ It is non-Recursive.
3)	It is TOP-down approach.	→ It is Bottom-up approach.
4)	Subproblems are independent of each other.	→ Subproblems are Interdependent.
5)	Eg: MergeSort, Binary Search	→ Matrix multiplication.

★	Dynamic Programming	Greedy approach
1)	DP is used to obtain optimal solution.	→ It is also used to get optimal solution.
2)	In DP, we choose at each step, but choice depends on the sol^{n}. of subproblems.	→ We choose immediate best solution at that moment.
3)	It guranteed an optimal solution.	→ It doesnot guranteed for an optimal solution.
4)	less efficient as compared to Greedy algo.	→ More efficient.
5)	Eg: 0/1 Knapsack	→ Eg: Fractional Knapsack.

UNIT: 2.3 Dashrath Nandan Date :

Backtracking:

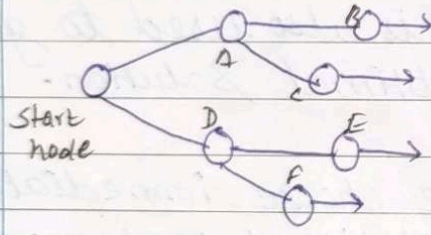
It is one of the technique that can be used to solve the problem. It uses the Brute force search to solve the problem.

↳ DFS approach.

* General step of backtracking:

- Start with a sub-solution.
- Check if this sub-solution will lead to the sol^n. or not.
- If not, then come back and change the sub-solution.

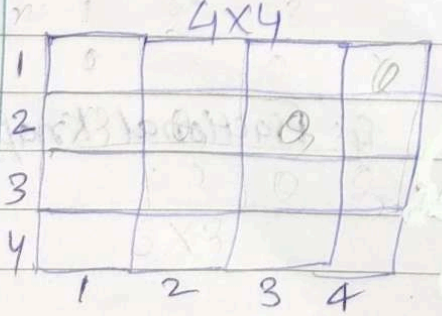
* State Space Tree

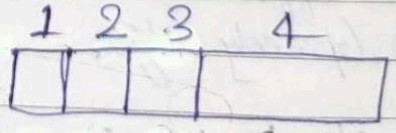


- Live node: The node that can be further generated.
- Success node: Node that provides a feasible sol^n.
- Dead node: Node that cannot be further generated.

☆ N Queens on N×N chessboard:

The N queen is the problem of placing N queens on a N×N chessboard so that no two queen attack each other.



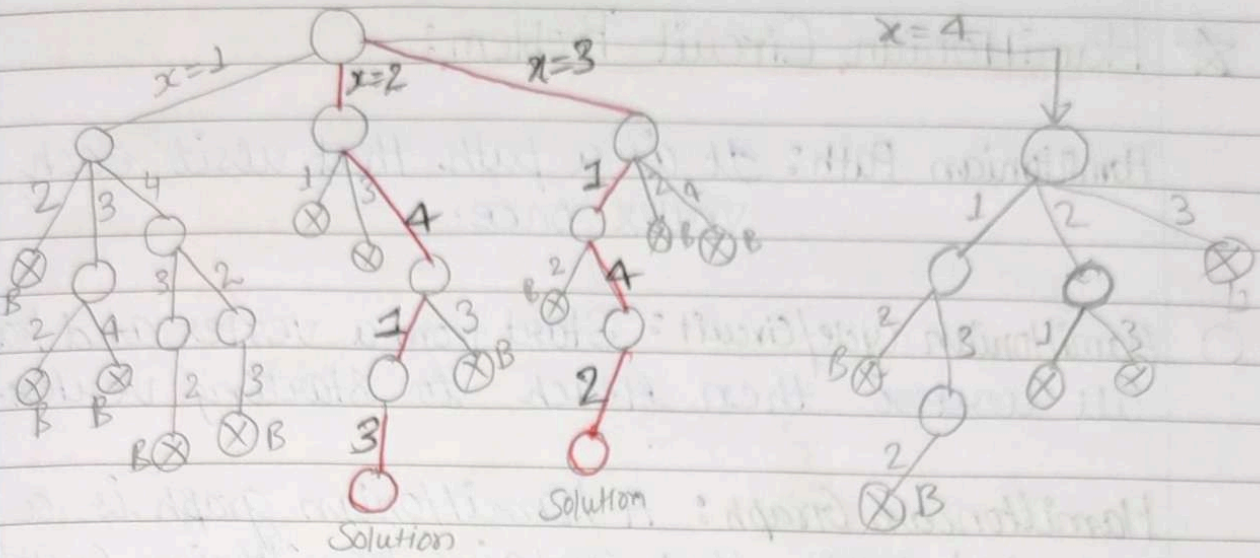


x ⇒ Column number

→ It is a depth wise traversal.

→ if we place any 'Q' in a col. then next 'Q' cannot be placed in same col.

Date :



