DATA STRUCTURES AND ALGORITHMS

VOLUME - 2

Tree traversals

* The process by which we can reach Each and Every node of a Tree exactly once in some order.
* There are 4 Methods which we can traverse a tree and these 4 Methods are further divided into two categories.

1) Breadth first search / Level order search (BFS)
2) Depth first search (DFS)

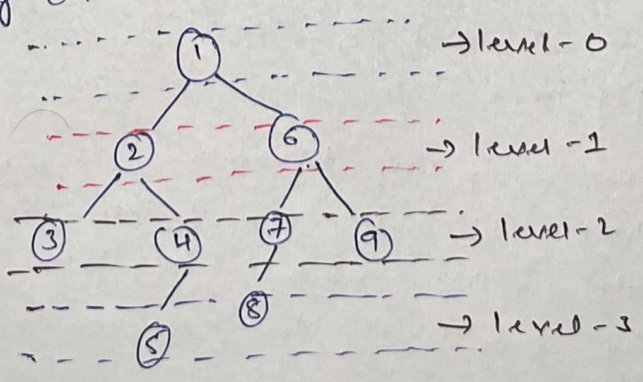
* we use the term search, in searching and in traversing we have different objective, but the task is going to remain same i.e reaching every node
* we perform ~~depth~~ breadth first search with the help of level order traversal

↳ What is Level order Traversal?

BFS ↳ A Binary tree is organized in different levels where the root node is at the topmost level (0th element). So the idea of the level order traversal is

↳ we first start processing from the root node, then we process all the nodes at the first level, 2nd, 3rd ....

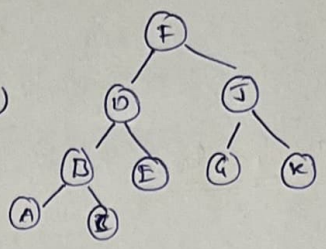
→ i.e we Explore all the nodes at the current level before moving on to the nodes at the next level.



BFS / level order Traversal 1, 2, 6, 3, 4, 7, 9, 5, 8

F, D, J, B, E, G, K, A, R

=>



DFS:-

Depth first search (DFS)

for any node

-> In BFS we visit all it's children before visiting any of it's grand children

↳ Suppose in the Above Example, we are visiting "F" & then we are visiting "D" and we are not going to any child of "D" like B/E along the Depth

DFS:- Unlike BFS in DFS approach if we would go to child, we would complete the whole subtree of the child before going to the next child

↳ In the above Example, from "F" the root node if we are going left to "D" then we should visit all the nodes in the left subtree. that is we should finish this left subtree in its complete depth (or) in other words we should finish all the grand children of "F" along the path. before going to right child of F i.e J

-> There are 3 popular Strategies of DFS

1) Root, left, right => Pre order Traversal
2) left, Root, right => In order Traversal
3) left, right, Root => post order Traversal

-> In total there are 6 possible permutations but conventionally left subtree is always visited before the right subtree, so above are 3 strategies that we use only the position of the "root" is changing here

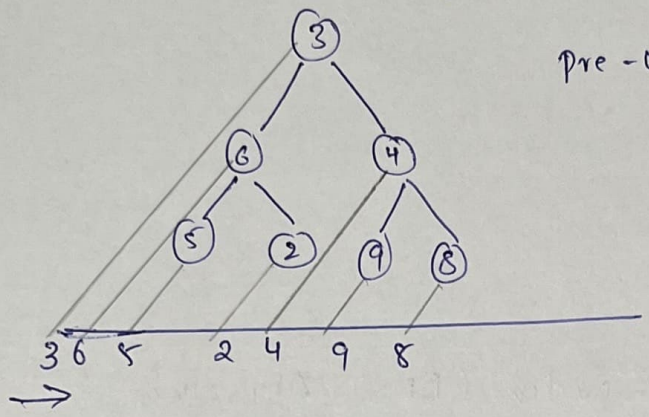
-> There an Easy way to ~~remember~~ remember these 3 depth first algorithms. If we can denote visiting a node (or) reading the data in that node with letter "D" going to the left subtree as "L", going to the Right subtree as R

D L R -> Pre order
L D R -> In order
L R D -> post order.

1) Pre-order (D, L, R)

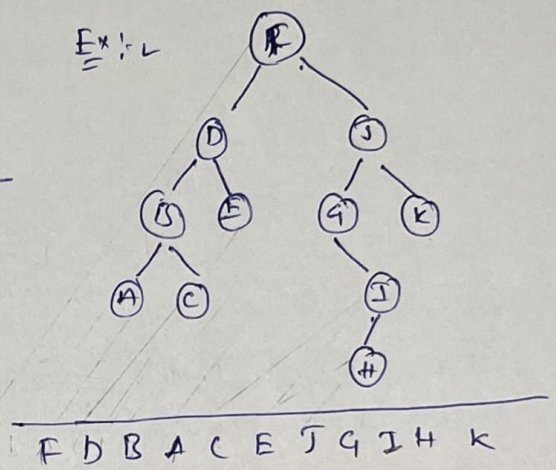
→ Take a straight line at the bottom of Tree

→ Then Draw a line from left hand side of the Every node so that it connects the bottom line. Make sure no - Two lines intercating.



Pre-order → 3, 6, 5, 2, 4, 9, 8

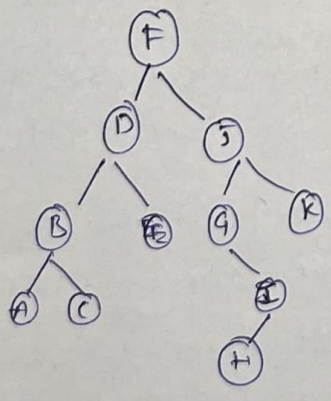
Ex:-



2) In-order (LDR)

↳ In - in order traversal we will first finish visiting the left subtree then visit the current node and then go right

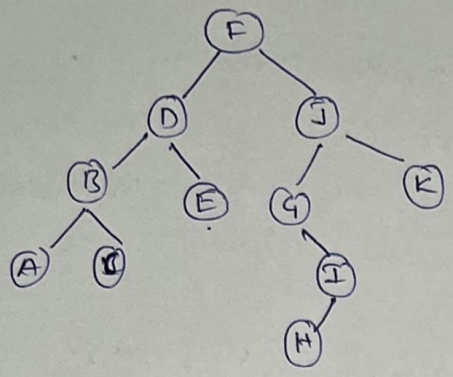
↳ We will start at the root and we first go left, once again for D we will first go left i.e to "B" & From B we will go left i.e "A" → Now for A there is no left so we can read the data. and now we can go right but there is no right Either, i.e done with this node.



=) In order (LDR) =) A, B, C, D, E, F, G, H, I, J, K

↑ no D-left

3) POST-order (L R D)



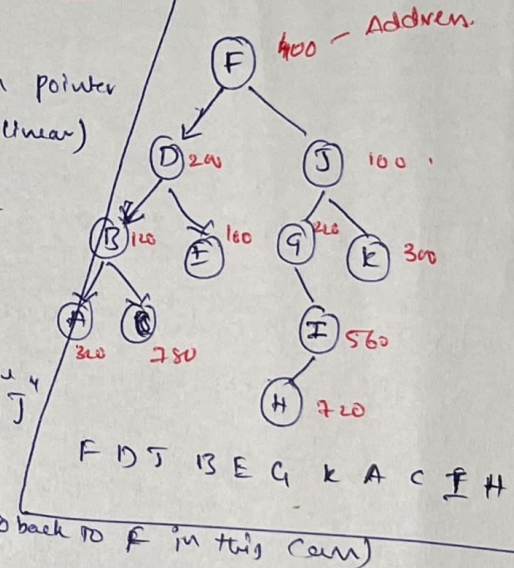
A, C, B, E, D, H, I, G, K, J, F

Level-order (BFS) Algorithm

=) Like linked list, we can't just have an pointer and keep moving it (as the Tree is non-linear)

=) let's say we have a point at F, it possible to move to nodes "D" / "J"

↳ As a conventional let say I moved to left node "D" → No I can not move to "J" (next Element in the same level) as the Tree ~~dare~~ Uni-directional. (we can't go back to F in this case)



→ So what we can do is, as we visit a node, we can reference (or address of all it's children in a "queue" so we can visit them later.

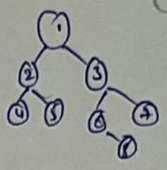
↳ A node in the queue can be called discovered node whose address is known to us. but we have not visited ~~at~~ it yet

↳ Initially we can start with the address of root node in the queue, to mean that initially this is the only discovered node.

↳ let's say in the Ex:- address of the root node(F) = 400 and Assume some random address for others

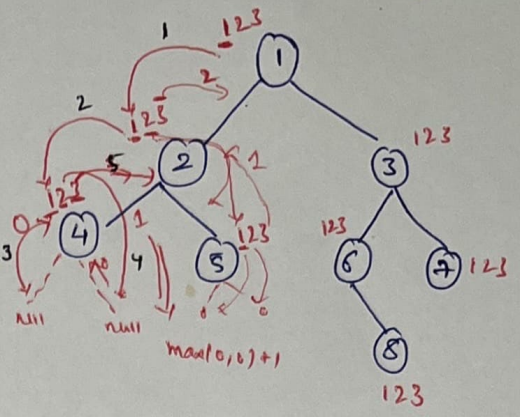
→ Finding height of node

→ or max depth



```
Public int getHeight (TNode node){
    if(node == null) return -1;
①   int lh = this.getHeight(node.left)
②   int rh = this.getHeight(node.Right)
③   return max(lh, rh) + 1;
}
```

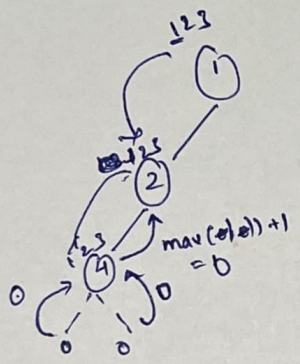
how many nodes from the ground level.
↳ 8 x 1 - 3 edges ⇒ 4 nodes



→ The 3 steps marked above should be performed for all the nodes. (so marking 1 2 3 at every node i.e these steps should be performed in seq manner for all node

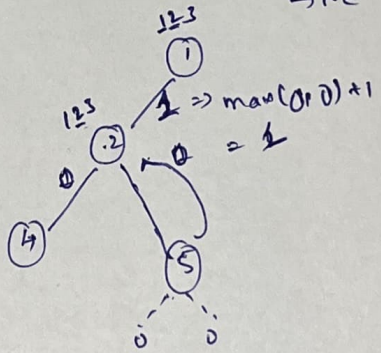
↳ Height (1) = 3

↳ Start with node (1)



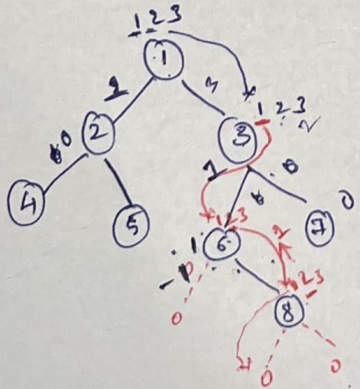
⇒

now we are step 2 of node 2
↳ i.e go to right sub Tree



⇒ now we are at step 2 of node 1

↳ i.e go to Right subtree of node 1.



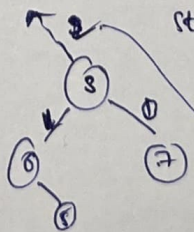
⇓

Step 2 of node 6 left Right

↳ max(-1, 0) + 1 ⇒ 1

⇓

This will pass node ③ Step 2 (visit right node) i.e 7

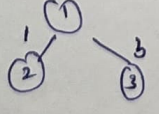


→ max(1, 2) + 1 = 2

now we are Step 3 of node 1

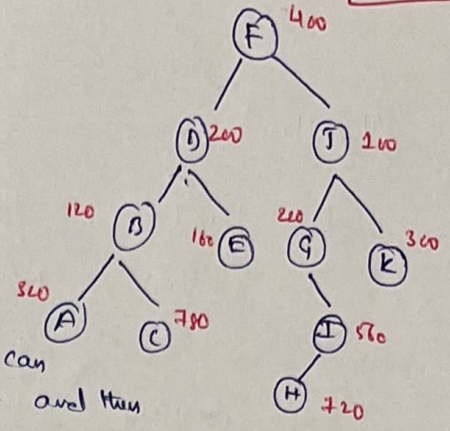
⇓

max



⇐

max(1, 2) + 1 ⇒ 3 ⇒ Height of node 1



→ Enqueue Root node (F).

↳ Storing node in the queue means Storing Address of it (or) reference.

→ Logic

↳ As long as the queue is not Empty, we can take out a node from the front, visit it and then Enqueue it's children.

Visit ⇒ Printing the value (In our case)

Step 1

Queue ← | 400 F | ←

Traversal (or) print | F |

F - Enqued & visited

~~visited~~ ⇒ After visit dequeue the node "F" & Enqueue it's Children.

Step 2 :- Enqueue the children of node "F"

Q | 200 D | 100 J |

⇒ Dequeue D → And Enqueue it's Children

↳ D children ⇒ B, E

T | ~~F~~ D |

⇒ Q | 100 J | 120 B | 160 E |

T = | F | D |

Step 3 → Dequeue J → And Enqueue it's Children

Q | 100 J | 120 B | 160 E | 220 G | 300 K | ← (G, K)

out

T | F | D | J |

⇒ Q | B | E | G | K | A | C | ← (A, C)

T | F | D | J | B |

Step 4

⇒ Q | E | G | K | A | C | (Dequeue E → There are no children so nothing to Enqueue)

T | F | D | J | B | E |

⇒ Q | G | K | A | C | I | ← I

T | F | D | J | B | I | G |

⇒ Q | K | A | C | I | ← no Elements to Enqueue

T | F | D | J | B | I | G | K |

S/g For others till the queue is empty

Algorithm

```
def levelorder (self, start):
    if start is None:
        return
    q = Queue()
    q.enqueue(start)
    traversal = []
    while len(q.items) > 0:
        traversal.append(q.peek())
        node = queue.dequeue()
        if node.left:
            q.enqueue(node.left)
        if node.right:
            q.enqueue(node.right)
    return traversal
```

→ Here Traversal we are not considering as we are using this for printing (used for internal logic not for actual requirement)

→ O(n)

→ Here Instead of append we can simply use print

all cases

Time complexity = O(n)

Space - complexity

↳ Here q will shrink and expand i.e q is dynamic

O(1) – best

O(n) → worst / Avg

main

```
tree = BinaryTree(5)
tree.root.left = Node(4)
tree.root.right = Node(6)
tree.root.left.left = Node(7)
```

=) BFS Time complexity = O(n)

BFS Space complexity = O(n)

↳ max no. of nodes in queue at at time.

↳) Best case => O(n) (for degenerated BinaryTree)

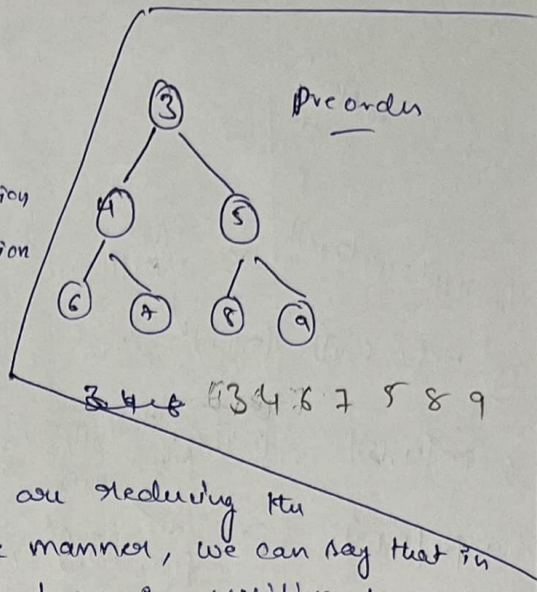
Worst/Avg case = O(n)

DS-50

Depth first Traversal (DFS)

- Pre-order
- In-order
- Post order

1) Pre-order (D, L, R)



* In DFS, if we go in one direction then we visit all the nodes in that direction (or), in other words we visit the complete sub-tree in that direction. And then only we go in other direction

* So if we see in this Approach we are reducing the problem in a self similar (or) recursive manner, we can say that in total visiting all the nodes in the tree is visiting the root node, visiting left subtree and visiting right subtree (or) other ways. (DLR, LDR, LRD)

1) Pre Order (D L R)

1) visit root
2) visit left sub tree
3) visit Right Sub tree

!! Continue this process in recursive => Break recursive for the subtree if root is null

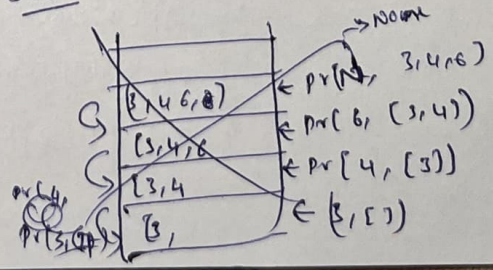
② // say If want to get the List of Elements then ↑ List

```
def preorder(self, root, Traversal):
    # Base case To break the recursion loop
    if root is none:
        return
    else:
        Traversal.append(root.value)
        self.preorder(root.left, Traversal)
        self.preorder(root.right, Traversal)
    return Traversal
```

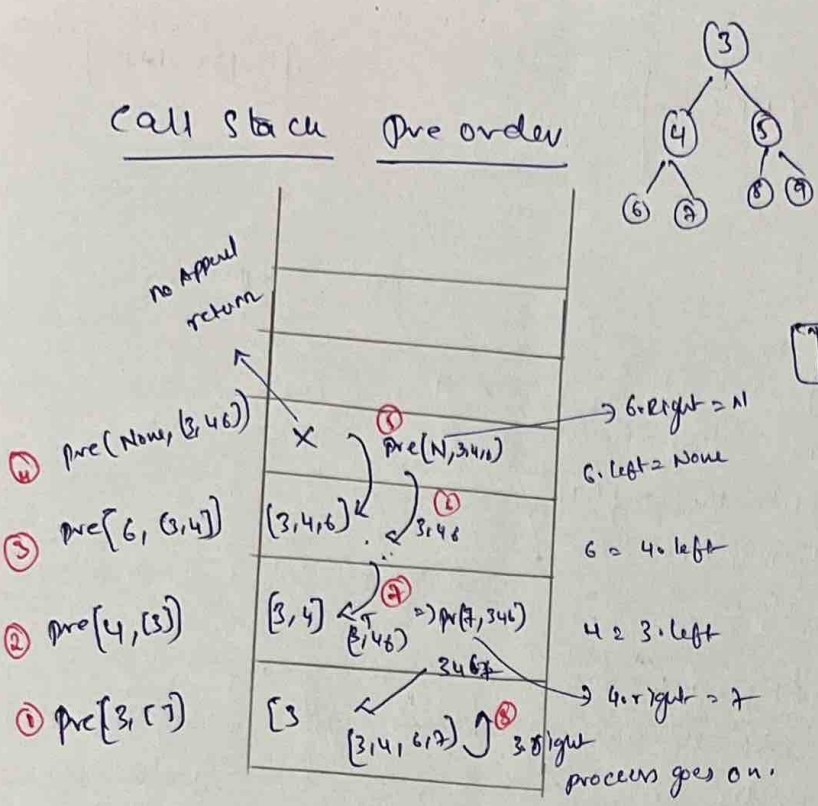
①

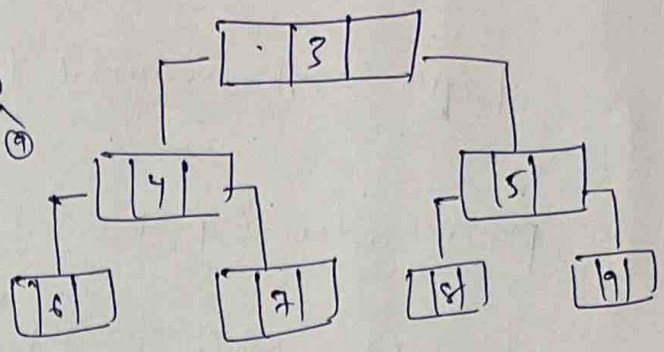
```
class Node():
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self, value):
        self.root = Node(value)

    def preOrder(self, root):
        if root is none:
            return
        # print the root
        print(root.value)
        self.preorder(root.left)
        self.preorder(root.right)
```

Stack



Call Stack Pre order






```
def pre order ( self, root, Traversal):
    if root is None:
        return
    else;
        Traversal.append (root.value)
        preorder (root.left, T)
        preorder (root.right, T)
```

(T)

2) In-order

```
def inorder ( self, start, traversal):
    # Left → Root → Right
    if start is None:
        return
    else:
        self.inorder (root.left, Traversal)
        # append Root
        Traversal.append (start.value)
        self.inorder (root.right, Traversal)
```

3) Post order

```
def Post order ( self, start, T )        Start = Root
    # Left → Right → Root
    if start is None:
        return
    else:
        self.postorder ( start.left , T )
        self.postorder ( start.right, T )
        self.T.append (start.value)
```

Pre order Iterative method → Right child must be pushed first so that left child is processed first (LIFO order).

```
Public static void PreorderIterative (Node root)
{
    if (root == null) {
        return;
    }
    // Create an Empty stack and push the root node
    stack <node> stack = new stack <> ();
    stack.push(root)
    // Loop till the stack is empty
    While (! stack.Empty()) {
        // Pop a node from stack & print it
        Node currentNode = stack.pop();
        System.out.println (currentnode.data);
        // Push the right child of the poped node
        if (currentNode.right != null) {
            stack.push(currentnode.right)
        }
        // Push the left child of the piped node
        if(currentNode.left != null){
            stack.push (currentnode.left)
        }
    }
}
```

⇒ Above solution can be further optimised by pushing only "Right" Child to the stack.

```
Node curr = root
While (! stack.Empty()) {
    if (curr != null) {
        Print (curr.data)
        if (curr.right != null){
            stack.push (curr.right)
        }
        curr = curr.left
    }
    else { curr = stack.pop() }
}
```

→ If the current node is null, pop a node from stack & set the current node to the popped one.

→ If the current node exits, print it and push it's right to the stack before moving to it's left

Iterative Inorder

```
Public static void inorderIterative (Node root)
{
    Stack <Node> stack = new Stack <> ();
    Node curr = root
// If the current node is null & the stack is also Empty we are done
    While ( !stack.Empty() || curr != null) {
                    * → If current node exists, push it into the stack and move to left child
        if (curr != null){
            stack.push (curr)
            curr = curr.left
        }
        else {   @ → Otherwise, if the current node is null, pop an element
                    From the stack, print it and finally
                    Set the current node to it's right child
            curr = stack.pop();
            System.out.print (curr.data);
            Curr = curr.right
        }
    }
}
```

Iterative - Post Order (we need an explicit ~~out~~ stack)

```
postOrderIterative (Node root) {
    if (root == null){
        return;
    }
    Stack <Node> stack = new Stack ();
    stack.push (root);
    Stack <Integer> out = new Stack <> ();
    while ( !stack.Empty() ){
        Node curr = stack.pop()
        out.push (curr.data)
        if (curr.left != null){
            stack.push (curr.left);
        }
        if (curr.right != null) {
            stack.push (curr.right);
        }
```

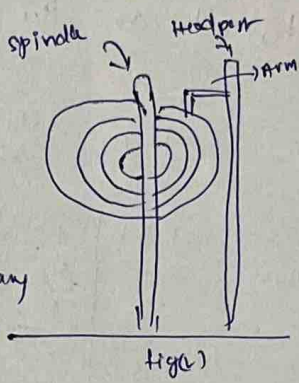
```
While ( !out.Empty ();){
    Print (out.pop());
}
```

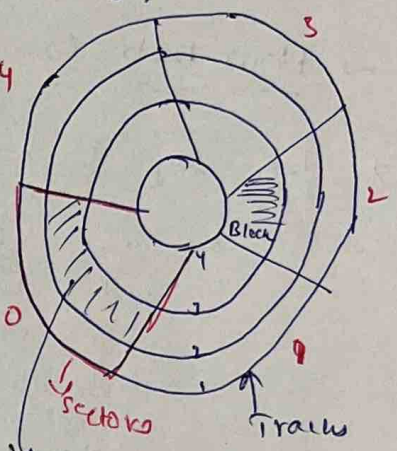
B-Trees and B+-Trees

→ Before we go to B & B+ trees we need to understand the below things.

1) Disk Structure
2) How data is stored on Disk
3) What is Indexing
4) What is Multi level Indexing
5) M-way search Trees
6) B-Trees
7) Insertion and Deletion B Trees
8) B+-Trees.

1) Disk Structure :-





→ Concentric logical circles are called Tracks

→ Tracks are divided into many Sectors (Red colour) Here we have 5 sectors (0 to 4)

→ Intersection portion of track and sector is Block

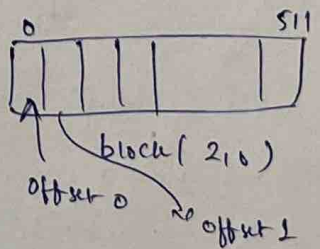
→ So Any location on the disk can be addressed with Track no. & sector number

i.e Address of the Block = (Track number, sector Number) ⇒

→ let's consider block size as 512 bytes (depends on manufacturers)

↳ when we read and write on a disk we always write in terms of blocks

↳ let us consider the block divided into multiple bytes Here 512 bytes i.e 0 to 511 bytes, Then each byte can have it's own address that is called "offset"



→ So To reach Each byte we should know ⇒ Track no, sector Number, offset

→ From Fig(1) → disk is mounted like shown on a spindle and the head post will also be there so by spinning it will change the sectors by the Arm movement of the head rest (i.e reading data (or) writing data) by Arm Movent Tracks are changed.

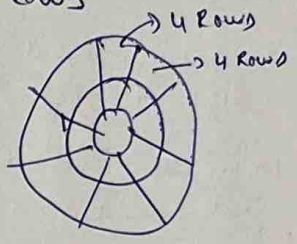
→ The Data can not be processed directly on disk it has to brought on to the main memory (RAM) and then access and process → once processed again the data copied to disk

→ Organizing the data that is directly used by the program that is in the memory is the JOb of data structure

→ Organizing the data Efficiently so that it can be easily utilized that is DBMS

→ How Data is stored on the Disk

* let say I have Employee table with 100 Rows



Block Size = 512 Bytes

Schema — Size

eid — 10 Bytes
name — 50
dept — 10
section — 10
addr — 10

Size = 128 Bytes

EId	name	...
1	A	128 Bytes
2	B	128 Bytes
3	C	128 Bytes
4	D	

100 Rows

↳ It means Each Row Size is 128 Bytes

→ When we want to store the table on the Disk / DB then Each Block How many rows can be stored?

