Graph

(1) Basic Theory

(2) Own Implemented Graph Nodes & BFS / DFS traversal.

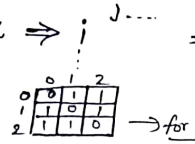
Basics Theory of Graph

Graph (Data Structure)

⇒ For basic defn. and terms refer to Coding Blocks booklet of Java.
(It's very enough to understand all the terms)

⇒ Terms like :- ① Vertex ② Edge (undirected edge / directed edge) ③ Degree (No. of vertex) ④ Adjacency ⑤ Path

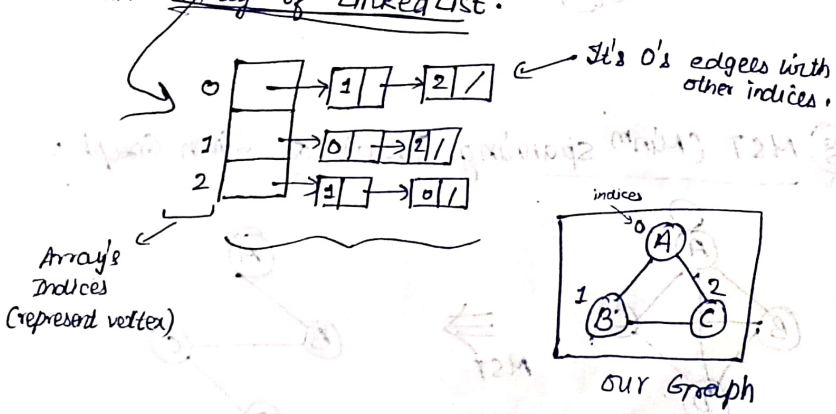
⇒ Adjacency Matrix ⇒ 2D Matrix



⇒ tells if there is edge b/w i or j or not. 0 → No, 1 → Yes.

→ for graph below → It's always symmetric for undirected Graphs.

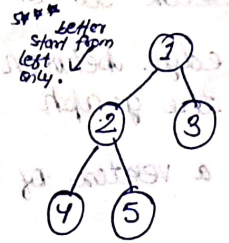
⇒ Adjacency List ⇒ It will be used like same above thing to show connection b/w vertex. But here it does not have to store things like '0' → No.
It will show exact adjacent edges of that vertex.
It is an array of LinkedList.



Graph Traversals :-

① Depth First Search → start from starting vertex and explore as fare as possible along each branch before backtracking.

e.g.

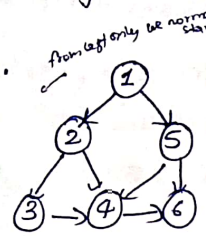


⇒ DFS traversal

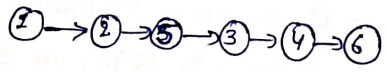
1 → 2 → 4 → 5 → 3

② Breadth First Search → we try to work closer to the starting vertex. explore all neighbour vertices before moving to the next level neighbours.

e.g.



bfs traversal



Starts

It's like level order traversal in tree

Implementation of Graph :-

Basic operations

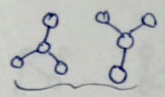
following are the primary operations of a Graph-

- **Add Vertex** → Adds a vertex to the graph
- **Add Edge** → Adds an edge between the two vertices of the graph
- **Display vertex** → Displays a vertex of the graph.

Miscellaneous :-

→ A connected acylic graph is called a tree.

→ A disconnected acyclic graph is called a forest. In other words, a disjoint collection of trees is called a forest e.g.



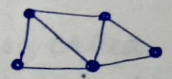
Together called as forest. since disconnected and acyclic.

→ Spanning trees (you know already)

⇒ **Circuit rank** :- Let "G_1" be a connected graph with "n" vertices and "m" edges. A spanning tree "T" of G_1 contains $(n-1)$ edges.

Therefore, the number of edges you need to delete from "G_1" in order to get a spanning tree = $m-(n-1)$, which is called the circuit rank of G_1.

e.g.



⇒ $m=7$ edges

⇒ $n=5$ ~~edges~~. vertex

Then the circuit rank is

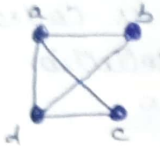
$G_1 = m-(n-1)$

$= 7-(5-1)$

$= 3$ → means we've to remove three edges to make our Graph as MST.

Kirchoff's Theorem :- It is useful in finding the number of spanning trees that can be formed from a connected graph.

e.g.



The matrix 'A' be filled as, if there is an edge between two vertices, then it should be given as '1', else '0'.