Two Sol^n :→ | 2 | 4 | 1 | 3 | and | 3 | 1 | 4 | 2 |

1st Q 2nd Q 3rd Q 4th Q

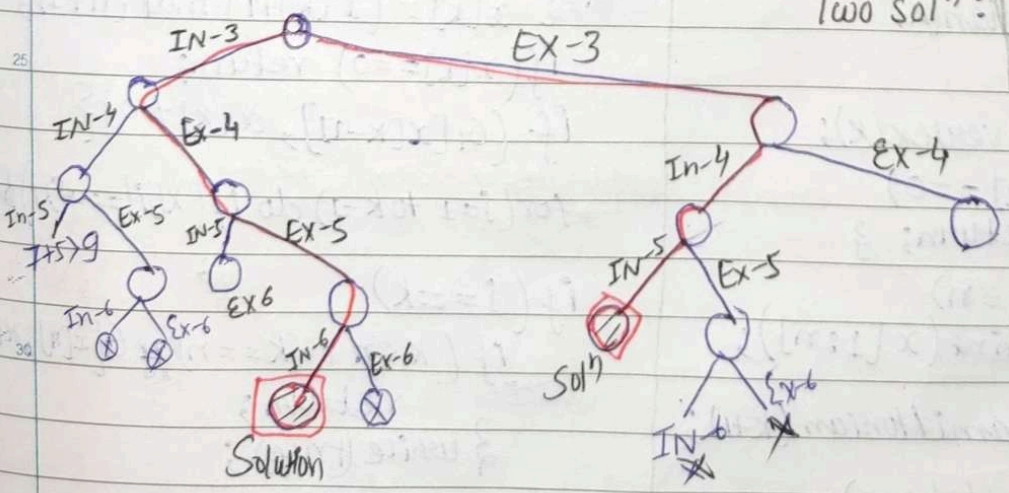
col. no

☆ Subset Sum Problem [Using Backtracking]

Find the subsets of the given set 'S' where the elements of the set S are no positive integers, such that sum of all element is equal to some 'X'.

Eg: S = (3, 4, 5, 6) and X = 9

	3	4	5	6
Two Sol^n :	1	0	0	1
	0	1	1	0



Date :

☆ Hamiltonian Circuit Problem:

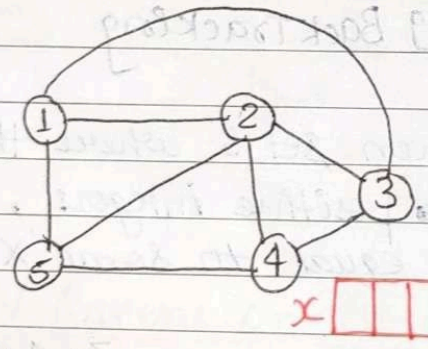
Hamiltonian Path: It is a path that visit each vertex once.

Hamiltonian Cycle/Circuit: Start from a vertex and traverse all vertex then reach to starting vertex.

Hamiltonian Graph: A hamiltonian graph is a connected graph that contains hamiltonian cycle/circuit.

* Problem statement:
Given a graph $G=(V,E)$ we have to find the Hamiltonian circuit using Backtracking.

Eg



$G =$

	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	1	1
3	1	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

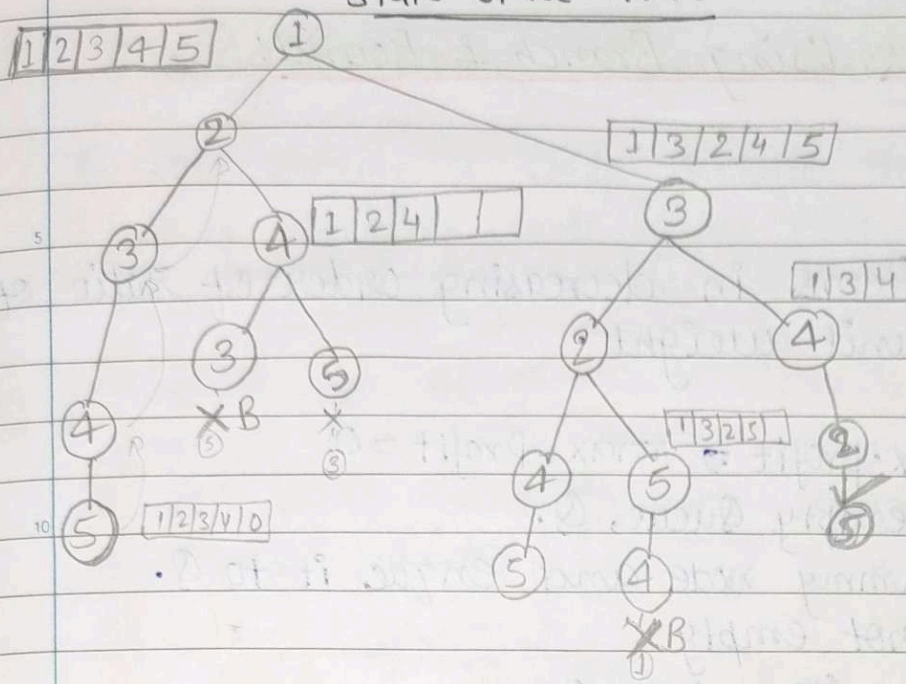
Algo:

```
Hamiltonian(k)
{
do{ Next vertex(k);
   if(x[k]==0)
      return; }
   if(k==n)
      Print(x[1:n]);
   else
      Hamiltonian(k+1);
} while(true);
```

```
Algorithm Next vertex(k){
do{ x[k]=(x[k]+1) mod (n+1);
   if(x[k]==0) return;
   if(G[x[k-1], x[k]≠0){ edge b/w x[k-1] & x[k]
duplicate vertex check-
   for(j=1 to k-1) do if(x[j]==x[k]) break;
   if(j==k)
      if(k<n or (k==n) && G[x[n],x[1]]≠0)
         return;
} while(true);
}
```

Date :

State Space Tree



Solutions: 1 2 3 4 5 , 1 3 2 4 5 , 1 3 4 2 5 , ...