No. of Records / Block = $\frac{512}{128}$ => 4 => Each Block can store 4 Records / Rows

for 100 Records = $\frac{100}{4}$ => 25 Blocks.

→ let's say now we have written a select query where eId =

↳ If I am searching then how much time it takes

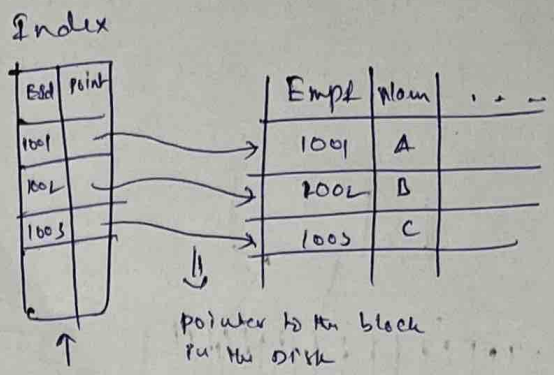
↳ i.e Time depends on the no. of blocks you are accessing Here we need to Access 25 blocks bcoz record may present any where

Records ↑ => Blocks ↑ => Search Time ↑

How can be reduced => using Indexes

→ In Index → we will store key (Emp's) and row pointer.

3) Indexing



→ Where do we store Index
  ↳ we store Index also in the Blocks in the Disk.

→ How much space Index taking?

Index → Empl = 10 Bytes
  ↳ Record point = 16 Bytes (Assume)

Each Index record will take ⇒ 16 Bytes

For Each Record we will a Entry

→ Say we have 100 Records in Employee Table. So 100 records in Index table ⇒ ~~in the 100~~

No. of Entries / Block = 512 (Bytes Block size) / 16 ⇒ 32

→ i.e per block we can store 32 - Entries

For 100 Entries ⇒ 100/32 ≈ 3.2 ≈ 4 Blocks
  ↳ So we required 4 blocks to store 100 Records

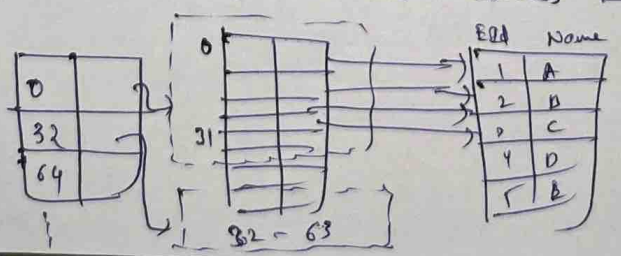
→ Now perform searching. → So Here Instead of searching directly into the Employee table we search in Index Table → get Row pointer → And then Access the record ~~in~~ in that Memory (Row pointer)

  ↳ So Total 4 Blocks + 1 Block to visit in Emp Table = 5 Blocks

Not 25 Blocks as in Earlier case

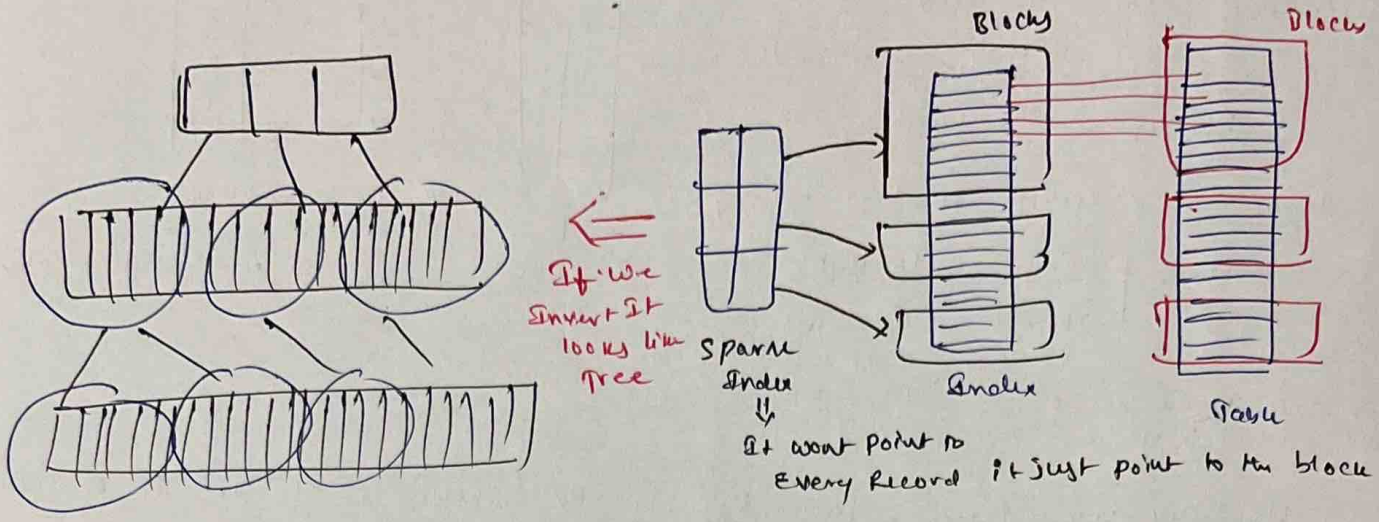
→ What if Index size also growing?
  ↳ Then perform Index on an Index (Multi Level Indexing)
  ↳ Now How do we prepare this second Index?
    ↳ For Every Entry in Index → No (As size becomes same and no - use.)
    ↳ we perform Index for every block in the Index Table.
      ↳ For 1000 records → 40 Blocks
      ↳ So For 2nd level Index 40/32 = 1.... ≈ 2 Blocks.

Sparse Index



→ So this is the basic Idea for orginating B & B+ trees.

↳ If we observe this multi level index looks like a Tree.



⇒ What is the requirement of Indexing ?

↳ Shall we check dB → growing / shrinking and manually and perform Indexing?

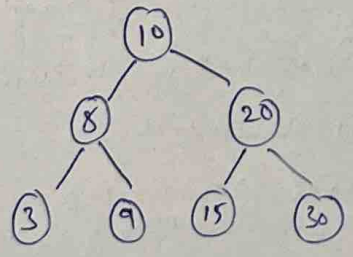
↳ It should be done Automatically so we want Self-managed multilevel Indexing. That is where B & B+ Trees comes into picture.

→ B & B+ Trees are actually originating from M-way Search Trees.

* M-Way-Search Tree :-

→ Like in Binary Search Tree we know that the Smaller Elements arranged towards left side and larger Elements towards right side.

BST



⇒ no. of key per node = 1

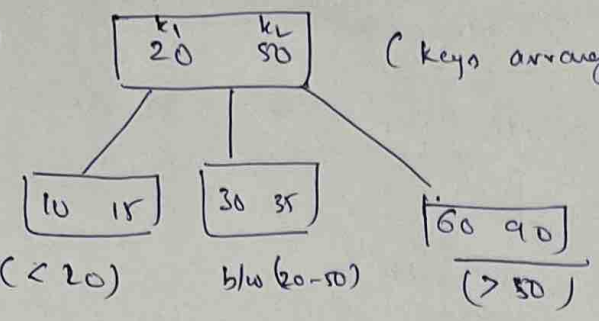
⇒ no. of Childrens (max) per node = 2

⇓

Can we have node with More than 1- key and more than 2 Children ?

$k_1 < k_2 < k_3 \ldots$

=)



(Keys arranged in increasing order)

=) Amount of time taken to search a node is more than BST but the approach is same.

=) These Trees are called as M-way search Tree

No. of key per node => 2 keys

No. of Childrens per node = 3 Childrens

|| So

3-way search Tree

So M-way search Tree

i.e ↳ Each node can have at most "m" Childrens

↳ i.e (m-1) key per node

=) M- is based on no. of childrens = degree of a node.

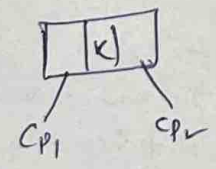
↳ i.e Binary search tree (BST) is a type of m-way search Tree with degree 2

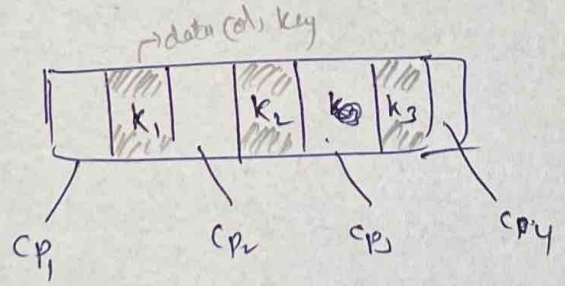
=) How we represent a node in M-way search Tree.?

for Example M-way search Tree = 4-way search Tree

4-way search → 3 keys - 4 childrens

2-way

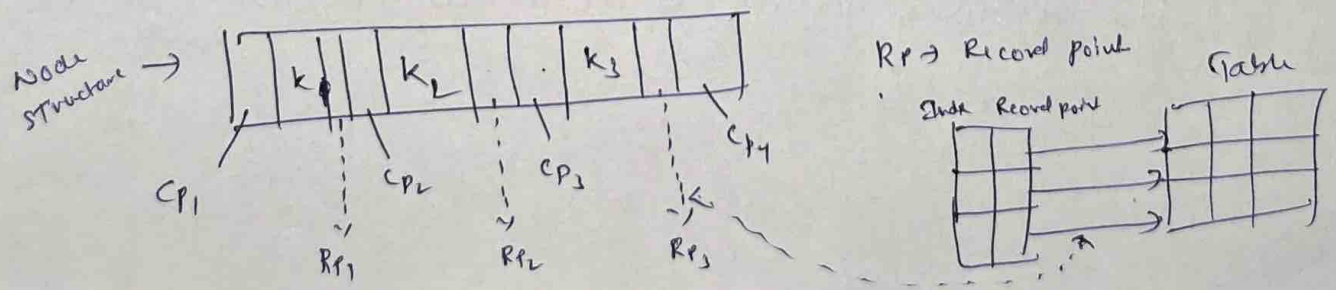




cp - child pointer

→ Can we utilize M-way search Tree for Indexing

↳ Take 4-way as Example



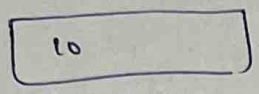
→ Creation process is not in control there is no guidelines

→ Problems with M-Way Search

→ Insertion → 10 way search Tree i.e 10 children 9 keys.

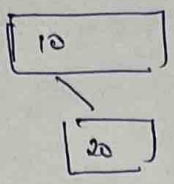
10, 20, 30

1) Insert 10

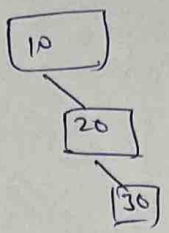


8 more keys can Add here

2) Insert 20



3) Insert 30



→ I can Insert like above but this is wrong because we have lots of memory wasting at each node. First fill up the previous node and fill up other nodes

↳ But there is no control on M-way Search Tree we can Insert as we like

→ TO overcome this problem, we use B & B+ Tree with some guidelines.

B-Tree

→ B-Tree is a M-way Search Tree with some Rules

→ B-Tree is a self-balancing Search Tree. ~~like AVL, RB tree~~

→ In most of the other self balancing Search Tree (like AVL & Red black Trees), It is assumed that Everything is in main Memory.

→ TO understand the use of B-Trees, we must think of the huge ~~memory~~ amt. of data that can not fit in the main memory (RAM). When the no. of keys are high, the data is read from the disk in the form of blocks. Disk access time is very high, compared to main Memory Access time. The main idea of using B-Tree is to reduce the disk Access time.

B-Tree Rules

* Every node in B-Tree must be filled by half i.e

1) If degree is M → then ceil[m/2] children must be there

Ex:- If degree is 10 → Then Every node (10/2) = 5 children must be there

↳ Then only think of creating new node.

2) We can not impose the above condition on a Root, If we impose we can not even insert one key. So

↳ Root can have minimum Two Children.

3) All leaf nodes must be at same level.

4) Creation process is bottom up Approach. i.e Tree Size will grow from bottom.

B-Tree Insertion:

Ex:- m = 4 (4-way-ST) : keys = 10, 20, 40, 50

Steps

1) If the tree is Empty → Allocate a root node & insert a key
2) Update the allowed no. of keys in the node
3) Search the Appropriate node for Insertion.
   ↳ If the node is Full Follow below steps
   1) ↳ Insert the Elements in Increasing Order
   2) ↳ Now there are Elements greater than limit, so split at the median (middle).
   3) ↳ Push the median key upwards and make left keys as left Child and right keys as right Child

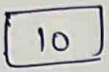
↳ If node is not Full
↳ Insert the node in increasing order.

Insertion Example 1

m = 4 (4-way search Tree) ⇒ 4 children per node (max)
3 keys per node (max)

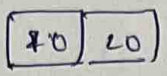
Keys :- 10, 20, 40, 50, 60, 70, 80, 30, 35, 5

1) Insert 10



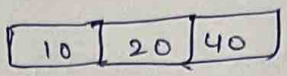
↳ 2 more key can Insert

2) Insert 20



↳ 1 more key can Insert

3) Insert 40



↳ No more key can Insert

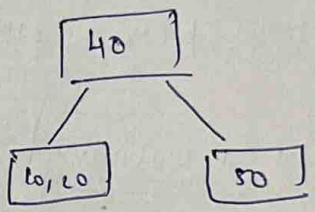
Note :- Insertion the keys in a node should be in increasing order.

4) Insert – 50

↳ there is no space to add it in Root node, then we should split the node and create one more node

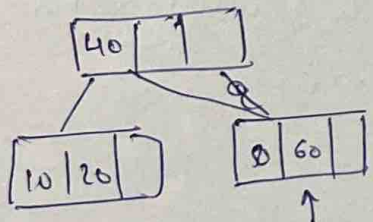
→ Take Median → and push it to upwards

10, 20, 40, 50
left ↑ median right ⇒



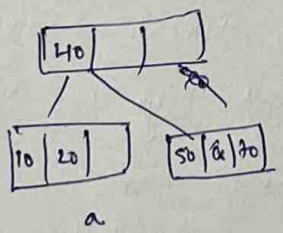
→ If we observe, the Tree is growing upwards → i.e bottom up approach.

5) Insert 60



Right side Recent
60 > 40

6) Insert 70

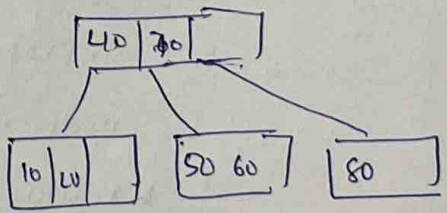


a

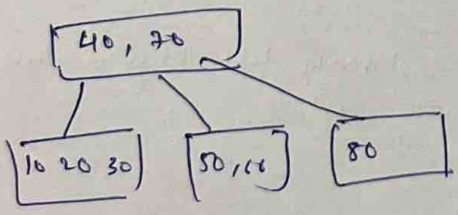
7) Insert 80

↳ no space so split the nodes

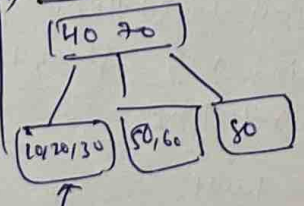
50 60 70 80
↑ median → move up



8) Insert 30



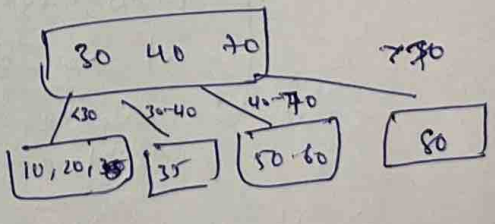
9) Insert 35



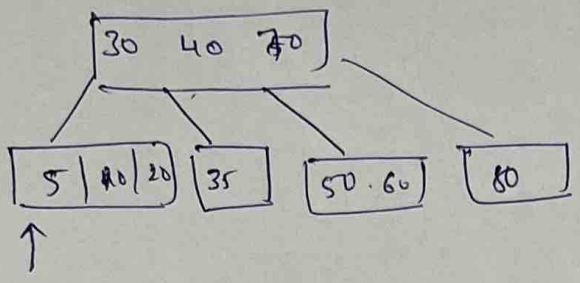
35 ⇒ Overflow

10, 20, 30, 35
↑ M

⇒



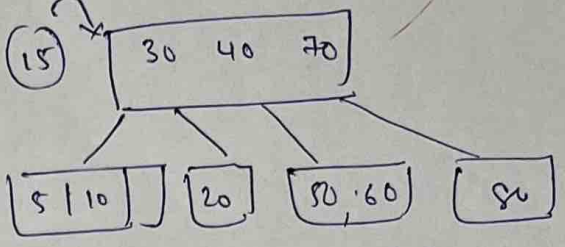
10) Insert 5



11) Insert 15

5 10 ~~15~~ 20

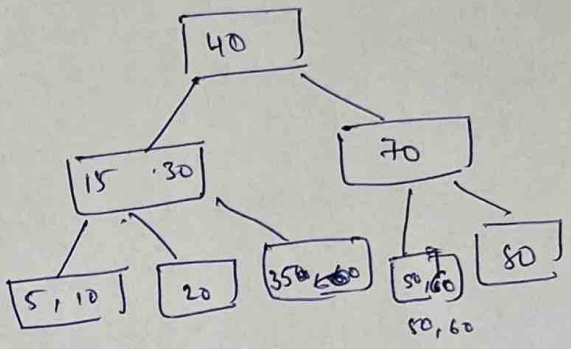
↑ m



2) Even there is no space to insert 15 at root node so split it

15 30 40 ~~70~~

↑ m



B-Tree complexity

1) Search – O(log(n))
2) Insert – O(log n)
3) Delete – O(log(n))

n = Total no. of Elements in B-Tree

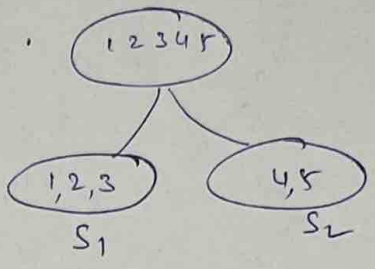
B-Tree deletion

Disjoint Set

→ 1) Basics 2) Union & Find 3) cycle detection.

→ Two (or) more sets with nothing in common are called disjoint sets

$$S_1 \cap S_2 = \emptyset$$



→ Use of disjoint set

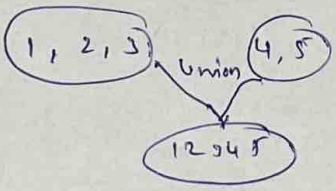
① ↳ keeps track of the set that an element belongs to Therefore it is easier to check, given 2 elements, whether they belong to the same subset (i.e Find operation)

S(1); S(2) ⇒ Belongs to same set i.e S_1

↳ True

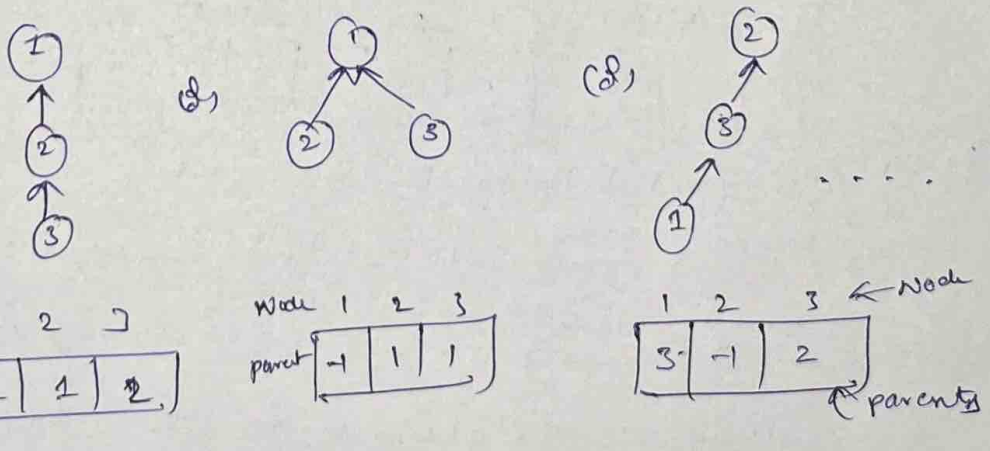
② It is used to merge 2 sets into one i.e (Union operation)

↳ Union of 2 Elements is same as union of 2 sets



→ Disjoint sets (DS) uses chaining to define a set, The Chaining is defined by a parent - child relationship.

Ex:- 1, 2, 3 → This can be represented below



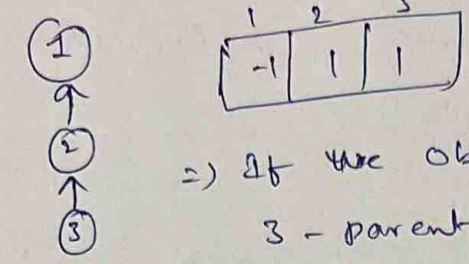
~~Indices~~ Array Indices ⇒ nodes

Array value ⇒ Hold the parent

→ How can we find that the two elements belongs to same set ?

↳ We can tell that finding it's Absoulte parent (A·P).

Q:-

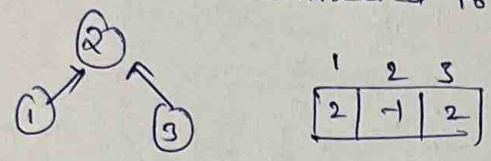


=) If we observe here, 2 -parent is 1 &
3 - parent is (1) => i.e "2" parent again is

=) So the absolute parent of 2, 3 => 1

=) & Absoulte parent of 1 is "-1" (Root not connected to Any)

S/Ly



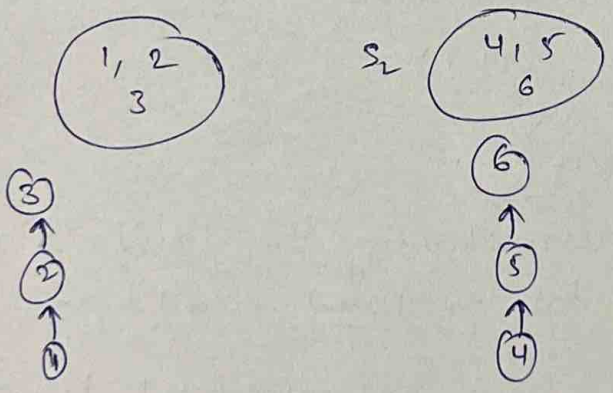
=) Same absolute parent so Belong to same set

→ The Structure of the Tree does not matter, it just matters for the optimization purpose.

2- sets Representation

Q_1 : Are (2, 5) same set ?

Q_2 Are (1, 2) same set.



Nodes

1	2	3	4	5	6
2	3	-1	5	6	-1

Q_1 → Are (2, 5) Belong to same set ?

2 $\xrightarrow{\text{parent}}$ 3 $\xrightarrow{\text{parent}}$ -1 (i.e Root) => i.e 2 Absolute parent (A·P) is "3"

5 $\xrightarrow{\text{parent}}$ 6 $\xrightarrow{\text{parent}}$ -1 (i.e Root) => i.e 5 Absolute parent is "6"

→ So Here → The absolute parent of 2 & 5 is different

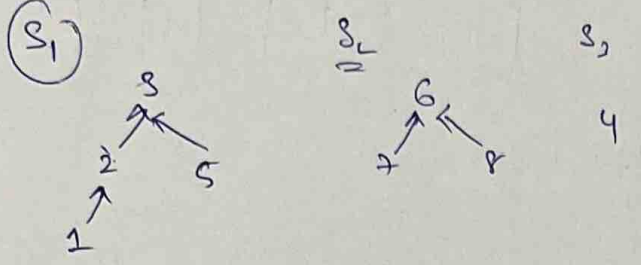
↳ So they are not belong to same set.

S/Ly (1, 2) → 1 $\xrightarrow{\text{Absolute parent}}$ 3 ; 2 → 3 => So same set as A·P are same.

DS-55.2

Find operation

* If root parent (or) Absolute parent of A & B are same then they must be in the same set.



Find → (1,4) => F
(2,5) => T
(5,8) => F

Node:	1	2	3	4	5	6	7	8
parents:	2	3	-1	-1	3	-1	6	6

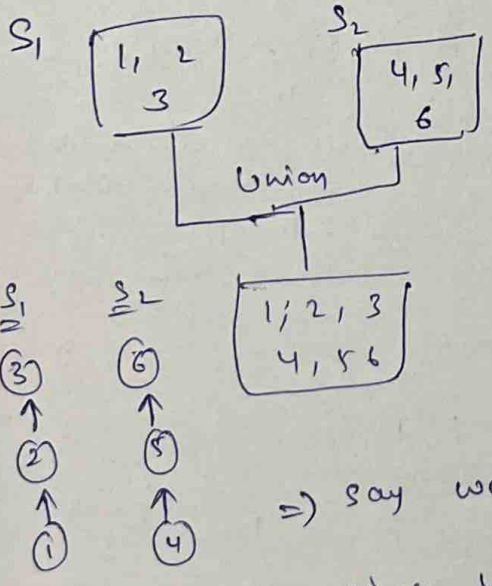
(1,4) => 1 —Absolute parent→ 2 —parent→ 3 —parent→ -1 (Root) => i.e 3 is absolute parent

4 ⟶ -1 =>(Root)

(1,4) => No belong to same sets as their absolute parents are different

Union Operation

=> Union of (1,5)



Taking of one Element is nothing but the merging the Element into the other set i.e nothing but the merging of two sets

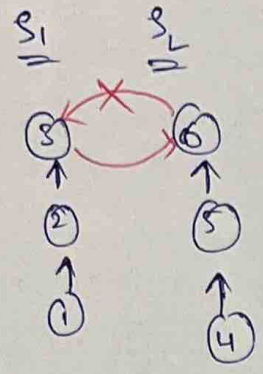
Node =>	1	2	3	4	5	6
parent;	2	3	-1	5	6	-1

=> say we want to find union of (1,5)

→ To do merging what can we do is take absolute parent of "5" ie "6"

↳ Now what we can do is point the absolute root of 1 to the absolute root of 5 (or) Vise Versa

↳ In order to optimize it we can take union by rank → but now lets go with brutal force Method



Node	1	2	3	4	5	6
parents	2	3	-1	5	6	-1

Now made 3 point to 6

1	2	3	4	5	6	
2	3	6	5	6	-1	

(1, 5) => 1 → 2 → 3 → 6 → -1

5 → 6 → -1

For both Absolute parent will be "6" now

⊢ O(E * V)

Ex :- Detect cycle in an undirected graph.