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left|\begin{array}{cccc} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{array}\right| \end{array} = \left|\begin{array}{cccc} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{array}\right|$$

By using Kirchoff's theorem, it should be changed as replacing the principle diagonal values with the degree of vertices and all other elements with (-1 . A)

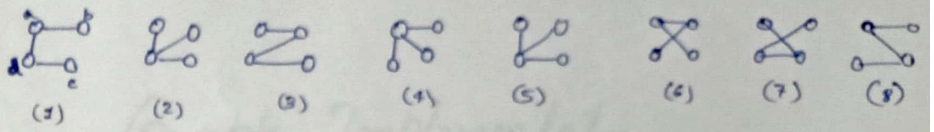
degree of 'a' is 3 as 3 edges connected with it.

$$= \left|\begin{array}{cccc} 3 & -1 & -1 & -1 \\ -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ -1 & -1 & -1 & 3 \end{array}\right| = M$$

Now remains -1.

cofactor of $m_1 = \left|\begin{array}{ccc} 2 & 0 & -1 \\ 0 & 2 & -1 \\ -1 & -1 & 3 \end{array}\right| = 8$

Thus, the number of spanning trees = 8.

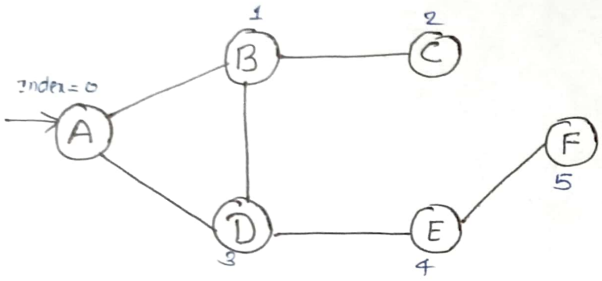


Cut edge or cut vertex → removing that edge or vertex results in a disconnected graph.

Graph theory — coloring → Graph coloring is nothing but a simple way of labelling graph components such as vertices, edges, and regions under some constraints. In a graph no two adjacent vertices, adjacent edges, or adjacent regions are colored with minimum number of colors.
This number is called the chromatic number and the graph is called a properly colored graph.

Graph Implementation In JAVA

source → 4t video :- Breadth First Search (BFS) on Graph with implementation in Java
channel → Coding Simplified.



You can construct graph own way also
If you understand clearly

⑤ In the same BFSApp.java (All class in one file only)

It's like node only with data = (label)

```
//Let's created vertex (like Node in LinkedList)
public class Vertex {

    public char label;
    public boolean wasVisited;

    public Vertex(char lab) {
        label = lab;
        wasVisited = false;
    }
}
```

false by default

G.K :-
If writing in same file then no need to mention public since. because it gives compile time error saying ("class vertex is public, should be declared in a file named vertex.java") so.

class vertex in Graph similar similar to Node in free

#In Main.java :-

For our Graph to be like that

(main)

```
public class BFSApp {

    p s v M {
        //Let's build the Graph structure with given details
        Graph theGraph = new Graph();
        //add the vertex
        theGraph.addVertex('A');   A=0
        theGraph.addVertex('B');   B=1
        " "      ('C');            C=2
        " "      ('D');
        " "      ('E');
        " "      ('F');
        //Let's add the edges
        theGraph.addEdge(0,1);
        theGraph.addEdge(1,2);
        " "      (0,3);
        " "      (3,4);
        " "      (4,5);
        " "      (1,3);
    }
}
```

and so on

Graph build here.

Instead of this now we can build Graph matrix directly as only we'll use remove use labels

Graph class (our implemented Graph class in BfsApp.java file only).

```
public class Graph {

    private final int MAX_VERTS = 20;
    private Vertex vertexList[];
    private int adjMat[][];
    private int nVerts;                 // no. of vertex
    private Queue<Integer> q;           // → To be used during bfs only
    private Stack<Integer> s;           // → To be used for dfs

    public Graph() {
        vertexList = new Vertex[MAX_VERTS];     // Make array (ref) initialised
        adjMat = new int[MAX_VERTS][MAX_VERTS]; // Make 2D array initialised automatically
        nVerts = 0;

        q = new LinkedList<Integer>();  // Queue in java is interface implemented by LinkedList
        s = new Stack<Integer>();
    }

    // Basic operations: addVertex, addEdge, displayVertex.

    public void addVertex(char lab) {
        vertexList[nVerts++] = new Vertex(lab);
    }

    public void addEdge(int start, int end) {
        adjMat[start][end] = 1;
        adjMat[end][start] = 1;
    }

    public void displayVertex(int v){
        System.out.print(vertexList[v].label);
    }
```

Store // Array of Vertex; without using ArrayList // you could have written it also as Vertex[] vertexList; & int[][] adjMat;

→ It's for non directed graph. So A-B !!! B-A so-

⇒ Adjacency Matrix used to denote Graph edges

```
    // let's build this get adjacent unvisited edge in separate fxn. used by bfs
    public int getAdjUnvisitedVertex(int v) {
        for(int j=0; j<nVerts; j++) {
            if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
                return j;
        }
        return -1;
    }
```

If there's any edge from vertex 'v' to some vertex 'j' and was not visited then return 'j' vertex.

