# Graphs level 1

## Data Structures

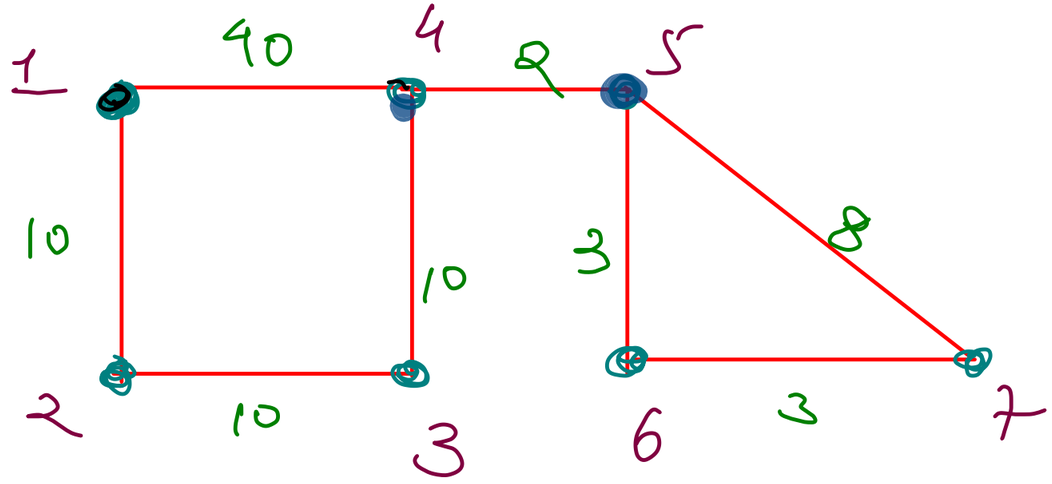
- → Arrays & Strings
- → Array list
- → Stack
- → Queue
- → Linked list
- → Generic Tree
- → Binary Tree
- → Binary Search Tree
- → Hashmap
- → Heap

# Graphy

- → graph algorithms
- → graph application

# Terminologies

1. Node / vertices
2. Edges
3. Undirected / directed
   - undirected → bidirectional
   - directed → unidirectional
4. weighted / unweighted
5. Incoming Edge / Outgoing Edge

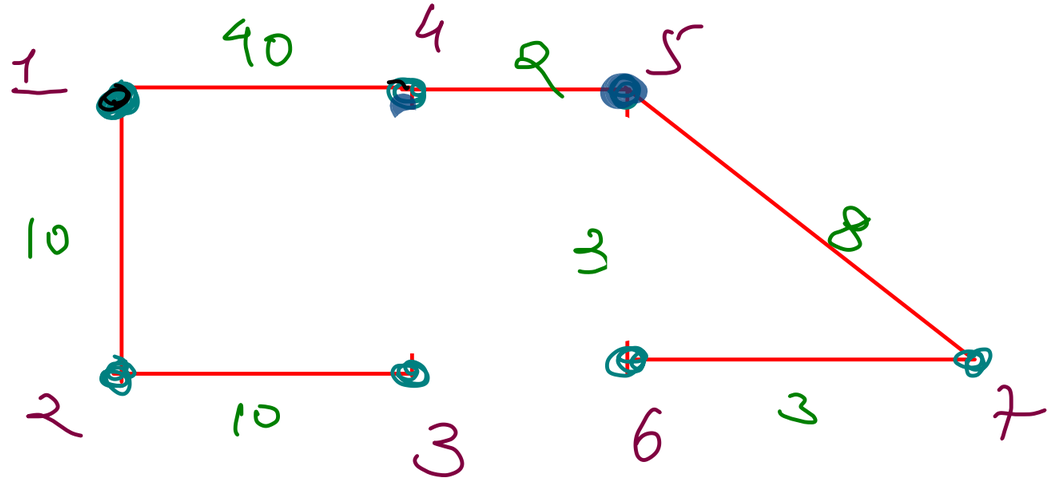



# Applications

1. Google maps → Shortest distance / Mincost to connect all dist
2. Social Media
3. Airline Management

# Terminologies

1. Node / vertices
2. Edges
3. Undirected / directed
   - undirected → bidirectional
   - directed → unidirectional
4. weighted / unweighted
5. Incoming Edge / Outgoing Edge



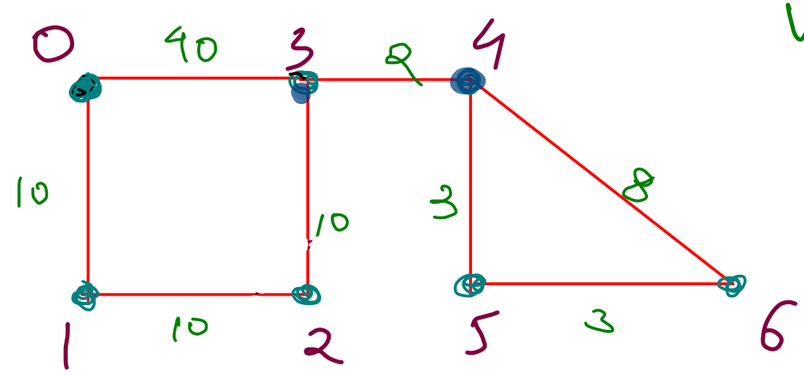

# Applications

1. Google maps → Shortest distance / Mincost to connect all dist
2. Social Media
3. Airline Management

Implementation

## ② Adjacency matrix

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | -1 | 10 | -1 | 40 | -1 | -1 | -1 |
| 1 | 10 | -1 | 10 | -1 | -1 | -1 | -1 |
| 2 | -1 | 10 | -1 | 10 | -1 | -1 | -1 |
| 3 | 40 | -1 | 10 | -1 | 2 | -1 | -1 |
| 4 | -1 | -1 | -1 | 2 | -1 | 3 | 8 |
| 5 | -1 | -1 | -1 | -1 | 3 | -1 | 3 |
| 6 | -1 | -1 | -1 | -1 | 8 | 3 | -1 |




$O(n)$

Vertices → $n$

Edges → $^nC_2$

$= \frac{n*(n-1)}{2}$

$O(n^2)$

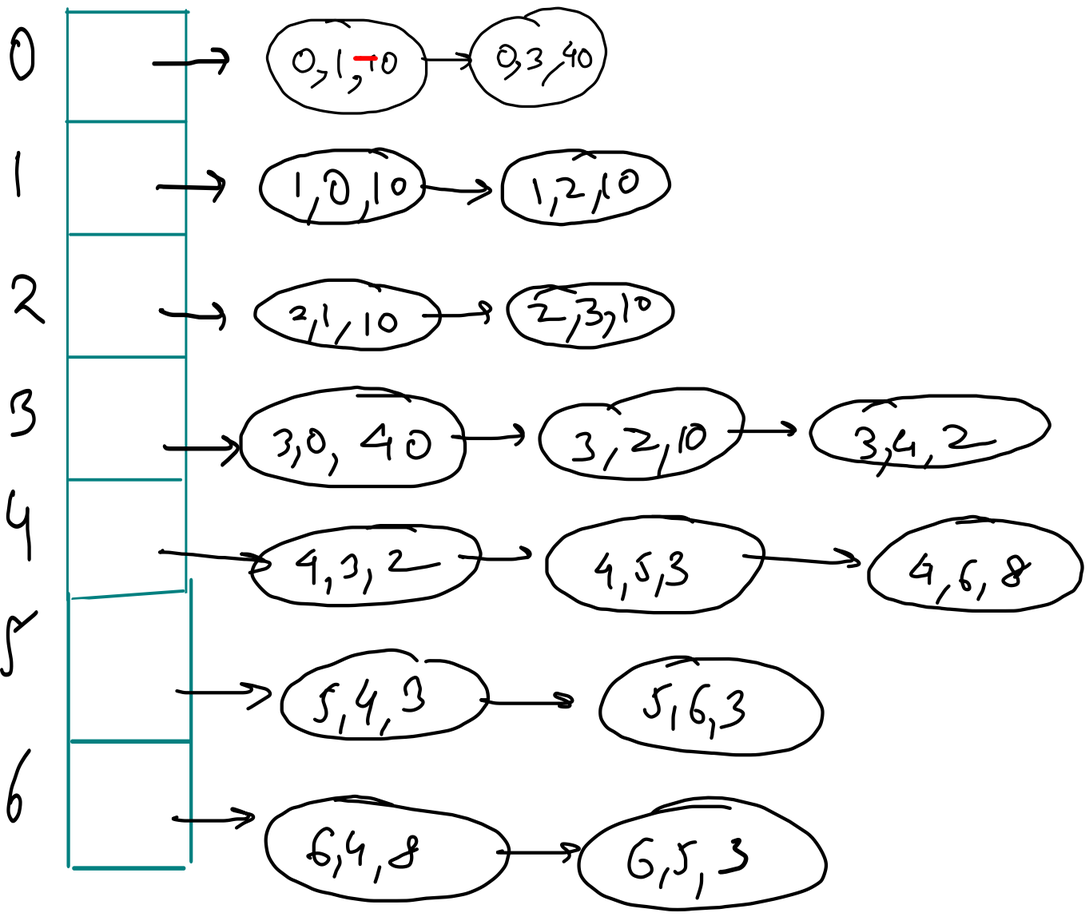
## ① Edge list

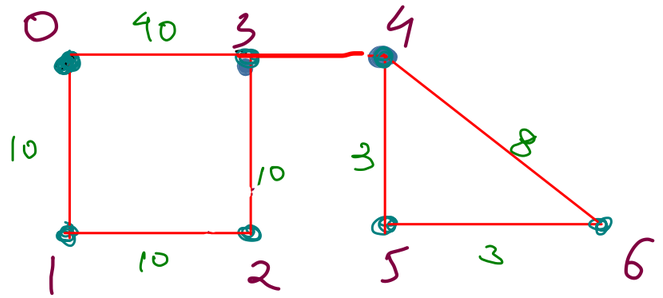
{ 0,3,40}, { 0,1,10}, { 1,2,10},

{ 2,3,10}, { 3,4,2}, { 4,5,3},

{ 5,6,3} , { 4,6,8}

## ③ Adjacency list




```java
static class Edge {
    int src;
    int nbr;
    int wt;
    Edge(int src, int nbr){ }
    Edge(int src, int nbr, int wt){ }
}
```

Edge class

```java
ArrayList<Edge>[] graph = new ArrayList[vts];
for(int i=0; i<vts; i++){
    graph[i] = new ArrayList<>();
}
```

Initialization

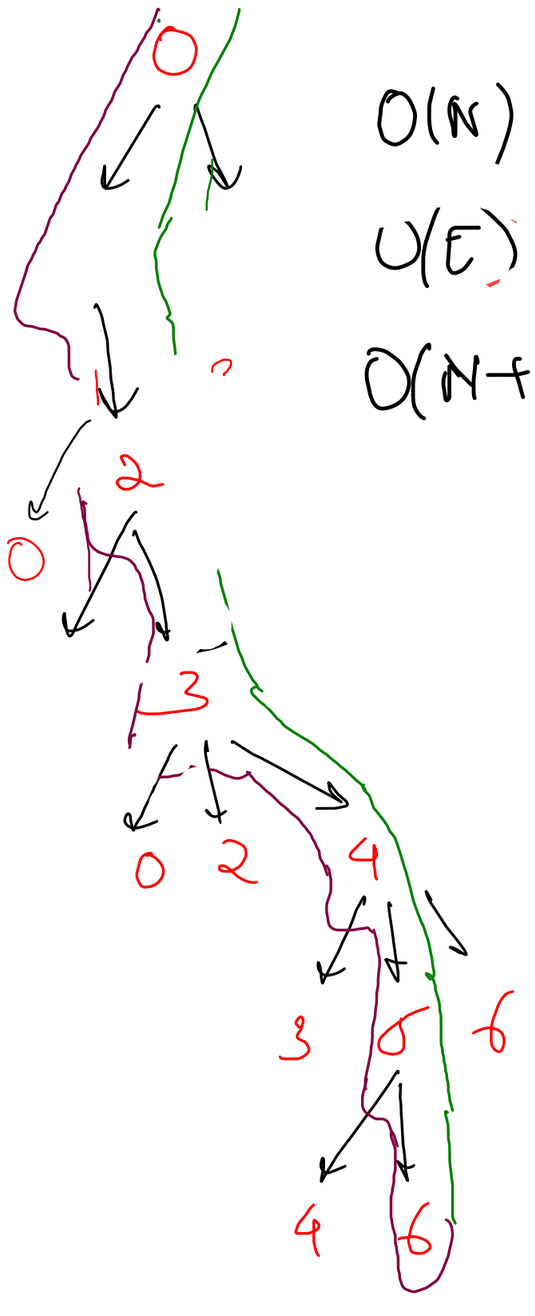
```java
int edges = scn.nextInt();

for(int i=0; i<edges; i++){
    int v1 = scn.nextInt();
    int v2 = scn.nextInt();
    int wt = scn.nextInt();

    graph[v1].add(new Edge(v1, v2, wt));
    graph[v2].add(new Edge(v2, v1, wt));
}
```

Construct

```java
for(int i=0; i<vts; i++){

    System.out.print(i + ": ");

    // Adjacency List of Vertex i
    for(Edge e: graph[i]){
        System.out.print("{ " + e.src + ", " + e.nbr + " @ " + e.wt + "}, ");
    }

    System.out.println();
}
```

display

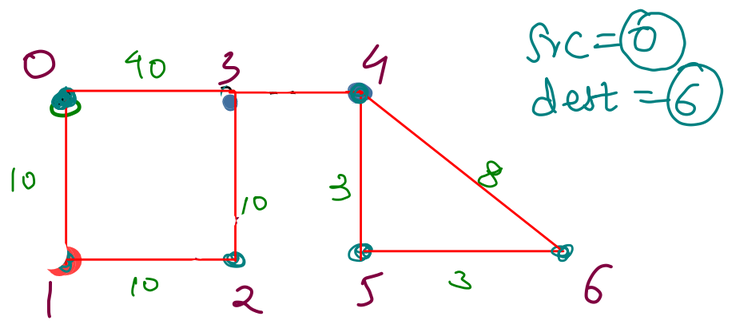
# has Path

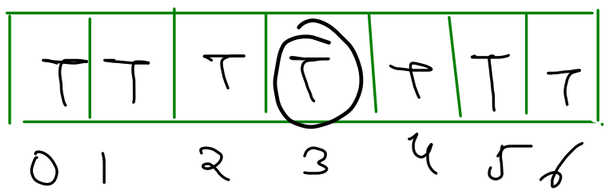



O(N)

O(E)

O(N+E)




2f 2f 3p

...