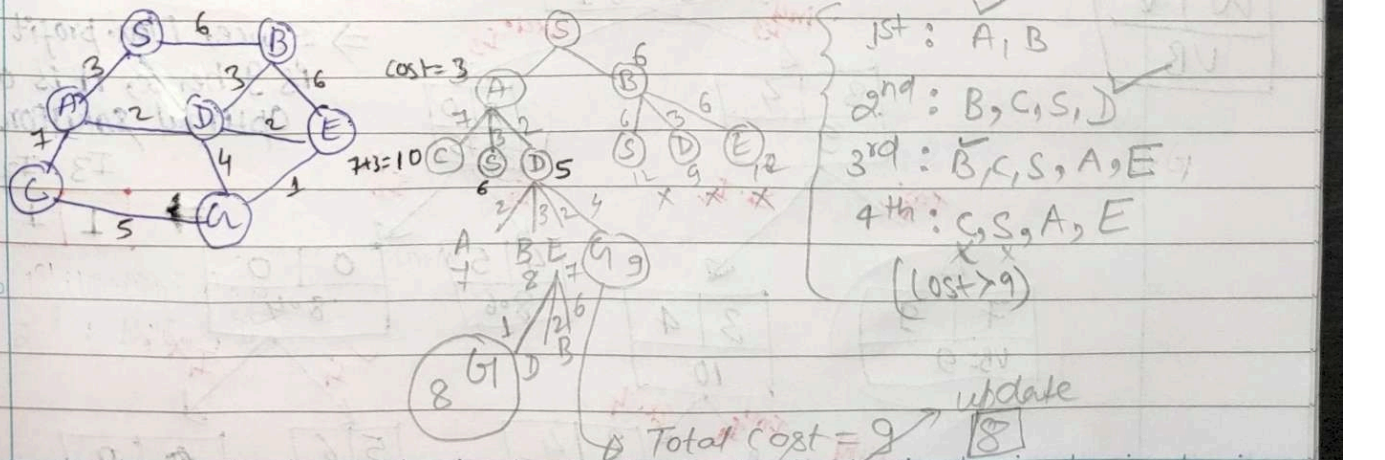
☆ Branch and Bound:

→ It is an algorithm design paradigm which is generally used for solving combinatorial optimization problems.

↳ Similar to Backtracking but have BFS approach.

↳ State space tree

Problem: Reach goal state from start in minimum cost.



Date :

✱ 0/1 Knapsack Using Branch & Bound:

Algorithm:-

↳ Sort all the items in decreasing order of ratio of (Profit) Value per unit weight.

↳ initalize max.profit ⇒ max_Profit = 0
↳ Create an empty Queue, Q.
↳ Create a dummy node and enque it to Q.
↳ While Q is not empty
→ Extract an item from Q.
→ Compute profit of next level node. if P > Max P, Update max, Profit.

Eg

	Weight	Value	Value/Weight
I_1	5	6	6/5 = 1.2
I_2	4	5	5/4 = 1.25
I_3	3	4	4/3 = 1.33

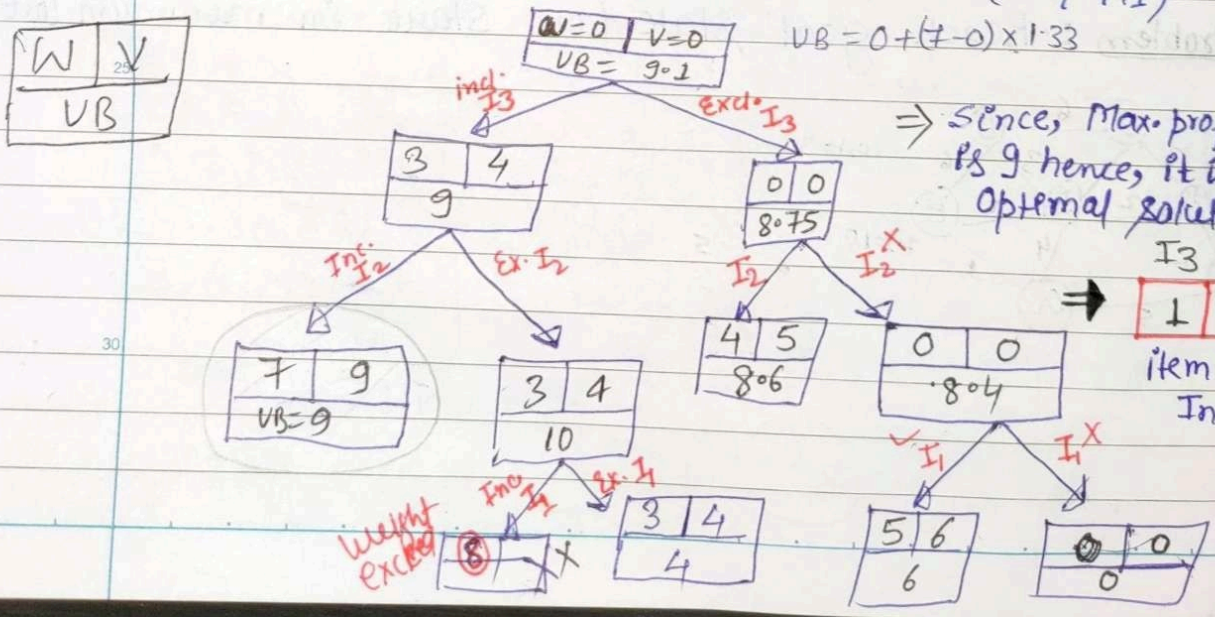
⇒ Sorting ⇒ Weight value Val/weight

I_3	3	4	1.33
I_2	4	5	1.25
I_1	5	6	1.2

capacity = 7.

Upper Bound (UB) = $V + (W-w) * (v_{i+1}/w_{i+1})$

W	V
UB	



⇒ Since, Max. profit obtained is 9 hence, it is only optimal solution.

I_3	I_2	I_1
1	1	0

⇒ item 3 and 1 are Included.