This will be used during traversal.

Breadth First Search Implementation in our Graph

in Graph class only do follow method → (bfs)

```
public void bfs() {
    // let's begin from the first node as root of the graph.
    vertexList[0].wasVisited = true; // Mark it as visited
    displayVertex(0); // print the vertex
    q.add(0); → root node say 0th index // Add in the queue.
    int v2; // used to store adjacent vertex of given vertex.

    while(!q.isEmpty()) {
        int v1 = q.remove();
        while((v2 = getAdjUnvisitedVertex(v1)) != -1) {
            vertexList[v2].wasVisited = true;
            displayVertex(v2);
            q.add(v2);
        }
    }

}
```

→ In some cases you can get here first vertex

→ Repeatedly & then condition checked

→ This while loop works until the value of v2 is not equal to (-1).

O(V+E)

→ It's easy straightforward code

⇒ In Main / BalBFSApp (..)?
Add following code to get graph's BFS result.

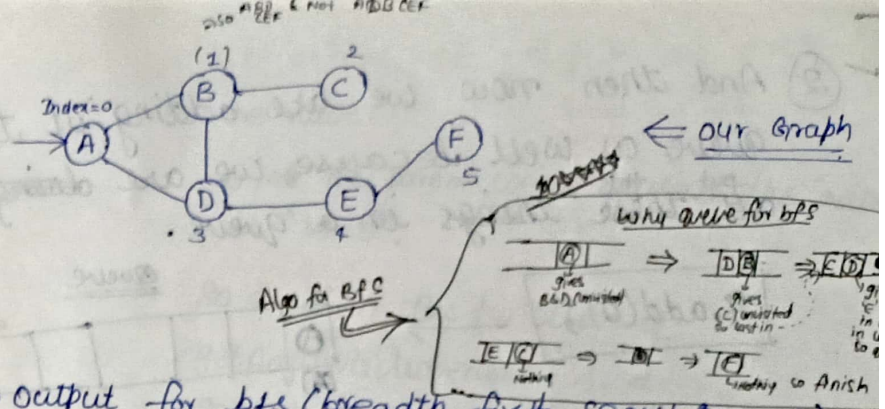
```
System.out.println("Visits : ");
theGraph.bfs();
System.out.println();
```

→ This single line only

→ You can remove this if's just not required.

we don't want to write lengthy graph code.

See different ways & solve bfs approach in competitive programming



← our Graph

⇒ Output for bfs (breadth first search traversal)

ABDCEF.

Now let's see how our bfs method working

① First we are starting from (A) (Index=0)
so let's make it visited graph node.
so let's make the index 0, visited as true.
and display that vertext.
so code as:-

```
vertexList[0].wasVisited = true;
```

→ first make it's property true

and then display it

```
displayVertex(0);
```

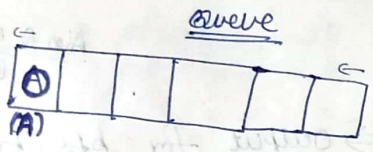
→ so calls display vertex method, which simply prints the label of the vertex. (char[0].label).

⇒ so for now our output as:-

```
Visits:
A
```

② And then now we are adding it to queue as well because we are ~~doing~~ initially ~~add these things~~ put this in a queue.

```
q.add(0);
```



{int v2;}

③ Now we are checking for until queue is not empty.

```
while (!q.isEmpty()) {
```

queue

① first we will remove the item

```
int v1 = q.remove();
```

→ v1=0 → removed item

So it has removed the item which is our case for now is (v1=0) zero; which is the first value which is 'A'. (0 is Mapped with 'A')

② After this we are calling a function 'getAdjUnvisitedVertex'.

```
while ((v2 = getAdjUnvisitedVertex(v1)) != -1) {
```

so this function what will do is it will check all neighbours of A(0) which are not visited. If it has some value then return me the value.

So int it's param we've passed the index as 'v1=0'.

So now in the function of getAdjUnvisitedVertex(int v); it So code as :- In this fn

```
for(int j=0; j<nVerts; j++) {
    if (adjMat[v][j]==1 && vertexList[j].wasVisited==false)
        return j;
}
return -1;
```

So it will check if it has same edges and if it has wasVisited as false. It means return j.

means it is not visited so we can use it.

So in above code :-

for j=0 → Nothing

for j=1 → Yes because 'B' not visited and edge==1.

So now it will return 'B'.