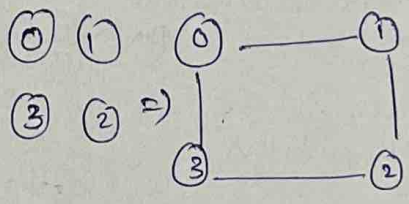
Note :- disjoint sets wont work fit directed graph as the union won't follow direction. In above case

} Any way / direction merge can be done.

From → To

(0, 1)
(0, 3)
(2, 3)
(1, 2)

} Edges undirected graph so vice versa also consider



Node	0	1	2	3
parents	-1	-1	-1	-1

(Initial No nodes are connected i.e disjoint)

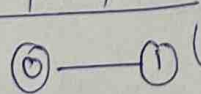
=>

1) (0, 1) => Edge

Since it is undirected we can consider 0 to 1 (or) 1 to 0, teh consider 0 to 1 i.e Take union

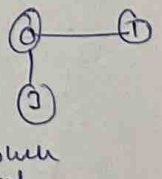
=> (0, 1) => Take union

0	1	2	3
1	-1	-1	-1



(0, 3) union

0	1	2	3
1	3	-1	-1



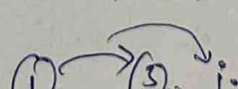
Absolute Parent

0 —parent→ 1 → -1 → 1

3 → -1 → 3. (Absolute parent)

=> (2, 3) => Both are disjoint as it pointing to -1 so perform union

0	1	2	3
1	3	3	-1



i.e 1 pointing 3

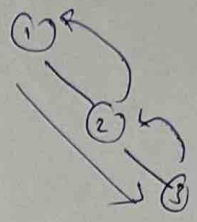
(1, 2) => It has same absolute Root i.e 3 i.e

↳ Belong to same set → If you add it creats cycle

DS-55.3

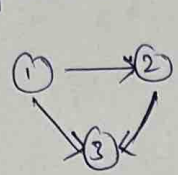
Time complexity :- $O(E * V)$

- E → Process All edges
- V → Find (DS) Union : Worst case in a Degenerated Tree / Skewed Tree



→ Disjoint set can't detect cycle in a directed graph because union operation has no directionality

Ex :-



(1, 2)
(1, 3)
(2, 3)

⇒ If we see here, there is no cycle, but DS will detect cycle

1	2	3
-1	-1	-1

⇒

1	2	3
2	-1	-1

1	2	3
2	3	-1

1, 3

1 → 2 → 3 → -1

3 → -1

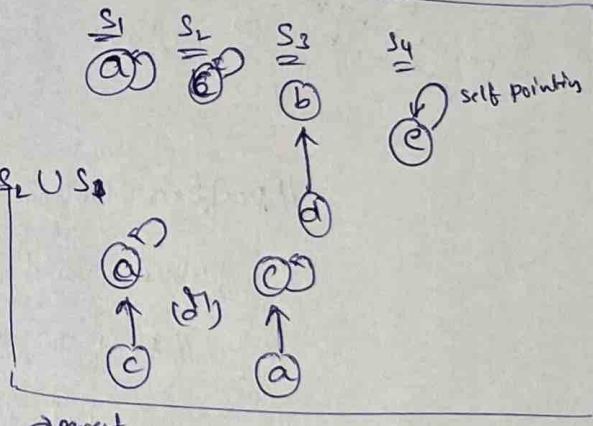
Both pointing to 3 → Cycle

Program

find (d) → Takes one param

⇒ S3 (d) b (parent)

union (S2, S4)

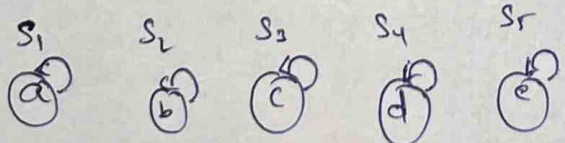


→ let Elements = {"a", "b", "c", "d", "e"}

Sets ⇒ S_1 = { key "a" : parent "a" } // pointing to self

S_2 = { "b" : "b" }

S_3 = { "c" : "c" } ; S_4 = { "d" : "d" } ; S_5 = { "e" : "e" }



⇒ Form

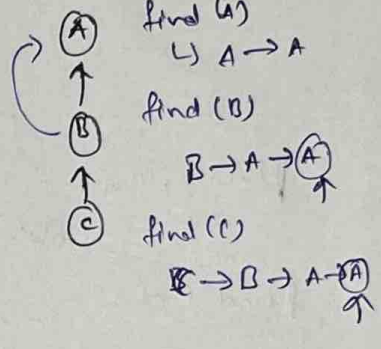
⇒ i.e make parent of d as b

{ "d" : "b" }

Program

```
Class DisjointSet {

    Private Map<Integer, Integer> parent = new HashMap<>();

    // perform make set operation
        to create "n" disjoint sets (one for each item)

    public void makeSet(int[] universe) {
        for(int i : universe) {
            parent.put(i, i)
        }
    }

    // Find the root of the set in which Element "k" belongs
    public int Find(int k) {
        if (parent.get(k) == k) {
            return k
        }
        return find(parent.get(k))
    }
```

Recur for the parent until we find the root




```
    // performs union of two subsets
    public void union(int a, int b) {
        // find the root of the sets in which elements "x" & "y" belongs
        int x = Find(a)
        int y = find(b)
        parent.put(x, y)   (or) parent.put(y, x)
    }
}
main() {
    int[] universe = {1, 2, 3, 4, 5}
    DisjointSet DS = new disjointSet();
    ds.makeSet(universe)  ⟶  1, 2, 3, 4, 5
    ds.union(4, 3)  => 1 2 3 3 5
    ds.union(2, 1)  => 1 1 3 3 5
    ds.union(1, 3)  => 3 3 3 3 5
```

DS-55.4

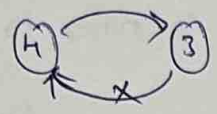
Universe = [1, 2, 3, 4, 5]

Parent = {1:1, 2:2, 3:3, 4:4, 5:5}

Union(4,3)

Union(4, 3)

4 3 => Both parents are different so we can apply union



=> i.e mark 4 parent as 3

Parent.get(4) => 3 => {1:1, 2:2, 3:3, 4:3, 5:5}

Sly Union(2, 1) & Union(1, 3)

Union(2,1) parent: {1:1, 2:1, 3:3, 4:3, 5:5}

Union(1,3): {1:3, 2:1, 3:3, 4:3, 5:5}

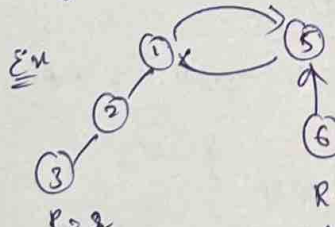
↳ now Find(1) => 1 → 3 ← Parent

Find(2) => 2 → 1 → 3 → 3 (Parent)

→ In the above approach, Tree can be highly balanced, so we can enhanced by below Improvements

1) Union-by-Rank :- Always attach smaller trees to the root of larger tree

Ex



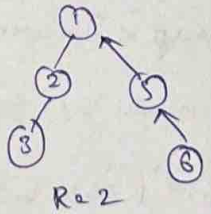
R=2 depth = 2

R=1 dep=1

Find(3) → 2 steps

=>

If we attach 5 to 1

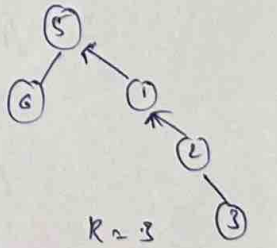


R=2

(Attaching smaller to larger)

Find(3) => 2 steps

If we attach to 1 to 5



R=3

Find(3) → 3 steps

Worst case run time improves to O(log(n))

2) Path compression

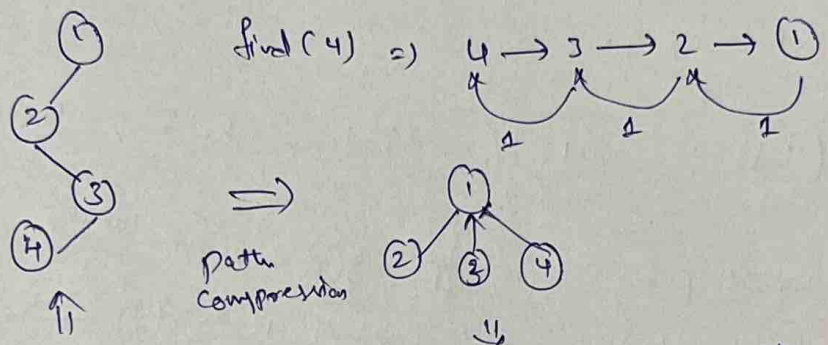
↳ flattening the Tree's structure when ever find is used on it.

↳ The Idea is that Each node visited heading to a Absolute Root

↳ so the Element can directly attached / Linked to absolute root

→ TO Effect, this as "Find" recursively traverse up the Tree, it changes Each node parent reference to point to the root that is Found

→ The resulting tree is Much flatter, speeding in future operations not only on these Elements but also on those referencing them, directly (or) Indirectly.



Program

```
class Disjoint {

    private map <Integer, Integer> parent = new HashMap<>();

    // Stores the depth of trees
    private map <Integer, Integer> rank = new HashMap();

    // Performs make set operation
    public void MakeSet (int[] universe) {
        for(int i : universe){
            parent.put(i, i)
            rank.put(i, 0)
        }
    }
}
```

```
Public find (int k) {
    // If k is not root
    if (parent.get(k) != k) {
        // path compression
        parent.put(k, Find(parent.get(k))
    }
    return parent.get(k)
}

Public void union (int a, int b) {
    // Find the Roots of a, & b
    int x = Find(a)
    int y = find(b)
    // If x & y are Equal → i.e if already present in same set
                            no need of union
    if (x == y) { return; }
    // Always Attach a smaller depth tree to larger depth Tree
    if (rank.get(x) > rank.get(y)) {
        parent.put(y, x)
    }
    else if (rank.get(x) < rank.get(y)) {
        parent.put(x, y)
    }
    else {
        parent.put(x, y)
        rank.put(y, rank.get(y) + 1)
    }
```

main

```
int[] universe = {1, 2, 3, 4, 5}
DisjointSet ds = new DisjointSet(5);
ds.makeSet(universe)
ds.union(4,3) → {1 2 3 3 5}      ds.union(1,3) → {3 3 3 3 5}
ds.union(2,1) → {1 1 3 3 5}
```

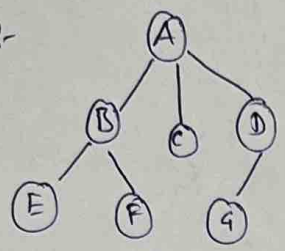
→ Disjoint sets will be useful for finding cycles in undirected graph

↳ This will be used in Kruskals Algorithm, (finding minimum spanning tree.

Graph Traversal

* Graph won't have root unlike in B tree, so we have to consider one starting node to start all the properties of BFS and DFS

Ex:-



(Graph)

=) This looks like tree, but it is a graph for easy to understand graph is visually represented as Tree in this case.

=) Unlike Tree, In graph there is no concept like parent, child etc

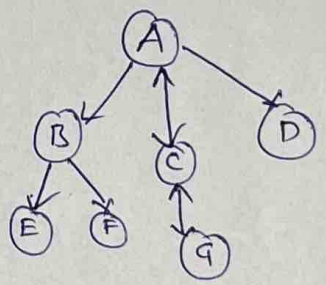
=) In graph we will have Vertex and adjacent node, for Ex:-

A → vertex =) B & D =) adjacent nodes

=) We have to keep a track that if we have visited an Element or not. because there are chances that several nodes might be connected with Each other.

→ Sample graph implementation :-

```
Class Node:
    def __init__(self, value):
        self.value = value
        self.adjacent_list = []
        self.visited = False

Class Graph:
    pass

nodeA = Node('A')
nodeB = Node('B')
nodeC = Node('C')
nodeD = Node('D')
nodeE = Node('E')
nodeF = Node('F')
nodeG = Node('G')
```



Adjacent nodes

A → B, C, D
B → E, F
C → A, G
D → no - adjacent nodes
E → no - adjacent nodes
F → no - adjacent nodes
G → C

→ nodeA.adjacent_nodes.append(nodeB, nodeC, nodeD)

→ nodeB.adjacent_nodes.appendall(nodeE, nodeF)

→ nodeC.adjacent_nodes.append(nodeA, nodeG)

→ nodeG.adjacent_nodes.append(nodeC)

1) Graph BFS

→ Since we don't have parent-child relation ship in graphs we will make use of graph visited property → as soon as we visited node we will set this flag to True → else we will struck in infinite loop

```
Public List<T> BFS (GNode<T> node){

    List<T> traversal = new ArrayList<>();

    Queue<GNode<T>> q = new Queue();

    q.enqueue(node)

    while(!q.isempty()){                          → O(n)
        GNode<T> currentnode = q.dequeue();
        traversal.add(currentnode)
        currentnode.visited = True;                → O(E)
        for(GNode<T> gnode : currentnode.adjacentNodes){
            if(gnode.visited == False){
                gnode.visited = True
                q.Enqueue(gnode);
            }
        }
    }
    return traversal;
}
```

This step can be here

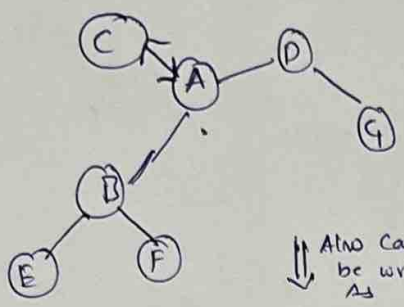
BFS
So Time complexity
O(n + E)
O(V + E)
V => node/vertex
n = node/vertex

Space complexity
O(V)
Log-size

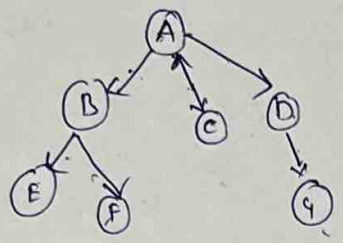
BFS - Application

- TO build index by search index
- for GPS navigation
- Path finding Algorithms
- In Ford-Fulkerson Algorithm to find max flow in a N/w
- Cycle detection in an undirected graph.
- In minimum Spanning Tree.

Graph BFS



Also can be written as



→ Step 1 :- We can start from any node say we have started with A

↳ Q → A | | | | ← Enqueue(A)

→ mark visited = True

↓↓

Traverse it's Adjacent node

T | | | |

A => B, C, D => Adjacent nodes

!! loop it

(2) delete

→ Q A | B | C | D ← B, C, D

T A |

(3)

=> Q A | B | C | D | E | F ← E, F (B Adjacent nodes)

=>

T A | B |

(4)

→ Q C | D | E | F ← C => (A)

↳ But it is already visited in step 1

↳ So we will not push to Queue

T A | B | C |

(5)

Q D | E | F | G ← G

↑↑ D adj nodes

T A | B | C | D |

(6) Q E | F | G ← no Adj nodes

T A | B | C | D | E |

(7) F | G (No Adj nodes to F)

A | B | C | D | E | F |

(8) G

A | B | C | D | E | F | G |

(9) Q is Empty Come out

→ Time complexity => O(V + E)

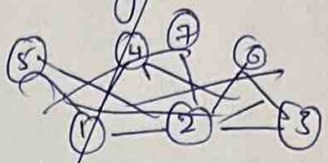
V => no. of nodes / vertexs

E => no. of edges

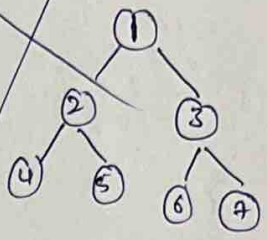
Space - complexity => O(V) → Q - length.

Depth - first Search (Graph)

* In DFS, once we visited a node / vertex we will suspend this vertex and start visiting it's adjacent nodes in that path.



DFS is like Pre order Traversal in Tree.



DFS

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

↳ Start From (1) → we got (2) then start Exploring "2" → then we got (4) → then stop Exploring (2) and continue Exploring (4) → So there is nothing so goback (2) → (which is already visited → then go to 5 ... Process continue

---

Ex :- 2 (BFS)

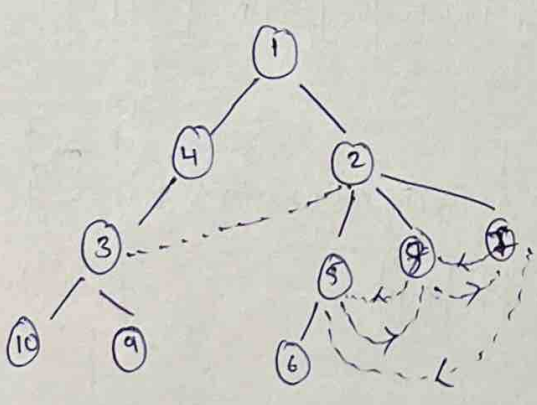
~~2~~ ~~3~~ 3

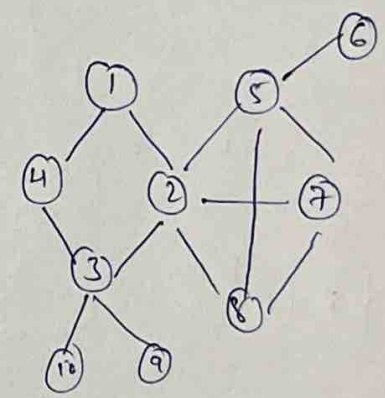
Q = 1, 4, 2, 3, 5, 8, 7, 10, 9, 6

⇓

at this point we already visited "3" so dotted line

Start :- from 1





Dotted line → If vertex already visited

↳ Tree we got while we Exploring the Each Vertex / node is ~~Spa~~ BFS Spanning - Tree

→ Dotted Edges are called as cross Edges.

→ In BFS :-

BFS - Rules

1) You can start BFS from vertex

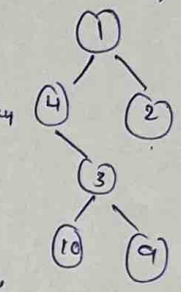
2) Let's say when we are exploring the given vertex, we can choose any Adjacent vertexes (order does not matter)

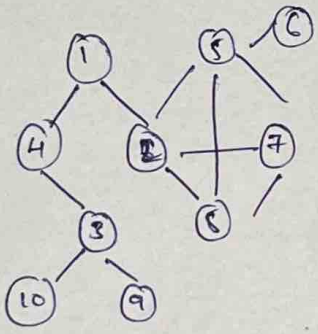
3) When you are ~~selecting~~ selected the node for Exploration then you have to visit all its adjacent nodes, And then only we should go to next vertex for Exploration.

↳ This next vertex should be picked From the Queue

Exploration ⇒ Reading value, print value etc. passing to Traversal Array

→ Say we have choosen "1" for Exploration then we have to visit adjacent nodes of 1 → (4, 2) then only we need to choose 4 - for Exploration.



* 

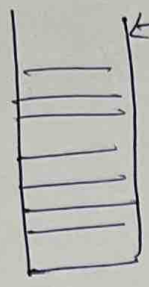
⇒ possible BFS

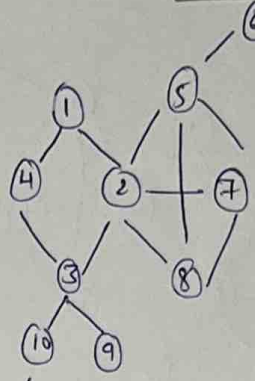
1) 1, 2, 4, 8, 5, 7, 3, 6, 10, 9 (start)

2) 5, 2, 8, 7, 6, 3, 1, 9, 10, 4 (start)

↳ Like this we can draw many Combinations.

Depth first search (Equivalent to pre order)

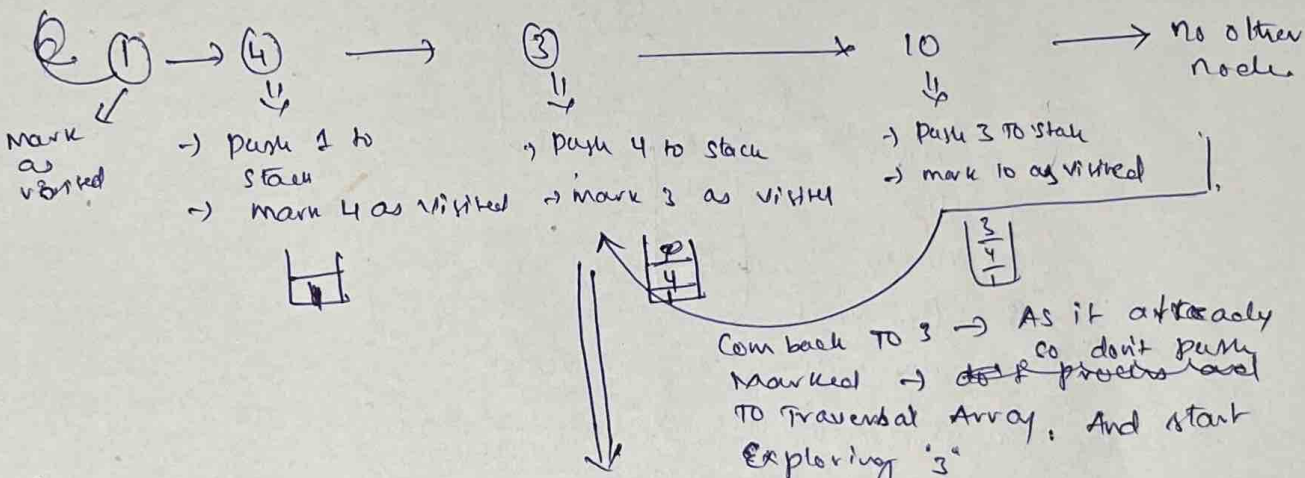


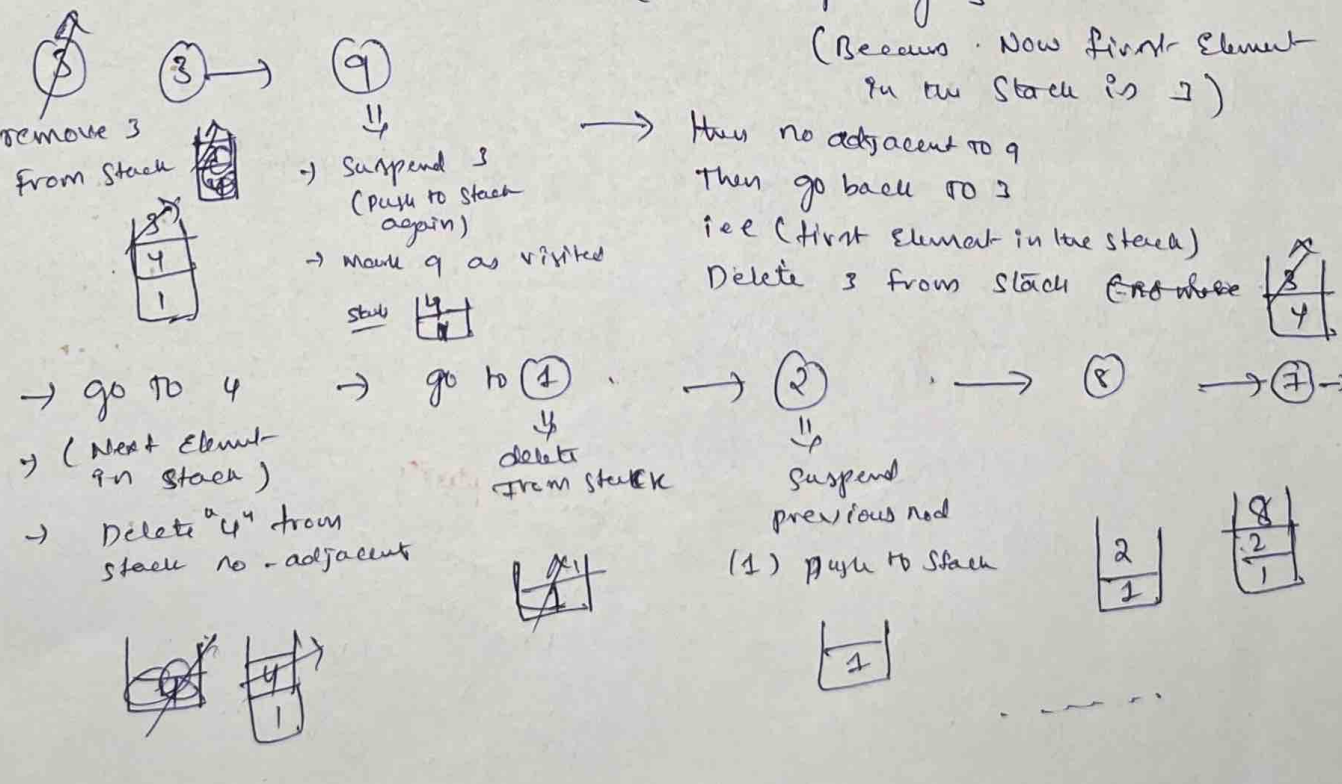


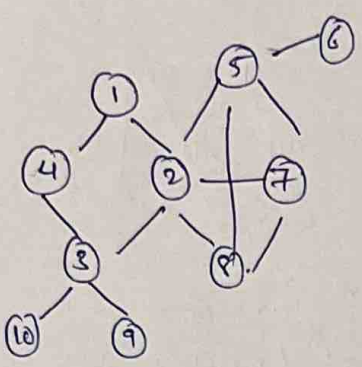
* In BFS, One we visited a node / vertex we will suspend that vertex / node and start visiting it's adjacent nodes in that path.