Date :

☆ Traveling Salesman Problem using B & B: [youtube]

The cost through a node include two cost —
(i) Cost of reaching node from root.
(ii) Cost of reaching an answer from current node.

eg

	1	2	3	4	5	
1	∞	20	30	10	11	10
2	15	∞	16	4	2	2
3	3	5	∞	2	4	2
4	19	6	18	∞	3	3
5	16	4	7	16	∞	4
	3					21

→

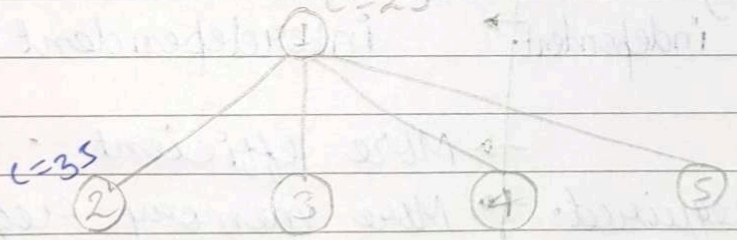
	1	2	3	4	5	
1	∞	10	20	0	1	0
2	13	∞	14	2	0	0
3	1	3	∞	0	2	0
4	16	3	15	∞	0	0
5	12	0	3	12	∞	0
	1	0	3	0	0	21

↳ Select the min values within row and write them
↳ Subtract the min. within the row and generate new matrix.

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	10	12	∞

reduced matrix of (1)

21+4=25



1-2:
Make first row 2nd col. ∞ and 2→1 also ∞.

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	∞	∞	11	2	0	0
3	0	∞	∞	0	2	0
4	15	∞	12	∞	0	0
5	11	∞	0	12	∞	0
	0	0	0	0	0	0

∴ $CP_2 = CP(1,2) + r + r$ (r in reduced matrix of 1)
$= 10 + 25 + 0 = 35$

Date :

☆ Backtracking	Branch & Bound
i) It solves decision problem.	→ It solve optimisation problem.
ii) It uses DFS.	→ It uses BFS or DFS.
iii) More efficient.	→ Less efficient.
iv) It contains feasibility func^n.	→ It contains Bounding func^n.
v) It can solve almost any Problem( )	→ Cannot solve almost any problem.
vi) Applic^n: N-Queen, Subset Sum, etc.	→ 0/1 Knapsack, TSP.

☆ Divide & Conquer	DP
i) Follow TOP down approach.	→ Bottom-up approach
ii) Used to solve decision Problem.	→ Solve optimization problem.
iii) Time Consuming.	→ less Time consuming.
iv) Subproblems are independent.	→ interdependent.
v) less efficient	→ More efficient.
vi) Less memory required.	→ More memory req.
vii) Ex: Merge Sort	→ 0/1 Knapsack.

UNIT: 3

Dashrath Nandan

Date :

★ Graphs:

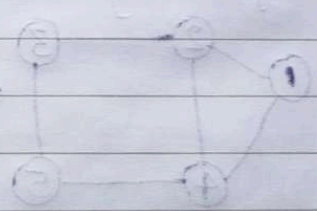
A graph is a non-linear data structure consisting of nodes and edges (vertices and edges).

$G = (V, E)$

* Terminologies:

- Vertex; Edge; Adjacency: Two nodes or vertices are adjacent if they are connect to each other through an edge.
- Path: Path represent a sequence of edges b/w two vertices.
- Cycle; Digraph: is also known as directed graph.

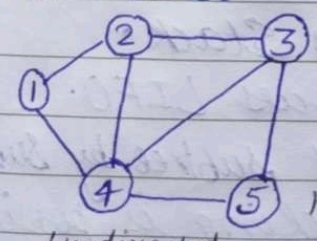
* Basic Operations → Add Vertex, Add Edge, Display Vertex



★ Graph Representation:

A graph representation is a technique to store graph into the memory of computer.

1) Adjacency Matrix: It is a sequential representation. It is used to represent which node are adjacent to each other.



Undirected graph

$n \times n$

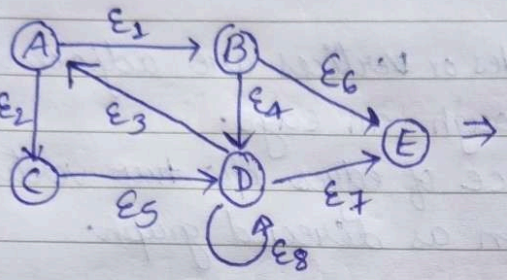
↓ no. of vertices

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

Adjacency matrix

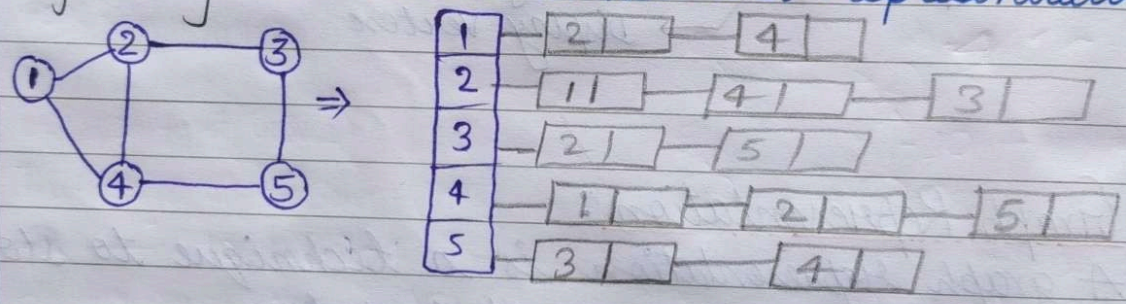
2.) **Incidence Matrix**: In this, a graph can be represented using a matrix of size $n \times m$; n → no. of vertices, m → no. of edges

0 → is used to represent which is not connected.
1 → is used to represent outgoing edge to column Vertex.
-1 → is used to represent Incoming edges.