So it will be our index now as wasVisited true and display it. code as :-

```
vertexList[v1].wasVisited = true;
displayVertex(v2);
```

so we display the vertex so our output now as

```
Visits:
AB
```

and then add it do the queue as well as `q.add(v2);`

our queue

again our while loop of

```
while ((v2 = getAdj(v1)) != -1)
```

↓

So in this fn: now it goes for

for j=0, Nod edge

j=1, edge but visited = true

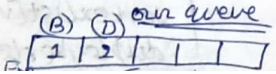
j=2, Edge & visited = false

so it will return 'D' which is index (2).

so now again do it's visited true and display it.

```
vertexList[v2].wasVisited = true;
displayVertex(v2);
```

and then add it do the queue as `q.add(v2);`



so again it will do while ((v2 = getAdj...)

↓

And for i=0 to i=nVerts.

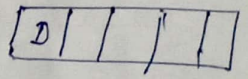
But now since no edge of 'A' anymore so it returns -1.

so whenever it returns -1, we are completed and done for the index.

Now again in our parent while loop it checks if q is empty then.

```
while (!q.isEmpty()) {
    int v1 = q.remove();
}
```

our queue



→ And remove 'B' = v1 from queue.

so now with v1 = 'B' (1) our while(v2 = get...)

↓

loop starts

and so on it will proceed on.

so this is the way and at end we've printed all the elements.

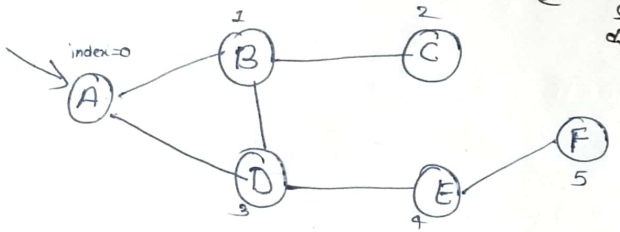
so our output = `Visits: ABDCEF`

⇒ Complexity of BFS $O(\text{vertex} + \text{Edge}) = O(V+E)$

Now comes depth first search

First we will go in depth of an element. and once we cover all its depth then we cover next neighbour.

e.g. graph :-



⇐ This numbering should be in dfs type. so it will go like dfs in case of dfs & bfs in case of bfs.

DFS output :-

A → B → E → F → B → E

or

A → B → C → D → E → F

0 1 2 3 4 5

There since in sequence (getAdjUnvisited) is checking in sequence so.

⇒ In depth first search we take use of stack.

code for method dfs in Graph class only.

```
public void dfs() {
    //Let's start with initial vertex as root vertex.
    VertexList[0].wasVisited = true;
    displayVertex(0);
    s.push(0);  → root node say 0th index.
    while(!s.isEmpty()) {
        int v = getAdjUnvisitedVertex(s.peek());
        if(v == -1) {
            s.pop();
        } else {
            VertexList[v].wasVisited = true;
            displayVertex(v);
            s.push(v);
        }
    }
}
```

→ In starting we will just start with 0th index vertex, print it and then push it to stack.

same as stack.top

very straight forward code

complexity :- O(V+E)

Now after the first base case & 0th index, printing & then pushed to check.

⇒ Our next code will be to check if stack is empty. Or to work until our stack is empty. So it comes in a while loop.

```
while (!s.isEmpty()) {
}
```

→ since it's not empty

Printed: A (printed value)

so it will check for

```
int v = getAdjUnvisitedVertex(s.peek());
```

A(0)

So now it will go in above unvisit.. fn. and return some int value:- so iteration in f'n:-

$j = 0$; $[v][j] = -1$ but $(v)(0)$ is visited (true)

so $j = 1$; $[v][j] = 1$ (True) and wasVisited = false

i.e. 'B' (return j) $(1=1)$ $(j=B)$

so $\boxed{v = B(1)}$ → Make note that 'v' is int. we are just using 'B' as it's label & that index in Graph.

So now next step:-

```
if (v == -1) {
}
```

since $\boxed{v=1}$ moves to else part.

```
else {
    vertexList(v).wasVisited = true;   B (was visited true)
    displayVertex(v);
    s.push();
```

→ Printed value overall: A B

→ stack overall.

So our first Iteration finished & while(not empty)

⇒ Now 2nd Iteration :-

```
while (!s.isEmpty()) {
    int v = getAdjUnvisitedVertex(s.peek());
```

B(1)

The function getAdjUnvisitedVertex(1) is called.

$j = 0$ (A) ⇒ wasVisited (true) (A)

$j = 1$ (B) ⇒ wasVisited (true) (B)

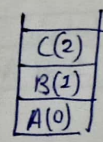
$j = 2$ (C) ⇒ wasVisited (false) and $[v][j] == 1$ so return j $\boxed{j=2}$ $\boxed{2 \to C}$

Now next :- ∴ $\boxed{v = 2\ (C)}$

again if (v == -1) { skip and go to else part.

```
else {
    vertexList[v].wasVisited = true;
    displayVertex
    s.push(v);
```

→ Printed list: A B C



our stack

Now 2nd Iteration completed.