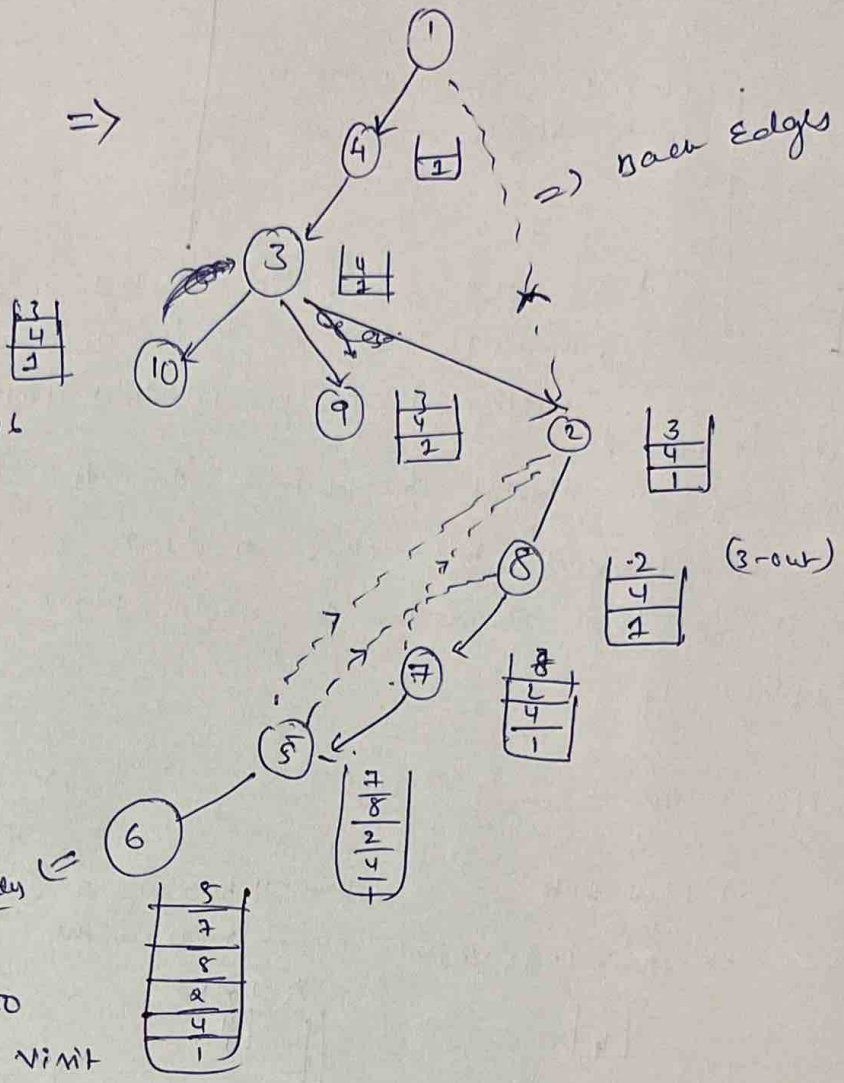
→ ~~Start from "1"~~

→ let's say we started from node "1" then mark it as visited and we moved to either 4 (or) 2, let say we moved to "4", then suspend "1" → i.e place it in the stack. and Explore 4.







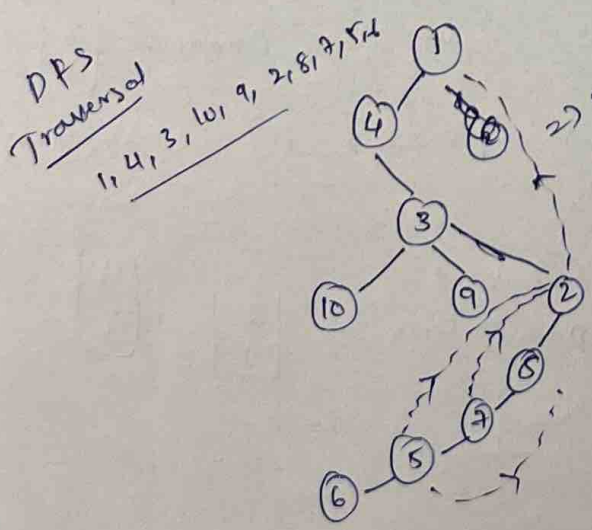


1, 4, 3, 10, 9, 2, 8, 7, 5, 6

=> s/by Back Track the Stack → 5, 7, 8, 2, 4, 1 Exit

=> final graph (DFS)

DFS - Spanning Tree



Other possible Traversals

1) 1, 2, 8, 7, 5, 6, 3, 9, 10, 4

2) 3, 4, 1, 2, 5, 6, 7, 8, 10, 9

Time complexity

=> The same can also achieved by Recursion without using

Time complexity → O(V+E)

Space → O(V)

DS- 59.1

Graph DFS Iterative Method

* It is same as Graph non-recursive BFS. but differs from it in ~~too~~ 3 ways.

1) It uses a stack instead of queue
2) The DFS should mark discovered / visited only after popping the vertex, not before pushing it.
3) It uses a reverse iterator instead of an forward iterator, to produce the same results of recursive DFS

```
List<T>                                  GraphNode<T>
Public ~~List<T>~~ iterativeDFS (~~GraphNode<T>~~ start) {

    list<T> traversal = new ArrayList<>();
    stack<T> stack    = new stack<>();
    stack.push(start)
    while (!stack.Empty()) {
        // pop a vertex from stack
        v = stack.pop();
        if (v.visited) {
            continue;
        }                          ← we will reach here, if the popped vertex "v" is not visited yet.
        v.visited = True;          ↳ Add v to Traversal and process it's undiscovered adj nodes into the stack
        Traversal.Add(v);

    ~~for~~ List<GNode<T>> adjList = v.adjList();
    for(int i = adjList.size()-1; i>=0; i--) {
        Gnode<T> u = adjList.get(i);
        if (!u.visited) {
            stack.push(u)
        }
    }
}
```

DFS Algorithm → Equivalent to pre order

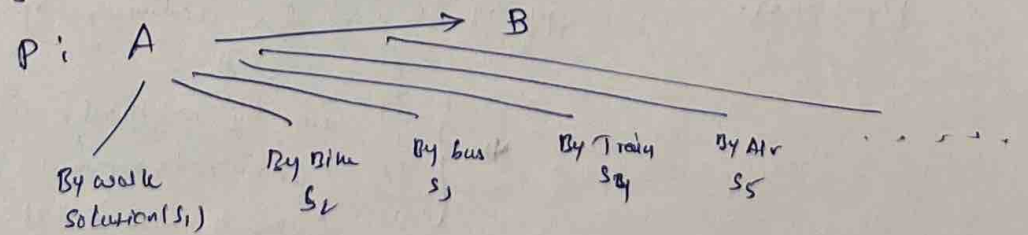
```
public List<T> DFS(GNode<T> Startnode, List<T> traversal) {
    if (startnode != null) {
        Startnode.visited = True
        traversal.add(startnodeValue);
        for (GNode<T> node : Startnode.adjacentnodes) {
            if (!node.visited) {
                this.DFS(node, traversal)
            }
        }
    return traversal
}
```

Greedy Method

* Greedy method is also one of the approach for solving a problem (or) design which we can adopt for solving similar problem that fit's into this
* This Method is used for solving optimization problem,

Optimization problem
↳ The problem that demands either minimum results (or) maximum results.

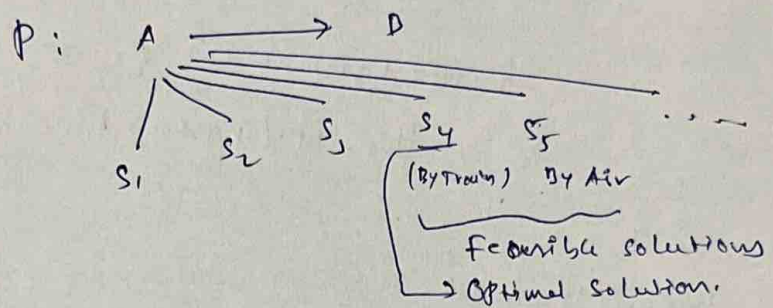
Ex:- Suppose I have problem (P) i.e I want to travel from A to B



→ To Solve a problem there are many solutions (In our case To reach from A to B we have solutions like S_1, S_2, S_3, S_4, S_5

* ↳ Now I will add a constrain, that I have to cover this journey with in 12hrs.

↳ Assume that I can not cover it by walk (or) Bus, bike. And I can cover it only by ~~car~~ train (or) Air.



→ The solution which is satisfying the condition in problem are called Feasible Solutions → There can be 1 (or) more Feasible solution.

→ Now I will Add one more condition that I want to spend less money. then out of two feasible solution → one solution would be cost Efficient (i.e S_4) → so this is called as

↳ i.e minimization problem → Optimal solution

↳ for any problem there can be only one optimal Solution

→ This problem requires minimum result and some other problem requires Maximum result. So if a problem requires either minimum / max result, then we call that type of problems as Optimization problem

→ So "Greedy Method" is used for solving optimization problems.

=> Diff Strategies for solving optimization problems

⇓

* { 1) Greedy method
  2) Dynamic Program
  3) Branch and Bound

→ Greedy Method says that the problem should be solved in stages, in each stage, we will consider one I/p from a given problem and if that I/p is ~~feas~~ feasible; then we include it in the solution → Now from all these feasible solutions we will get an optimal solution.

Greedy Algorithm

1) TO begin with, the solution set is Empty
2) At each step, an item is added to the solution set until a solution is reached.
3) If the solution ~~set~~ is ~~Empty~~ ~~feas~~ feasible the current item is kept
5) Else, the item is rejected and never considered again.

~~Algorithm(a, n)~~

Ex:- You have to make a change of Amount using the smallest possible number of coins,

Amount :- 18 → Available coins => 5, 2, 1

There is no limit to the no. of each coin you can use.

Solution :-

1) Create a Empty solution set = { } Available coins = {5, 2, 1}

↳ Always start from larger value (i.e) until the Sum > 18

↳ Here we select the largest value at each step, we hope to reach the destination faster. This concept is called as the Greedy choice property

1st iteration → solution set (S) = {5} Sum = 5

2nd → S = {5, 5} Sum = 10

3rd → S = {5, 5, 5} Sum = 15

4th → S = {5, 5, 5, 2} Sum = 17

↳ 2nd largest coin If we choose 2 (5) 20 > 18 So choose 2

5th → 5, 5, 5, 2, 1

↳ 3rd largest

Ex 1.2 Knapsack problem

* Here knapsack is like a container (or) a bag. Given weights and values of n items, put them items in a knapsack of capacity "W" to get the maximum total value (max cost/profit) in the knapsack.

		(x1)	(x2)	(x3)	(x4)	(x5)	(x6)	(x7)	
Object (O)	=	1	2	3	4	5	6	7	(Total, 7 objects
profit (P)	=	10	5	15	7	6	18	3	W = 15 kg
weight (W)	=	2	3	5	7	1	4	1	
(P/W)		5 (10/2)	1.5 (5/3)	3 (15/5)	1	6	4.5	3	

Constraint :- Fill the knapsack with objects, so that total weight is ≤ 15

i.e $\sum (x_i \times w_i) \leq$ bag

(sum of, object count, weight of per 1 obj)

Objective :- get maximum profit $\Rightarrow \max(\sum x_i P_i)$

Select max (P/W) → pick one after other

~~$\sum (x_i w_i) \Rightarrow$~~ $x = [1, 2/3, 1, 0, 1, 1, 1)$

$x_1, x_2, x_3, x_4, x_5, x_6, x_7$

0 to 1

$\sum (x_i w_i) = (1 \times 2) + (\frac{2}{3} \times 3) + (1 \times 5) + (0 \times 7) + (1 \times 1) + (1 \times 4) + (1 \times 1)$

$= 2 + 2 + 5 + 0 + 1 + 4 + 1 = 15$ ↑ constraint met

$\max(\sum x_i P_i) = (1 \times 10) + (\frac{2}{3} \times 5) + (1 \times 15) + (0 \times 7) + (1 \times 6) + (1 \times 18) + (1 \times 3)$

$= 54.6$

Ex:-3

* Job sequencing with deadlines

↳ Here 5 tasks are given, for each job there is some profit, but that job must be finished with in the deadline.

↳ TYPE writer → Each job couples in 1 hr.

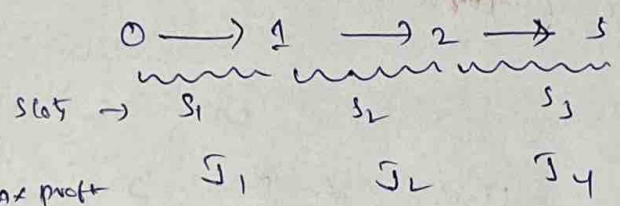
→ Constraint ↓ Deadline

→ Objective max profit

Jobs	J_1	J_2	J_3	J_4	J_5
Profits	20	15	10	5	1
deadlines	2	2	1	3	3

→ It is also given that, Every job takes a single unit of time, so the min possible deadline of any job is 1.

↳ Here Max time is 3 → so we have 3 slots



Consider Job in max profit order → J_1 J_2 J_4

Job consider	slot Asign	Solution	profit
–	–	∅	0
J_1	1 to 2 as it can wait 2 hr	J_1	20
J_2	1 to 2 ↓ Already occupied so (0 to 1)	J_2	15 + 20 = 35
J_3	0 to 1 ↓ Already occupied so Reject as it can wait more than 1 hr	✗	✗
J_4	(2,3) → free Allocate	J_5	15 + 20 + 5 = 40
J_5	(2,3) → Already Occupied → min Profit so Reject	✗	✗

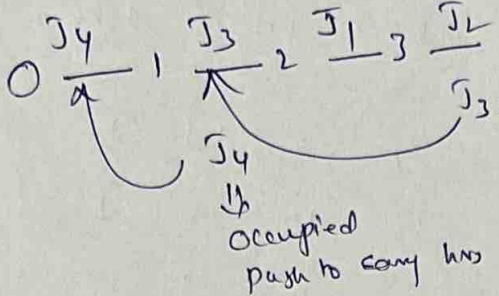
Job order

J_1 J_2 J_4 (or) J_2 J_1 J_4

So total profit = 40

Ex:-

Job's	J_1	J_2	J_3	J_4	J_5	J_6	J_7
profits	35	30	25	20	15	10	5
deadline	3	4	4	2	3	1	2



Job seq → $J_4 - J_3 - J_1 - J_2$ Max profit

$20 + 25 + 35 + 30 = 110$

→ Optimal Merge pattern (Greedy Method)

↳ Will be used by Huffman coding Greedy Method.

↳ Merging → can be done only on sorted list

sorted list → A B C (Result)

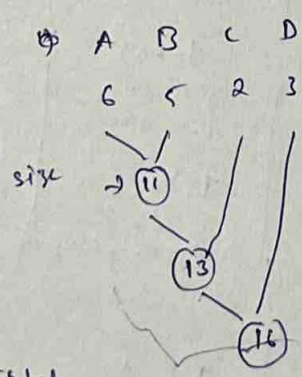
A	B	C (Result)
5	05	3
8	9	5
12	15	8
20	16	9
		11
		12
		16
		20

O(m+n)

Lists = A B C D

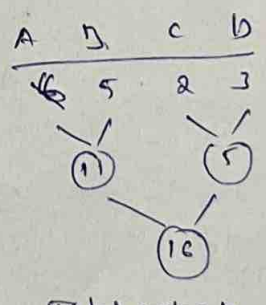
Sizes = 6 5 2 3

Merge

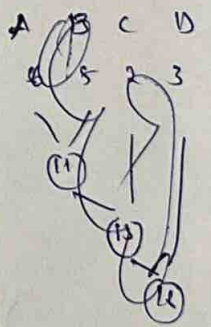


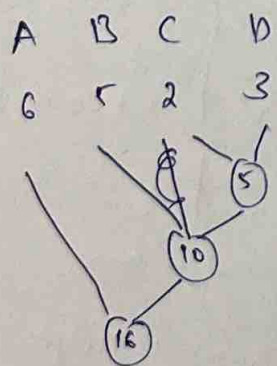
Total Cost = 11 + 13 + 16 = 40

2nd way



Total cost = 7 + 10 + 16 = 33





16 + 10 + 5 = 31 ⇒ Best Method

⇒ Here How we achieved Best Method?

↳ By Merging

The greed method we should follow is always merge a small size list to get the best result.

→ Optimal Merge Pattern Example

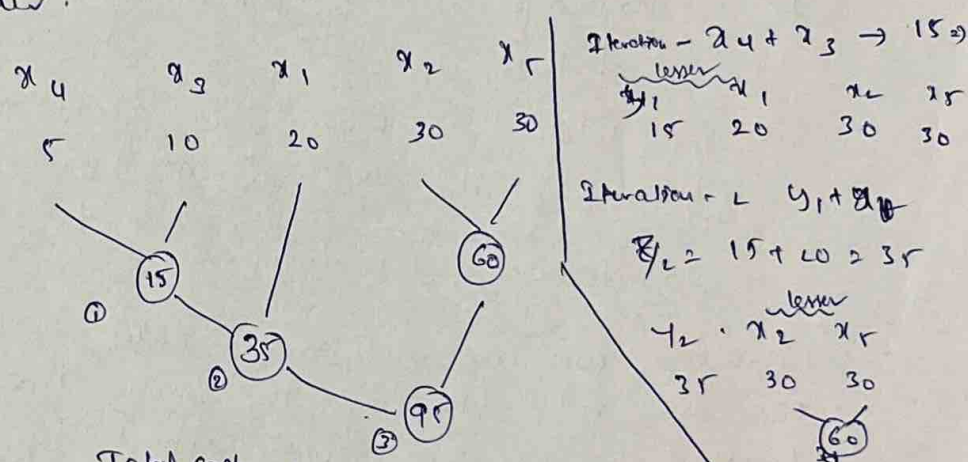
→ Finde the pattern in which the total cost of merging is minimized.

Lists/files	x_1	x_2	x_3	x_4	x_5
Size.	20	30	10	5	30

Optimal Merge Pattern

↳ Always select a smaller size of list to get the minimized time solution

→ In the above Example files are not sorted let's place them in sorted order.



Total cost => 15 + 35 + 60 + 95 => 205

=> Total cost can also measure as → no. of times the list merged.

↳ For Example x_4 → merged 3 times

$(3 \times 5) + (3 \times 10) + 2(20) + 2(30) + 2(30) = 205$

(5: distance, 10: size)

i.e $\sum d_i \times x_i$

Optimal Merge pattern formula = $\sum d_i \times x_i$

d_i => distance of each node

x_i => size of each node

↳ This pattern will be used by · Huffman coding.

Huffman - coding (Greedy Method)

* Huffaman Coding is a technique of compressing data to reduce the size without losing any of the details.
* Huffaman coding is generally useful to compress the data in which there are frequently occuring characters.

Ex:- 1) Suppose storing data in a file in compressed form. to reduce the size of a file

2) Transmitting data over a N/w in compressed form to reduce the bandwidth.

Ex:- message → BCEABBDDAECCBBAEDDCC
↳ length = 20

↳ Each letter will have an ASCII mapping, and ASCII if of 8 bit Ex:- A → 65 → 01000001

20 - letters → 160 bits

→ Now can we use our own codes instead of ASCII codes as we are not utilizing the whole 128 characters in our message.
↳ Here I am using just Capital English Alphabets

→ Fix Size Encoding

Character	Count	Frequency	Code
A	3	3/20	000
B	5	5/20	001
C	6	6/20	010
D	4	4/20	011
E	2	2/20	100
	20		

Assume own codes using 3 bits

→ say if I have 1 bit
↳ Then I can write 0/1 → 2 chars

→ 2 bits → 00, 01, 10, 11 } 4 characters

→ 3 bits → 0-7 } 8 combinations = 8 characters

=> now size of Message => 20 × 3 = 60 bits

→ Now, as we not using the standard ASCII codes, so along with the message we also need to send our own ASCII table for decode the msg at receiver side

→ Encoded Table

	ASCII value	New Code
A	65-01 00001 01	000
B		001
C		00C
D		00J
E		00J
	⇓ 8 bit	⇓ 3 bit

5 × 8 + 5 × 3 ⇒ 40 + 15 = 55

(5 ↑ Characters (A to E))

New TOTal Msg size

⇓

660 bit + 55 bit

(660 bit ~ Msg Light; 55 bit ~ Encoded table length)

= 115 bit

Previous size = 160 bit

After fixed Encoding = 115 bit. → So the size of the msg reduced

→ Now let us perform Variable size code and let us follow Huffman coding, which will still reduce the size of the msg

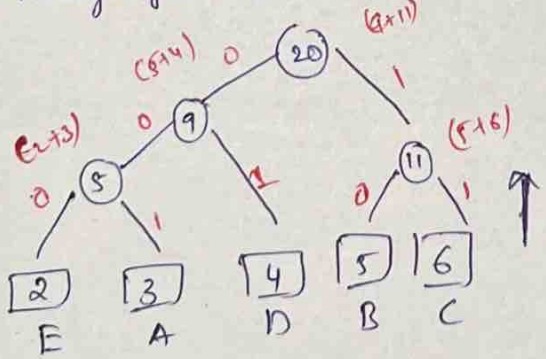
→ Huffman

↳ says that we don't have to say, take fixed size Codes for the Alphabets, Some Characters or Alphabets may be appearing few times, and some may be more times → So If we small size code for the more appearing Characters then the size of msg will be further reduced.

Message → B C C A B B D D A E C C B B A E D D C C

↳ length 20

↳ Arrange the Character count in Ascending Order → & perform Merging using Optimal merge pattern



Character	Count	Code
A	3	000
B	5	10
C	6	11
D	4	01
E	2	000
	20	

→ on the left hand side Edges Mark them as "Zero" & Right hand side Edges mark them as "One" (As per Huffman)

↳ For Each Alphabet Follow From Root to node A → 000 1

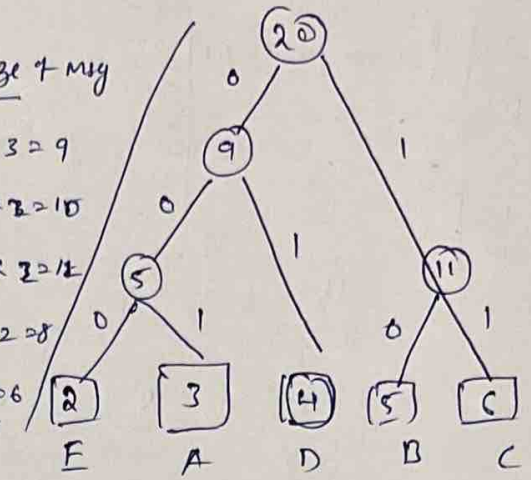
Say For others

Message :- BCCABBDDDAECCBBDAEDDCC

Huffman Encoding table

Char	Count	Code	Size * Msg
A	3	001	3*3=9
B	5	10	5*2=10
C	6	11	6*2=12
D	4	01	4*2=8
E	2	000	2*3=6
	20	1 bit	45

Total → 20



97 bits => Size of Msg

New → Message length => 5*8 => 15 bit

→ 3 bit

Non-to Repeat Chars

Character

Total Message size = (3*3) + (5*2) + (6*2) + (4*2) + (2*3)

9 + 10 + 12 + 8 + 6 => 45 bit

Encoding Table size => (5 * 8) + 12 bit => 40 + 12 => 52 bit

5 characters ↑ ASCII ↑ New codes

Total size of the msg => 45 + 52 = 97 bit

→ we can also get the size of the msg from the Tree as well

$\sum d_i f_i$ => (3 * 3) + (2 * 5) + (2 * 6) + (2 * 4) + 2(3)

f_i → frequency / count

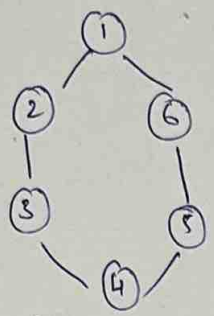
d_i → distance from root node

A frequency, B frequency

9 + 10 + 12 + 8 + 6 => 45

Minimum cost Spanning Tree :-

↳ As we know graph is represented as a set of vertexs and Edges



$G = (V, E)$

$V = \{1, 2, 3, 4, 5, 6\}$

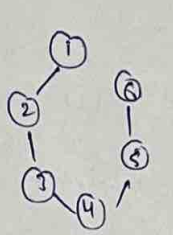
$E = \{(1,2), (2,5), (3,4), (4,5) \ldots\}$

=> Now spanning tree is a sub-graph of a graph, As I should take the subset of the graph.

↳ subset of Edges, vertex must be as it is.

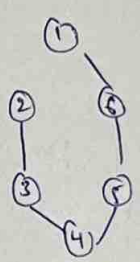
no. of vertexs $= (n) = 6$
no. of Edges $= n-1 = 5$ } For spanning Tree

⇓ Sample spanning trees ↳ can not Form a cycle.



$V = 6$
$E = 6-1 = 5$

(or)



$V = 6$
$E = 5$

→ So we can define the spanning tree as set of all nodes in a graph and $(n-1)$ Edges

↳ $S \subseteq G$ => $S = (V', E')$
↳ Subset

where $V' = V$; $E' = V-1$

→ Now For a given graph How many diff spanning Trees can be drawn ?

$$\frac{|E|_C}{|V|-1|} \quad - \text{ no. of cycles}$$

$$n_{C_r} = \frac{n!}{(n-r)! \times r!}$$

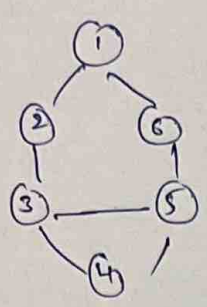
⇓ Observation

above Example
7 - vertices
6 - Edges ≅ 6 possible spanning Trees

$\Rightarrow {}^{6}C_5 \Rightarrow \frac{6!}{1! \times 5!} \Rightarrow \frac{6 \times 5!}{5!} = 6$ ⇓ possible spanning Trees

↳ Remember we can not have cyclic circle in Tree
↳ So always one vertex path will remain open.
↳ So we need to remove cycles in the above formula.