= O(E)

```java
public static boolean dfs(ArrayList<Edge>[] graph, int src, int dest, boolean[] vis){
    if(src == dest){
        return true;
    }

    vis[src] = true;

    for(Edge e: graph[src]){
        if(vis[e.nbr] == false){ // already visited
            boolean res = dfs(graph, e.nbr, dest, vis);
            if(res == true) return true;
        }
    }

    return false;
}
```

1. Time Complexity of DFS (Single Choice) *

30/30 (100%) answered

A Has Path - O(N + E), Print All Paths - O(N + E) (14/30) 47%

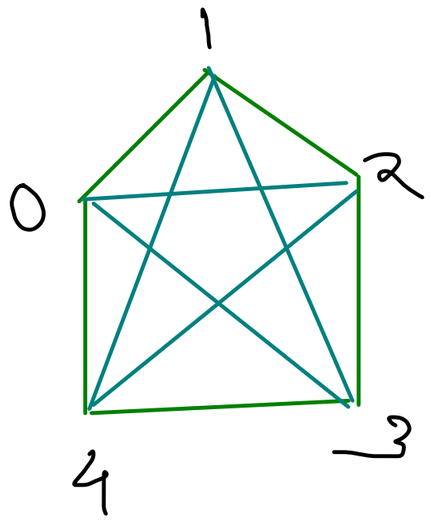
B Has path - O(N + E), Print All Paths - Exponential (14/30) 47%

C Has Path - Exponential, Print All Paths - Exponential (2/30) 7%

Backtracking ⇒ ~~polynomial~~ Exponential

Recursion

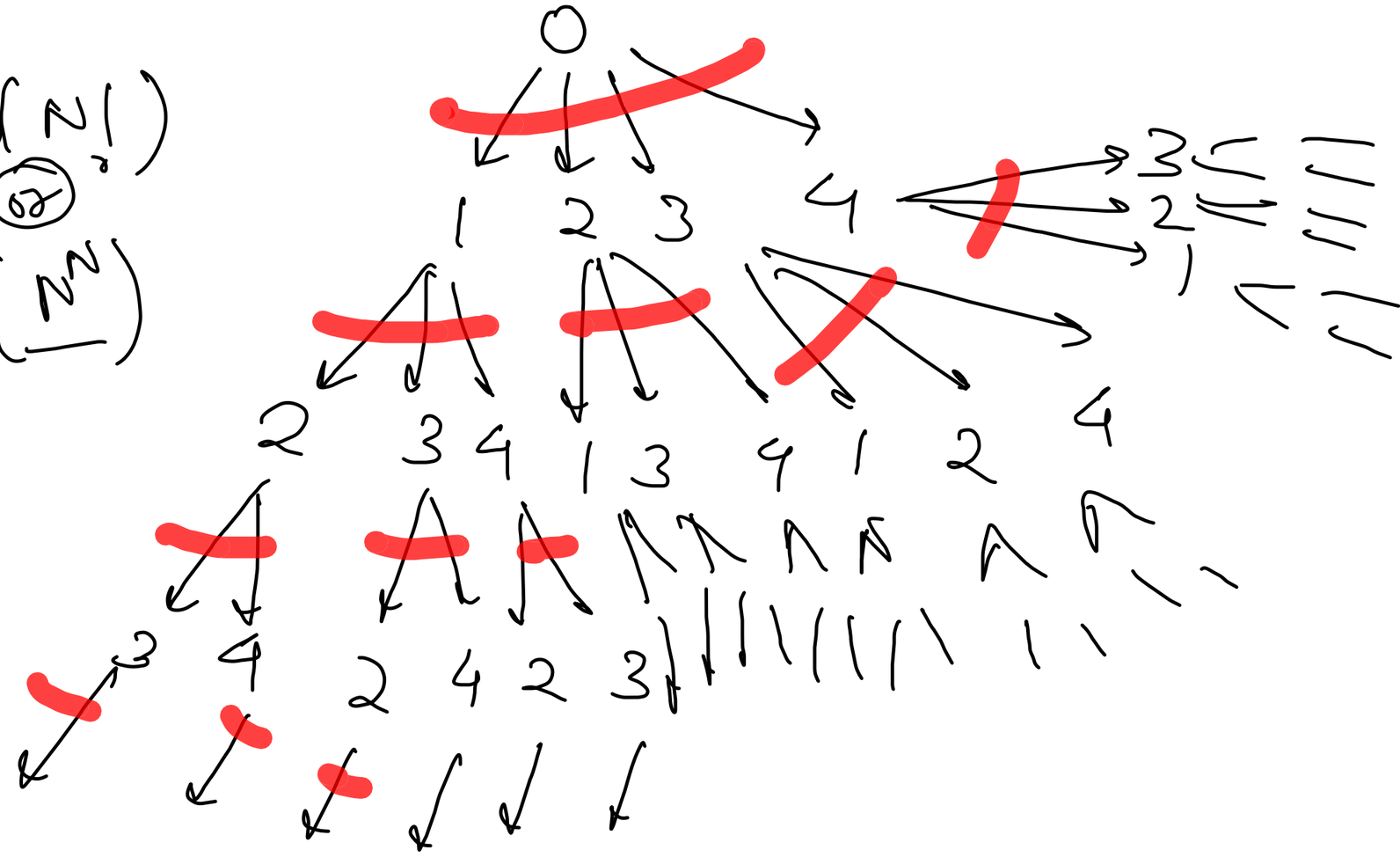
$(\text{calls})^{\text{height}} = \left(\frac{E}{N}\right)^{N} = \left(\frac{N^2}{N}\right)^{N} = O(N^N)$



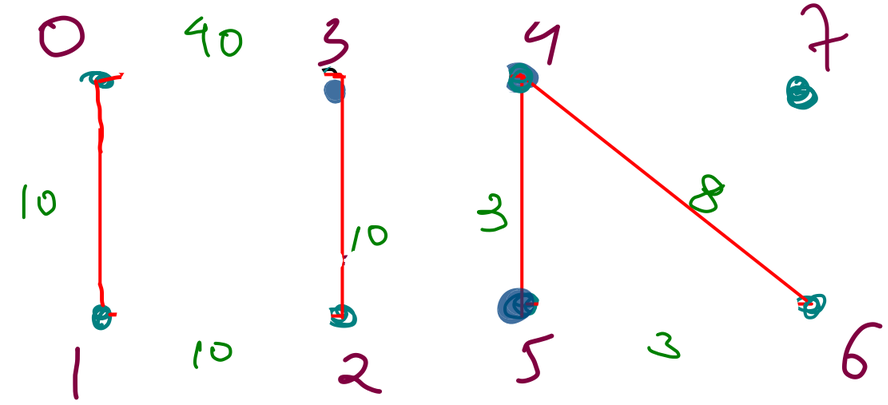

$O(N!)$

or

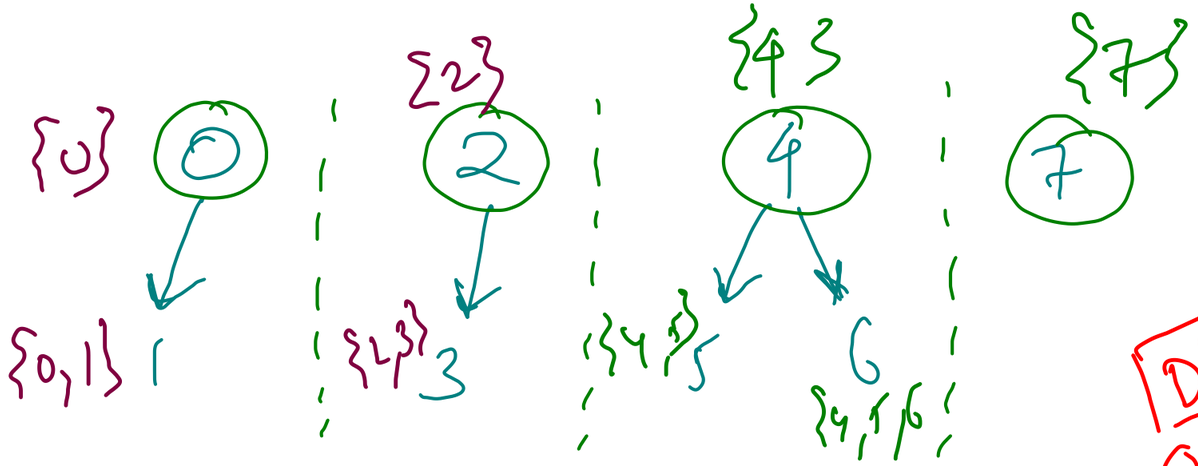
$O(N^N)$

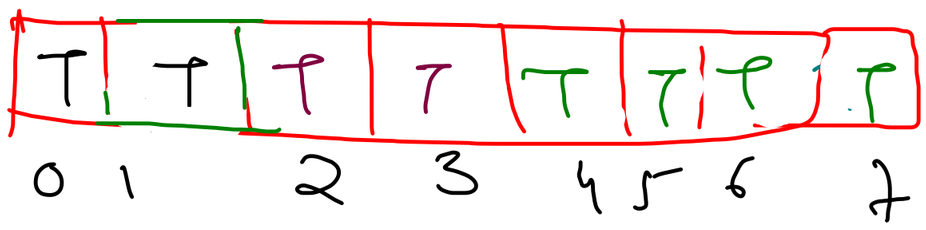


# connected components {undirected}




{{0,1}, {2,3}, {4,5,6}, {7}}




DFS
O(N+E)

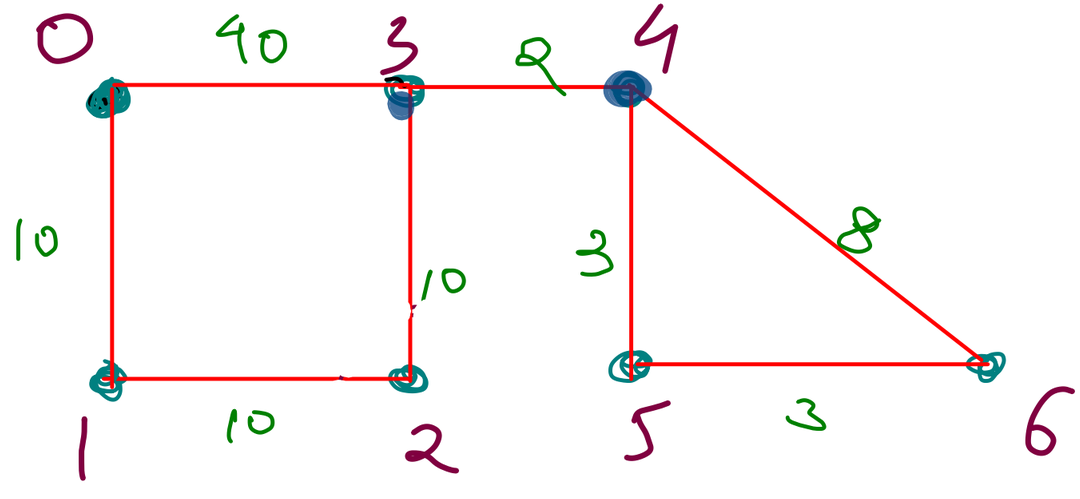
```java
public static void dfs(ArrayList<Edge>[] graph,
        int src, ArrayList<Integer> comp, boolean[] vis){
    vis[src] = true;
    comp.add(src);

    for(Edge e: graph[src]){
        if(vis[e.nbr] == false){ // already visited
            dfs(graph, e.nbr, comp, vis);
        }
    }
}
```

```java
for(int i=0; i<vtces; i++){
    if(vis[i] == false){
        ArrayList<Integer> comp = new ArrayList<>();
        dfs(graph, i, comp, vis);
        comps.add(comp);
    }
}
```

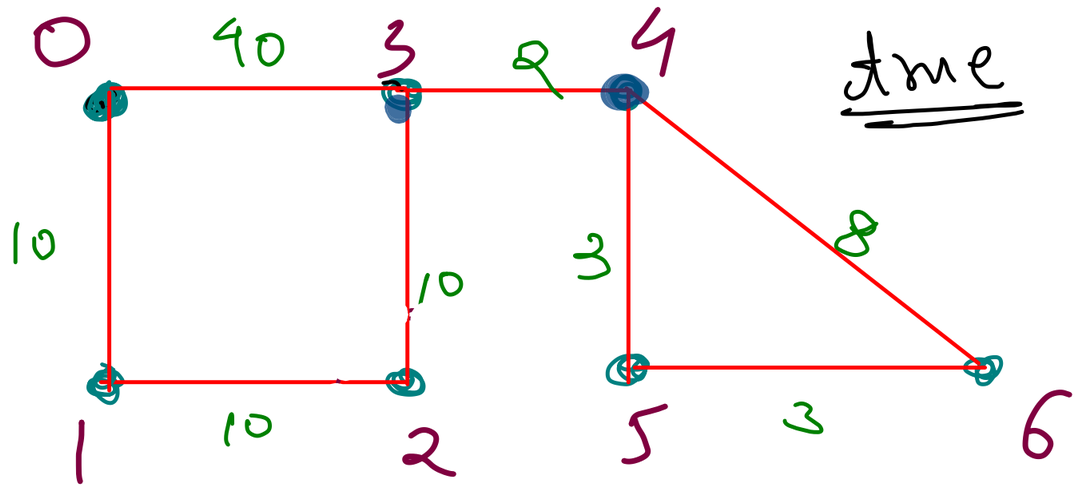
# Today's Questions

- Multisolver - Smallest, Longest, Ceil, Floor, Kthlargest Path
- Is Graph Connected
- Number Of Islands
- Perfect Friends
- ~~Hamiltonian Path And Cycle~~

} ⇒ multisource DFS



# Is Graph connected




component

=1 : connected

>1 : disconnected

=0 component {not possible} { atleast one vertex should be present in graph }

# Number of Islands

Prerequisite :- Floodfill, Connected components.

O(r*c)

```
public static void dfs(int[][] arr, int srcRow, int srcCol){
    if(srcRow < 0 || srcRow >= arr.length || srcCol < 0
        || srcCol >= arr[0].length || arr[srcRow][srcCol] != 0){
        // invalid cell or already visited island cell or water cell
        return;
    }

    arr[srcRow][srcCol] = -1;

    dfs(arr, srcRow + 1, srcCol);
    dfs(arr, srcRow - 1, srcCol);
    dfs(arr, srcRow, srcCol + 1);
    dfs(arr, srcRow, srcCol - 1);
}
```

```
int islands = 0;
for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){

        if(arr[i][j] != -1 && arr[i][j] != 1)
        {
            // unvisited island cell
            dfs(arr, i, j);
            islands++;
        }

    }
}
System.out.println(islands);
```

for each r*c cell

no of vertices

c

r

```
0 0 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0
1 1 0 0 0 1 1 0
1 1 1 1 0 1 1 0
1 1 1 1 0 1 1 0
1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 0
```

$\phi$ X 3

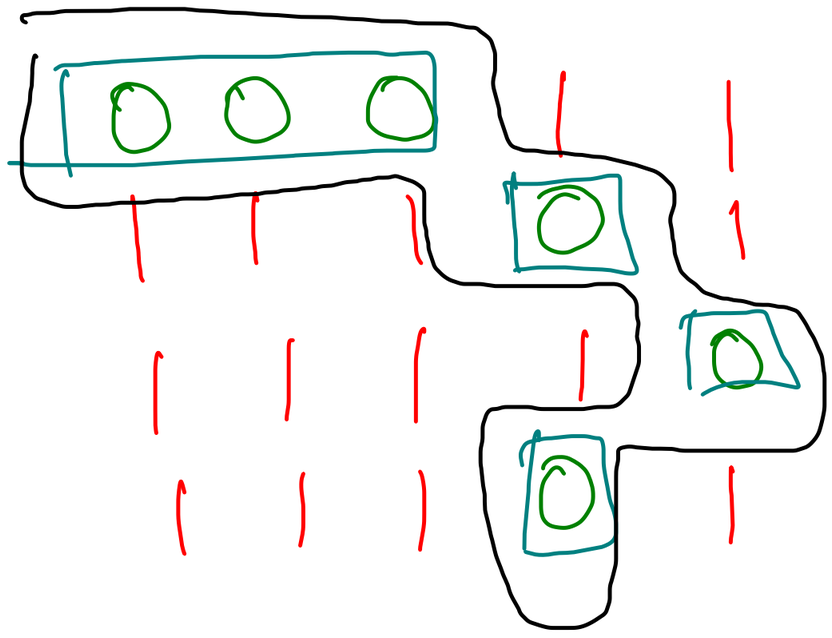
0 ⇒ island

1 ⇒ water

4 way connected

Note

↙ This matrix is not adjacency matrix.