⇒ Now 3rd iteration :-


```
while (!s.isEmpty()) {
    int v = getAdjUnvisitedVertex(s.peek());
                                  v = 3 (C)
```

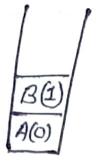
This fn call :-

j=0 ; [v][i] ≠ 1
j=1 ; [v][j] = 1 but wasVisited (true)
j=2 ; [v][j] ≠ 1
j=3 ; [v][j] ≠ 1
j=4 ; [v][j] ≠ 1
j=5 ; [v][j] ≠ 1

So it has no adjacent unvisited vertex
so 'returns -1'
so v = -1

⇒ Now

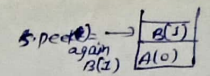
```
if (v==-1) {        true
    s.pop()
}
```



C(2) is now popped out.

else part won't be executed.

⇒ Now 4th Iteration :-




```
while (!s.isEmpty()) {
    int v = getAdjUnvisitedVertex(s.peek())
                                  B(1).
```

This fn call :-

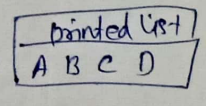
(A) j=0 ; [v][j]=1 ; But wasVisited (true)
(B) j=1 ; wasVisited=(true) [v][j]=1 ✓ (course)
(C) j=2 ; wasVisited=(true) [v][j]=1 ✓ (course)
(D) j=3 ; [v][j]==1 && wasVisited (false)

So 'return j' j=4 4→D

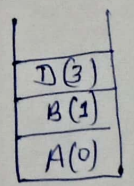
∴ V=4(D)

Next :- if (v==-1) { skip.

else {
    vertexList[v].wasVisited = true;
    displayVertex(v);
    s.push();
}



stack



and so on it iterates and results = ABCDEF with at end stack will be fully empty and it comes out of while loop.

Graph coding and Implementation stuff Based on PSS.

(#1) Very basic Graph Implemented you will find it in Graph-2 notebook.

Array Matrix to store edges & Node to store Nod

⇒ In Competitive programming mostly tree is done using Arrays or ArrayList concept of storing nodes (vertex) and edges. There can be weighted edges also.

Using 2D matrix we try to build our approach.

⇒ So normally in competitive programmi You will be given N*N 2-D symmetric matrices & edges. You have to get yourself the vertex array, wasVisited array, VertexLable array and then implement BFS and DFS or do manipulation.

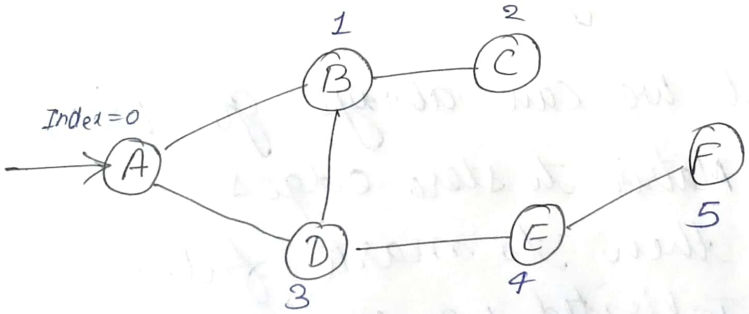
But we can always go with 2D Matrix to store edges and then to mark false as Is Visited we can use 2D matrix kind of thing.

2 We know how to use bfs and dfs for given vertex and edges. vertex as an class earlier.

⇒ But now we will see how to implement that same Graph in Crux-2 book without using nodes vertex class.

⇒ We will do it by using only arrays.

our Graph was



⇒ code

```
public class Main {

    static char[] vertexLabel = {'A','B','C','D','E','F'};
    static int[] vertexCount = {0,1,2,3,4,5};
    static boolean[] wasVisited = {false, false, false, false, false, false};

    static int[][] edges = {{0,1,0,1,0,0},
                            {0,0,1,1,0,0},
                            {0,1,0,0,0,0},
                            {1,1,0,0,1,0},
                            {0,0,0,1,0,1},
                            {0,0,0,0,1,0}};

    static Queue<Integer> queue = new LinkedList<>();
    static Stack<Integer> stack = new Stack<>();
```

```
static void bfs() {
    wasVisited[0] = true;          // or start from 1st vertex in index
    System.out.println(vertexLabel[0]);   // Track the '0' first found in 2D matrix
    queue.add(vertexCount[0]);     // Print it
    int nextVertex;                // Add in queue
                                   // make record of it's '1'
    while (!queue.isEmpty()) {     // Now until queue is not empty
        int currVertex = queue.remove();   // Remove that element in queue & track it's adjacent.
        while ((nextVertex = getAdjacentVertex(currVertex)) != -1) {   // find all it's adjacent.
            wasVisited[nextVertex] = true;   // set all visited as true.
            System.out.println(vertexLabel[nextVertex]);   // Print all it's adjacent
            queue.add(nextVertex);         // Put all it's adjacent in queue.
        }
    }
}
```

```
static int getAdjacentVertex(int currVertex) {
    for(int j=0; j<vertexCount.length; j++) {
        if(edges[currVertex][j]==1 && wasVisited[j]==false)
            return j;
    }
    return -1;
}

public static void main(String[] args) {
    bfs();
}
}
```