Ex:- 2



=) V = 6

E = 7

cycles = 2 (1,2,3,5,6) , (3,4,5)

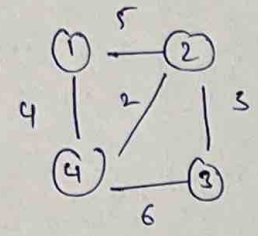
No. of subgraphs $^{7}C_{5} - 2$ =) $\left(\frac{7!}{2! \times 5!}\right) - 2$

↑ By formula

=) $\left(\frac{7 \times 6 \times 5!}{2! \, 5!}\right) - 2$

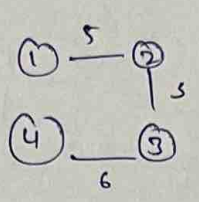
=) $(7 \times 3) - 2$ =) 21 - 2 =) 19

→ Now let's see in case of weighted graph

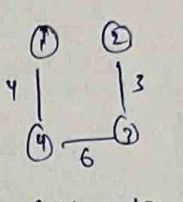


=>

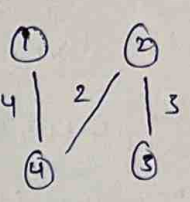
possible spanning tree



cost = 14



cost = 13



cost = 9

→ So for every possible combination spanning tree cost will be different → So, Now out of all spanning tree, How do we find minimum cost spanning tree

↳ i.e By visiting all the possible spanning tree

↳ But this will be lengthy and costly

⇓

So, this can be found using some Greedy methods

1) Prims Algorithm

2) Kruskals Algorithm.

PTO

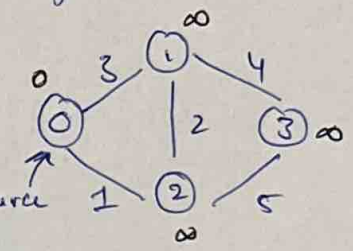
Prims Algorithm

→ Prims Algorithm is a Greedy Method used to identify the Minimum cost spanning tree (MST)

→ This algorithm takes graph as an I/p and finds the subset of the Edges of that graph which
  → Form a tree that includes Every Vertex
  → Has the min sum of weights among all the trees that can be formed from the graph

Steps

1) Select Node with min-weight (start at source)
   ↳ How will you find the min-weight at start?
   ↳ choose a vertex as Random and mark as source with weight 0 and all the other nodes mark weights as ∞

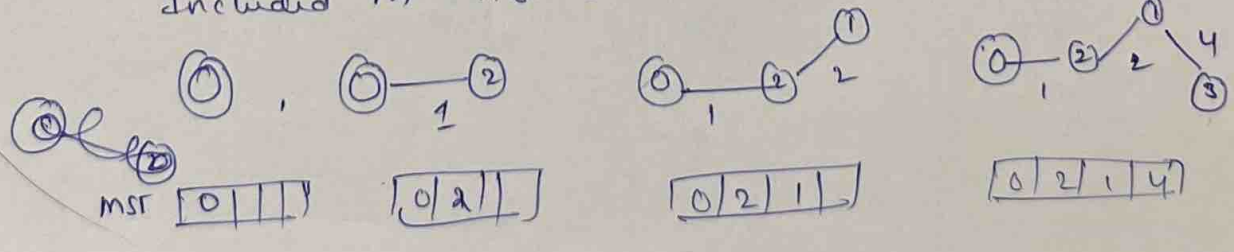


i.e at begining, all nodes weights are ∞, expcet source therefore we need to select source first

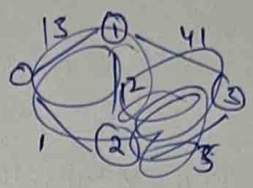
2) Include selected node in MST set, ie
   ↳ why? → what ever node is processed we will not process it again, so that the duplicates node wont Exist in MST

3) Relax/compute all adjacent Edges
   ↳ say the adjacent Edges of "0" (source) is 1 & 2
   we need to update their weights from ∞ to (1)→3, (2)→1

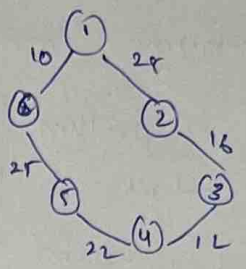
→ Repeat all the 3 steps unless all the vertices are Included in mst set.



→ Observations



→ First we selected (0) —10— (6) and the next min is (2) —12— (3) 2) But we should not select this as it not connected to either 1 (or) 6



↳ Prims thought is if some father away edge is selected then it may not be forming a Tree finally.

↳ So Always select minimum which are connected to the ~~edges~~ vertices in MST

→ How to code ? How to print MST ?

↓↓

Requirements

1) we need to keep track of weights values assigned to each node.

↳ use → Array (Int) size = V

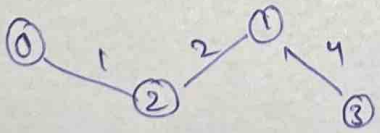
↳ Initially we ~~updated~~ marked as ∞ for all nodes and source as 0 → And then we updating.

2) We need to know what all vertices are included in MST set

↳ use → MST set → Array (boolean) size = V

3) We need to remember the edges of MST to print

↳ use → parent → Array (Int) size = V

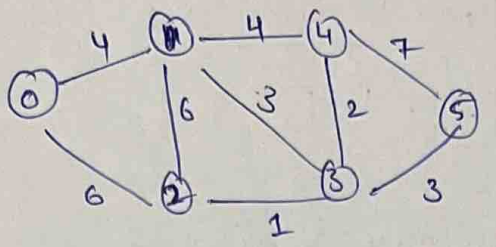


Order in which MST formed =>

2) parent of 2 → 0
parent of 1 → 2
parent of 3 → 1

(Prims Algorithm)

Ex:-2



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

Adjacency Matrix

Distance

0	1	2	3	4	5
0	∞	∞	∞	∞	∞

MST Set

F	F	F	F	F	F

Parent

-1					

Since source will have no parent in MST so -1

Pass-0

→ Pick min node value i.e 0

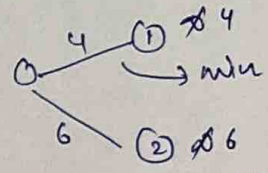
	0	1	2	3	4	5
Dist	0	4	6	∞	∞	∞

MST	T

Parent

Adjacent nodes

0 =>



=> Relax the Edge weights

min(∞, 6) => 6

min(∞, 4) => 4

=) Update parent Array as

parent[1] = 0

parent[2] = 0

Because 1 & 2 got relaxed by node 0.

Pass 1

→ Now pick min from distance Array

min(0, 4, 6, ∞, ∞, ∞) => 0 i.e Node 0 => But it is Already selected

0 1 2 3 4 5

i.e MST[0] = True

nextmin

min(4, 6, ∞, ∞, ∞)

=> 4 => Node 1 & MST[1] = False

Not Included so select

	0	1	2	3	4	5
Dist	0	4	6	∅3	∅4	∞

MST	T	T	F	F	F	F

Parent	-1	0				

0 —4— 1 (Already visited)

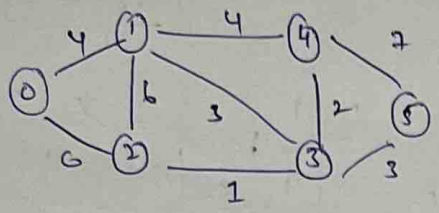
1 —6— 2 6

1 —4— 4 ∅4

0 —6— 2 6

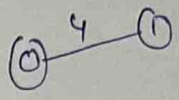
1 — 3 ∅3

weight



	0	1	2	3	4	5
Dist	0	4	6	3	4	∞
MST set	T	T	F	F	F	F
Parent	-1	0				

(MST)



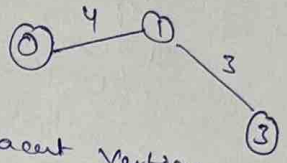
pass - 2

min (0, 4, 6, 3, 4, ∞) => Ignore 0 & 4 as they are Already processed => 3 i.e => Node 3

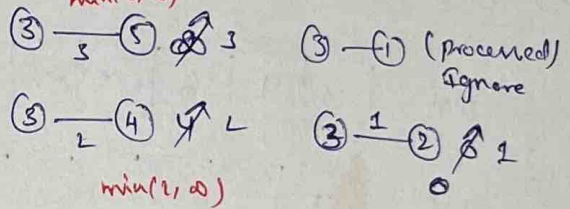
MST[3] = False so we can pick

	0	1	2	3	4	5
Dist	0	4	6	3	4	∞
MST	T	T	F	T	F	F
Parent	-1	0		1		

-> We jumped from node 1 to Node - 3 so parent of node 1 is "3"



-> Relax Adjacent Vertices



pass - 3

ignore Already visited

min (0, 4, 1, 3, 2, 3) => min (1, 2, 3) => 1

i.e node 2

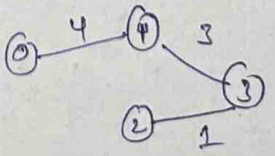
2 -> got a call from 3 so 3 will be its parent

MST[2] = F => so pick

- (2) — 1 — (3) (Process Already)
- (2) — (1) (Done)
- (2) — 0 (Done)

=> No relaxation

	0	1	2	3	4	5
Dist	0	4	1	3	2	3
MST	T	T	T	T		
Parent	-1	0	3	1		



This value 2 got for Node 4 when we Relaxing Node(3) - so Parent is 3 (pass 2)

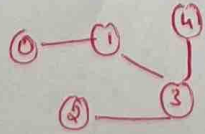
pass 4

index: 4, 5

min (2, 3) => 2 => 4 => MST(4) = F => so pick

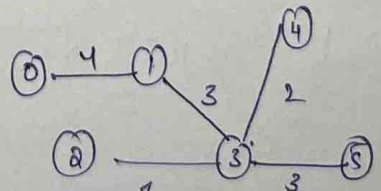
-> Parent[4] = 3

	-1	0	3	1	3	



pass - 5

Last min-value = 3 => Node 5



	-1	0	3	1	3	3

Two!

Pass

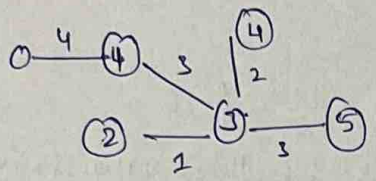
→ If we observe a pass-4 parent of node 4 marked as "3" Even though we jump from node 2 → Because the node 4 weight was relaxed by node 3 in pass-2

so ↳ When Ever an adjacent-vertex is relaxed from the current vertex → update parent Array. as

parent [adj_vertex] = current_vertex

This is How we need to do it in code.

Final spanning Tree



⇒ 4+3+2+1+1 = 13

Total cost

→ 6-vertices → 5 passes i.e (6-1)

Program:

```
Class Prims Algorithm {
    Private int vertices;
    Private int[][] graph;
    Private Integer[] values;  → used fore Eedge relaxation
    Private Integer[] parent;  → Used to store Parent of the given Vertex
    Private Boolean[] mstSet;  - it could be hashmap as well like disjoint set
                               ↳ used to store which are included in MST
    //Constructor.
    public Prims Algorithm (int vertices, int[][] graph) {
        this.vertices = vertices
        this.graph = graph
        this.values = new Integer[vertices]
        this.parent = new Integer[vertices]
        this.mstSet = new Boolean[vertices]
```

```
//Initially fill all the vertices weight as ∞ Expept source
                                                (Node 0)
    Arrays.fill(values, Integer.MAX_VALUE);
    Arrays.fill(mstSet, false);
    values[0] = 0;
    //source has no parent i.e Self pointing in case of
                                                Hash Table
    parent[0] = -1
} //Construct End.

private Integer selectMinVertex() {
    int min = Integer.MAX_VALUE;
    int vertex = -1
    for(int i=0; i<vertices; i++) {
        if(mstSet[i] == false && this.values[i] < min) {
            vertex = i;
            min = this.values[i];
        }
    }
    return vertex;
}

private void printMST() {
    int totalWeight = 0
    for(int i=0; i< this.parent.length; i++) {
        if(this.parent[i] != -1) {
            print(this.parent[i] + " - " + i);
            print("weight: " + this.graph[i][parent[i]][i]
        totalweight += this.graph[parent[i]][i]}
    }
    print("Total weight", totalweight);
}
```

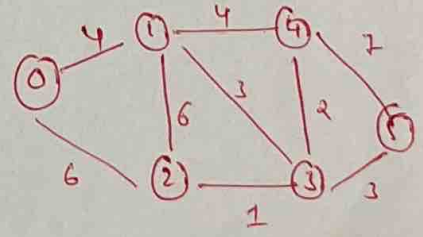
DS-65-4

```
public void findMst() {
    // Form MST with (v-1) Edges
  for(int i=0 ; i<vertices-1 ; i++){
     int minvertex = this.selectminvertex();
     // Include new vertex in MST
   if (minvertex >=0 && minvertex <vertices){
          this.mstSet[minvertex] = true;
     for(int j=0 ; j<vertices ;j++){
       int currentEdgeweight = values[j]
       int Edgeweight = graph[minvertex][j]
```

Relaxation Constraints

1) Edge is present b/w min

```
       if( Edgeweight !=0 &&        → Edge is present b/w min vertex & j
          mstSet[j] == False &&     → vertex j not included in MST
          Edgeweight < currentEdgeweight){
                                    → Edge weight is smaller than the current weight
            values[j] = Edgeweight
            Parent[j] = minvertex
       }
     }
  }
```

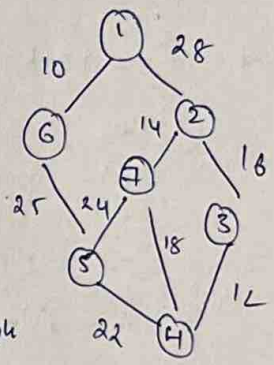



```
main {
  int v = 6
  int[][] G = { {0,4,6,0,0,0}, {4,0,6,3,4,0}
                {6,6,0,1,0,0}, {0,3,1,0,2,3},
                {0,4,0,2,0,7}, {0,0,0,3,7,0} };
  PrimsAlgorithm pg = new PrimsAlgorithm(v, G)
  pg.findMst()
  pg.printMst()
}
```

Prim's Algorithm

Uses
- → Laying cables of Electric wiring
- → In network design
- → To make protocols in N/w cycles.

* Prims algorithm is a greedy method used to identify the minimum spanning tree algorithm that takes a graph as I/p and finds the subsets of the Edges of that graph which
  1) Forms a Tree that includes Every Vertex
  2) Has minimum sum of weights among all the Trees that can be formed from the graph.

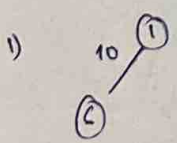


→ According To Prim.

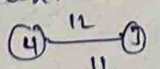
1) Select a minimum cost Edge. (Initial)
2) Then for rest of the procedure, always select a minimum cost Edge from the graph but make sure that it is connected to already selected vertex

Algorithm

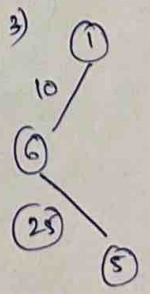
1) Initialize a minimum spanning tree with a vertex choosen at random
2) Find all the Edges that connect the tree to new vertices, find the minimum and add it to the tree.
3) Keep repeating steps → until we get a minimum spanning Tree

1)


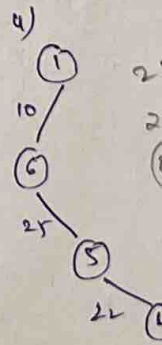
Minium cost Edge in whole graph

2) next minimum


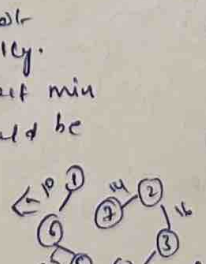
But don't choose this as it not connected to 1(d) 6 vertices
↳ Prims thought is if some Father away Edge is selected then it may not be forming a Tree finally.
↳ So Always select min Edge which should be Connected

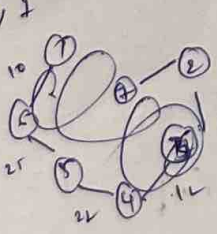
=>


(28725)

=)


!)

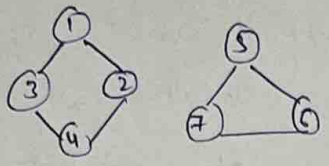


6, 7


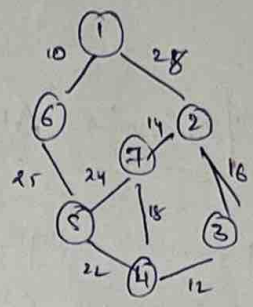
So total cost = 99

(6,7 steps)

↳ let's say there are two graphs which are not connected



} Then we can not form a minimum spanning tree using Any algorithm.



⇓

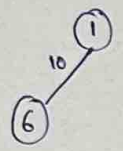
To form a spanning tree vertecies must be connected

$O(E \log V)$ Time complexity

Kruskals Algorithm

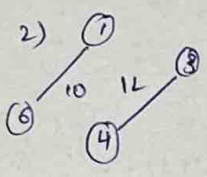
* Kruskals Algorithm is a minimum spanning tree algorithm that takes a graph as I/p and finds the subset of the edges of that graph which
  - → Form a tree that includes every vertex
  - → Always select a minimum cost edge

1)



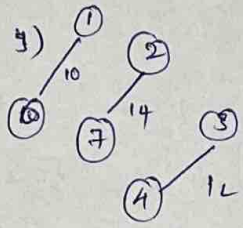
1st min 10

2)



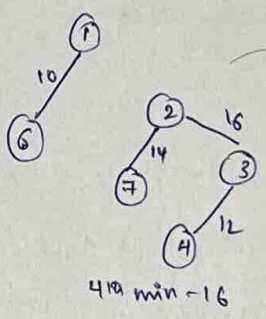
2nd min 12

3)



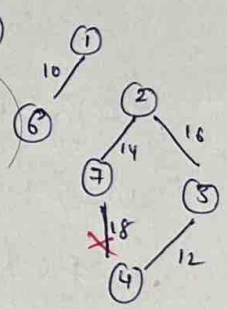
3rd min 14

4)



4th min - 16

5)

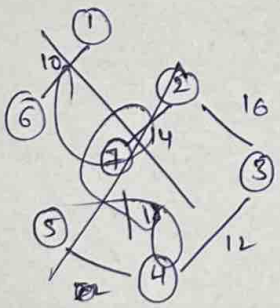


5th min - 18

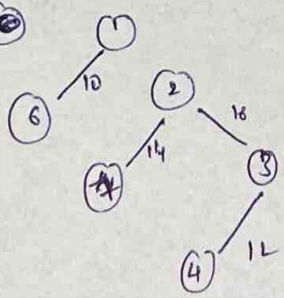
⇓

But it is forming cycle so reject

6)



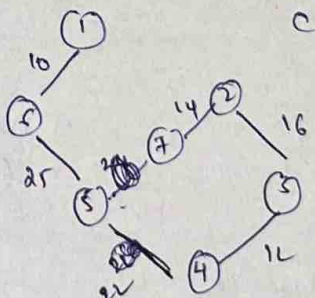
6th step



next min - 12

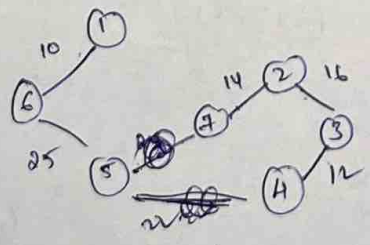
↳ But it will form cycle so reject it

7th step



next - 25 → so reject it forms cycle

8th step (Final)



10 + 25 + 14 + 16 + 12 = 99

=) All nodes present in tree

=) $E = V - 1$ =) $7 = 8 - 1 = 6$ ✓ condition satisfied

$T = O(VE)$ =) $O(n^2)$ (If $V \geq E$)

→ Time taken by the Kruskals algorithm is

$O(VE) \simeq O(n^2)$ if $(V = E)$

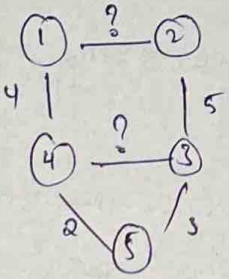
↳ But this can be Improved, if we always take min Element → so we can use min heap

↳ From min-heap when ever you delete it gives min Element → and the time complexity to delete min Element in min heap is $\log(n)$

$n \log(n)$ ← we need to perform for n nodes } For Kruskal algorithm time complexity $\simeq n \log(n)$

→ Kruskal algorithm may find the spanning tree of the non connected graphs, but not of the entire graph together

Problem :- What could be the min value of the Edges



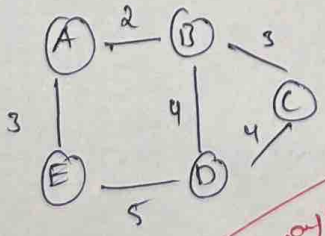
=> Edge (4) —4→ (5) ← min value

=> Edge (1) —6— (2)

=> For forming the spanning tree, if we select 4 & 3 it will form cycle i.e It may consider after 2 & 3 so the value will be more than 2 & 3 so Min would be "4"

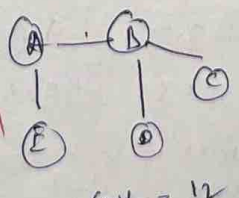
Silly for (1) & (2) → It may form at the end as it also create cycle → so min value would be 6

Problem - 3 → find min cost spanning tree



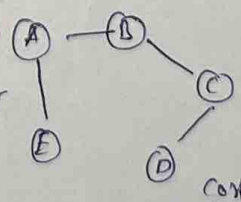
=> As the weights of B & D Edge & D & C Edge is same so we can Form 2 min span tree but eventually same cost

So Always Only one min cycle Even through Multiple possible spanning Tree.



Cost = 12

(or)



Cost = 12

Note :- The optimal cost of the minimum spanning tree will have only one same value → But the I/p to get the value may ~~vary~~ vary → Here we have 2 spanning tree but final cost is same. i.e 12

Algorithm

1) Sort all the edges from low weight to low
2) Take the edge with the lowest weight and add it to the spanning tree. if adding the Edge created a cycle, then reject this edge.
3) Keep adding edges until we reach all vertices.

Kruskal's Algorithm Program → It uses Disjoint sets

```
Class Graph {                                  → for sorting
    Class Edge implements Comparable <Edge> {
        int src, dest, weight
        Public int compareTO (Edge compareEdge) {
            return this.weight - compareEdge.weight
        }
    };

//Union
Class Subset {
    int parent, rank;
}
int vertices; edges;
Edge[] edge;
//constructed of graph (graph creation)
    Graph (int v, int e) {
        vertices = v;
        edges = e;   edges = new Edge[edges];
```

PTO

```
for (int i = 0; i < e; i++) {
        edge[i] = new Edge
}

// Find (Disjoint set)
int find (Subset[] subsets, int i) {
        if (subsets[i].parent != i) {          → Recursion
                subsets[i].parent = find (subsets, subsets[i].parent));
        }
        return subsets[i].parent;
}

// Union (Disjoint set)
void Union (Subset subsets[], int x, int y) {
        int xroot = find (subsets, x);
        int yroot = find (subsets, y);
        if (subsets[xroot].rank < subsets[yroot].rank) {
                subsets[xroot].parent = yroot
        else if (subsets[xroot].rank > subsets[yroot].rank) {
                subsets[yroot].parent = xroot
        }
        else {
                subsets[yroot].parent = xroot
                subsets[xroot].Rank++;
        }
}
```

```
//Applying kruskkal Algorithm

void kruskkalAlgo () {
    Edge result[] = new Edge[vertices];
    int e = 0;
    int i = 0;
    for(i=0; i< vertices; i++){
        result[i] = new Edge()
    }
    //sorting the edges (step 1)
    Arrays.sort(edges)
    // creating subsets for disjoint find & union
    for(i=0; i< vertices; i++){
        subset[i] = new subset();
    }
    for(int v=0; v< vertices; v++){
        subsets[v].parent = v
        subsets[v].rank = 0;
    }
    i = 0;
    while(e < vertices-1){
        Edge nextedge = new Edge();
        nextedge = edge(i++);
        int x = find(subsets, nextEdge.src)
        int y = find(subsets, nextedge.dest);
        if(x != y){                    ---> no cycle (not belong to same set)
            result[e++] = nextEdge;
            Union(subsets, x, y)
        }
    }
    //print
    for(i=0; i<e; i++){
        print(result[i].src + "-" result.dest + result.(i).weight);
    }
```

main

```
int v = 6
int e = 8
Graph g = new graph(v,e)
g.edge(0).src = 0
g.Edge[0].dest = 1
g.Edge[0].weight = 4
. . . etc
g.kruskalAlgo()
```

Dijkstra Algorithm → Same as prim's Algorithm

↳ Single source Shortest path – Greedy method.

* If the weighted graph is given, we have to find the shortest path from the single source.
  ↳ say I have choosen vertex "1" then I have to find out the shortest path to all the vertices. may be a direct path (or) by a indirect path. via other vertices.
  ↳ As we have to find out a shortest path, so it's a minimization problem → And as we know minimization problem is an optimization problem → so optimization problem can be solved using Greedy method.

→ As we know Greedy methods says that a problem should be solved in stages by taking one step at a time and considering one I/p at a time. to get a optimal solution

→ Dijkstra Algorithm is a procedure to get an optimal solution that is min result that is shortest path.

→ Dijkstra Algorithm can work for directed (or) a non-directed graph.

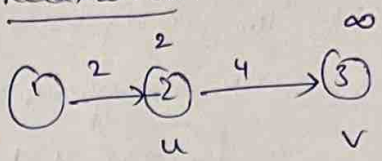
→ Approach :-

① $\xrightarrow{2}$ ② $\xrightarrow{4}$ ③ => If I say vertex "1" as the starting vertex and I want to find the shortest path to vertex ② & vertex (3)

→ If we observe the above graph there is a directed path From ① → ② and cost "2". And there is no direct path to "3" so we don't know what is the path ?
  ↳ consider as ∞
  ↳ first we consider a direct path, but now If we follow dijkstra Algorithm, first we will be selecting the shortest path vertex i.e "2" (as 1 to 3 is ∞)
  ↳ Now check from that vertex there is any shortest path To other ~~vertices~~ vertices 2 $\xrightarrow{4}$ 3 => Now Infinity can change To "6"

⇒ Now, there is no direct path to vertex "3", but there is a path coming via vertex "2" and update the distance by sum up the weights.

↳ This process is called as the "Relaxation".