4 way connected != 4

8 way connected != 1

# Perfect friends

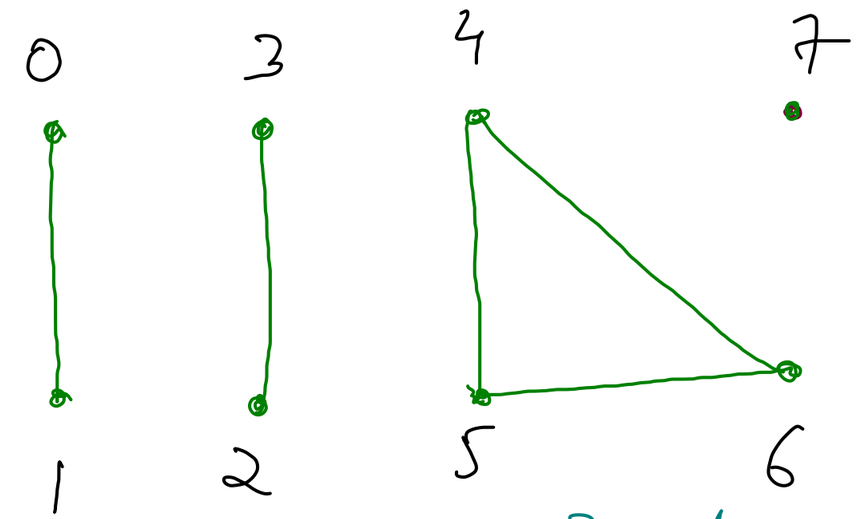
```
8
5
01
23
45
56
46
```

| vl=0 | vl=1 |
|---|---|
| (0,2) | (1,2) |
| (0,3) | (1,3) |
| (0,4) | (1,4) |
| (0,5) | (1,5) |
| (0,6) | (1,6) |
| (0,7) | (1,7) |

| vl=2 | vl=3 |
|---|---|
| (2,4) | (3,4) |
| (2,5) | (3,5) |
| (2,6) | (3,6) |
| (2,7) | (3,7) |

| vl=4 | vl=5 | vl=6 |
|---|---|---|
| (4,7) | (5,7) | (6,7) |

vl=7 no pair




{ {0,1,3}, {2,4}, {5,6}, {7} }

2 * 6 + 2 * 4 + 3 * 1 + 1 * 0

= 12 + 8 + 3 + 0 = 23

Approach 1

```
int countWays = 0;
int remVtces = vtces;

for(int size: comps){
    remVtces -= size;
    countWays += remVtces * size;
}
System.out.println(countWays);
```

Approach 2

```
int countWays = 0;
for(int c1 = 0; c1 < comps.size(); c1++){
    for(int c2 = c1 + 1; c2 < comps.size(); c2++){
        countWays = countWays + comps.get(c1) * comps.get(c2);
    }
}
System.out.println(countWays);
```

# Multisolver - Smallest, Longest, Ceil, Floor, Kthlargest Path

smallest :→ 0123456 @ 38

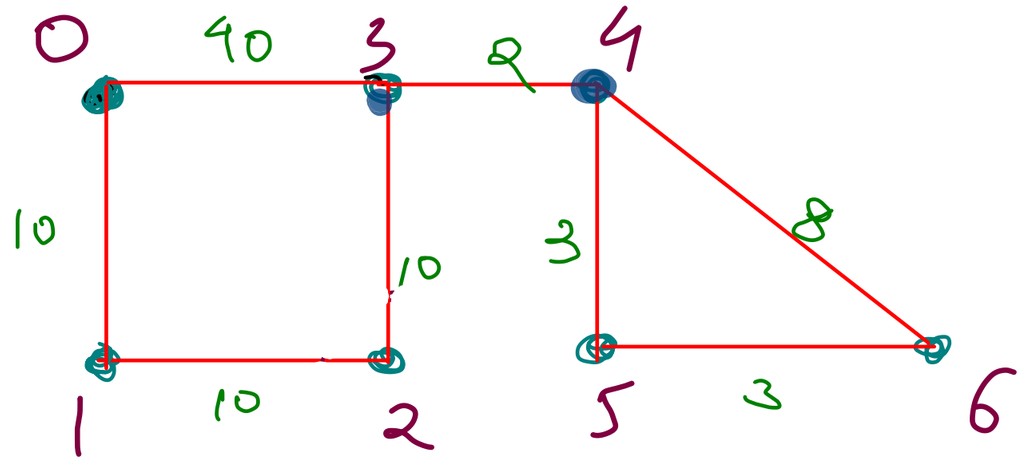
longest :→ 0346 @ 50

ceil(40) :→ 0345 @ 48

floor(40) :→ 012345 @ 38

kthlargest :
k = 3
012346 @ 40

src = 0, dest = 6
val = 40




0123456 :→ 38

01246 :→ 40

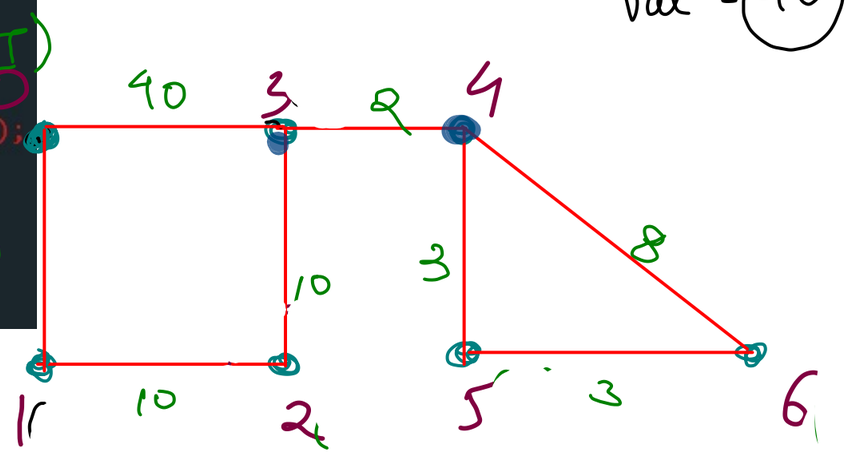
0346 :→ 48

0346 :→ 50

```java
public static void dfs(ArrayList<Edge>[] graph, int src, int dest,
        boolean[] vis, String pathSofar, int weightSofar){
    if(src == dest){ }

    vis[src] = true;

    for(Edge e: graph[src]){
        if(vis[e.nbr] == false){
            dfs(graph, e.nbr, dest, vis, pathSofar + e.nbr, weightSofar + e.wt);
        }
    }

    vis[src] = false;
}
```

src = 0, dest = 6

val = 40




0, "0", 0

1, "01", 10 — 3, "03", 40

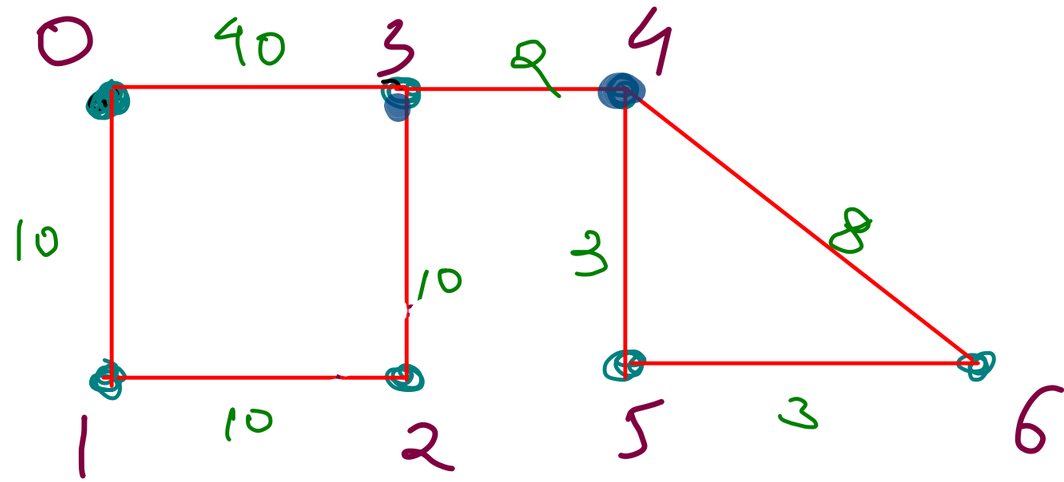
2, "012", 20

3, "0123", 30 → 4, "01234", 32 → 5, "012345", 35 — 6, "0123456", 38

6, "012346", 40

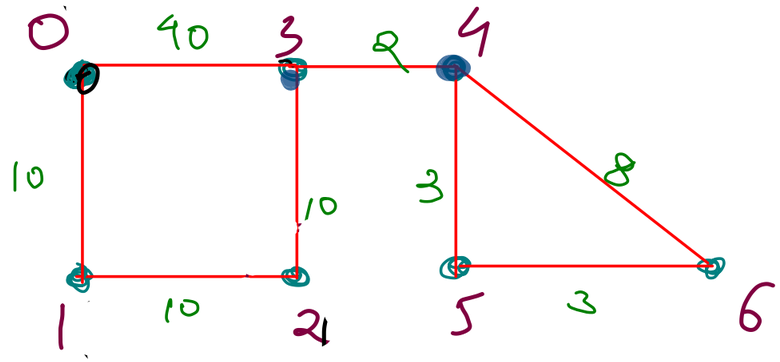
# Today's Target { lecture-3 Graphs }

- </> Hamiltonian Path And Cycle
- </> Breadth First Traversal
- </> Is Graph Cyclic
- </> Is Graph Bipartite



# worst case {Exponential} → O(N^N) → O(N!)

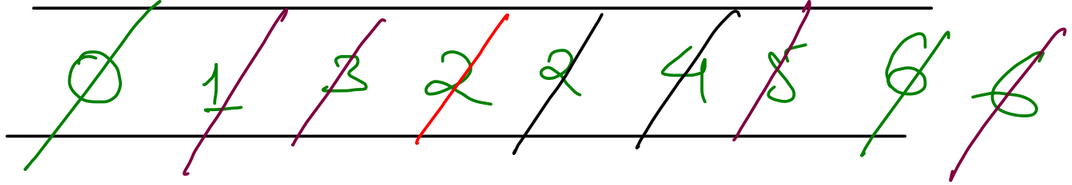
```java
public static void dfs(ArrayList<Edge>[] graph, int node, int visCount, boolean[] vis, String pathSofar){

    if(visCount == graph.length - 1)
    {
        // hamiltonian path or cycle
        if(isEdgePresent(graph, pathSofar.charAt(0) - '0', node)){
            // cycle check -> is there a edge between src and 0
            System.out.println(pathSofar + "*");
        } else {
            System.out.println(pathSofar + ".");
        }
        return;
    }

    visCount++;
    vis[node] = true;

    for(Edge e: graph[node]){
        if(vis[e.nbr] == false){
            dfs(graph, e.nbr, visCount, vis, pathSofar + e.nbr);
        }
    }

    vis[node] = false;
}
```



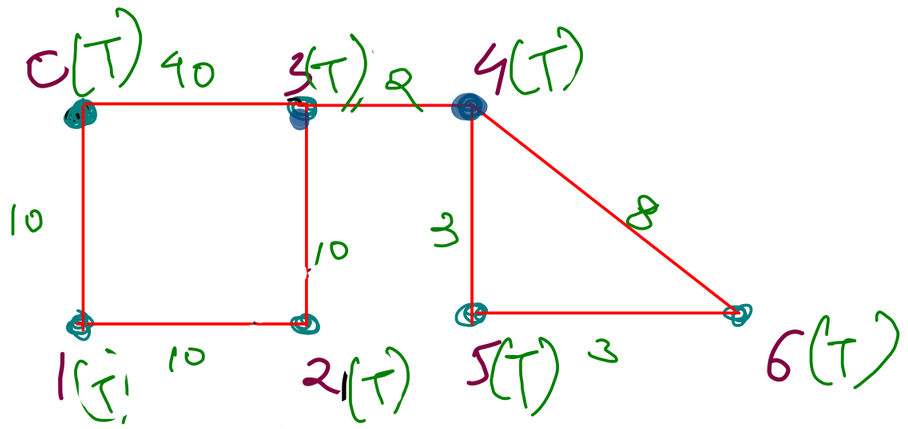

## Hamiltonian Path & Cycle

- visits all vertices in its path without repeating any vertex
- Edge b/w Starting & ending Vertex