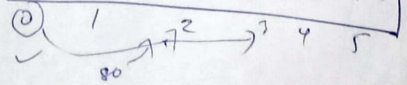
Output = As expected → A B D C E F

(5) Similarly for dfs as :-

```
static void dfs() {
    wasVisited[0] = true;
    System.out.println(vertexLabel[0]);
    stack.push(vertexCount[0]);

    while(!stack.isEmpty()) {
        int nextVertex = getAdjacentVertex(stack.peek());

        if(nextVertex == -1) {
            stack.pop();
        }
        else {
            wasVisited[nextVertex] = true;
            System.out.println(vertexLabel[nextVertex]);
            stack.push(vertexCount[nextVertex]);
        }
    }
}
```

Output = as expected

A B C D E F



You can/should be able to write its notes by first thinking the code in mind and then write it.

#③ → 2(B)

⇒ We could also have put edges matrix as input and do our bfs in it.

like:-

```
p s v m ( ) {
    Scanner sc = new Scanner (System.in);
    int testCase = sc.nextInt();
    for (int t=0; t<testCase; t++) {
        int len = sc.nextInt();
        edges = new int[len][len];
        for (int i=0; i<len; i++) {
            for (int j=0; j<len; j++) {
                edges[i][j] = sc.nextInt();
            }
        }
        bfs();
    }
}
```

⇒ We just have to declare edge 2D matrix in as static in Global as:-

```
public class Main {
    static edges[][];
    // rest All wasVisited etc remains as usual of 6x6 edges
}
```

⇐ ps vm

So input ⇒

```
1
6
0 1 0 1 0 0 1 0 1 1 0 0 0 1 0 0 0 0 1 1 0 0 . .
```

Same our matrix of abcdef.

output = A B D C E F

But this was hardcoded of 6×6.

⇒ Now what if we want for any N×N. So lets check for that also.

#④ So now we can generate all vertexCount, wasVisited, vertexLabel, etc our self only for given edges 2-D matrix and can do bfs and dfs traversal in it code as :-

⇒ code :-

```
public class Main {

    static char[] vertexLabel;
    static int[] vertexCount;
    static boolean[] wasVisited;
    static int[][] edges;
    static Queue<Integer> queue = new LinkedList<>();
    static Stack<Integer> stack = new Stack<>();

    static void bfs() {...} //same as code as in bejiy #1

    static void dfs() {...} // same code

    static ~~void~~ int getAdjacentVertex(int currentVertex) {...} Next
```

```
p s v m {

    Scanner sc = new Scanner(System.in);
    int testCase = sc.nextInt();
    for(int t=0; t<testCase; t++){
        int len = sc.nextInt();
        edges = new int[len][len];
        for(int i=0; i<len; i++){
            for(int j=0; j<len; j++){
                edges[i][j] = sc.nextInt();
            }
        }
        vertexCount = new int[len];
        wasVisited = new boolean[len];
        vertexLabel = new char[len];
        Arrays.fill(wasVisited, false);
        for(int i=0; i<len; i++){
            vertexCount[i] = i;
            vertexLabel[i] = (char)(i+65);
        }
        bfs();
    }
}
}
```

⇒ Input :-

```
1
6
0 1 0 1 0 0 1 0 1 1 0 0 0 1 0 0 0 0 1 1 - - -
```

output

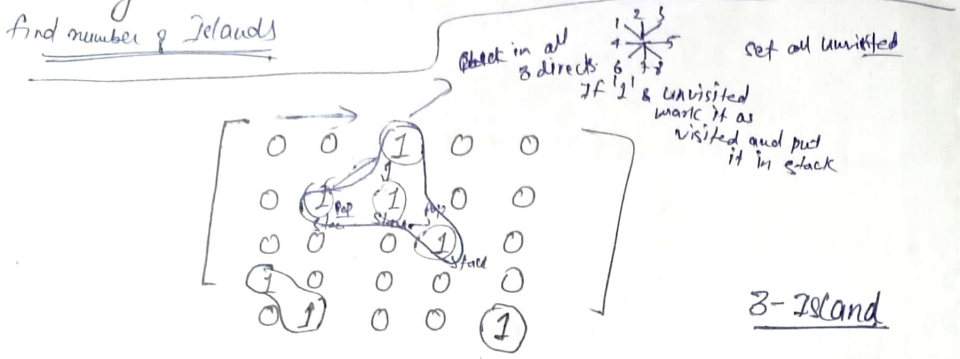
```
A B D E F
```

As expected

Question :- Given a binary 2D Matrix, find the number of Islands.