Initially (Relaxation)

```
if (d(u) + c(u,v) < d(v)) {
        d[v] = d(u) + c(u,v)
}
```

⇒

distance of u ⇒ d(u) = 2

cost of U (u) ⇒ 2 ; cost of V (v) ⇒ 4 ; } c(u,v) = 2+4 = 6

distance of V ⇒ d(v) = ∞ (Initial → as we don't have direct path)

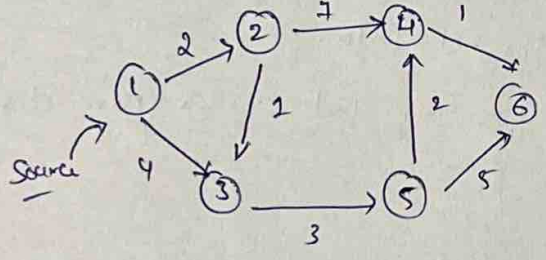
As per above formula

(2+4) ≤ ∞

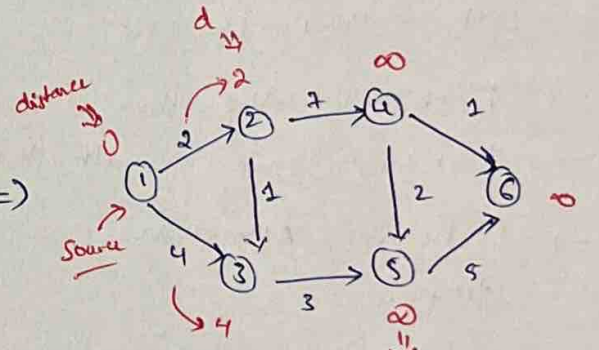
⇓

d(v) = 6

→



⇒



⇓

As there is no direct connection with "4" Initially treat it as ∞

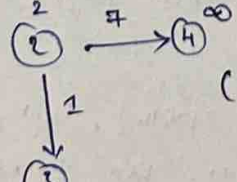
Step - 2

⇒ Now select the shortest distance node

↳ out of (2, 4, ∞, ∞, ∞) ⇒ 2 is the smallest distance

So consider Vertex 2

↳ Check who are connect to Vertex(2) and apply relaxation.



(2+1) < 4

3 < 4

So replace 4 with 3

(2+7) < ∞ } Relaxation

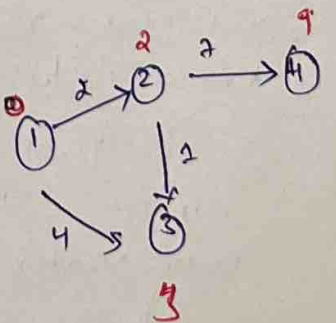
9 < ∞

⇓

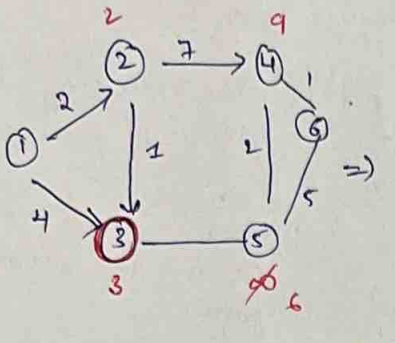
So replace ∞ with 9

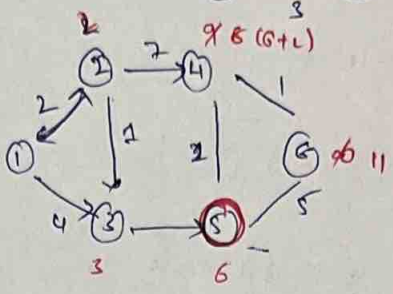
i.e distance to node 4 from 1 is 9

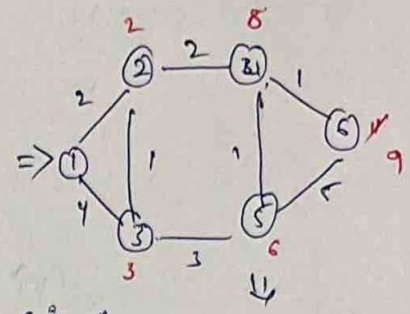
⇒



Step 3







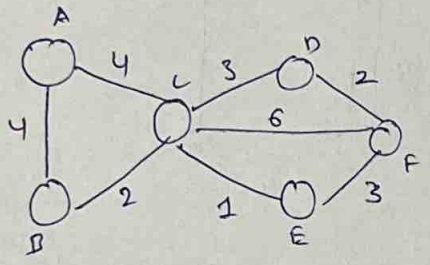
remaining nodes 5, 4, 6

min → (6, ∞ 9) => 6 is min

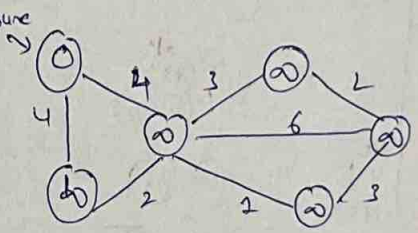
↳ connected nodes of 5

V	d(v)
2	2
3	3
4	8
5	6
6	9

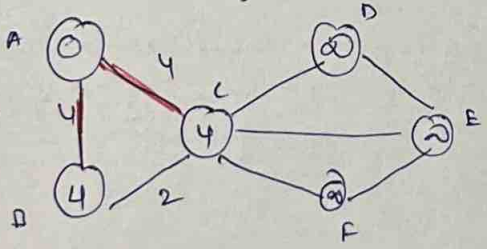
Example 2



1) Choose A vertex and Assign Infinity path values to all other vertex

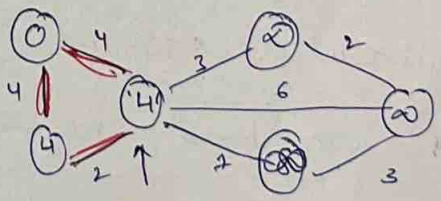


2) Go to Each vertex and update it's path length



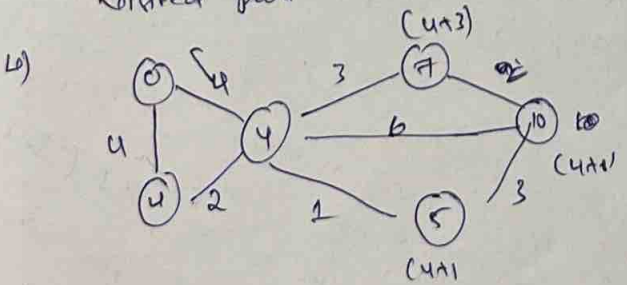
3) Choose next min path out of connected vertex to source i.e B & C and it's distance are 4, 4 => so we can choose any one.

↳ If the path length of the adjacent vertex is lesser than new path length, don't update it

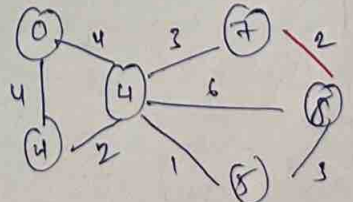


(4+2) = 6 > 4

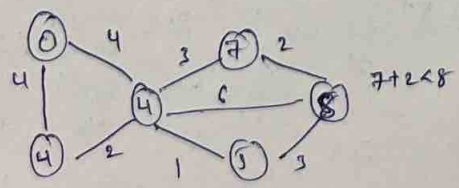
4) Avoid updating path lengths of already visited path



=>

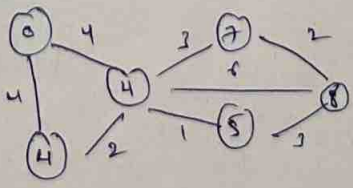


(5+3) < 10



7+2 < 8

=>



Observations

* In Every Iteration - it is finding Shortest path

no. of vertices => n = V → and it is relaxing the other vertices at each step

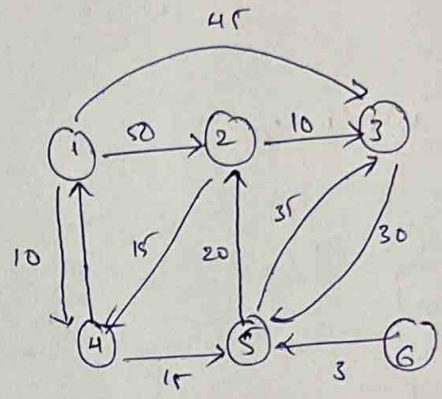
max = 2^V

↳ Because a given node is connected all nodes in graph

$T = O(V \times V)$ (or) $O(n \times n)$

$T = O(n^2)$ ← worst case

Ex :- 3



Starting vertex (1)

vertices

Selected Vertex	2	3	4	5	6
1	50	45	(10)	∞	∞
4	50	45	(10)	(25) (10+15)	∞
5	(45) (25+20)	45	(10)	(25)	∞
2	(45)	45	(10)	(25)	∞
3	(45)	(45)	(10)	(25)	∞
6					

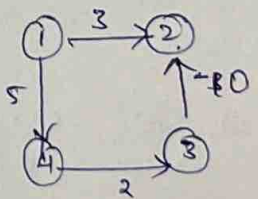
→ Smallest one for next Iteration

→ we can not Reach Vertex "6" because There is no Incoming Edge on "6" so we can't reach.

If already vertex is selected Then don't select again.

Drawback of Dijkstra Algorithm

→ let's we have negative value on Edge



vertex	2	3	4
1	(3)	∞	5
2	(3)	∞	(5)
4	(3)	7 (5+2)	(5)
(7)			

It may not work for negative values

No need to select Because Remaining nodes 2, 4 → already visited

But DG select the minimum value to node (2) is -3 (-10+7) => -3

↳ so the better value is -3 (-3<3)

↳ But Dijkstra says that don't visit the node if already visited

↳ so the Algorithm may not work for negative values

Dijkstra's Algorithm Steps

1) Mark the ending vertex with a distance of zero. Designate this vertex as current

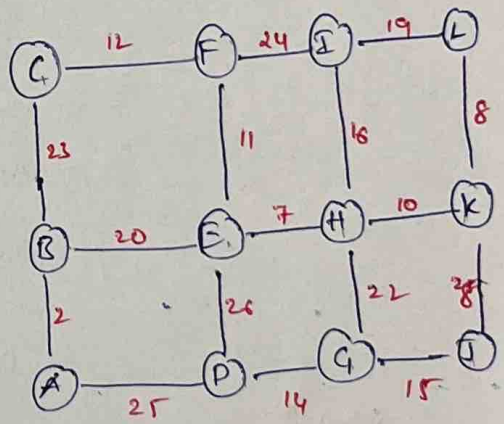
2) Find all vertices leading to the current vertex. Calculate their distances to the End. Since we already know the distance the current vertex is from the End, this will require adding the most recent edge.
   ↳ Don't record the distance if it is longer than a previously recorded distance

3) Mark the current vertex as visited. We will never look at the vertex again

4) Mark the vertex with the smallest distance as current, and repeat from step 2

Example :- find the length of the shortest path from vertex A to vertex L.

Ans :- 47



Circled → means visited Already

⊙ ⇒ Means main Element For next iteration

Start vertex	B	C	D	E	F	G	H	I	J	K	L
A	(2)	∞	25	∞	∞	∞	∞	∞	∞	∞	∞
B	(2)	25 (2+23)	25	(22) (2+20)	∞	∞	∞	∞	∞	∞	∞
E	(2)	(25)	25	(22)	33	∞	29 (22+7)	∞	∞	∞	∞
C	(2)	(25)	(25)	(22)	33 (23+33)	∞	29	∞	∞	∞	∞
D	(2)	(25)	(25)	(22)	33	39	(29)	∞	∞	∞	∞
H	(2)	(25)	(25)	(22)	(33)	(39)	(29)	45	∞	39	∞
F	(2)	(25)	(25)	(22)	(33)	(39)	(29)	45 (57 > 45)	∞	(39)	∞
K	(2)	(25)	(25)	(22)	(33)	(39)	(29)	45 !!	54	(39)	47

Next Smallest → But all the nodes connected TO (I) i.e F, H, Already marked as visited.
∴ for L ⇒ 47 K (45+19) ⇒ L = 47

left

Program

```
Public class Dijkstra {

    Public static void dijkstra(int[][] graph, int src) {
                                                  ↑ source

        int count = graph.length

        boolean[] visitedVertex = new boolean[count];
        int distance[] = new int[count];

    // Set All distances initially as ∞ Except source
       Also Set visited as false for all nodes

        Arrays.fill(distance, Integer.MAX_VALUE);
        Arrays.fill(visitedVertex, False);

    // Set source distance as 0
        distance[source] = 0

    for(int i=0; i<count; i++) {

        // Update the distance b/w neighbouring
           Vertex & source Vertex for Relaxation.
        int u = findMinDistance(distance, visitedVertex)
        visitedVertex[u] = true;        → same as prims

        // Update all the neighbouring vertex distances
                          ↳ i.e Relaxation
        for(int v=0; v<count; v++) {

            if( visitedVertex[i] = false &&
                graph[u][v] != 0    &&
                distance[u] + graph[u][v] < distance[v]){

                distance[v] = distance[u] + graph[u][v];
            }
        }

    // Print
        for(int i=0; i<distance.length; i++) {
            print(source, i, distance[i])
        }
```

Relaxation

③ Constraints

1) Edge must exist b/w u & v
2) vertex "v" is not included in MST
3) edge weight is current ③

Dynamic Programming

* Difference b/w Greedy Method and dynamic programing (DP)
(GM)

1) GM, DP both are used fr solving optimization problems

a) In GM → we tried to follow the defined procedure TO get the best result like, Krushkal Algo,

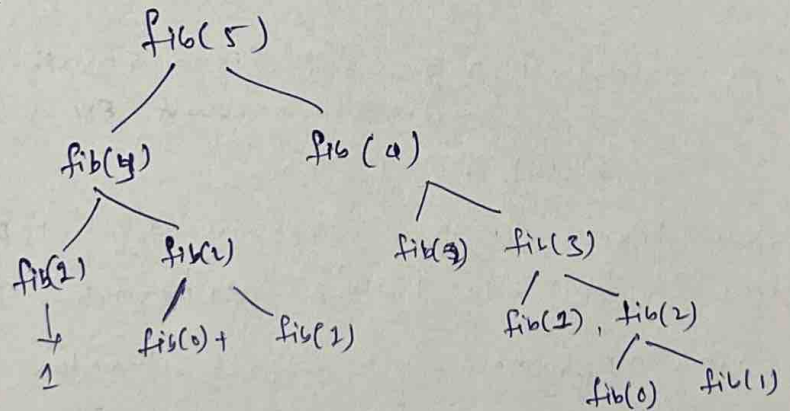
3) In dyp → we find all possible solutions and then pick up an optimal solution out of possible solutions

→ Time consuming compared to D·P
→ Mostly D·P problems are solved using recursive formulas.
→ It follows principal of optimality

Two Methods
1) Memoization Method
2) Tabulization Method
↳ widely used

It states that the problem can be solved by taking the sequence of desicions to get the optimal solution.

Ex:-




```
int fib(int n) {
  if(n<=1)
    return 1
  return fib(n-2) + fib(n-1)
```

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

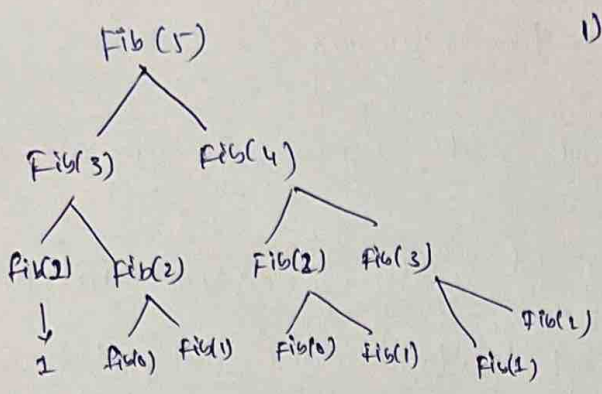
Assume => $(n-1)$ => $2T(n-1)$

=> Every time it adding the result => $2T(n-1)+1$

$= T(2^n)$ (By Recurrence)

→ If we observe the same fn is being called again & again for same parameter → say f(2) 3times

↳ So more fn calling → more time → so this can be reduced by storing values.



1) Initialize Array with (-1) a

T =

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

fib(5) => There is no value at index 5 -> so continue

fib(3) -> There is no value at index 3 -> so continue

Silly -> fib(2) ; fib(1) => 1

so store T[1] = 1

→

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

left side

=>

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

→ If we see here we have called fn fib => Fib(5), fib(4), fib(3), fib(2), fib(1), fib(0) → only once.

So total calls => O(n+1) => O(n)

→ The above Methodology is called as "Memoization". By storing the results of the fn we are avoiding the same call once again

↳ Here Fib(2) → called once and stored result -> so don't call again when Evaluating Fib(3), Fib(4) ...

→ If we see the above tree, The tree growing from TOP to bottom i.e Memoization works in TOP-to bottom Approach. If we want bottom-up approach we follow iterative Method i.e Tabularization Method

↳ widely used Method in D.P

Ex :-

F

0	1	1	2	3	.
0	1	2	3	4	

F =

0	1	1	2	3	5
0	1	2	3	4	

```
int fib(int n) {          // size = n
    F = [ ]
    if (n<=1) { return n; }
    F[0] = 0; F[1] = 1
    for (int i=2; i<=n; i++)
    {
        F[i] = F[i-2] + F[i-1]
    }
    return F[n]
}
```

↳ If you see Here the iteration starts from Smaller number → i.e from 0 where as in Recursion it starts from Bigger i.e @ 5 => n.

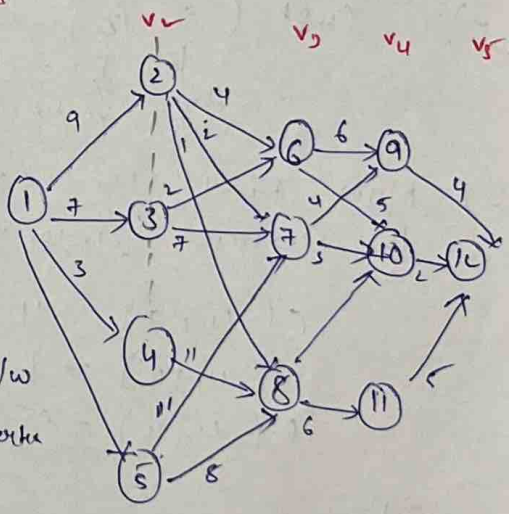
* Mostly dynamic problems solved using 2 Methods
  1) Memoization Method
     - Using recursion
     - Top - down Approach
  2) Tabulization Method
     - Using Iteration Method
     - Bottom - up Approch.

* Tabulization is widely used in D.P
* The main problem is identifying the solution (or) approach for solving a problem.

Multi-Staging

Time complexity $O(n^2)$ → Multi Stage Graph

* A multi stage graph is a directed graph, in which the nodes can be divided into a set of stages, such that all Edges are from a stage to stage only (In other words, There is no Edge b/w vertices of same stage and from a vertex of current stage to previous stage)



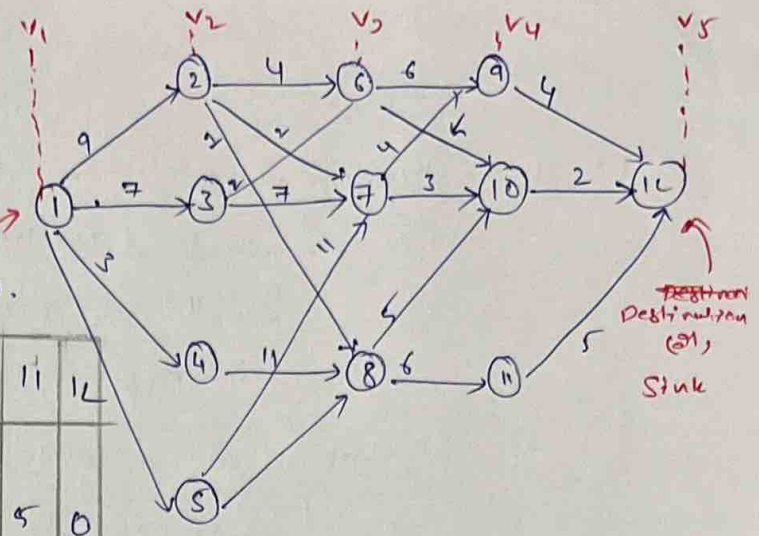
* We need to find the shortest path from source to destination By convention we consider source at stage 1 and destination at stage 5

V -> Vertex

C => Cost

D => ~~Distance~~ destination vertex

d => vertex giving the minimum value.



V	1	2	3	4	5	6	7	8	9	10	11	12
C	16	7	6	18	15	7	5	7	4	2	5	0
D	2 (or) 3	7	9	8	8	10	10	10	12	12	12	12

↳ vertex giving the min value

* If I want find the cost of vertex "1" then I should consider all ~~the~~ paths reaching sink
* Silly if want to find for vertex "3" then I need to consider paths from 3 to sink.

-> So, I will choose from sink, because cost will be zero, as it is the last vertex (or) (destination).

-> Cost (5, 12) = 0
(stage, vertex)

-> Now Find the cost of all vertices in the 4th stage i.e for the vertices (9, 10, 11)

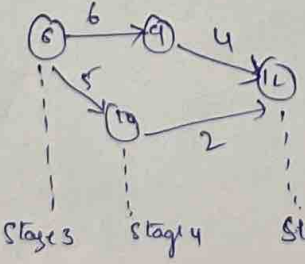
C(4, 9) = 4 =>
C(4, 10) = 2 =>
C(4, 11) = 5 =>
} By taking the above path we are reaching to node/vertex = 12
So d for all these vertices will be "12"

-> 3rd stage

$$c(3,6) = \min \{ c(6,9) + c(4,9),\ c(6,10) + c(4,10) \}$$

c(6,9) ⇓ cost of 6 to 9
c(4,9) ⇓ cost of 4th stage 9th vertex
c(6,10) ⇓ cost of 6 to 10
c(4,10) ⇓ cost of 4th stage vertex 10

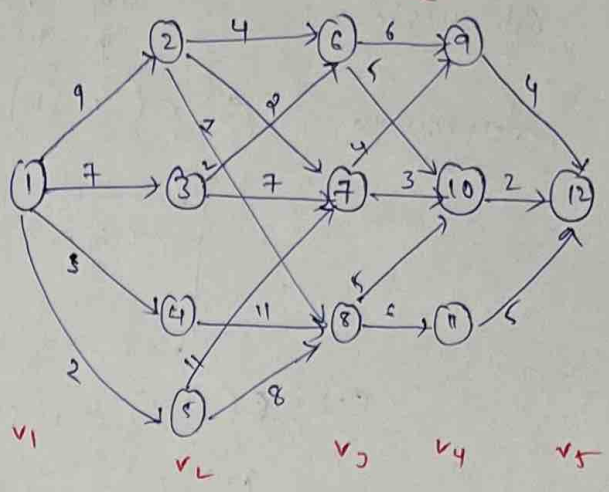
From vertex "6" we have two paths to reach "12"



$$c(3,6) = \min \{ (6+4), (5+2) \}$$
$$= \min \{ 10, 7 \}$$
$$= 7$$

↑ And vertex 6 to 10 is giving that min value.

DS-73



→ $c(3,7) = \min\{ c(7,9) + c(s_4, 9),\ c(7,10) + c(s_4, 10) \}$ (stage-4)

$= \min\{ (4+4), (3+2) \}$ ↳ Already found in table for previous stage

$= \min\{ 8, 5 \}$

$c(3,7) = 5$

↳ This min value is getting by vertex 10

$\therefore c(3,7) = 5, \quad d = 10$

→ $c(3,8) \Rightarrow \min\{ c(8,10) + c(s_4, 10),\ c(8,11) + c(s_4, 11) \}$

$= \min\{ (5+2), (6+5) \} \Rightarrow \min\{ 7, 11 \} \Rightarrow 7$

$c(3,8) = 7 ; \quad d = 10$ ↳ It is getting by node 10

Stage 2 (stage, vertex)

→ $c(2,2) \Rightarrow \min\{ c(2,6) + c(3,6),\ c(2,7) + c(3,7),\ c(2,8) + c(3,8) \}$ (This calculated in previous step; stages, vertex)

$\Rightarrow \min\{ (4+7), (2+5), (1+7) \} \Rightarrow \min\{ 11, 7, 8 \}$ ↳ It is getting because of 7 vertex 7

$= 7$

$c(2,2) = 7 ; \quad d = 7$

→ $c(2,4) = \min\{ c(4,8) + c(s_3, 8) \}$

$= \min\{ 11+7 \} = 18 ; \quad d = 8$

→ $c(2,5) = \min\{ c(5,7) + c(3,7) ; c(5,8) + c(3,8) \}$

$= \min\{ (11+5), (8+7) \} = \min\{ 16, 15 \} \Rightarrow 15.$

$d = 8$

Stage 1

→ $c(1,1) = \min\{ c(1,2) + c(2,2),\ c(1,3) + c(2,3),\ c(1,4) + c(2,4),\ c(1,5) + c(2,5) \}$

$= \min\{ (9+7), (7+9), (3+18), (2+15) \}$

$= \min\{ 16, 16, 21, 17 \} \Rightarrow 16$ ✓

So d can take 2, (2) 3.

Observation

formulae

stage, vertex no, some vertex in next stage

$$c(i,j) = \min\{ c(j,l) + c(l) \}$$

$$\boxed{c(i,j) = \min\{ c(j,l) + c(i+1,l) \}}$$

Ex:- $c(2,3) \Rightarrow \min\{ c(2,6) + c(3,6) \}$

$i=2; j=3$

$c(j,l)$: $l=6$; $c(i+1,l)$: $(i+1)$

$\therefore cost(i,j) = \min\{ c(j,l) + cost(i+1,l) \}$

where i = stage ; j, l = vertex no.

$(l = V_{i+1})$

→ Now Apply dynamic programming on the tabular data we have

↳ And we are going to solve it by taking sequence of decision and we will start from ~~star~~ source vertex (1) to sink vertex (12)

↳ so this is forward, the decisions will be taking in forward direction.