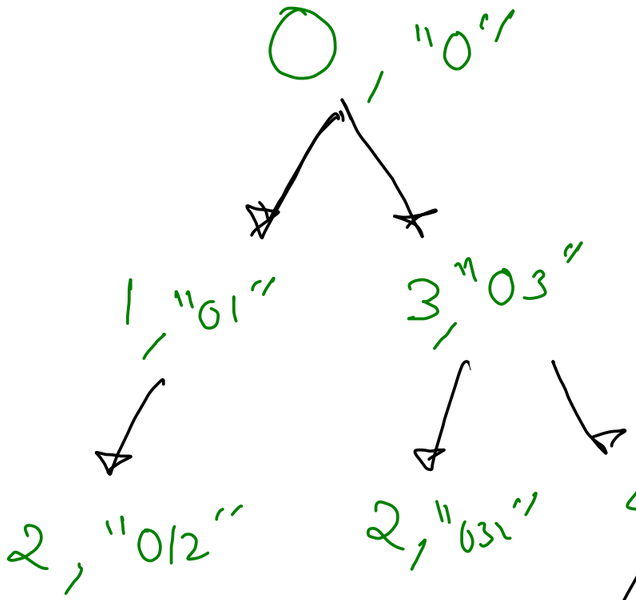
# Breadth First Traversal

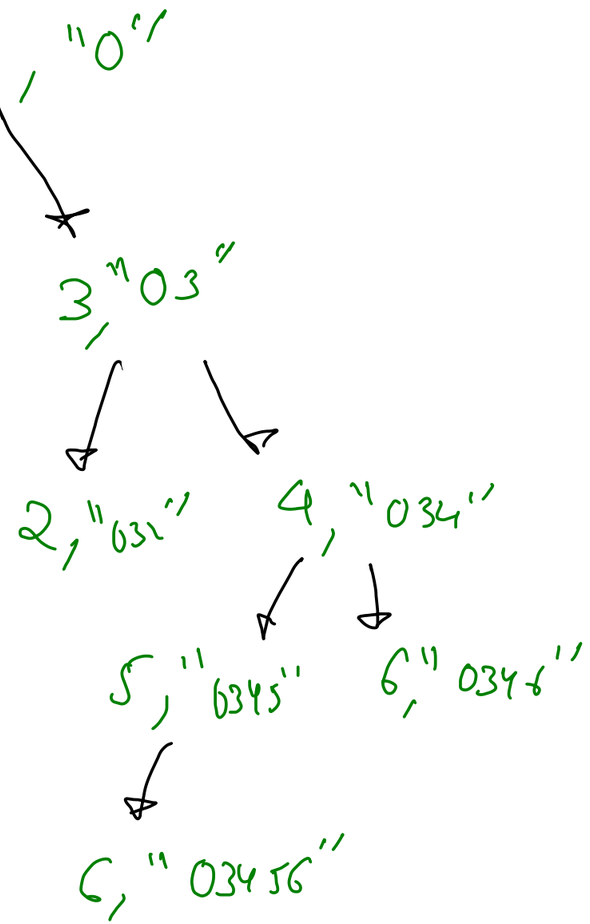


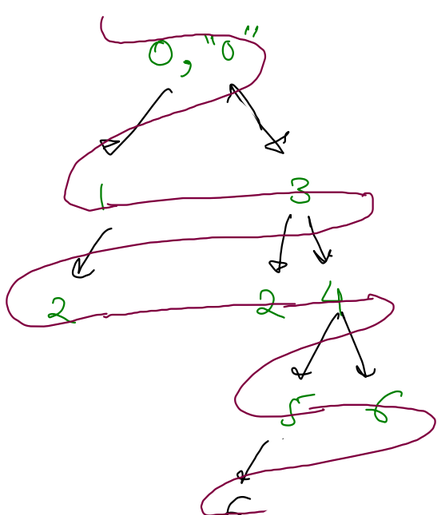

level order

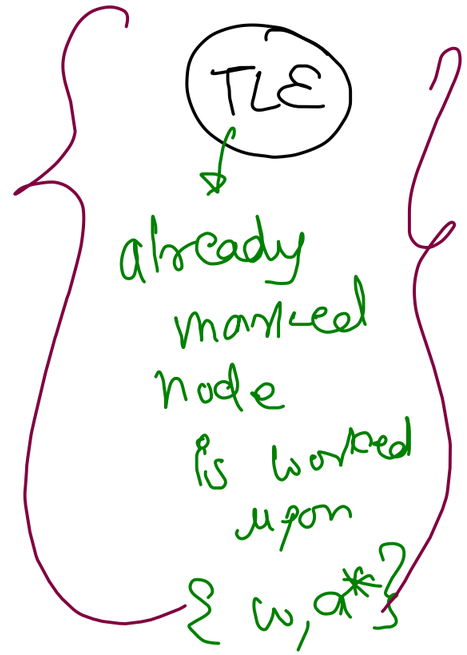



0 @ 0, 1 @ 01, 2 @ 012, 3 @ 034
5 @ 0345
6 @ 03456

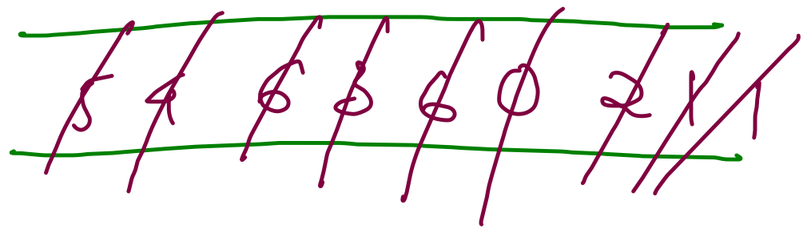


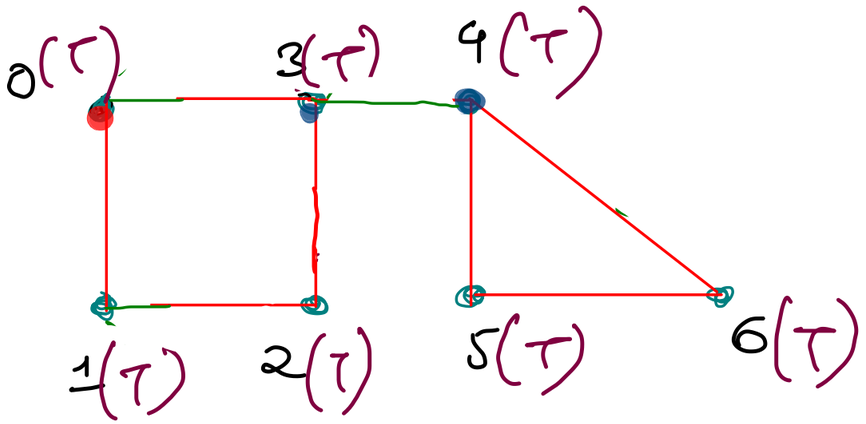



src = 5



## ① Component BFS




```java
Queue<Pair> q = new LinkedList<>();
boolean[] vis = new boolean[vtces];

q.add(new Pair(src, "" + src));

while(q.size() > 0){
    // remove
    Pair curr = q.remove();

    if(vis[curr.node] == true) continue;

    // mark*
    vis[curr.node] = true;

    // work
    System.out.println(curr.node + "@" + curr.pathSofar);

    // add*
    for(Edge e: graph[curr.node]){
        if(vis[e.nbr] == false){
            q.add(new Pair(e.nbr, curr.pathSofar + e.nbr));
        }
    }
}
```

return true → Cycle present

return false; ↔ Cycle not present

Time

O(N+E)

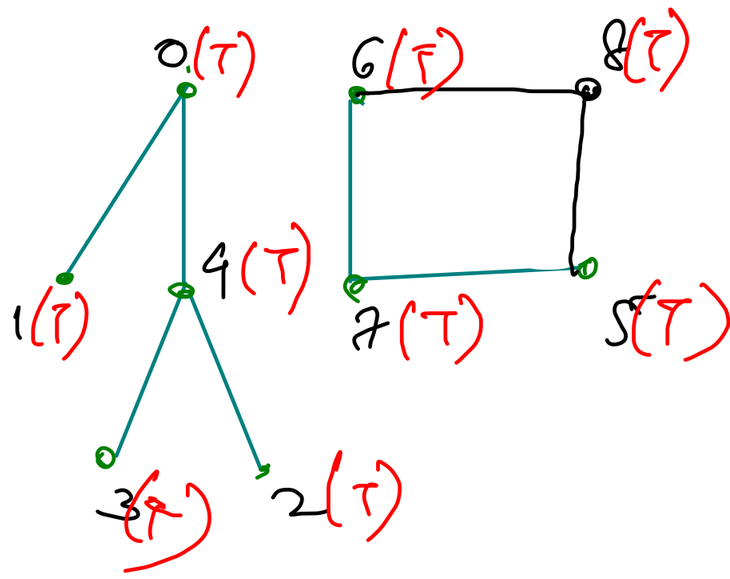
## Is cycle Present in Component

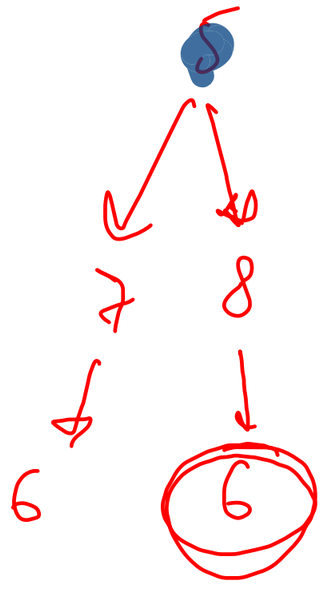
```java
public static boolean BFS(ArrayList<Edge>[] graph, boolean[] vis, int src){
    Queue<Pair> q = new LinkedList<>();
    q.add(new Pair(src, "" + src));

    while(q.size() > 0){
        // remove
        Pair curr = q.remove();

        if(vis[curr.node] == true) return true;

        // mark*
        vis[curr.node] = true;

        // add*
        for(Edge e: graph[curr.node]){
            if(vis[e.nbr] == false){
                q.add(new Pair(e.nbr, curr.pathSofar + e.nbr));
            }
        }
    }

    return false;
}
```

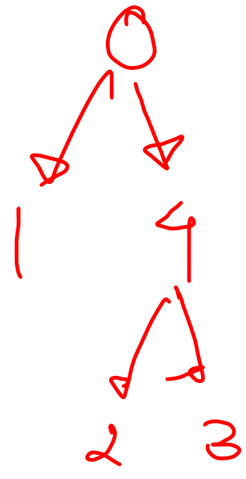
## BFS on each component

```java
boolean[] vis = new boolean[vtces];
for(int i=0; i<vtces; i++){
    if(vis[i] == false){
        // new Component
        boolean isCycle = BFS(graph, vis, i);
        if(isCycle == true) {
            System.out.println(true);
            return;
        }
    }
}
System.out.println(false);
```

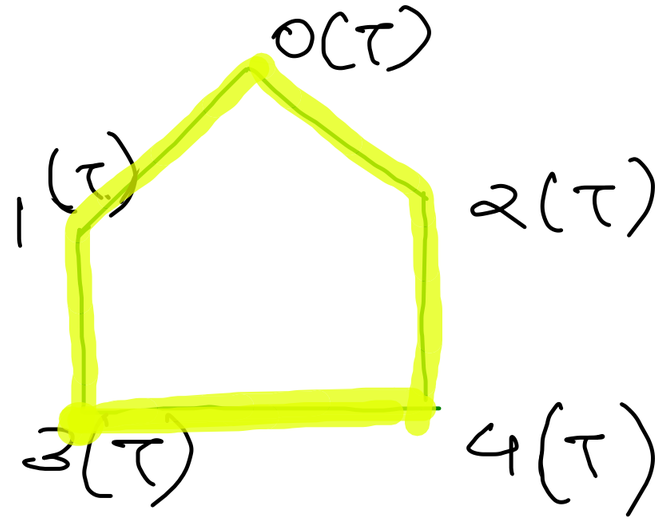


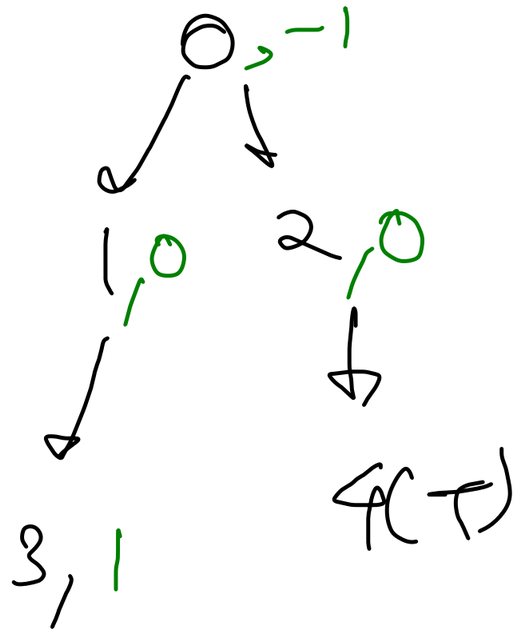


```java
public static boolean BFS(ArrayList<Edge>[] graph, boolean[] vis, int src){
    Queue<Pair> q = new LinkedList<>();
    q.add(new Pair(src, "" + src, -1));
    vis[src] = true;

    while(q.size() > 0){
        // remove
        Pair curr = q.remove();

        // add*
        for(Edge e: graph[curr.node]){
            if(vis[e.nbr] == false){
                vis[e.nbr] = true;
                q.add(new Pair(e.nbr, curr.pathSofar + e.nbr, curr.node));
            } else {
                if(curr.parent != -1 && e.nbr != curr.parent){
                    return true;
                }
            }
        }
    }

    return false;
}
```



Interview - Tip

45 - 60 mins

Editor

↳ Test case

① Introduce yourself

↳ 2-3 mins 5-10

② DSA

Brute force { code }

↳ thinking loud { Hints }

↳ logic { comm skill }

↳ code

↳ optimization

↳ 30

→ Pepcoding

→ Interview Bit

→ Ramp

↑

Mock Interviews

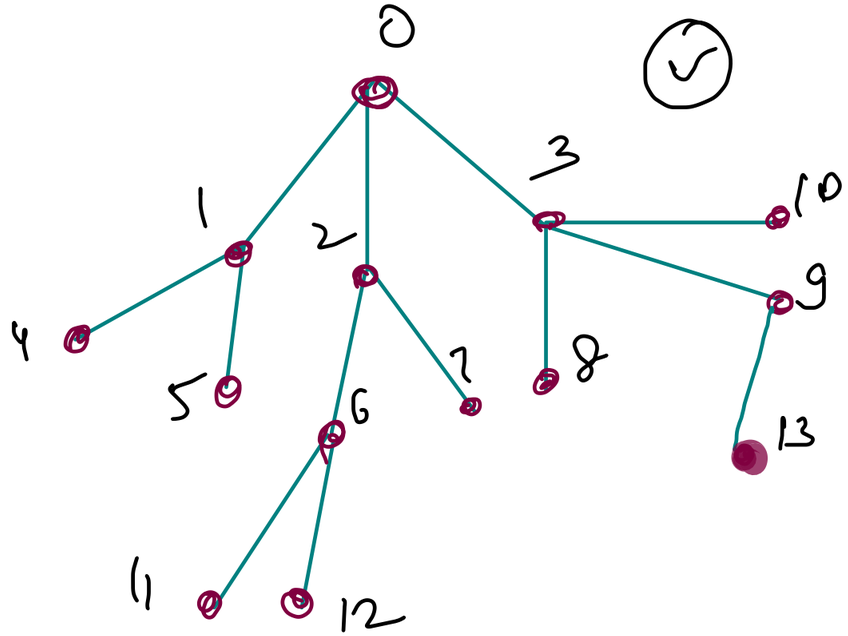
10 mins

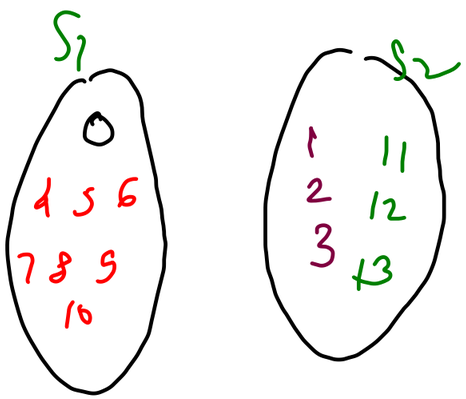
↳ follow ups

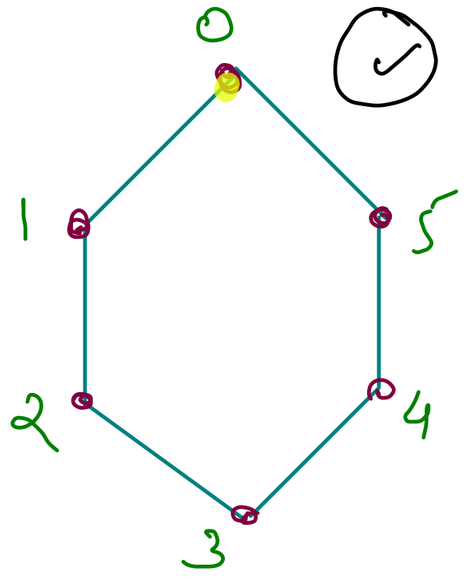
# Graphs

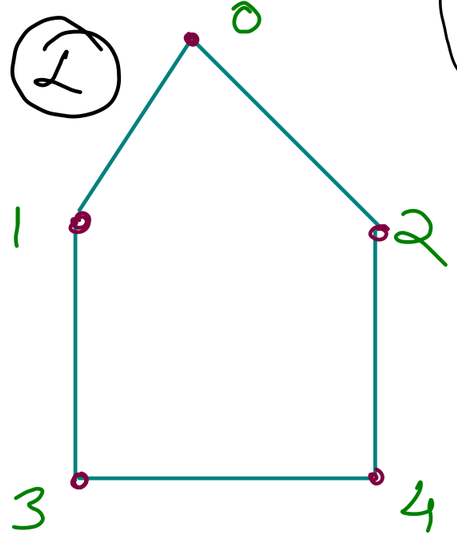
- Is Graph Bipartite
- Spread Of Infection
- Shortest Path In Weights
- Minimum Wire Required To Connect All Pcs