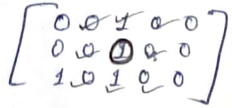
How to solve island type question by mush :-

Find number of Islands



Make another boolean 2D Matrix. Initially all false

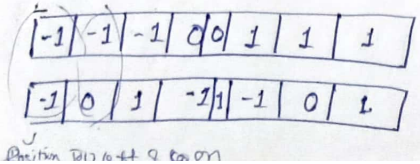
and 2D Array as 8 points surrounding that point to check all its adjacent position as :-

e.g. 

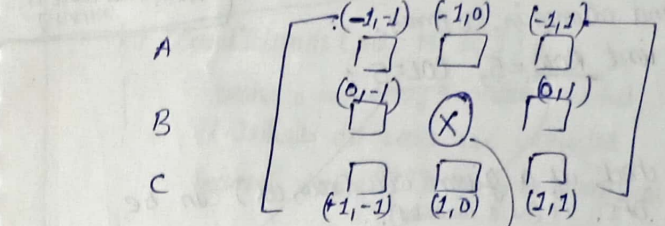
If value=0 then set True only.

If value=1 first put in stack and then mark it as True for visited.

So to track this maintain it's position & adjacent of that point in 2D as :- or 2 Array as.



Position Pair of i & j



So check for all it's adjacent and put in stack.

Remember they using BFS only.

It's always sure that you will ~~give~~ get ans in BFS but in DFS sometimes it may skip the answer.

Code in next page :-

It's not actually overall apply DFS or BFS but just to every adjacent in 8 direction if 1, then make the visited as true.

We use stack thus DFS we use queue thus BFS.

And everytime while traversing in 2D array if we found new '1' which is unvisited again the number of Island count increases.

It just finding number of disconnected graphs

#

Time: O(Row x Col) :- Using DFS() you can also check BFS() :-

OK Now After writing code and debugging understanding everything How this code is running

```
//No. of rows and columns in 2D Array.
static final int ROW = 5, COL = 5;

//A function to check if a given cell (row, col) can be
//included in DFS. i.e (=1 & unvisited).

boolean isSafe(int M[][], int row, int col, boolean visited[][]){
    //row number is in range, column number is in range
    // and value is 1 and not yet visited.
    return (row >= 0) && (row < ROW) &&
           (col >= 0) && (col < COL) &&
           (M[row][col] == 1 && !visited[row][col]);
}

//A utility function to do DFS for a 2D boolean matrix
//It only considers 8 neighbours as adjacent vertices
void DFS(int M[][], int row, int col, boolean visited[][]){
    //These arrays are used to get row and column numbers
    //of 8 neighbours of a given cell.
    int rowNbr[] = new int[]{-1, -1, -1, 0, 0, 1, 1, 1};
    int colNbr[] = new int[]{-1, 0, 1, -1, 1, -1, 0, 1};

    //Mark this cell as visited
    visited[row][col] = true;
    //Recur for all connected neighbours
    for(int k=0; k<8; ++k){
        if(isSafe(M, row+rowNbr[k], col+colNbr[k], visited)){
            DFS(M, row+rowNbr[k], col+colNbr[k], visited); //<-Recursive
        }
    }
}
```

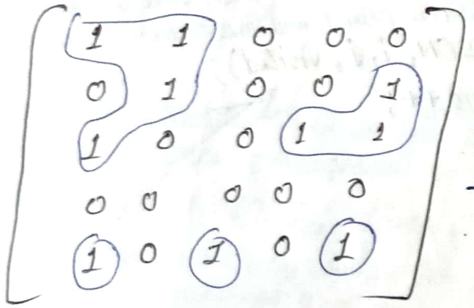
```
//This main function that returns count of islands in a given
//boolean 2D matrix

int countIslands(int M[][]){
    //Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    boolean visited[][] = new boolean[ROW][COL]; (By Default all false)

    //Initialize count as 0 and traverse through the all cells
    //of given matrix
    int count = 0;
    for(int i=0; i<ROW; ++i){
        for(int j=0; j<COL; ++j){
            if(M[i][j]==1 && !visited[i][j]){
                //If a cell with value 1 is not visited yet,
                //then new island found, visit all cells
                //in this island and increment island count.
                DFS(M, i, j, visited);
                count++;
            }
        }
    }

    return count;
}
```

Can we've now recursive solution for this.

Driver code :-

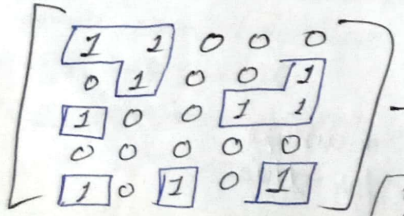
```
int M[][] = new int[][] {{1,1,0,0,0},
                         {0,1,0,0,1},
                         {1,0,0,1,1},
                         {0,0,0,0,0},
                         {1,0,1,0,1}};

System.out.println(countIslands(M));
```

↳ output = 5



→ 5 islands (Disconnected graphs)

if we can traverse in 4 direction (up, left, down, right) then it will have 6 islands. e.g.