	ε1	ε2	ε3	ε4	ε5	ε6	ε7	ε8
A	1	1	-1	0	0	0	0	0
B	-1	0	0	1	0	1	0	0
C	0	-1	0	0	1	0	0	0
D	0	0	1	-1	-1	0	1	1
E	0	0	0	0	0	-1	-1	0

3.) **Adjacency List**: It is a linked list representation.



★ **Graph Traversal**: → BFS, → DFS

Example:- PPt or Word.

BFS	DFS
i. Breadth First Search	• Depth First Search.
ii. It uses Queue	• It uses Stack.
iii. It follows FIFO.	• It follows LIFO.
iv. It build tree level by level.	• It built subtree by subtree.
v. No backtracking.	• Backtracking is possible.
vi. Require more memory.	• Require less memory.
vii. Slow.	• Fast.

Date :

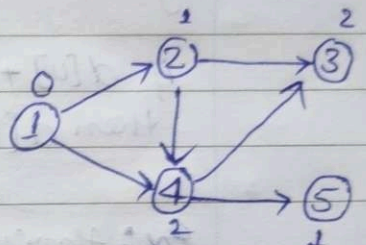
★ Topological Sort:

It is a linear ordering of the vertices in such a way that if there is an edge in the DAG going from vertex 'u' to vertex 'v', then 'u' comes before 'v' in ordering.

↳ Graph should be Directed Acyclic Graph (DAG).
↳ Every DAG will have atleast one topological ordering.

Algorithm:

Topological_Sort (G)
1. Call DFS (G) to compute finishing time v.f for each vertex v.
2. as each vertex is finished, insert it onto the front of a linked list.
3. return the linked list of vertices.



⇒ 1 2 4 3 5 or 1 2 4 5 3

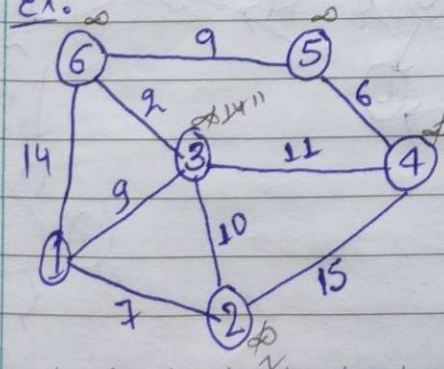
★ Dijkstra's Algorithm: Single Source Shortest Path

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G=(V, E) with non-negative edge weights.

- Relaxation:
  if $d(u) + C(u,v) < d(v)$
  $d(v) = d(u) + C(u,v)$

(1→2) $d(1) + C(1,2) < d(v)$
$0 + 7 < \infty$
$7 < \infty$ → So relax (2)

Ex:

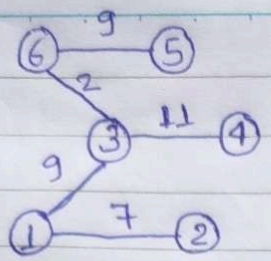


Source	Destination 2	3	4	5	6
	∞	∞	∞	∞	∞
1	(7)	9	∞	∞	14
1,2	(7)	(9)	22	∞	14
1,2,3	(7)	(9)	20	∞	(11)
1,2,3,6	(7)	(9)	(20)	20	(11)
1,2,3,6,4					

5 ∴ 1, 2, 3, 6, 4, 5

Date :

Sol:



Disadvantage of Dijkstra:

i) It does blind search, so waste lots of time.
ii) It can't handle negative edges.
iii) We need to keep track of visited nodes.

★ Bellman-Ford Algorithm : T.C. = O(E|V|-1)

Solve Single Source shortest path problem in which edge weight may be negative but no negative cycle exist (if total edge's weight is -ive).

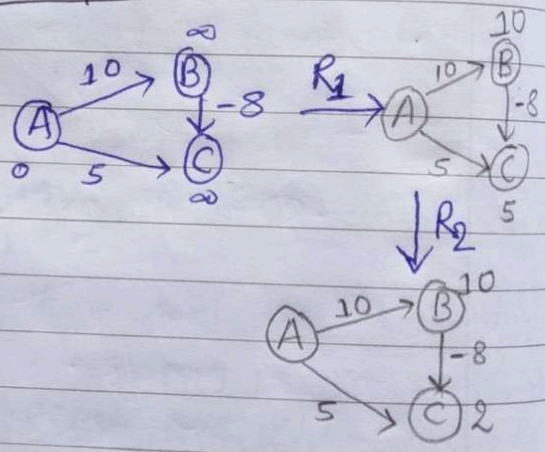
↳ go on relaxing all the edges (n-1) times.
↳ Used to detect negative cycle.

if $d[u] + c(u,v) < d[v]$ ⇒ then, $d[v] = d[u] + c(u,v)$

*** Algorithm:**

Step 1) Initialize distance and predecessors
Step 2) Relax edges (V-1) times.
Step 3) Check for negative cycles.
    if $d[v] > d[u] + c(u,v)$
        Print("Neg. Cycle Present");
Step 4) If No negative cycle present, return the shortest path

Example:

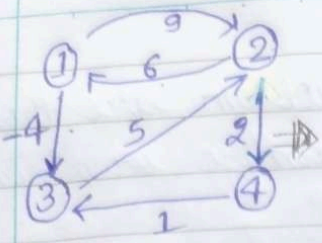


↳ It is slower than dijkstra algo.

★ Flyod Warshall Algorithm : All pair Shortest path

↳ It is used to find the shortest paths between all pairs.
↳ This algo. is highly efficient and can handle graphs with both ⊕ve and ⊖ive edge weights.
↳ It doesn't work for the graph with negative cycle.
↳ It follows D.P. approach to check every possible path.