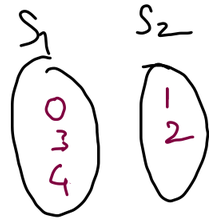
# Bipartite Graph



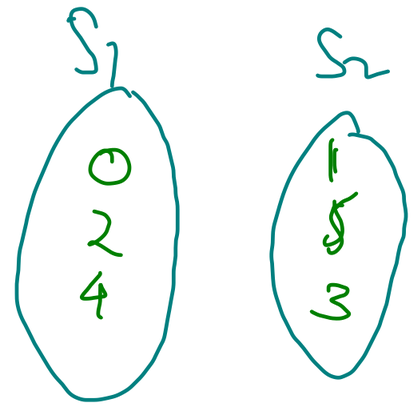




① mutually exclusive
exhaustive

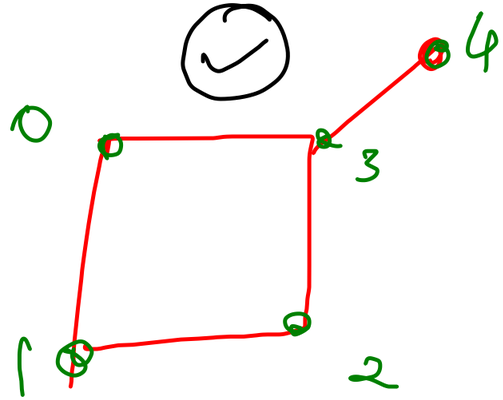
② within set, any edge must not exist

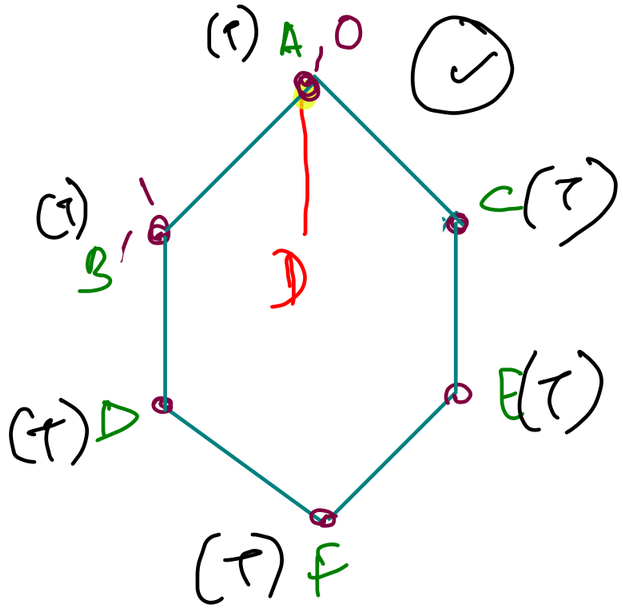


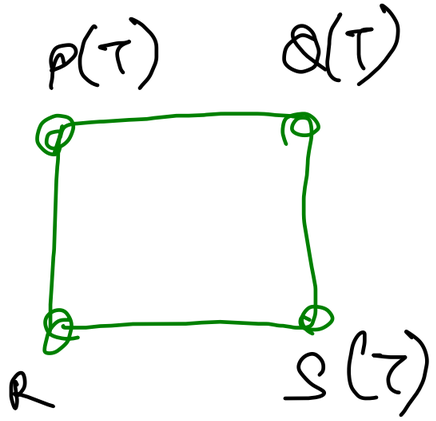
① Cycle does not exist → Bipartite

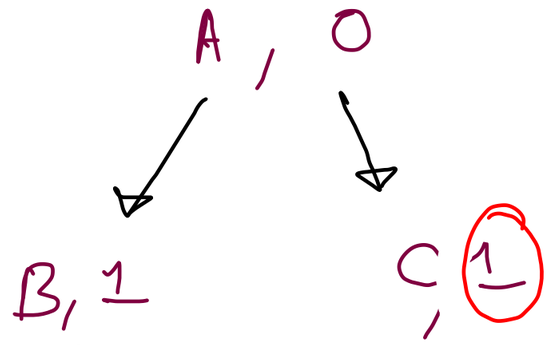
② Cycle(s) are of even length → Bipartite

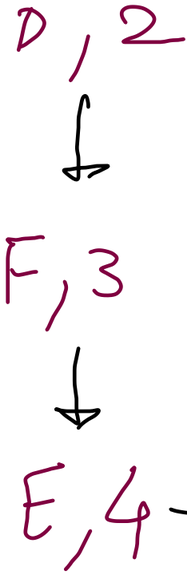
③ Atleast 1 cycle in any component is of odd length
→ Non-Bipartite.



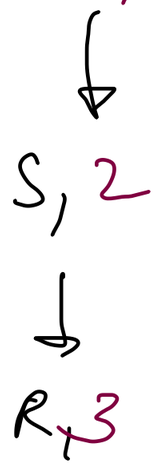



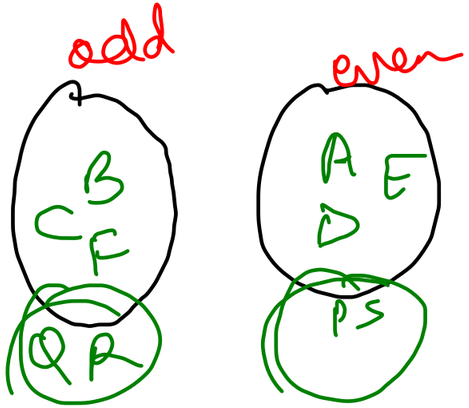





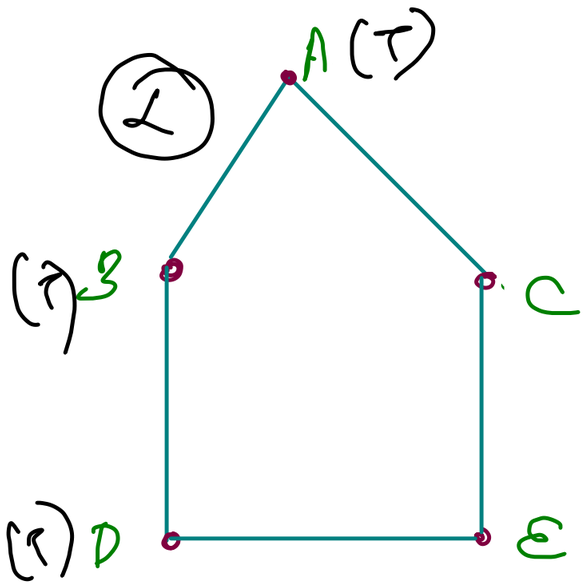


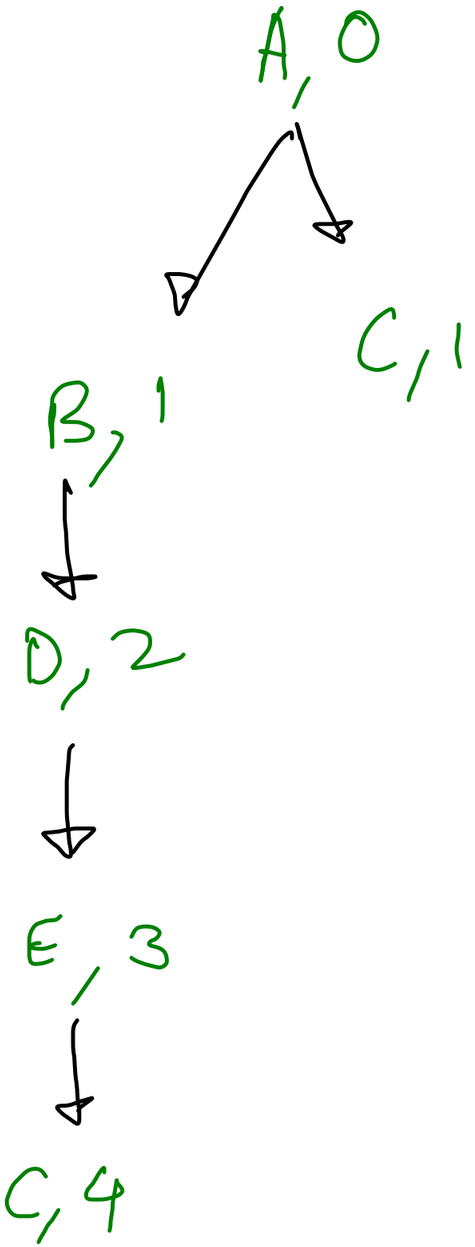

HashSet < int > odd, even;



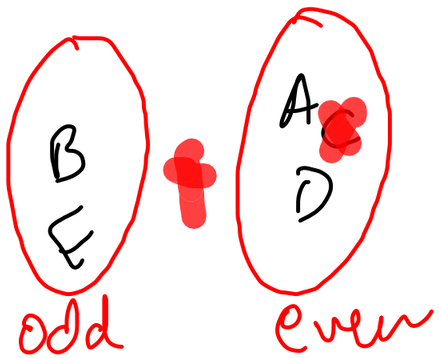

boolean

DFS

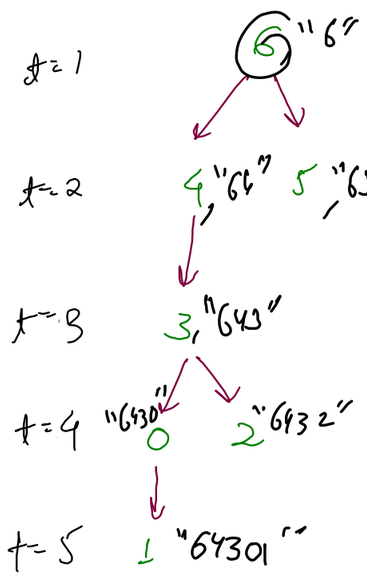


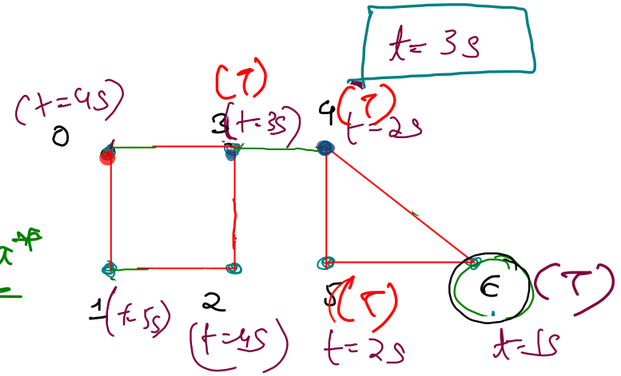

DFS



# Spread of Infection




{BFS}



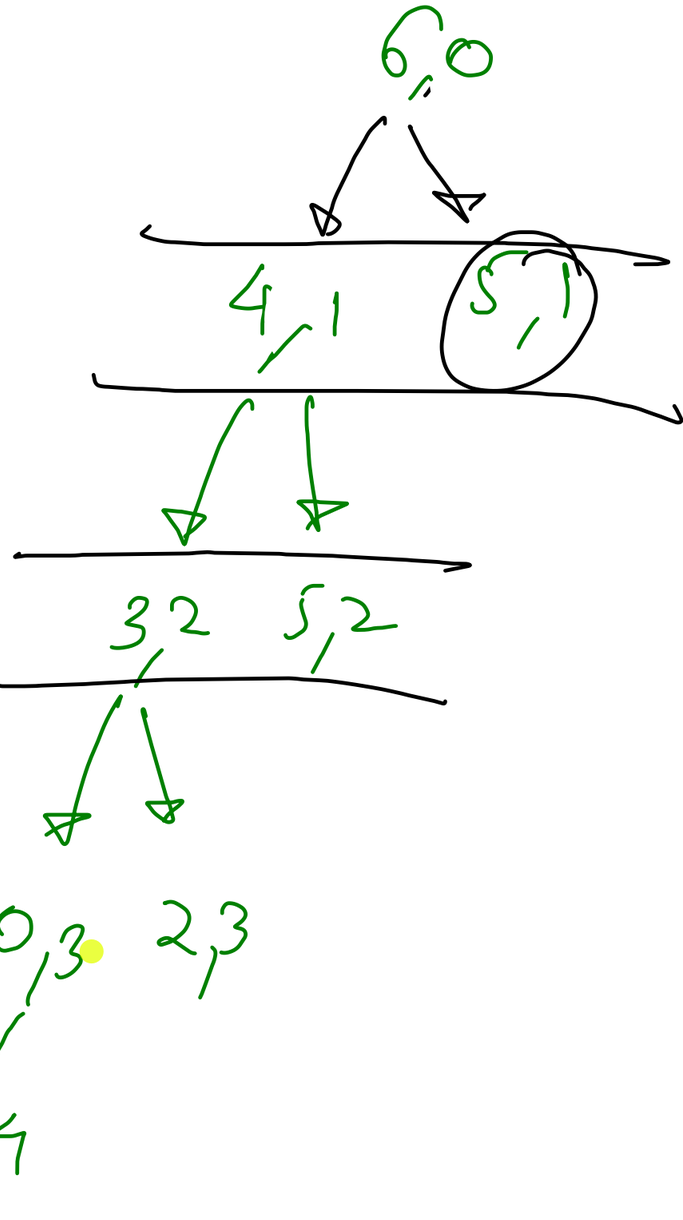

count ↓

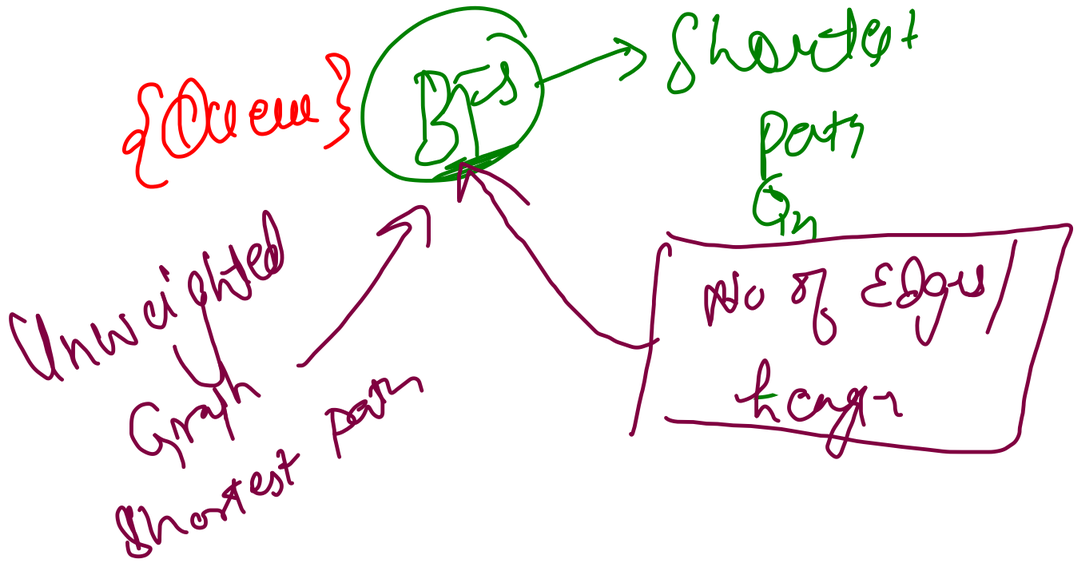
0 → 1 → 4

# Rotten oranges

Multi-source BFS

```
Queue<Pair> q = new LinkedList<>();
boolean[] vis = new boolean[vtces];

q.add(new Pair(src, 1));
int count = 0;

while(q.size() > 0){
    // remove
    Pair curr = q.remove();


    if(vis[curr.node] == true) continue;
    vis[curr.node] = true;

    // work
    if(curr.time > t) break;

    count++;

    // add*
    for(Edge e: graph[curr.node]){
        if(vis[e.nbr] == false){
            q.add(new Pair(e.nbr, curr.time + 1));
        }
    }
}

System.out.println(count);
```