→ six islands.

For this just change

```
rowNbr = {-1, 0, 0, 1}
colNbr = {0, -1, 1, 0}
```

→ after 4 element

understood

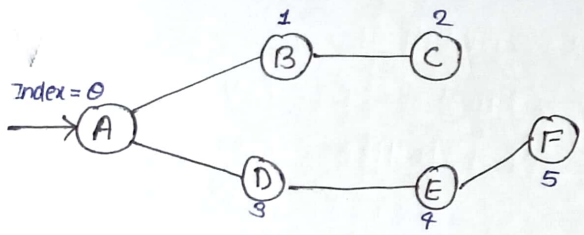
Graph Implementation In Java

(Just put all in same place)

#12 Graph Implementation In Java :-

(Used Adjacency Matrix to store vertex's edges and not array & LinkedList)

for DFS very better understanding by Iteration see in Crux 2 Notebook.



This index numbering is also very imp - you may see in dfs crux - 2 Notebook explained there.

Properties of vertex. Like Node in tree.

① In Vertex.java

```
public class Vertex {
    public char label;
    public boolean wasVisited;

    public Vertex (char lab) {
        label = lab;
        wasVisited = false;
    }
}
```

Graph is afterall an Array representation only used.

2D for ~~ve~~ edges

& 1D array for vertex. (vertex Array type)

② In Main.java :-

```
public class Main {

    P S V M {

        Graph theGraph = new Graph();

        theGraph.addVertex ('A');
        theGraph.addVertex ('B');
        theGraph.addVertex ('C');
        theGraph.addVertex ('D');
        theGraph.addVertex ('E');
        theGraph.addVertex ('F');

        theGraph.addEdge (0, 1);
        theGraph.addEdge (1, 2);
        theGraph.addEdge (0, 3);
        theGraph.addEdge (3, 4);
        theGraph.addEdge (4, 5);
        theGraph.addEdge (1, 3);

        Syso ("BFS traversal");
        theGraph.bfs ();

        Syso ("DFS traversal");
        theGraph.dfs ();
    }
}
```

In Graph.java :-

```
public class Graph {
    private final int MAX_VERTS = 20;
    private Vertex[] vertexList;
    private int adjMat[][];
    private int nVerts;
    private Queue<Integer> q;
    private Stack<Integer> s;

    public Graph() {
        vertexList = new Vertex[MAX_VERTS];
        adjMat = new int[MAX_VERTS][MAX_VERTS];
        nVerts = 0;
        q = new LinkedList<Integer>();
        s = new Stack<Integer>();
    }
```

```
//Basic operations of Graph: ①addVertex ②addEdge ③display

public void addVertex(char lab) {
        vertexList[nVerts++] = new Vertex(lab);
}

public void addEdge(int start, int end) {
        adjMat[start][end] = 1;
        adjMat[end][start] = 1;
}

public void displayVertex(int v) {
        Syso(vertexList[v].label);
}

public int getAdjUnvisitedVertex(int v) {
    for(int j=0; j<nVerts; j++) {
        if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;
        }
    }
    return -1;
}
```

```
//BFS traversal of Graph

public void bfs() {
    vertexList[0].wasVisited = true;
    displayVertex(0);
    q.add(0);
    int v2;

    while(!q.isEmpty()) {
        int v1 = q.remove();

        while((v2 = getAdjUnvisitedVertex(v1)) != -1) {
            vertexList[v2].wasVisited = true;
            displayVertex(v2);
            q.add(v2);
        }
    }
}
```

```
//DFS traversal of Graph (for more better understanding see (num-) notebook with Prof Mam.

public void dfs() {
    vertexList[0].wasVisited = true;
    displayVertex(0);
    s.push(0);

    while(!s.isEmpty()){
        int v = getAdjUnvisitedVertex(s.peek());

        if(v==-1) {
            s.pop();
        }
        else {
            vertexList[v].wasVisited = true;
            displayVertex(v);
            s.push(v);
        }
    }
}
```

13K → EMI
8K → Bikes
5K → Ration
3K → Night
3K → Bijlee
5K → Cash
4K -

This is only Notes, Codes not totally written, You need to understand in Youtube & then write code in your own way.

(A) Printed Notes.

code with output will be helpful

Graph Data Structure

1. Graph Data Structure Intro
2. Adjacency Matrix
3. Adjacency List
4. Depth First Search (Add code page in one)
5. Breadth First Search (Add code page in one)
6. Bellman Ford Algo (Dijkstra's Algo)
7. Minimum spanning Tree
8. Prim's Algo
9. Kruskal's Algo
10. Strongly connected Graph.