Date :



Example:

$$D^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & \infty & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & 2 & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

D^1: 1 is the middle or going via 1.

$$D^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 9 & -4 & 11 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & -4 & 3 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & (12) & 6 & 1 & 0 \end{array}$$

$4 \to 3 \to 2 \to 1$

$$D^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & -4 & 3 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & 12 & 6 & 1 & 0 \end{array}$$

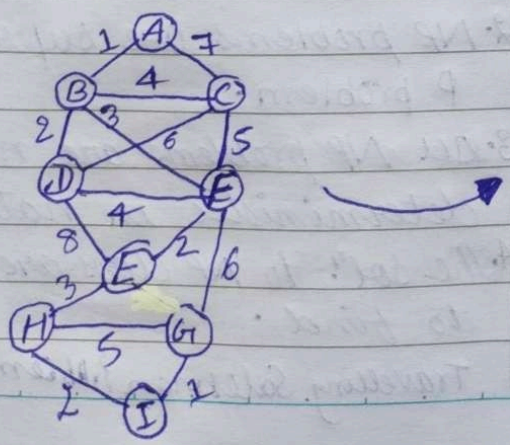
↳ No matter how many edges are there in graph the Flyod Warshall Algo. runs for $O(V^3)$ times, therefore it is best suited for dense graph.

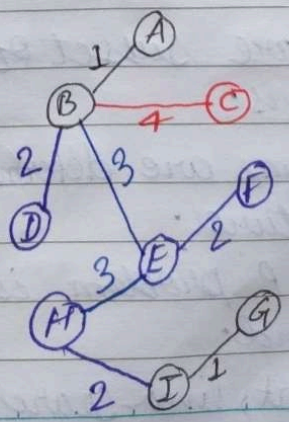
★ **Sollin's Algorithm**: Also Known as Boruvka's Algorithm

It is used to find MST.
It is a greedy algo. and Similar to prim's & Kruskal.

Steps:

1. Input a connected, weighted and undirected graph.
2. Initalize all vertices as individual sets.
3. Initalize MST as empty.
4. While there are more than one set, do following
   a) find the closet weight edge that connects this set to another.
   b) Add this closet edge to MST if not already.
5. Return MST.





Date :

* Optimization Problem : The optimization problem are those for which the objective is to maximize or minimize some values.

* Decision Problem: There are many problem for which the answer is Yes or a No. These are known as decision problem.

* Complexity classes: A complexity class is a set of problem with related complexity.
  Types : P class, NP class, CoNP class, NP hard, NP Complete.

* P Class : The P in the P class stand for Polynomial Time. It is a collection of decision problem that can solve by deterministic machine.

↳ P is often a class of computational problems that are solvable & traceable. Ex: Calculating the GCD.

* NP Class: The class consist of those problems that are verifiable in polynomial time. Every problem in this class can be solved in exponential time using exhaustive search.

P class Problems	NP Class Problems
1. Defination	1. Defination
2. P problems are subset of NP problems.	2. NP problems are superset of P problem.
3. All P problems are deterministic in nature.	3. All NP problems are non-deterministic in nature.
4. The sol^{n}. to P problem is easy to find.	4. The sol^{n}. to NP class are hard to find.
5. Ex: Selection sort, linear Search.	Travelling Salesman Problem.

Date :

* NP-Hard: An NP-hard problem is at least as hard as the hardest problem in NP and it a class of problems such that every problem in NP reduces to NP-hard.

* NP-Complete class: A problem is NP complete if it is both NP and NP-hard. They are the hard prob. in NP.
Ex. Hamiltonian Cycles, Vertex Cover.

★ Proof of NP-hard & NP-Complete:
⇒ A language B is NP complete if it satisfies the two cond^n:
- B is in NP.
- Every A in NP is polynomial time reducible to B.

If a language satisfies the second property, but not necessarily the first one, the language B is known as NP-hard.

★ Euclidean Algorithm for GCD:
We have to find GCD (Greatest common divisor) of two non-negative integers, gcd(a, b).
⇒ The algorithm is based on the fact that -
If we substract a smaller number from a larger number, GCD doesn't change. So, if we keep substracting repeatedly then we end up with GCD.

Pseudo Code:
1. Let a & b be two number.
2. a mod b = R
3. let a=b & b=R
4. Repeat steps 2 and 3 until a mod b > 0.
5. GCd = b
6. Finish

Ex gcd(12,33)

Q	A	B	R
2	33	12	9
1	12	9	3
	9	3	0
	3	0	

in this step B=0

∴ GCD=A

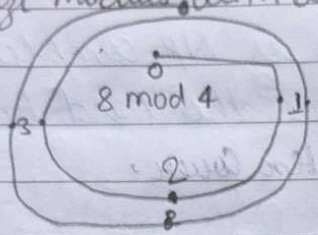
Date :

* Modular Arithmetic:

$\frac{A}{B}$ = Q remainder R ;

A is dividend
B is divisor
Q is Quotient
R is Remainder

Ex Visualize moduls with clock:



★ Chinese Remainder Theorem:

It states that there always exists an "x" that satisfies the given congruence -

$x = rem[0] \pmod{num[0]}$

$x = rem[1] \pmod{num[1]}$

and, (num[0], num[1], ... num[n-1]) → all must be coprime to one another.