V	1	2	3	4	5	6	7	8	9	10	11	12
C	12	7	6	18	15	7	5	7	4	2	5	0
D	2/3	7	9	8	8	10	10	10	12	12	12	12

stage → vertex

$d(1,1) = 2/3 \Rightarrow (2)$ ← source ; let's select

d in 2nd stage: $d(2,2) \Rightarrow 7$

7 in 3rd stage: $d(3,7) \Rightarrow 10$

10 in 4th stage: $d(4,10) \Rightarrow 12$

$d(5,12) \Rightarrow 12$

So the path from 1 to 12 (Shortest path)

$9+2+3+2 = 16$

(1) —9→ (2) —2→ (7) —3→ (10) —2→ (12)

(Source) (destination)

If we choose 3

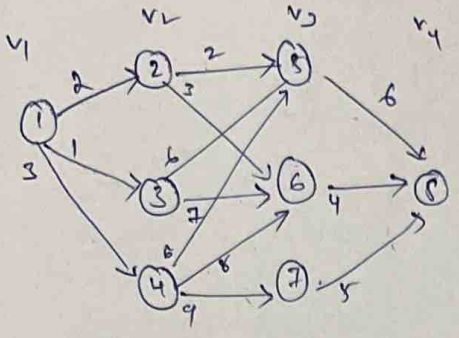
(1) —7→ (3) —2→ (6) —5→ (10) —2→ (12)

$7+2+5+2 \Rightarrow 16$

multistage graph program

DS-74

① → ② → ⑥ → ⑧ ← shortest path



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

Here vertex starts from 1 but Array starts from 0. So matrix consider from 0 to 8

```
int c[9][9] = {{ 0,0,0,0,0,0,0,0,0 },
               { 0,0,2,1,3,0,0,0,0 },
               { 0,0,0,0,0,2,3,0,0 },
               { 0,0,0,0,0,6,7,0,0 },
               { 0,0,0,0,0,6,8,9,0 },
               { 0,0,0,0,0,0,7,0,6 },
               { 0,0,0,0,0,0,0,6,4 },
               { 0,0,0,0,0,0,0,0,5 },
               { 0,0,0,0,0,0,0,0,0 }};
```

cost

0	1	2	3	4	5	6	7	8
	9	7	11		6	4	5	0
	12							n(8)=0

d

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

path

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

$cost[i] = \min\{ c[i][k] + cost[k] \}$ ← formula

i = 4

cost[4] = k = 5 ⇓ c[4,5) = 6 ∞, c[5] = 6

min{ 6, 6 }

k = 6 ⇒ c(4,6) + c(6) = 8 + 4 = 12

k = 7 ⇒ c(4,7) + c(7) = 9 + 5 = 14

k = 8 ⇒ It will not execute if statement → 5 vertex

⇒ min{ 12, 12, 14 }

```
main(){
  int stages = 4, min;
  int n = 8
  int cost[9], d[9], path[9];
  int c[9][9] = {{0,0,0,0,0,0,0,0,0},
                 {0,0,2,1,3,0,0,0,0},
                 . . . . . };
  cost[n] = 0;   // last node i.e destination
  for(int i = n-1; i >= 1; i--) {        → O(n-1) ≈ O(n)
    min = 32767;
    for(k = i+1; k <= n; k++) {          → n(n+1)/2 = O(n²)
      → if(c[i][k] != 0 && c[i][k] + c[k] < min){
          min = c[i][k] + c[k]
          d[i] = k
      }
    }
    cost[i] = min
  }
  p[1] = 1; p[stages] = n
  for(i = 2; i < stages; i++){
    p[i] = p[ d[i-1] ]
  }
```

finding path

DS-75

Matrix Chain Multiplication

1) What is Matrix multiplication → Time complexity $O(n^3)$
2) What is Matrix Chain multiplication
3) Formula using dynamic programming
4) How to use Dynamic programming Formula.
5) Example problem.

* Matrix Multiplication

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

Rows × Columns

2×3 ; 3×2 — Should be Equal for Matrix Multiplication

Rows — Columns

⇒ First Rule :- To perform a Matrix Multiplication, below condition should satisfy

⇓

No. of columns of first Matrix = No. of Rows of 2nd Matrix

$2 \times 3 = 3 \times 2$

⇒ How we Multiply :- All the Elements of row in first Matrix with all the Elements of single column in the second Matrix

$$\Rightarrow A \times B = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \end{bmatrix}$$

$A = 2 \times 3$; $B = 3 \times 2$ — equal

Resultant matrix (2×2) size / dimension

2×2

Total no. of Multiplications

$= 12$

$= 2 \times 3 \times 2 = 12$

Rows of 1st matrix — Columns of 2nd Matrix

Columns of 1st / Rows of 2nd matrix

Matrix Chain Multiplication

=) At a time we can multiply 2-matrix

d => dimension

$A_1 \times A_2 \times A_3$

$2 \times 3 \quad 3 \times 4 \quad 4 \times 2$

$d_0 \; d_1 \quad d_1 \; d_2 \quad d_2 \; d_3$

→ The above matrices can be multiplied in two ways, but the final result will be same.

$R = (A_1 \times A_2) A_3$ (1)

$2 \times 3 \quad 3 \times 4 \quad 4 \times 2$

Multiplication → $2 \times 3 \times 4 \Rightarrow 24$

$(A_1 \times A_2) \times A_3$

$2 \times 4 \quad 4 \times 2$

Result → 2×2

Total multiplication => $2 \times 4 \times 2$

$(A_1 \times A_2) \times A_3 = 16$

$2 \times 4 \quad 4 \times 2$

→ Total multiplications for which Result (R)

$24 + 16 = 40$

→ Resultant matrix dimension (R)

$R_1 \times A_3 \Rightarrow 2 \times 2$

$2 \times 4 \quad 4 \times 2$

$A_1 \times (A_2 \times A_3)$ (2)

$R_1 = A_2 \times A_3$

$2 \times 3 \quad 3 \times 4 \quad 4 \times 2$

→ $A_2 \times A_3 = R_1$

↳ Total multiplications

$3 \times 4 \times 2 = 24$

→ Result dimension $(R_1) = 3 \times 2$

$A_1 \times R_1$

$2 \times 3 \quad 3 \times 2$

=) Total multiplications $(A_1 \times R_1)$

$=) 2 \times 3 \times 2 =) 12$

→ Total multiplications $(A_1 \times (A_2 \times A_3))$

$12 + 24 =) 36$

→ Resultant matrix dimensions

$A \times R_1$

$2 \times 3 \quad 3 \times 2 \Rightarrow 2 \times 2$

Note :- If we observe, the no. of multiplication to perform to get the Result is not same in both the cases. So to get the optimal solution we need to choose the set, whose no. of multiplications will be less.

↳ But How do we identify the set, and which multiplication should be performed such that the total efforts put in for multiplying them should be minimized.

↳ This is matrix chain multiplication problem.

→ Finding the best possible paranthesisation such that the total cost of Multiplications will be less

↳ So dynamic programming approach says that, you should Try all possible paranthesisation and pick up the best.

↳ So we need some Formulae to do the JOB quick.

$A_1 \times A_2 \times A_3$

$2 \times 3 \quad 3 \times 4 \quad 4 \times 2$

$d_0 \quad d_1 \quad d_1 \quad d_2 \quad d_2 \quad d_3$

$(A_1 \times A_2) + A_3$ | $A_1 \times (A_2 \times A_3) \Leftarrow$

$2 \times 3 \quad 3 \times 4 \quad 4 \times 2$

$d_0 \quad d_1 \quad d_1 \quad d_2 \quad d_2 \quad d_3$

Cost [1,2] + Cost [3,3] + (i to j times) $C[1,1] + C[2,3]$ ← Step 2

↑ ↑ ↑ ↑

$A_1 \quad A_2 \quad A_3 \quad A_3$

$C[1,1] = 0$

$C[2,3] = 3 \times 4 \times 2 = 24$

$C[1,2] = 24 \quad (2 \times 3 \times 4)$

$C[3,3] = 0$

$A_1 \times [A_2 \times A_3]$

$d_0 \quad d_2$

$(A_1 \times A_2) \Rightarrow 2 \times 4$

$2 \times 3 + 3 \times 2$

$d_0 \quad d_1 \quad d_1 \quad d_3$

$\Rightarrow C[1,1] + C[2,3] + (2 \times 3 \times 2)$

$\Rightarrow C[1,1] + C[2,3] + (d_0 \times d_1 \times d_3)$

$\Rightarrow C[1,1] + C(2,3) + d_0 \times d_1 \times d_3 = 28$

$(A \times A_2) \times A_3$

$2 \times 4 \quad 4 \times 2$

$d_0 \quad d_2 \quad d_2 \quad d_3$

$C[1,2] + C(3,3) + (d_0 \times d_2 \times d_3)$

$24 + (2 \times 4 \times 2) = 40$

→ The Above Matrices $A_1 A_2 A_3$ can be Multiplied in 2 ways as below

i k → some next considerations in k

k+1 j

$d_{i-1} \quad d_k \quad d_j$

1) $C[1,2] + C(3,3) + (d_0 \times d_2 \times d_3) \Rightarrow (A_1 \times A_2) \times A_3$

consider as i ↑ consider as j

2) $C[1,1] + C(2,3) + (d_0 \times d_1 \times d_3) \Rightarrow A_1 (A_2 \times A_3)$

i k k+1 j $d_{i-1} \quad d_k \quad d_j$

observations

Formula :-

$$C[i,j] = \min_{k} \{ C[i,k] + C[k+1,j] + (d_{i-1} \times d_k \times d_j) \}$$

where $k \Rightarrow i \leq k < j$

↓ Here 3 matrices so k = 1, 2

Matrix chain Multiplication

$$C[i,j] = \min_{k=\ i \le k < j} \{ C[i,k] + C[k+1, j] + d_{i-1} + d_k + d_j \}$$

Let's look for 4 matrices

A_1	A_2	A_3	A_4
$d_0 \times d_1$	$d_1 \times d_2$	$d_2 \times d_3$	$d_3 \times d_4$

- Possibilities to multiply matrix.

1) $A_1 (A_2 (A_3 A_4))$
2) $A_1 ((A_2 A_3) A_4)$
3) $(A_1 A_2)(A_3 A_4)$
4) $(A_1 (A_2 A_3)) A_4$
5) $((A_1 A_2) A_3) A_4$

↳ So if we have 4 - matrix → then 5 - possibilities

$$\frac{2(n-1) C_{(n-1)}}{n} \quad (\text{or}) \quad \frac{2n C_n}{n+1}$$ where $n = (n-1)$

for 4 matrices ⇒ $n-1 \Rightarrow 4-1 = 3$

$$\frac{2(3) C_3}{4} \Rightarrow \frac{6 C_3}{4} \Rightarrow \frac{6!}{(6-3)! \times 3!} \times \frac{1}{4}$$

$$\Rightarrow \frac{6 \times 5 \times 4 \times 3!}{3! \times 3!} \times \frac{1}{4}$$

$$\frac{6 \times 5 \times 4}{3 \times 2 \times 1} \times \frac{1}{4} \Rightarrow \frac{5 \times 4 \times 2}{3 \times 2 \times 1} \times \frac{1}{4} \Rightarrow \frac{10}{3} \times \frac{1}{4}$$

$$= \frac{5}{3}$$

Now out of these 5 - this five the best possible way → now apply matrix chain multiplication formula.

Matrix Multiplication – formulae.

$$c[i,j] = \min_{i \le k < j} \{ c[i,k] + c[k+1,j] + d_{i-1} + d_k + d_j \}$$

→ for 4 matrices

A_1 A_2 A_3 A_4

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

$i=1$, $j=4$

$$c[1,4] = \min_{1 \le k < 4} \begin{cases} k=1 & c[1,1] + c[2,4] + (d_0 * d_1 * d_4), \\ k=2 & c[1,2] + c[3,4] + (d_0 \times d_2 \times d_4), \\ k=3 & c[1,3] + c[4,4] + d_0 \times d_3 \times d_4, \end{cases}$$

(i.e $k = 1, 2, 3$)

$\Downarrow$ i.e

$k=1$ $A_1 (A_2 * A_3 A_4)$

$k=2$ $(A_1 A_2)(A_3 A_4)$

$k=3$ $(A_1 A_2 A_3) A_4$

We know $c[1,1] = 0$, $c[4,4] = 0$ → now we need to find $c[2,4]$ & $c[3,4]$, $c[1,2]$ & $c[1,3]$

$$c(2,4) = \min_{2 \le k < 4} \begin{cases} k=2 & c(2,2) + c[3,4] + d_1 \times d_2 \times d_4 \\ k=3 & c(2,3) + c[4,4] + d_1 \times d_3 \times d_4 \end{cases}$$

↳ If we observe → above case $c[2,4]$ ⇒ again we need to find out the values for $c[2,3]$ $c[3,4]$ as $c(2,2)$ & $c[4,4] = 0$ → And it's not practical.

↳ And this process continues. → And How can we achieve arrive at the solution quickly. ?

And Also if we observe.

$c[1,4]$ ⇒ k = 3 possibilities → And then find min

$c[2,4]$ ⇒ $k=2$ possibilities → And then find min

Similarly for $c[i,j]$ ⇒ $(j-1)$ possibilities

k = 1 to $j-1$

↳ TO Arrive quickly at the solution

↳ First Find smaller of then go for larger values of k

i.e $k=0$, $k=1$, $k=2$ . . . .

4- matrices

$\rightarrow j$

i ↓	1	2	3	4
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

C =

↑ cost of matrix multiplication

1st min

i → Rows
j → columns

K =

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

$\Rightarrow k = j - i \checkmark \geq 0$ & $\leq j$

For 4 matrices possible k values → 0, 1, 2, 3

k=1

i	j
1	2
2	3
3	4

k=2

i	j
1	3
2	4

k=3

i	j
1	4

	1	2	3	4
1	0	24		
2		0	16	
3			0	
4				0

(c)

→ First fill diagonals of c & k as 0 => as

c[1,1] = 0 c[2,2] = 0

c[3,3] = 0 c[4,4] = 0

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

(k)

$$c[1,2] = \min_{1 \leq k < 2} \{ c[1,1] + c[2,2] + (d_0 \times d_1 \times d_2) \}$$

$k=1$

$0 + 0 + 3 \times 2 \times 4$

$= 24$

→ I got this value for k value 1

So $c[1,2] = 24$

$k(1,2) = 1$

So

→ $c[2,3] = \min_{2 \le k < 3} \{ k=2 \{ c[2,2] + c[3,3] + (d_1 \times d_2 \times d_3) \}$

$0 + 0 + (2 \times 4 \times 2)$

$c[2,3] = 16 \quad ; \quad k[2,3] = 2$

→ $c[3,4] = \min_{3 \le k \le 4} \{ k=3 \{ c[3,3] + c[4,4] + (d_2 \times d_3 \times d_4) \}$

$0 + 0 + 4 \times 2 \times 5 = 40$

$c[3,4] = 40 \quad ; \quad k[3,4] = 3$

→ $k=2$

$c[1,3] = \min_{1 \le k < 3} \{ k=1 \{ c[1,1] + c[2,3] + (d_0 \times d_1 \times d_3), \; k=2 \{ c[1,2] + c[3,3] + (d_0 \times d_2 \times d_3) \}$

$\Rightarrow \min \{ k=1 \{ 0 + 16 + (3 \times 2 \times 2), \; k=2 \{ 24 + 0 + (3 \times 4 \times 2) \}$

$\Rightarrow \min \{ 28, 48 \}$

$\uparrow$ min

$c[1,3] \Rightarrow k[1,3] = 1$

$c[2,4] = \min_{2 \le k < 4} \{ k=2 \{ c[2,2] + c[3,4] + (d_1 \times d_2 \times d_4), \; k=3 \{ c[2,3] + c[4,4] + (d_1 d_3 d_4) \}$

$= \{ 0 + 40 + (2 \times 4 \times 5), \; 16 + 0 + (2 \times 2 \times 5) \}$

$= \{ 80, 36 \}$ $\Rightarrow c[2,4] = 36, \; k[2,4] = 3$

↑ min

	1	2	3	4
1	0	24	28	58
2		0	16	36
3			0	40
4				0

(c)

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

k

$k=3$ Sly

$c[1,4] = \min_{1 \le k < 4} \{ k=1 \Rightarrow \min \{ c[1,1] + c[2,4] + d_0 \times d_1 \times d_4, \; k=2 \Rightarrow c[1,2] + c[3,4] + d_0 d_2 d_4, \; k=3 \Rightarrow c[1,3] + c[4,4] + d_0 d_3 d_4 \}$

$\Rightarrow \min(88, 124, 58)$

$c[1,4] = 58 \quad k[1,4] = 3$

$C[1,4] \Rightarrow 58$

↑ This is the min no. of Multiplications we need to perform for 4 matrices (in our example)

(C)

i \ j	1	2	3	4
1	0	24	28	58
2		0	16	36
3			0	40
4				0

→ How Paranthisation should be done

↳ Which should be multiply first and then next out of 5 possibilities what to choose

↳↳ This Information we get using the generated k-Table →

(k)

i \ j	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

↳ So I have Four matrices $A_1\ A_2\ A_3\ A_4$ ⇒ Our Final Answer for cost is $C(1,4) = 58$

↳↳ Similarly take the vertex in k-Table i.e $k[1,4] = 3$

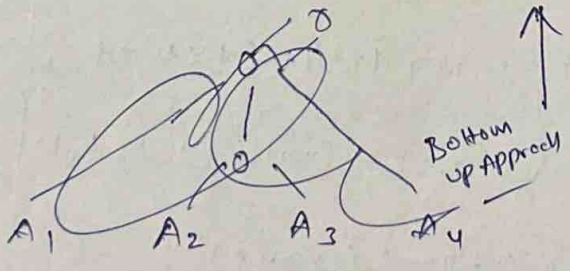
↳ Here do Paranthisation

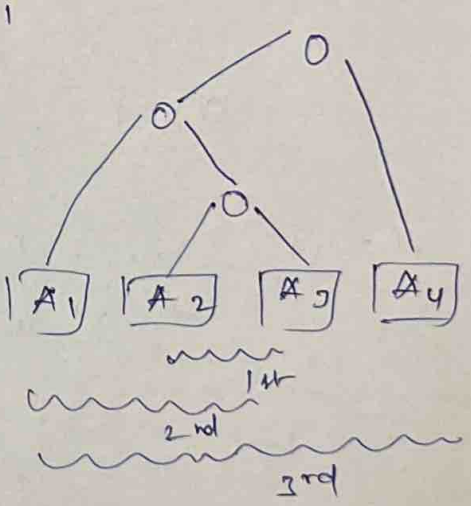
1) $(A_1 A_2 A_3) A_4$ ↳↳ $k(1,4) \Rightarrow$

2) $(A_1) A_2 A_3$ i.e $C[1,3] \Rightarrow 28$, $k(1,3) \Rightarrow 1$

↳↳

$((A_1) \times (A_2 A_3)) A_4$





→ Time complexity

↳ C Table ⇒ $1+2+3+4 \Rightarrow 4(5/2)$ (no. of values in C Table)

$= n(n+1)/2 = n^2$

↑

→ C-Table is computed diff k-values ⇒ so consider as n

$(n^2)\, n = n^3$

$= O(n^3)$

Matrix multiplication problem (program)

A_1 A_2 A_3 A_4

5 x 4 | 4 x 6 | 6 x 2 | 2 x 7

d_0 d_1 d_2 d_3 d_4

P =

0	1	2	3	4
5	4	6	2	7

m =

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

(m) - Table

S =

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

(S) - Table

Here we have FOUR matrices

But we choose n = 5 / As

Array index starts with 0, 4

we assign 0th columns & 0th Row as zero ↳ so d starts from 1 to 4

d = 1

i	j
1	2
2	3
3	4

d = 2

i	j
1	3
2	4

d = 3

i	j
1	3

If we observe j = d + i

```
void main() {
    int n = 5
d   => int p[7] = {5, 4, 6, 2, 7}
c   => int m[5][5] = {0}  => all places as zero and for value
k   => int s[5][5] = {0};
Table
    int j, min, q.
    for(int d = 1; d < n-1; d++){
        for(int i = 1; i < n-d; i++){
            j = i + d;
            min = 32767
        for(int k = i; k <= j-1; k++){
            q = m[i][k] + m[k+1][j] + (p[i-1] * p[k] * p[j])
            if (q < min){
                min = q;
                s[i,j] = k;
            }
        }
        m[i][j] = min;
    }
}
```

Ex: d = 2; i = 2; j = 4

then k = i ≤ k j 2) 2 ≤ k < 4

i.e 2, 3

for d = 2

i => 2 time loop

j = 3, 4

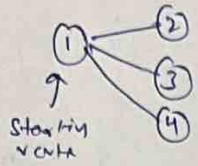
→ Dynamic Programing.

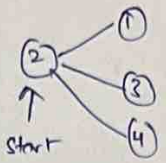
DS-80

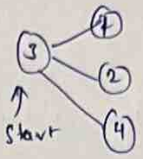
All Pairs Shortest Paths (Floyd-Warshall)

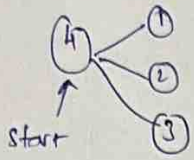
↳ Find Shortest path b/w Every pair of vertices.

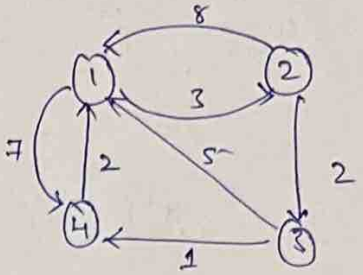
↳ let say the Starting vertex is 1 then I need TO find the path from, Slly For other





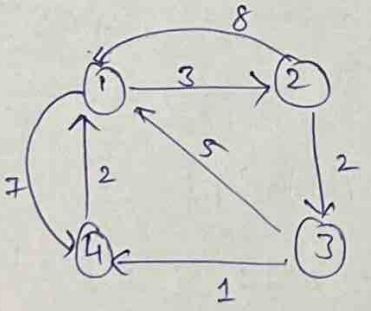






→ The above looks similar to single source shortest path that is is dijkstra Algorithm. but dijkstra algorithm finds shortest path From one of the vertex. As we dijkstra takes $O(n^2)$. so If I run dijkstra algorithm fo all vertices one by one. Then we get the result is $n \times n^2 \Rightarrow O(n^3)$. Now lets see How

→ Can b it solved using dynamic programing → i.e take decision at every step. → This can be done by Just preparing matrices



Adjacency matrix

$\Rightarrow A^0 =$ ↑ Original I/p (or) Original matrix

	1	2	3	4
1	0	3	∞	∞
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

→ no-direct Edge take it Infinity

→ And Self loop if it is not there we will mark as "0"

↳ Ex:- (1,1) (2,2) (3,3) (4,4)

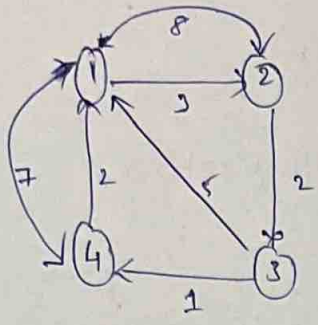
→ Now prepare Matrix For the Starting vertex (1) i.e A^1

$A^1 =$

	1	2	3	4
1	0	3	∞	7
2	8	0		
3	5		0	
4	2			0

⇒ When I say vertex "1" then all the paths that belongs to vertex 1 will remain unchanged. so I should not calculate them, directly I can take them.

→ no-self loops → so Fill diagonals with "0"



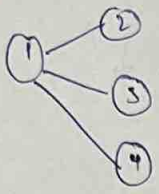
$A^0 =$

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

A^1

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

A^1



$A^1[2,3] = A^0[2,3] \quad A^0[2,1] \quad A^0[1,3]$

↳ In B/w I need to include vertex 1 as we are finding way from vertex 1

$2 < 8 + \infty = \infty$

$\min \Rightarrow 2$

$\therefore A^1(2,3) = 2$

$\rightarrow A^1[2,4] \Rightarrow \min\{A^0[2,4], A^0[2,1] + A[1,4]\}$

$\min(\infty, 8+7) \Rightarrow 15$

↳ Since means that there is no direct path from (2 to 4) $\Rightarrow$ as it is ∞; it has the path only 2 → 1 → 4

$\rightarrow A^1[3,2] \Rightarrow A^0[3,2], A[3,1] + A[1,2]$ ⇒ In b/w 3, 2 I need 1 As we are finding way for vertex 1

$\Rightarrow \min[\infty ; 5+3) \Rightarrow 8$

Slly for others.

$A^1(3,3) = 0$ (As no self loop)

$A^1(3,4) = A^0(3,4), A^0[3,1] + A^0[1,4]$

$\Rightarrow \min(1, 5+7)$

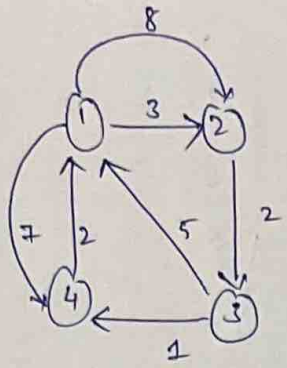
$\Rightarrow \min(1, 12) = 1$

$A^1(4,2) \Rightarrow \min\left(A^0(4,2), A^0[4,1] + A(1,2)\right)$

$\infty, 2+3 \Rightarrow 5$

$A^1(4,3) \Rightarrow \left(A^0[4,3), A^0(4,1) + A(1,3)\right) =$

$\infty, \infty + \infty \Rightarrow \infty, \infty = \infty$



A^0 =

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

A^1 =

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

=) Now Similarly Find ways from Vertex 2 i.e A_2

↳ when we are finding paths to other vertecies from vertex "2" then all the path that belong to vertex "2" will be unchanged

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

Remains same

values populated from A^1

=) NO self loop so diagonal are filled with "0"

→ $A_2(1,3)$ =) $A_1[1,3]$, $A_1[1,2] + A_1[2,3]$

↓↓

In b/w 1 & 3 include vertex 2

$$\min\{A_1[1,3], A_1(1,2) + A_1(2,3)\}$$

$$\min\{\infty, 3+2\} \Rightarrow 5$$

→ $A_2[1,4]$ =) $\min\{A_1[1,4], A_1[1,2] + A_1[2,4]\}$

$$\min\{7, 3+15\} \Rightarrow 7$$

=) Now similarly find out For Vertex "3"

A_3 =

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

→ Remains Same From previous value

Sily A_4 =

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

Observation

$$A^k[i,j] = \min\{A^{k-1}(i,j], A^{k-1}(i,k) + A^{k-1}[k,j]\}$$

Program (Floyd-Warshall)

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

↳ Every time I was generating matrices

So How many matrices => 4 => n

(A^1, A^2, A^3, A^4)

↳ i.e ↳ I need to generate for a each and every term. i.e $n \times n$ (Rows × columns)

↳ So Two loops

↳ for n - elements => $n \times n$ => no. of elements

③ → for(k = 1; k <= n; i++) {

① → for(i = 1; i <= n; i++) {

② → for(j = 1; j <= n; j++) {

Intermediate vertex

$A(i,j) = \min\{ A[i,j], (A[i,k] + A(k,j)) \}$

Not using diff Array (like A1, A2, A3, A4)

→ I took single Instance of 2D Array and updating the same at every Iteration

} } }

Scrap For understood Program

3 - loops => $O(n^3)$

k = 1 => A^1 | k = 3 => A_3

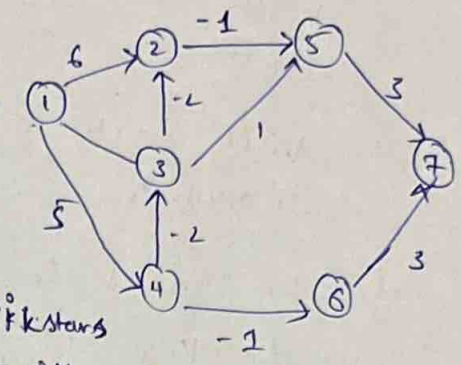
k = 2 => A^2 | k = 4 => A_4

```
for(k = 1; k <= n; k++) {
    for(i = 1; i <= n; i++) {
        for(j = 1; j <= n; j++) {
            A(i,j) = min(A(i,j),
                         A(i,k) + A(k,j))
        }
    }
}
```

Single Source Shortest Path (Dynamic Programming)

↳ Bellman - Ford Algorithm

* The problem is in a directed weight graph we have to find out the shortest path to all other vertices by choosing one vertex as source.