{Greedy}

Dijkstra's algorithm {shortest path in weight}

single source

BFS {queue}

priority queue

0 "0"

1 "01" 3 "03"

2 "012" 4 "034"

5 "0345" 6 "0346"

[0346 @ 50]

{0123456 @ 38}

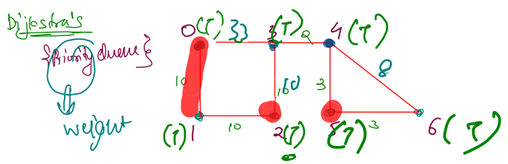
weight

TC :- O(V + ElogV)

```
PriorityQueue<Pair> q = new PriorityQueue<>();
boolean[] vis = new boolean[vtces];

q.add(new Pair(src, "" + src, 0));

while(q.size() > 0){
    // remove
    Pair curr = q.remove();

    if(vis[curr.node] == true) continue;

    // mark*
    vis[curr.node] = true;

    // work
    System.out.println(curr.node + " via " + curr.pathSofar + " @ " + curr.weightSofar);

    // add*
    for(Edge e: graph[curr.node]){
        if(vis[e.nbr] == false){
            q.add(new Pair(e.nbr, curr.pathSofar + e.nbr, curr.weightSofar + e.wt));
        }
    }
}
```

Dijkstra's priority queue

weight




0, "0", 0

1, "01", 10

3, "03", 30

2, "012", 20

3, "0123", 30

4, "01234", 32

5, "012345", 35

6, "0123456", 40

6, "0123456", 38

Dijkstra → -ve weight cycle → Bellman ford

Floyd's warshall → ② All pair shortest path

| | | |
|---|---|---|
| 0 | "0" | 0 |
| 1 | "01" | 10 |
| 2 | "012" | 20 |
| 3 | "0123" | 30 |
| 4 | "01234" | 32 |
| 5 | "012345" | 35 |
| 6 | "0123456" | 38 |

① multi-source → floyd's warshall

0

1

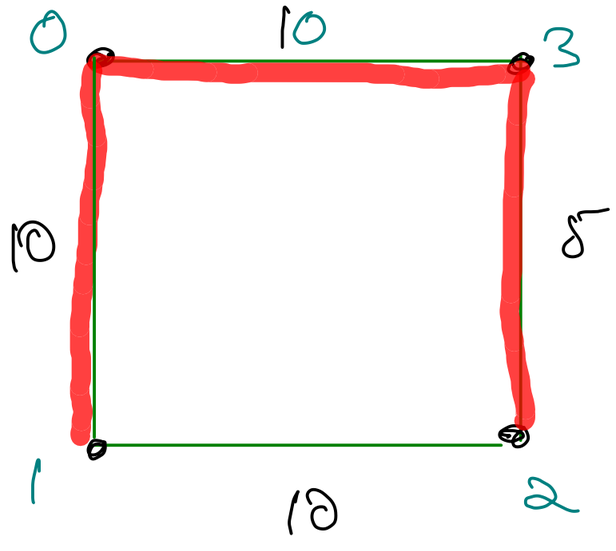
2

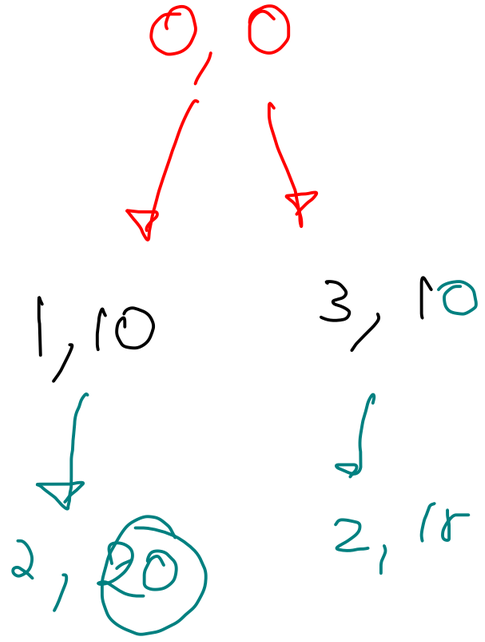
3

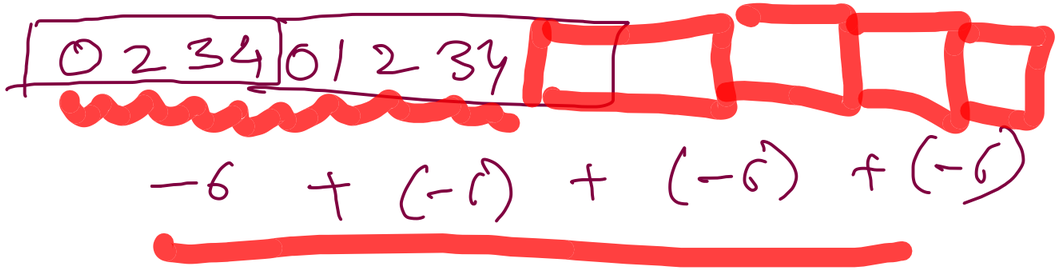
1

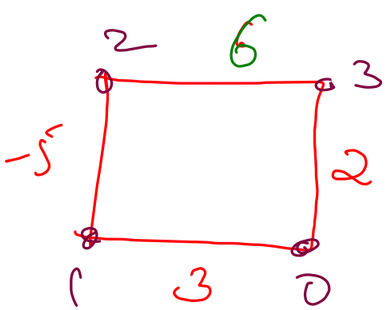
② Negative weight cycle → Bellman ford

U → pair

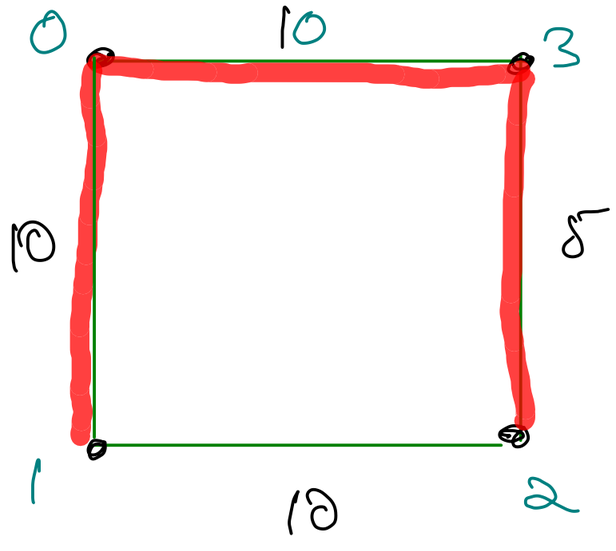

0
10
3
10
5
1
10
2

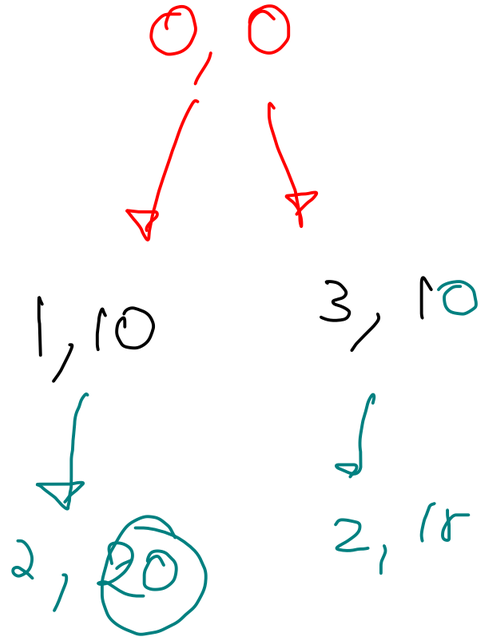

0, 0
1, 10
3, 10
2, 20
2, 15

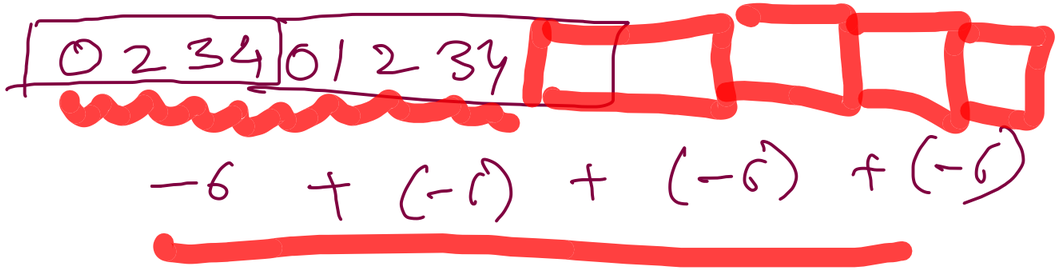

0 2 3 4 0 1 2 3 4
-6 + (-1) + (-6) + (-6)

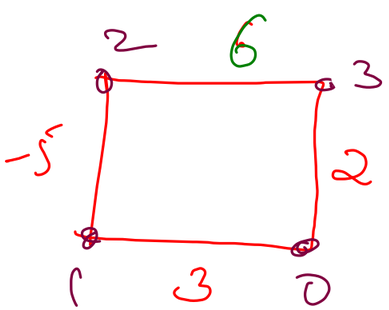

2
6
3
-5
2
1
3
0


1 (3) → 2 (-2)
0
3 (2) → 2 (8)


0
10
3
10
5
1
10
2


0, 0
1, 10
3, 10
2, 20
2, 15

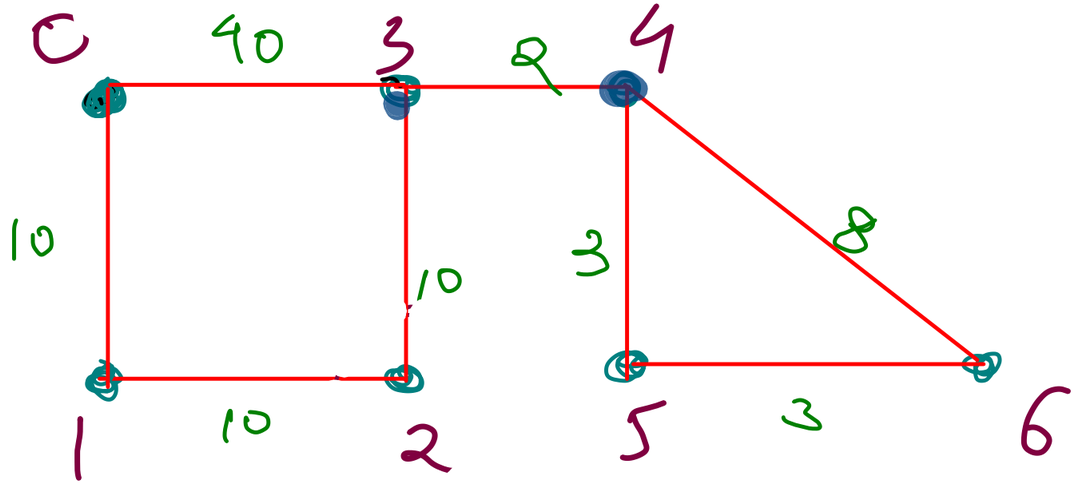

0 2 3 4 0 1 2 3 4
-6 + (-1) + (-6) + (-6)


2
6
3
-5
2
1
3
0


1 (3) → 2 (-2)
0
3 (2) → 2 (8)

# lecture 5 Graphs

1. </> Minimum Wire Required To Connect All Pcs
2. </> Order Of Compilation
3. </> Iterative Depth First Traversal

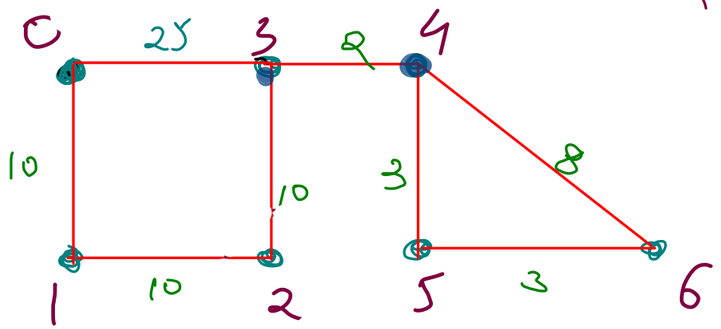


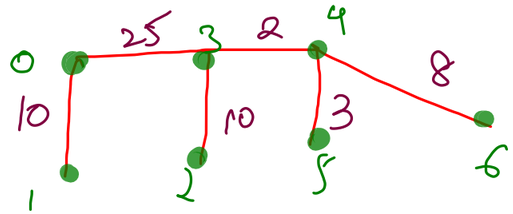
# ① Minimum Spanning Tree {undirected graphs}

{Prim's Algorithm}

→ subgraph

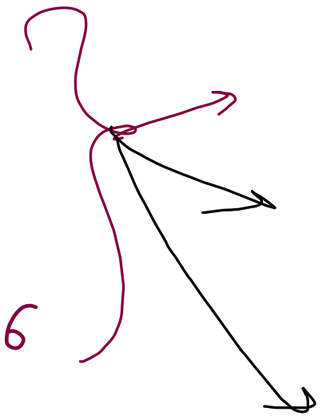
- → tree
  - → minimum edges
  - → Acyclic
  - → Connected
- → spanning
  - → all vertices will be there
- → minimum spanning tree
  - → edges' sum should be minimum