Working:

if $x = a_1 \pmod{m_1}$

$x = a_2 \pmod{m_2}$

$x = a_3 \pmod{m_3}$

i) $gcd(m_1, m_2) = gcd(m_2, m_3) = gcd(m_1, m_3) = 1$

ii) $x = (M_1 x_1 a_1 + M_2 x_2 a_2 + M_3 x_3 a_3 + \cdots + M_n x_n a_n) \bmod M$

- $M = m_1 \times m_2 \times m_3 \times \cdots \times m_n$
- $\boxed{M_i = \frac{M}{M_i}}$

To calculate x_i, $M_i x_i = 1 \bmod m_i$

Date :

Ex: $x \equiv 1 \pmod 5$, $x \equiv 1 \pmod 7$, $x \equiv 3 \pmod{11}$

Sol:- $a_1 = 1,\ a_2 = 1,\ a_3 = 3$; $[m_1 = 5,\ m_2 = 7,\ m_3 = 11]$ → They are all coprime

$M = m_1 \times m_2 \times m_3$

$M = 5 \times 7 \times 11 = 385$; $M_1 = \frac{M}{m_1} = \frac{385}{5} = 77$; $M_2 = 55,\ M_3 = 35$

Now,

$M_i x_i = 1 \bmod m_i \Rightarrow 77 \times x_1 = 1 \bmod 5$

$\Rightarrow \boxed{m_i \cdot x_i \pmod{m_1} = 1}$ | $77 \times x_1 \pmod 5 = 1$

$2 \times x_1 \pmod 5 = 1$ ($77 \div 5$, rem = 2)

$\therefore \boxed{x_1 = 3}$

$\Rightarrow M_2 x_2 = 1 \pmod{M_2} = 55 \times x_2 \pmod 7 = 1 \Rightarrow 6 \times x_2 \pmod 7 = 1$.

$\boxed{x_2 = 6}$

[$6 \times x_2$ = value of $7k$ min. multiple + 1]

$\Rightarrow M_3 x_3 \pmod{m_3} = 1 \Rightarrow 35 \times x_3 \pmod{11} = 1$

$1 \times x_3 \pmod{11} = 1$

$\Rightarrow \therefore \boxed{x_3 = 6}$

Now,

$x = (M_1 x_1 a_1 + M_2 x_2 a_2 + M_3 x_3 a_3) \bmod M$

$= (77 \times 3 \times 1 + 55 \times 6 \times 1 + 35 \times 6 \times 3) \bmod 385$

$= 1191 \bmod 385$

$= 36$ Ans

Verify:-

$36 \bmod 5 = 1$ ✓

$36 \bmod 7 = 1$ ✓

$36 \bmod 11 = 3$ ✓

☆ String Matching / Manipulation Algorithm:

1) Naive brute-force Algorithm: Naive pattern Searching is the simplest method among other pattern searching algorithm.

↳ It checks for all character of the main string to the pattern.

Algorithm: Naive-string-matcher(T, P)

1. $n = T.length$
2. $m = P.length$
3. for $S = 0$ to $n - m$
4. if $P[1 \ldots m] = T[S+1 \ldots S+M]$
5. Print "Pattern occur with shift S";

T.C. = $O(N^2)$ or $O(N \times m)$ (m: pattern length, N: string length)

∴ Best case = $O(N)$

Date :

2) Rabin-Karp Algorithm:

The Rabin-Karp algorithm is string matching algo. Calculates a hash value for pattern, as well as each M-character subsequence of text to be compared.

* Step-by-Step approach: (Algorithm)
  - Initially calculate the hash value of pattern.
  - Start iterating from the starting of the string.
    - Calculate the hash value of the current substring having length m.
    - If the hash value of the current substring and the pattern are same check if substring is same as the pattern.
    - If they are same, store the starting index as a valid answer. Otherwise, continue for next substring.
  - Return the starting indices as the required answer.

* Time Complexity: $\theta(n-m+1)$ (n: string length, m: pattern length)

  Worst Case: $O((n-m+1)m)$

Ex: Text/string: a a a a a b (1 2 3 4 5 6)

Pattern: a a b, m=3 (1 2 3)

Sol: Pattern = a a b

value = 1 + 1 + 2 = (4) ← Hash code

hash func^n → h(P)

assign, by own →

a = 1
b = 2
c = 3
d = 4
:

Substring length = 3

String: a a a a a b

(i) 1 + 1 + 1 = 3 → No match

(ii) a a a a a b

1 + 1 + 1 = 3 → No Match

(iii) a a a a a b

1 + 1 + 1 = 3 → no match

(iv) a a a a a b

1 + 1 + 2 = (4) ← Match

compare with pattern

Pattern: a a b

Yes it's a match.

Date :

3) KMP (Knuth-Morris-Pratt) algorithm : T.C. = O(N)

KMP is an algorithm, which checks the character from left to right. When a pattern has a subpattern appears more than one in the subpattern, it uses that property to improve time complexity.

* Component of KMP:

① The prefix function ($\sqrt{\Pi}$): The Π for a pattern encapsulate knowledge about how the pattern matches against the shift of itself. This enable avoiding backtracking of the string.

② The KMP Matcher: With string 's', pattern 'p' and prefix func^n. 'Π' as inputs, find the occurrence of 'p' in 's' and return the number of shifts of 'p' after which occurrences are found.

Ex: String:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern:

j	0	1	2	3	4	5
	a	b	a	b	d	
	0	0	1	2	0	← Π value

Steps:

Step 1. Initialize i at 1 of string and j at 0 of Pattern.
2. Start comparing i with j+1
3. If they match, then move j to j+1 and i to i+1.
4. Repeat step3, till the i & j are not mismatch.
5. When there is mismatch, then move j to corresponding value of Π table, where j is currently pointing.
6. If j reaches 0 again then repeat above steps till all character of string are traversed.

Note: The basic idea behind the KMP algo. is that if the begining part of a string is appearing again somewhere else in the string then don't again compare the character. Avoid the comparison and don't move i in backward direction.

Date :

4) The Boyer-Moore Algorithm : [Youtube]

The B-M algorithm takes a 'backward' approach : the pattern string (P) is aligned with the start of the text string (T), and then compare the characters of a pattern from right to left, beginning with rightmost char.