* For doing this we already have dijkstras algorithm. but dijkstras algorithm will not work for negative edges. (it may or may not give correct results)

* So the Bellman ford Algorithm will give us the correct result, and this Algorithm follows dynamic programming.
  ↳ As in dynamic programming we try all possible solutions and pick up the best one.

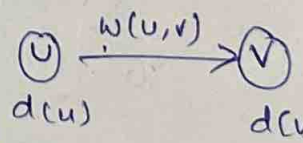
* But How do we try out the possible solutions ?
  ↳ This Algorithm says that go and relax all the Edges for (n-1) times where n is no. of nodes/vertices
  i.e there n = 7 i.e max possible Edges = 6 (7-1)

Edge Relaxation

↳ The Edge relaxation is the operation to calculate the reaching cost to the vertex lower. More concretely,

⇓

For the Edge from the vertex 'u' to the vertex 'v', if $d(u) + w(u,v) < d[v]$ is satisfied, update $d(v)$ to $d(u) + w(u,v)$



↳ The vertices u,v stand the neighbours in the graph and d[u], d[v] stand the reaching cost to the vertices u & v. Also w(u,v) stands the weight of the Edge from vertex "u" to vertex "v"

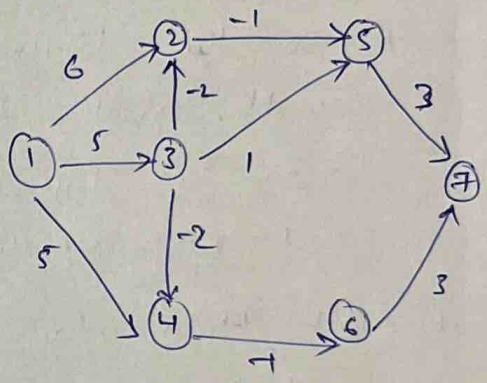
How Bellman-Ford Algorithm works?

1) Start with the weighted graph

2) Choose a starting vertex and mark the distances for that vertex "0" and assign infinity path values to all other vertices.

3) Visit each edge and relax the path distances if they are inaccurate

4) We need to do this for (n-1) times. but better to do this for "n" times in worst case (negative cycle graphs) a vertex path length might need to be readjusted n times



Possible edges from source (1)

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

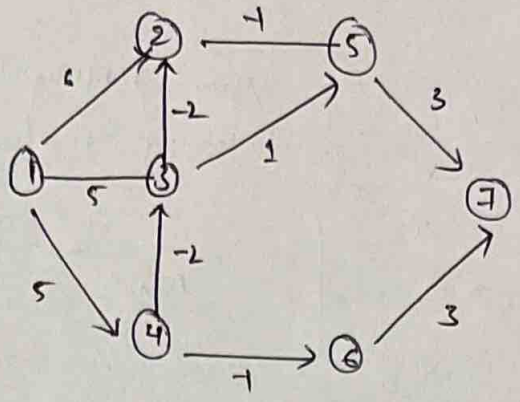
Iterations	A	B	C	D	E	F	G
	1	2	3	4	5	6	7
0	∞	∞	∞	∞	∞	∞	∞
1	0	6 (6<∞)	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	0	4	5
4	0	1	3	5	0	4	3
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

Now Shortest path. 1 to 7 A to G = 3

1 → 4 → 3 → 2 → 5 → 7
A D C B E G

∴ 5 + (-2) + (-2) + (-1) + 3 = 3

DS- 83



Edge list :- Any order.

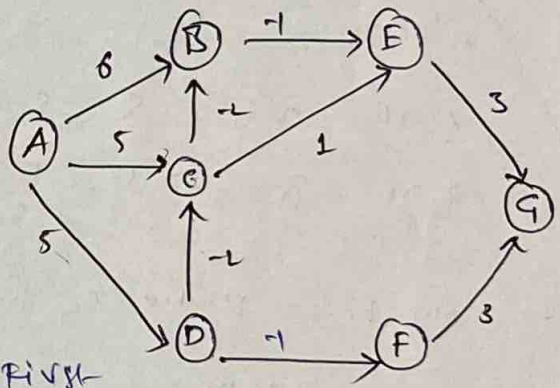
↳ (1,2) (1,3) (1,4) (2,5) (3,4) (3,5) (4,3) (4,6) (5,7) (6,7)

Iteration	(1,2)	(1,3)	(1,4)	(2,5)	(3,4)	(3,5)	(4,3)	(4,6)	(5,7)	(6,7)
Initial	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
1st	0	6								

Edge list

(A,B) (A,C) (A,D), (B,E) (C,B), (C,E)

(D,C) (D,F) (E,G) (F,G)

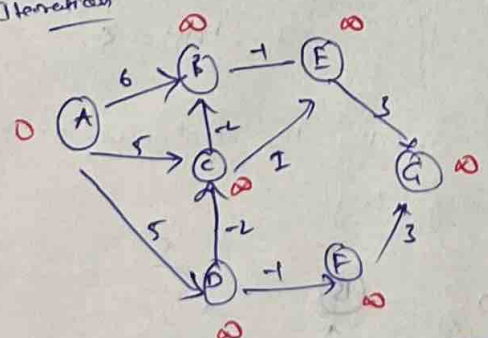


→ In order to find the shortest path, first we will initialize the source vertex (A) as "0" and other vertices with ∞.

→ After that, we will traverse towards each vertex from the source node. Update the value of the node during the traversal.

1st Iteration

1)



A	B	C	D	E	F	G
0	∞	∞	∞	∞	∞	∞

⇒ From Vertex "A" we can move to Vertex B & C & D

a) Moving A → B

A	B	C	D	E	F	G
0	∞	∞	∞	∞	∞	∞
0	6	∞	∞	∞	∞	∞

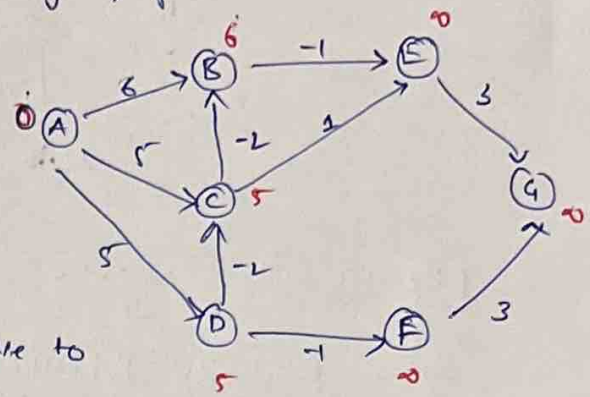
3) Moving A → C

A	B	C	D	E	F	G
0	∞	∞	∞	∞	∞	∞
0	6	5	∞	∞	∞	∞

A → C (Moving A to C)

	A	B	C	D	E	F	G
	0	∞	∞	∞	∞	∞	∞
(0+6) 6<∞	0	6	∞	∞	∞	∞	∞
(6+5)<∞ 5<∞	0	6	5	∞	∞	∞	∞
	0	6	5	5	∞	∞	∞

After updating the distances, we get the following graph.

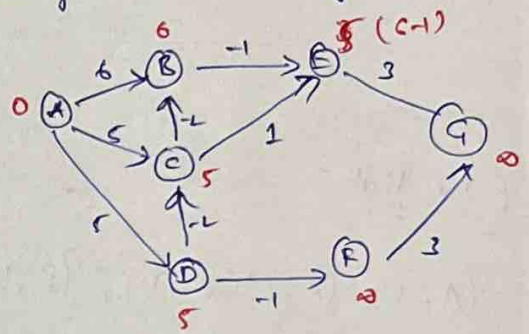


→ Now From Vertex B, we can move to vertex E

Moving B → E ⇒ (6-1) = 5

	A	B	C	D	E	F	G
(Initial)	0	∞	∞	∞	∞	∞	∞
A ⇒ B	0	6	∞	∞	∞	∞	∞
A → C	0	6	5	∞	∞	∞	∞
A → D	0	6	5	5	∞	∞	∞
B → E	0	6	5	5	5	∞	∞

After updating the distances we got the following graph



→ Now From Vertex C, we can move to E, B

Moving C to E ⇒ C (5) → E (5) ⇒ (5+1) < 5

6 < 5 ⇒ False so don't change

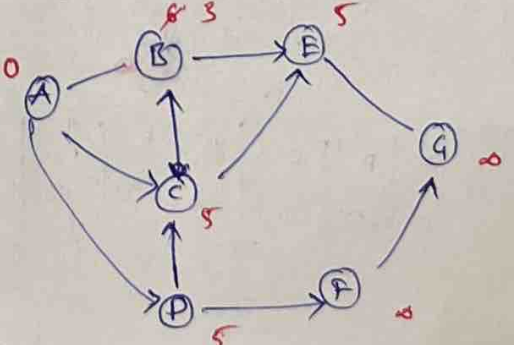
Moving C to B ⇒ C (5) → B (3) ⇒ (5 + (-2)) < 6

⇒ 3 < 6 ⇒ T

⇒ d(B) = 3

	A	B	C	D	E	F	G
Initial	0	∞	∞	∞	∞	∞	∞
(A, B)	0	6	∞	∞	∞	∞	∞
(A, C)	0	6	5	∞	∞	∞	∞
(A, D)	0	6	5	5	∞	∞	∞
(B, E)	0	6	5	5	5	∞	∞
(C, E)	0	6	5	5	5	∞	∞
(C, B)	0	3	5	5	5	∞	∞

After updating the distance, we get the following graph



-> Now from Vertex D we can moved to E, F

↳ moving (D) ——-2——> (C)
5

5 => 3 => ((5+(-2)) < 5)

(3 < 5) ✓ so update the min value.

↳ 3

↳ moving (D) ——-1——> (F)
5

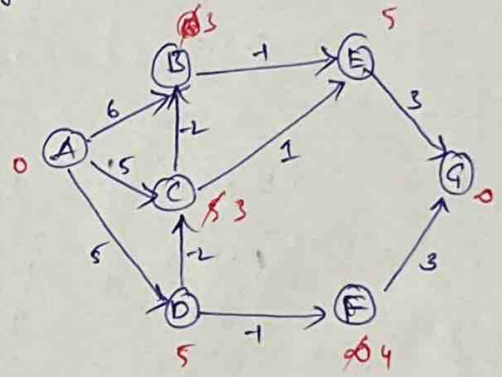
∞ => (5+(-1)) < ∞)
4

= 4 < ∞ => True

=> d(F) = 4

	A	B	C	D	E	F	G
	0	∞	∞	∞	∞	∞	∞
	0	6	∞	∞	∞	∞	∞
	0	6	5	∞	∞	∞	∞
	0	6	5	5	∞	∞	∞
	0	6	5	5	5	∞	∞
	0	6	5	5	5	∞	∞
	0	3	5	5	5	∞	∞
(D,E)	0	3	3	5	5	4	∞

After updating the distance graph looks like below



-> Now from Vertex "E" we can Move to G

(E) ——3——> (G)
5

∞ (5+3) = 8

last row from prev stg ↑

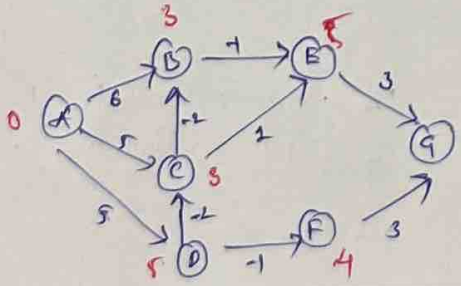
A	B	C	D	E	F	G	....
0	3	3	5	4	4	∞	← (D,E)
0	3	3	5	4	4	8	(E,4)

-> Now from Vertex "F" we can Move to G

(F) ——3——> (G)
4

8 => ((4+3) < 8)

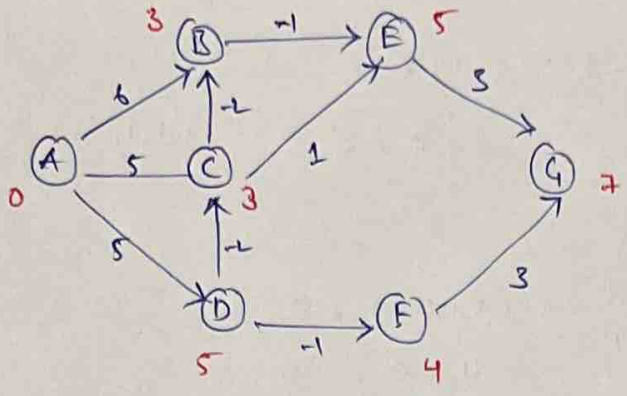
7 => 7 < 8 = True



=> At the end 1 iteration

A	B	C	D	E	F	G
0	∞	∞	∞	∞	∞	∞
0	6	∞	∞	∞	∞	∞
0	6	5	∞	∞	∞	∞
0	6	5	5	∞	∞	∞
0	6	5	5	5	∞	∞
0	6	5	5	5	∞	∞
0	3	5	5	5	4	∞
0	3	3	5	5	4	8
0	3	3	5	5	4	7

→ At the End of 1st iteration (i.e After traversing all the Edges in the graph)



A	B	C	D	E	F	G
0	3	3	5	5	4	7

distances of vertices to the source vertex

→ Follow the same procedure for the next iteration.

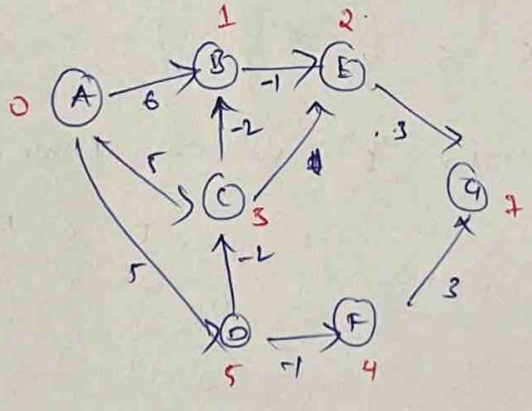
2nd Iteration i.e

(A, B)	(A, C)	(A, D)	(B, E)	(C, B)	(C, E)	(D, C)	(D, F)
3 (0+6)<3	3 (0+5)<3	5	(3-1)=2 2<5	(3-2)=1 1<3	(3+1)=4 4<5 =4	(5-2)=3 3=3 3	(5-1)=4 4=4 =4

(E, G)	(F, G)
(5+3)=8 8<7 × =7	(4+3)=7 7=7

=>

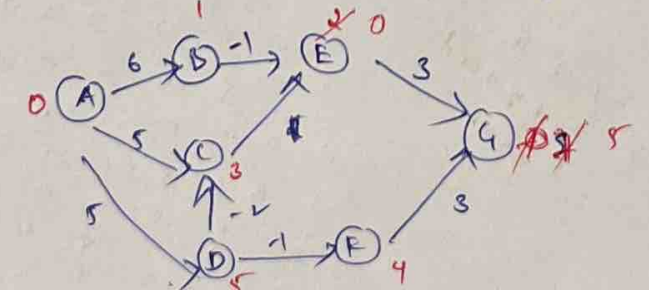
A	B	C	D	E	F	G
0	~~3~~ 1	~~3~~ 3	5	~~5~~ 2	4	7



S/Ly 3rd Iteration

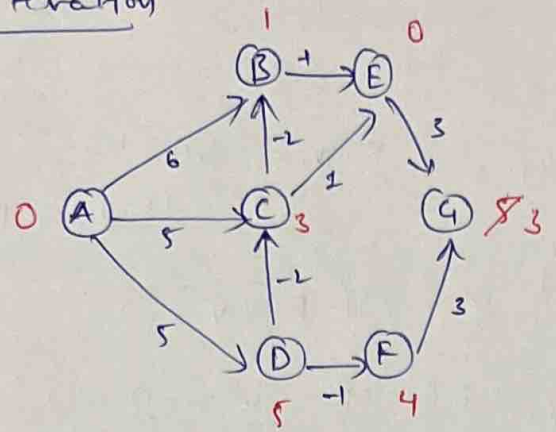
(A, B)	(A, C)	(A, D)	(B, E)	(C, B)	(C, E)	(D, C)	(D, F)	(E, G)	(F, G)
1	3	5	0 (2-1)<2 0<2 =0	1	2	3	4	(2+3) 5<7 ∴	5

So update G as 5



A	B	C	D	E	F	G
0	1	3	5	0	4	5

4th iteration



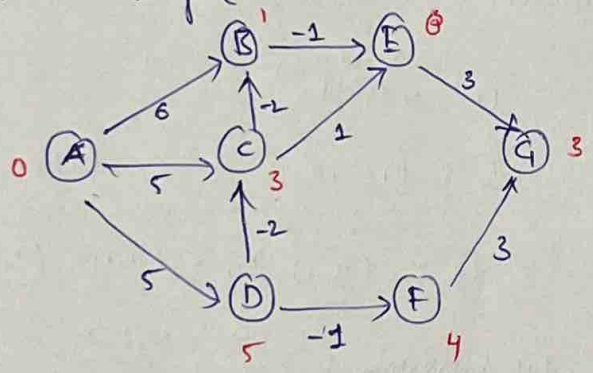
(A,B) → 1
(A,C) → 3
(A,D) → 5
(B,E) → 0
(C,B) → 1
(C,E) → 0
(D,F) → 4
(E,G) => 0+3
3 < 5
so → G = 3
(F,G) → 3

A	B	C	D	E	F	G
0	1	3	5	0	4	3

→ Break the loop as no relaxation is not performed means no other steps also relaxation won't performed.

5th iterations → No change ; 6th iteration no change.

↳ Final Graph



Vertex	Distance
A	0
B	1
C	3
D	5
E	0
F	4
G	3

Now if we want to ~~go~~ go from vertex A to vertex G

A → B → E → G => 0 + 1 + 0 + 3 => 4
A → D → C → E → G => 0 + 5 + 3 + 0 + 3 => 11
A → C → E → G => 0 + 3 + 0 + 3 => 6
A → C → B → E → G => 0 + 3 + 1 + 0 + 3 => 7
A → D → F → G => 0 + 5 + 4 + 3 => 12
A → D → C → B → E → G => 0 + 5 + 3 + 1 + 0 + 3 => 12

We get A to G => 3

i.e A → D → C → B → E → G
5 + (-2) + (-2) + (-1) + 3 = 3

Shortest Path
A → E => 0
A → D → C → B
↓
E
5 + (-2) + (-1) + (-1) = 0

Time complexity

↳ What it is doing → Relaxing all the Edges
!!, i.e E

$$O(E(V-1)) \simeq O(E \times V)$$

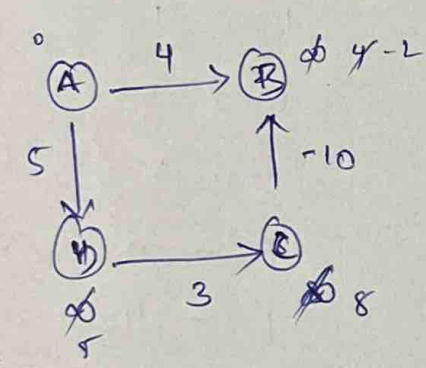
if $E = V = n \Rightarrow O(n^2)$
↳ min

→ for complete graph.

↳ no. of Edges $\Rightarrow n(n-1)/2$

$$n\left(\frac{n(n-1)}{2}\right) \Rightarrow O(n^3)$$
↳ max

Drawback of Bellmanford



Vertices

4 - ~~vertic~~

↳ (4-1) = 3 ⇒ Relaxation

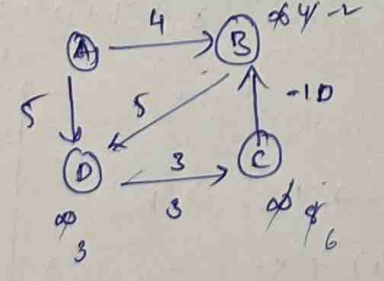
Edges :- (3,2) (4,3) (1,4) (1,2)

1st Relaxation :- (A,D) (A,D) (A,B) (D,C)

	A	B	C	D
	0	∞	∞	∞
1st	0	4	8	5
2nd Relaxation	0	-2	8	5
		(5-10)		
3rd	0	-2	8	5

vertex	distance
1	0
2	-2
3	8
4	5

→ Now I will add one more edge (B to D)



Edges (A,D) (C,B), (A,D) (D,C) (B,D)

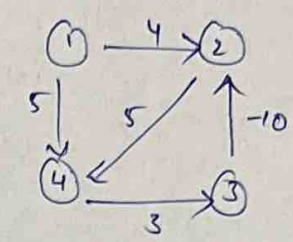
	A	B	C	D
1st	0	4	8	5
2nd	0	-2	8	3
3rd	0	-2	6	3
	0	-4	4	1

Ideally it should stop here

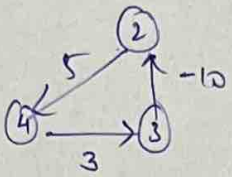
→ So, ideally when we perform (n-1) Relaxation we should get a Best Possible results

↳ But in 4th iteration if we observe it again relaxing. → i.e wrong → why is this Happening

↳ If we observe the graph, it is forming a negative cycle graph



i.e



⇒ 5 + 3 - 10 ⇒ -2

↑ weight of cycle is negative

↳ In this case relaxation would be Infinite.

↳ So Bellmanford Algorithm won't work if in case there is a negative cycle in graph.

→ Uses of Bellman ford

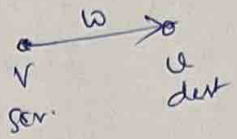
* Identifying negative weight cycle → perform (n-1) Relaxation & perform one more time, if relaxed it's negative cycle
* Identifying the most efficient currency conversion method

→ Applications

* Examining a graph for the presence of (-ve) weight cycle
* Using (-ve) edge weights, find the shortest path in a graph
* Routing is a concept used in data N/w

Bellman Ford Algorithm

```
// Represents connected, directed & weighted graph
Class Graph {
    // A class represents the weighted edge in graph
    class Edge {
        int src, dest, weight;
        Edge() {
            src = dest = weight = 0
        }
    }
    int V, E
    Edge edge[];
// Create a graph with V vertices & E edges
    Graph int V, int e) {
        V = v ; E = e
        edge = new Edge[e]
        for (int i=0 ; i<e ; i++) {
            edge[i] = new Edge();
        }
// Main Fn, that finds the shortest distances from "src" to all other
   Vertices using Bellman-Ford Algorithm. This Fn also detects Every
   cycle. Weighted cycle
void BellmanFord (Graph graph, int src) {
    int V = graph.V ;
    int E = graph.E ;
    int dist[] = new int[V] ;
// Step 1: Initialize distances from src to all other vertices as ∞
    for (int i = 0 ; i < V ; i++) {
        dist[i] = Integer.Max_Value;
        dist[src] = 0;
// Step 2: Relax all Edges |V|-1 times. A simple shortest path
   from src to any other vertex can have at most V-1
   Edges.
```

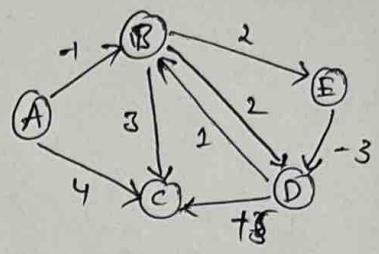


```
// Step 2 -> Relaxing Edges
for ( int i = 1 ;  i < V ; i++ ) {
    for( int j = 0 ; j < E ; j++ ) {
        int u = graph.Edge[j].src ;
        int v = graph.Edge[j].dest ;
        int weight = graph.Edge(j].weight ;
Relaxation {  if ( dist[u] != Integer.max_value && (dist[u] + weight
                                                    < dist[v]) {
                dist[v] = dist[u] + weight
        }
    }
}
```

// Step 3 : Check for -ve weight cycles. The above step guarantees shortest distances if graph doesn't contain (-ve) weight cycle. If we get a shorter path in this step means ~~then~~ there is a (-ve) cycle.

```
for (int j = 0 ; j < E ; j++ ) {
    int u = graph.Edge[j].src ;
    int v = graph.Edge(j].dest ;
    int weight = graph.Edge(j].weight ;
    if (dist[u] != Integer.max_value && dist[u]+ weight <
                                                dist[v]) {
        System.out.println("Graph contains negative cycle");
    }
}
```

```
Public static void main (String[] args)
    int V = 5;  // no. of vertices in graph
    int E = 8;  // no. of Edges in graph

    Graph graph = new Graph(V, E);
```




```
// Add Edge 0-1 (i.e A-B)
    graph.edge[0].src = 0
    graph.Edge[0].dest = 1
    graph.Edge[0].weight = -1
```

0	1	2	3	4
A	B	C	D	E

```
// Add edge (A -> C) i.e 0-2
    graph.Edge[1].src = 0
    graph.Edge[1].dest = 2
    graph.Edge[1].weight = 4

// Add Edge B-C (i.e) 1-2
    graph.Edge[2].src = 1
    graph.Edge[2].dest = 2
    graph.Edge[2].weight = 3
```

S/ly for other edges