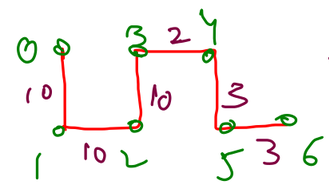


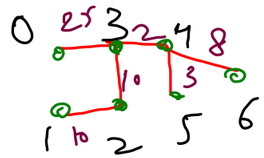

① sum = 61

{7 vertices, 6 edges + 7th edge}

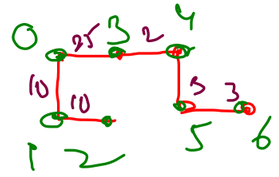





sum = 58




sum = 58




sum = 53

wt → replace edge weight

node  parent  edge wt

0, -1, 0

$O(V + E * \log V)$

1, 0, 10

3, 0, 25

2, 1, 10

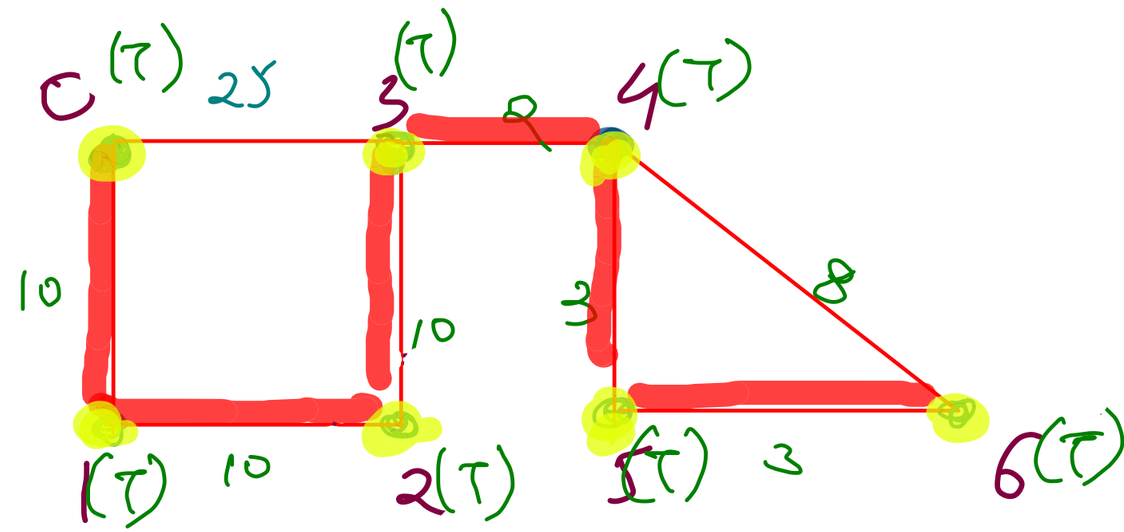
3, 2, 10

4, 3, 2

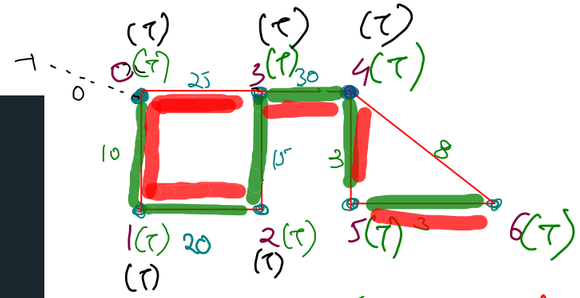
5, 4, 3

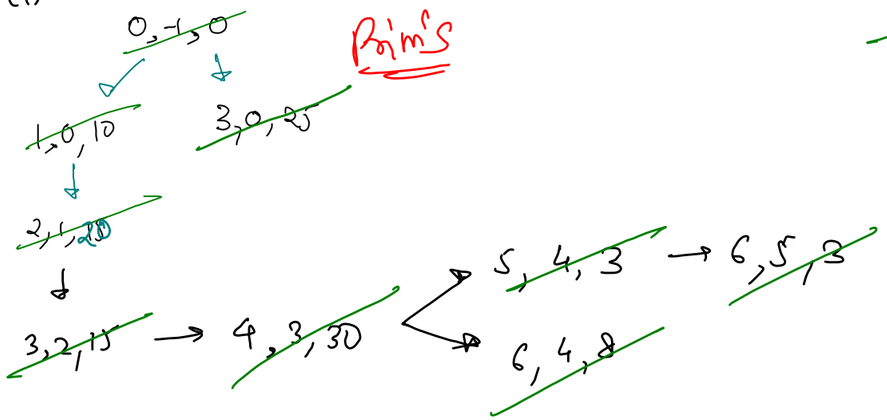
6, 5, 3

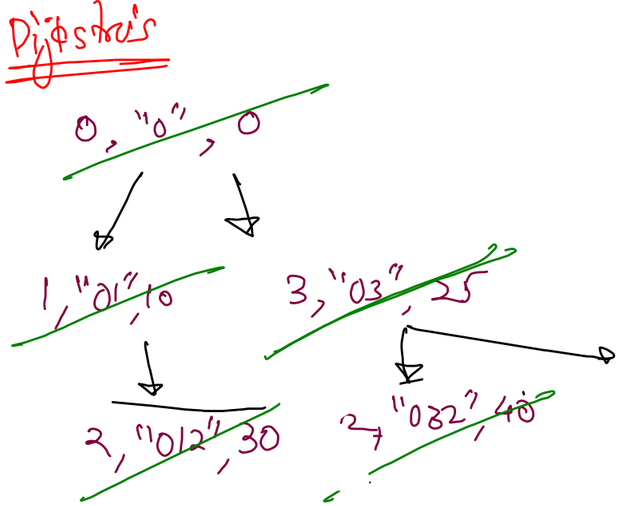
6, 4, 8

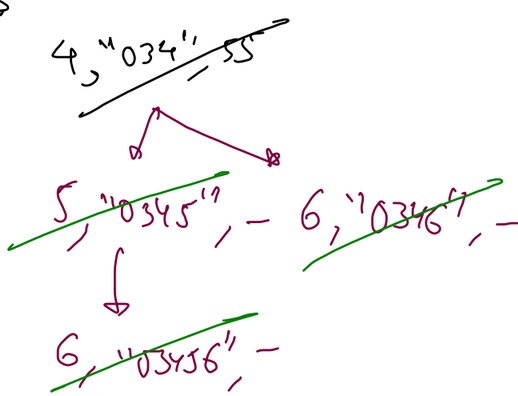


```
PriorityQueue<Pair> q = new PriorityQueue<>();
boolean[] vis = new boolean[vtces];

q.add(new Pair(0, -1, 0));

while(q.size() > 0){
    // remove
    Pair curr = q.remove();

    if(vis[curr.node] == true) continue;

    // mark*
    vis[curr.node] = true;

    // work
    if(curr.parent != -1){
      System.out.println("[" + curr.node + "-" + curr.parent
    }

    // add*
    for(Edge e: graph[curr.node]){
        if(vis[e.nbr] == false){
            q.add(new Pair(e.nbr, curr.node, e.wt));
        }
    }
}
```



# Application of MST

**Network design.**

*– telephone, electrical, hydraulic, TV cable, computer, road*

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

(1) Min cost of wire connect PC's

(2) Min cost of roads to connect all cities

{Google} (4) Optimize Water Distribution

{Min cost to supply water ⇒ well / tap}

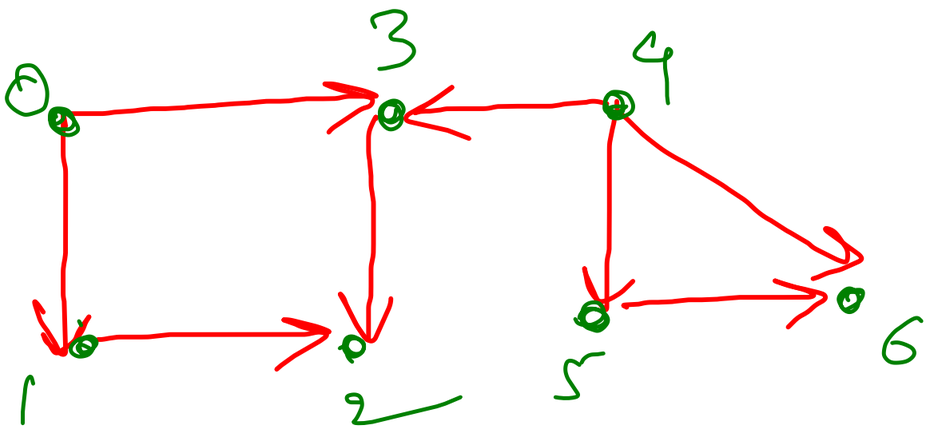
(3) Min cost to connect all 2D points

# Topological Sort { Order of Compilation }

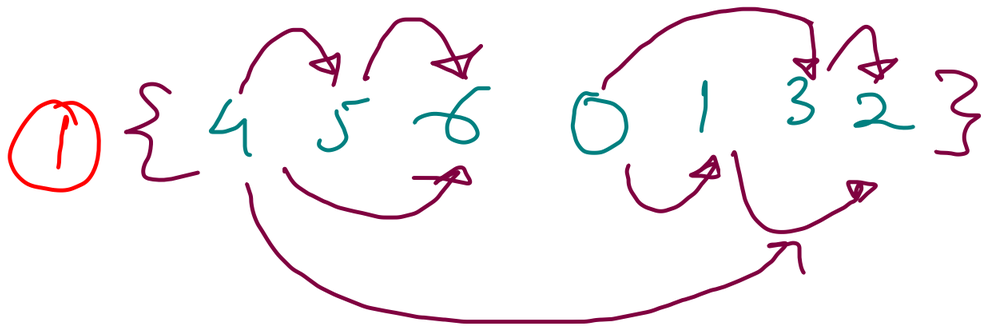
Visited

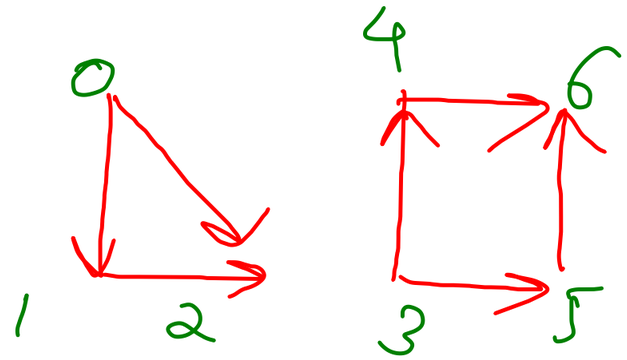
Directed Acyclic Graph (DAG)

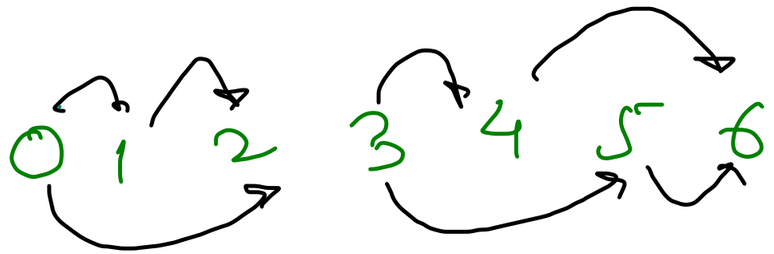
visited

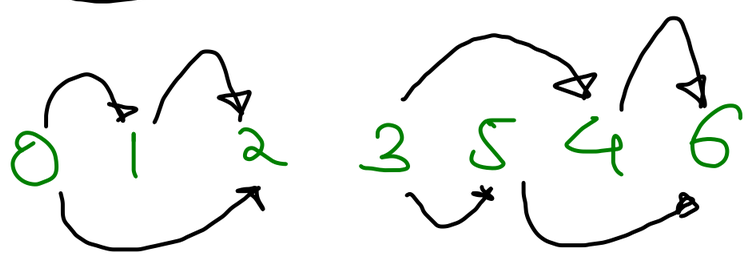


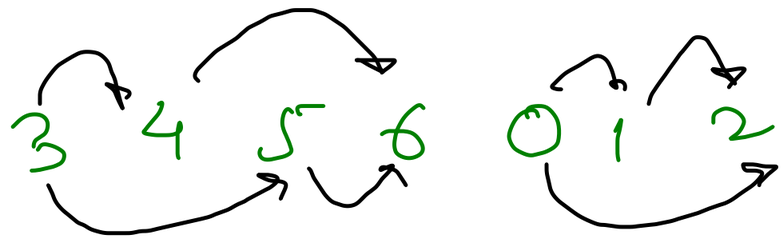

Permutation of vertices such that $u \to v$, u must occur before v

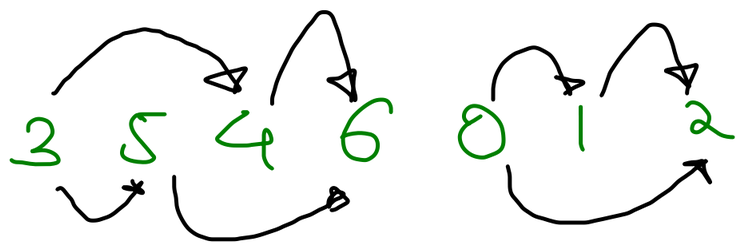


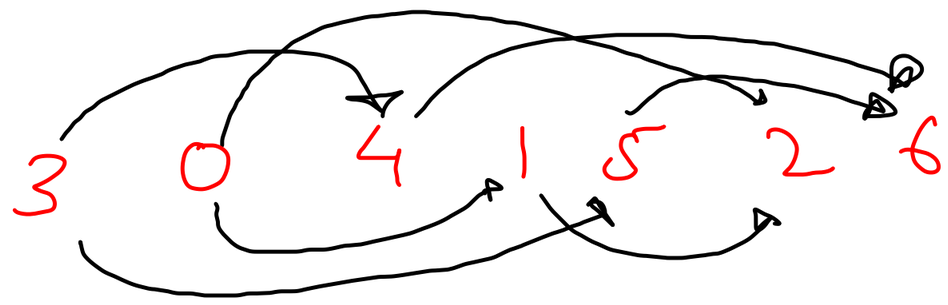



① 


② 


③ 


④ 


⑤ 

# Applications of Topological Sorting

FAANG

1. Prerequisites / Course Schedule {Medium}
2. Alien Dictionary {HARD}
3. Minimum Time Job Scheduling
4. Mother Vertex
5. Reconstruct Itenary {Hard}

1. Finding cycle in a graph {Cycle Detection}
2. Operation System deadlock detection
3. Dependency resolution
4. Sentence Ordering
5. Critical Path Analysis
6. Course Schedule problem
7. Other applications like manufacturing workflows, data serialization and context-free grammar.

# Algorithm {Depth First Search, Stack}

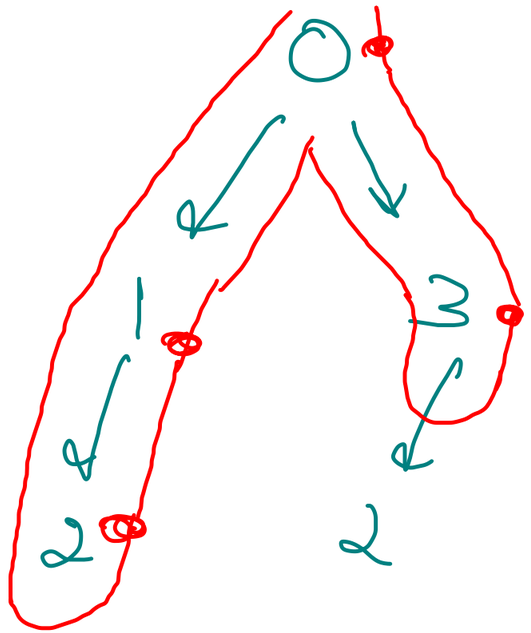
→ on connected components

→ postorder

```
public static void DFS(ArrayList<Edge>[] graph, int src,
            boolean[] vis, Stack<Integer> topoSort){
    vis[src] = true;

    for(Edge e: graph[src]){
        if(vis[e.nbr] == false){
            DFS(graph, e.nbr, vis, topoSort);
        }
    }

    topoSort.push(src);
}
```

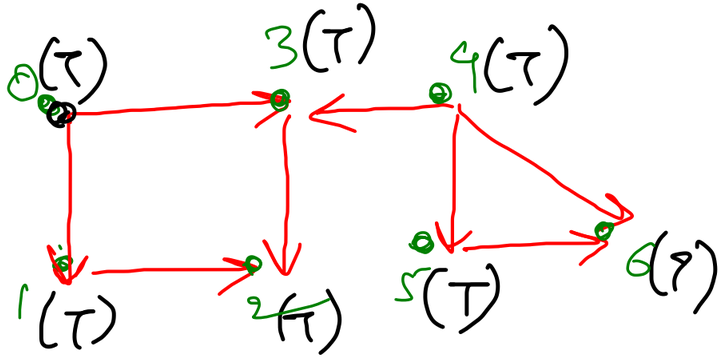
DFS order does not matter

{Any node can be source}

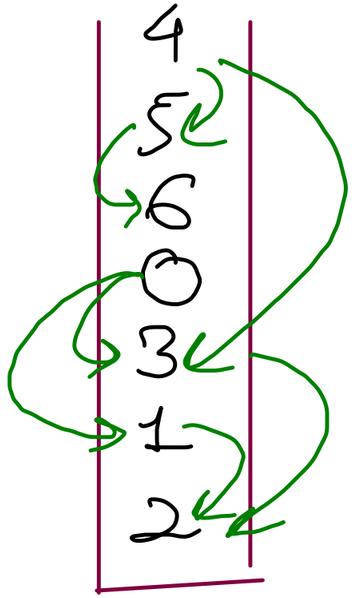


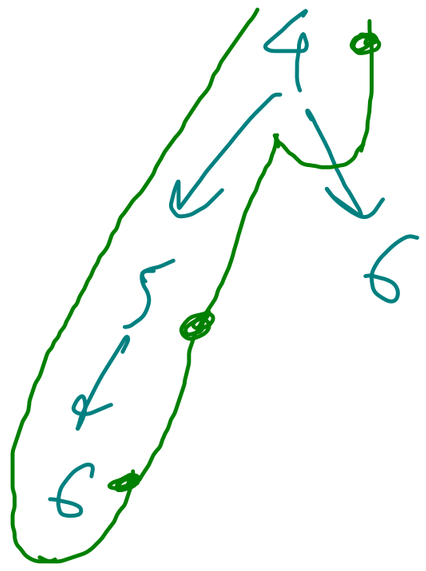

FAITH

hum jinpar dependent honge wo already stack me honge




```
boolean[] vis = new boolean[vtces];
Stack<Integer> topoSort = new Stack<>();

for(int i=0; i<vtces; i++){
    if(vis[i] == false){
        DFS(graph, i, vis, topoSort);
    }
}
```