The two strategies are called heuristics of B-M as they are used to reduce the search.

i) Bad character Heuristics
ii) Good Suffix Heuristics

i. **Bad character Heuristic** : The character of the text which doesn't match with the current character of pattern, is called Bad char. Upon Mismatch, we shift the pattern until -

- The mismatch become a match.
- The pattern P moves passed the mismatched character.

ii. **Good Suffix Heuristic** : After a mismatch which has a negative shift in bad char. heuristic, look if a substring of pattern matched till bad char. has good suffix in it, if it is so then onward jump equal to the length of suffix found.

Bad Heuristic.

Ex: T = A I N A I S E S T I Z A I N A I N E N , n = 19

P = A I N A I N E N (0 1 2 3 4 5 6 7) → m or length = 8

Sol:- Because B-M is a rightmost approach. So, we start from rightmost char. of the pattern P (i.e N) and its value is equal to value of ★ (length of P):

A	I	E	N	★ (for all characters that are not present in Pattern)
4	3	1	8	8

Bad Match Table

↳ for other character value will be -

Value = length − index − 1

→ Rightmost index of char.

Date :

$A = 8-3-1 = 4$, $I = 8-4-1 = 3$, $E = 8-6-1 = 1$

Now,

Bad match

T = A I N A I S E S T I Z A I N A I N E N

P = A I N A I N E N

↳ Compare the rightmost element of P to that of T.

↳ if they are mismatched then shift the pattern. The value of shifting is according to mismatched table and character is from T.

So,

mismatch value = 3

T = A I N A I S E S T I Z A I N A I N E N

P = A I N A I N E N

Shift by 8 b/c value of 'S' is ★.

So, shift by value of I, i.e 3.

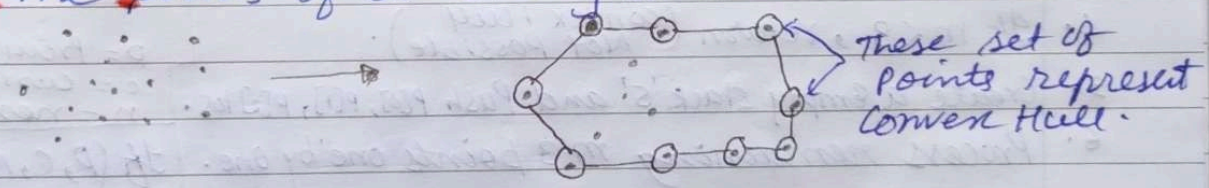
T = A I N A I S E S T I Z A I N A I N E N

P = A I N A I N E N

Shifted by 3

★ Convex Hull , (Geeks for Geeks)

Given a set of points in the plans, the Convex Hull of the set is the smallest convex polygon that contains all the points of it. It only consider outermost point.



1. Convex Hull using Jarvis's Algorithm or Wrapping

We select from the leftmost point (with min. x coord.) and we keep wrapping points in Counter-clockwise direc^n.

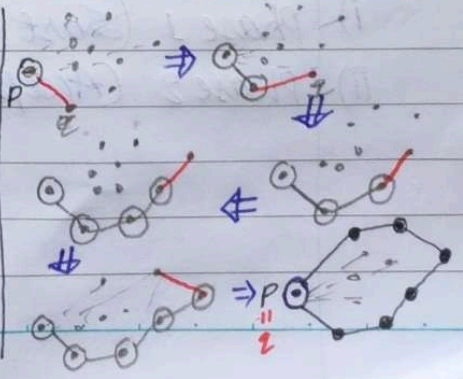
Algorithm : Step 1: Initalize P as leftmost point.

Step 2: Do following while we don't come back to P.

2.1: The next point q, such that the triplet (P, q, r) is counterclockwise for any point r.

2.2: next[P] = q (store q as next P in the output Convex Hull.)

2.3: P = q (Set P as q for next iteration)



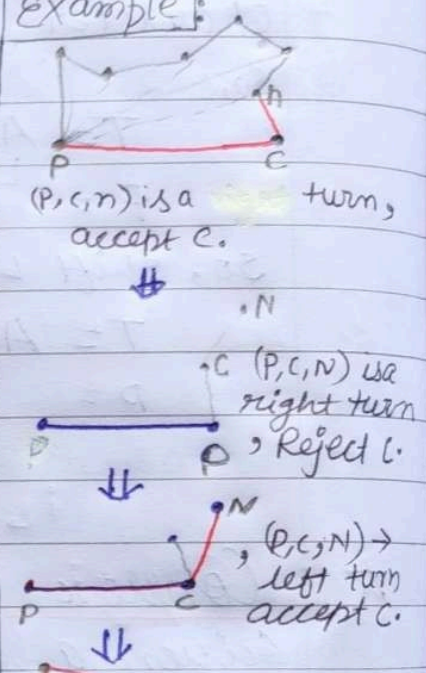
(2) Convex Hull Using Graham Scan Algorithm.

- The most case complexity of Jarvis's algo is $O(n^2)$
- Using Graham's scan algorithm, we can find convex Hull in $O(n \log n)$ time.

Algorithm:

1. Find the bottom most point by comparing coordinate of all points. If Y is same then compare X.
2. Consider the remaining n-1 point and sort them by polar angle in Counter CW order around points[0].
3. After sorting, if two points have same angle, then remove points except the point farthest from P0. let the size of new array be m.
4. If m < 3, return (convex Hull Not Possible).
5. Create a empty stack 'S' and Push P[0], P[1], P[2] to S.
6. Process remaining m-3 points one by one.

Example:



P = Previous
C = current
N = next

If (P, C, N) is a left-turn, then accept C, otherwise reject.

There are two phases-

i) Phase 1 (Sort Points)
ii) Phase 2 (Accept or Reject points):