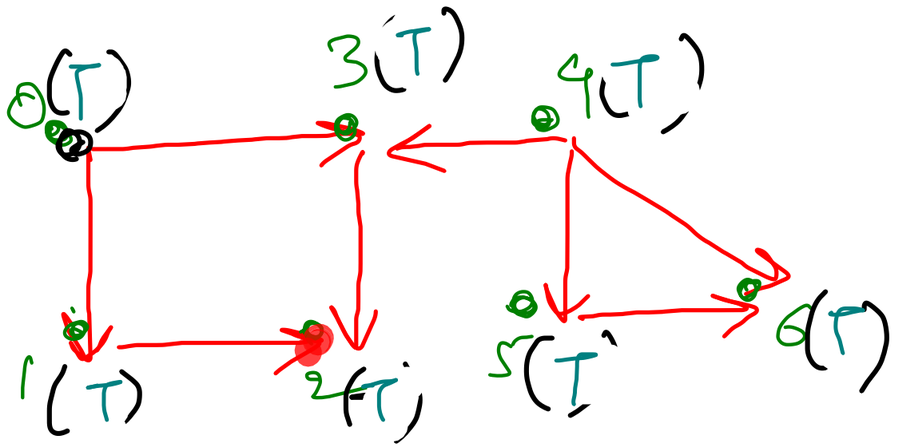

① Pre → Print

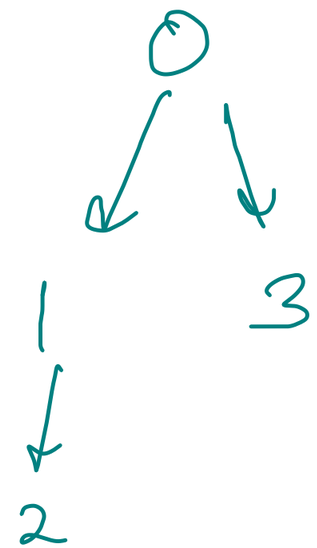
② Post → Print

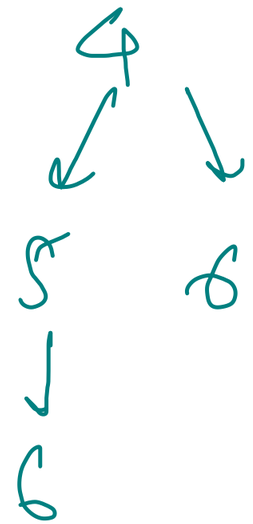
③ Stack → Pre

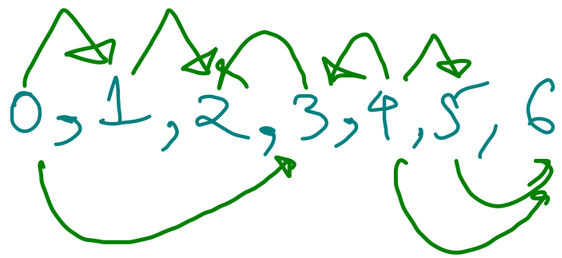
④ Stack → Post

# Preorder print

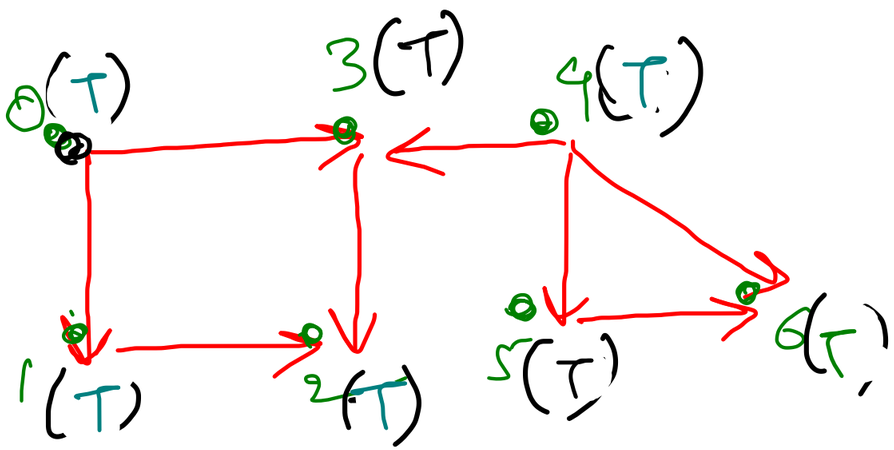


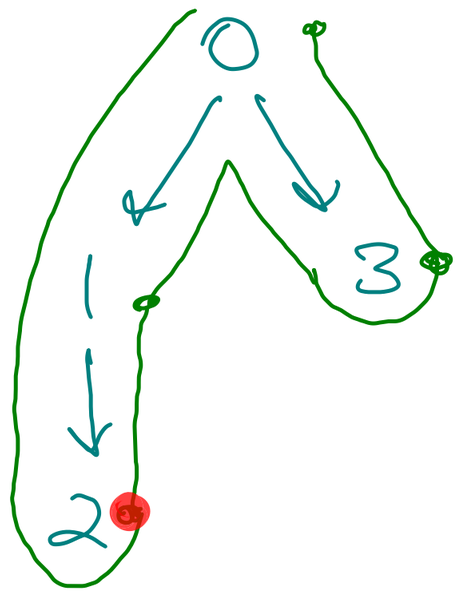



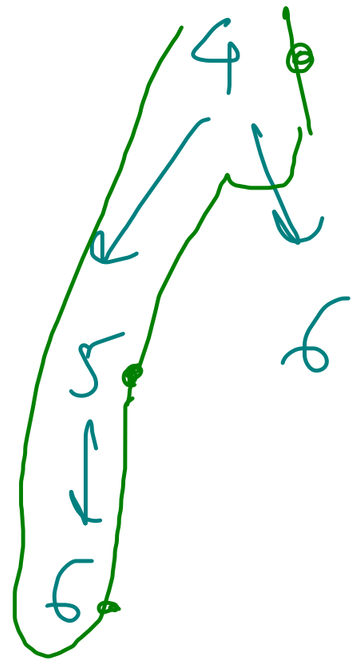



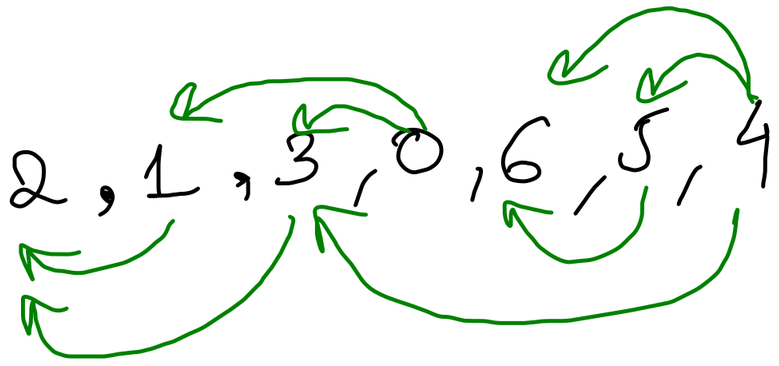



Postorder print

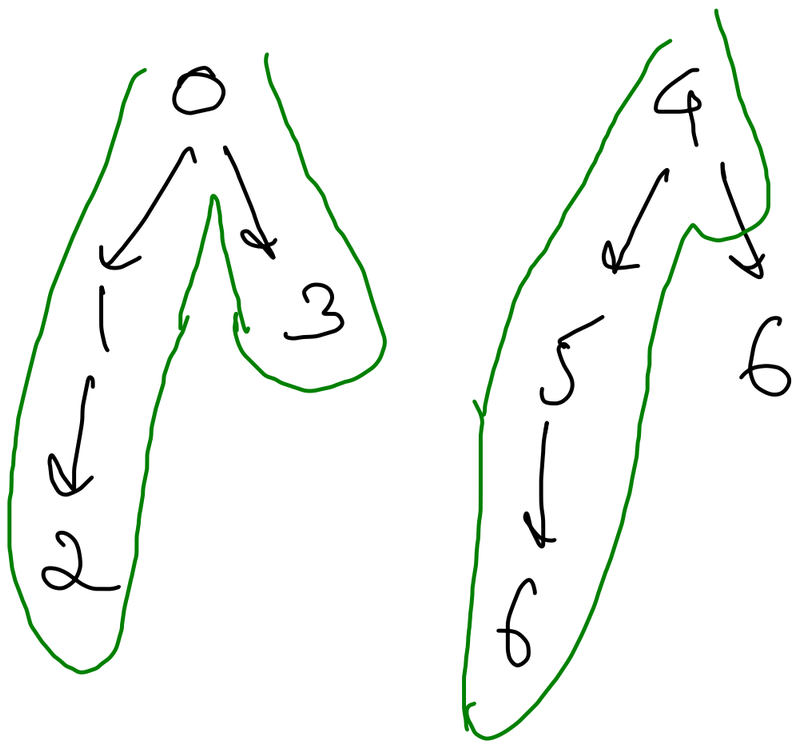


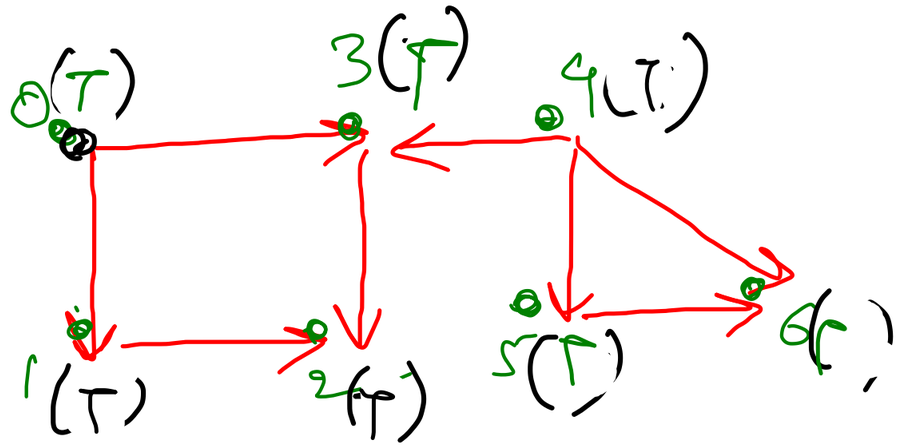



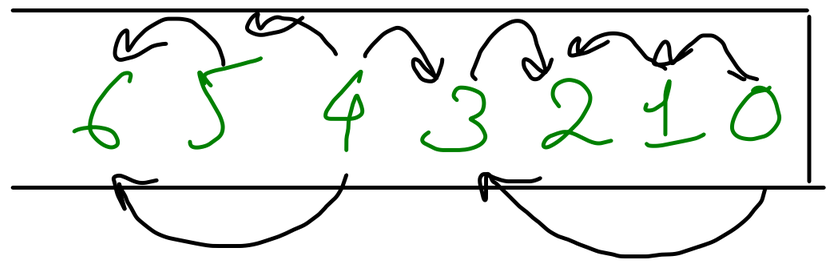



③ Stack push in preorder



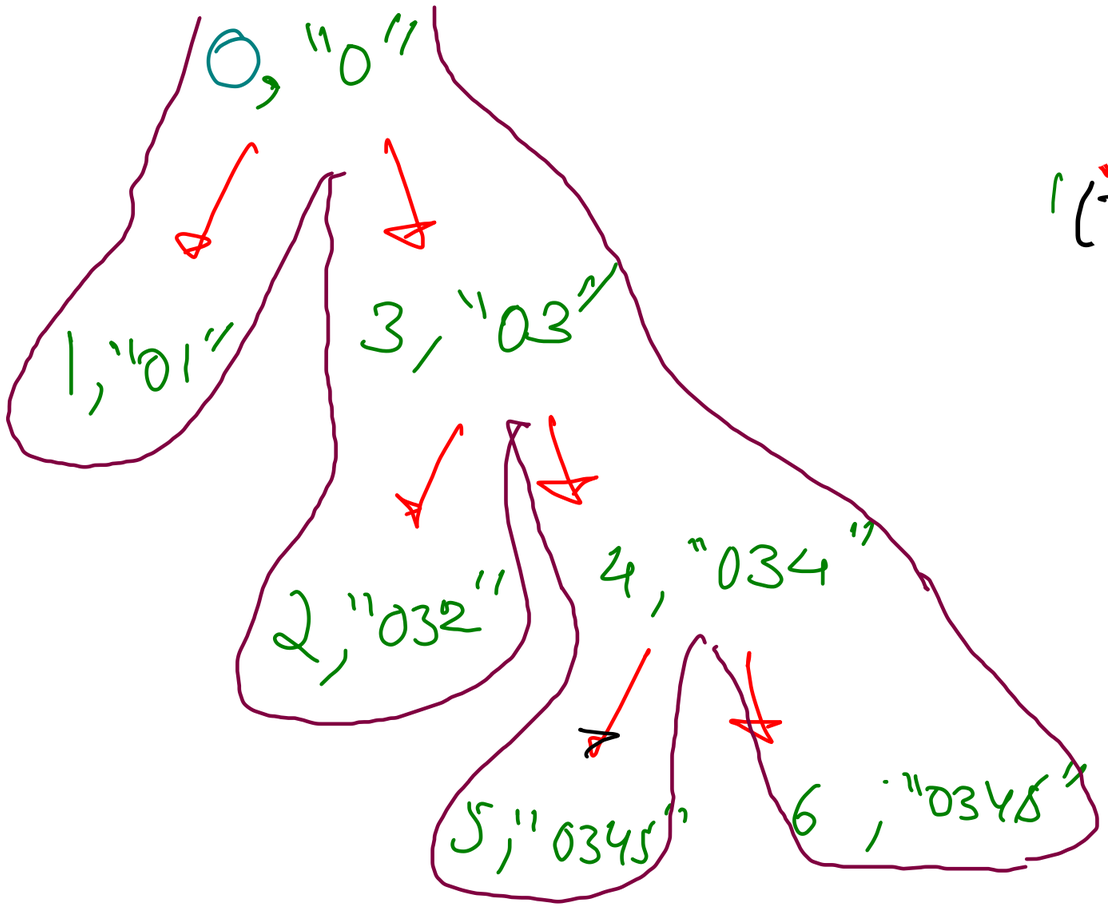



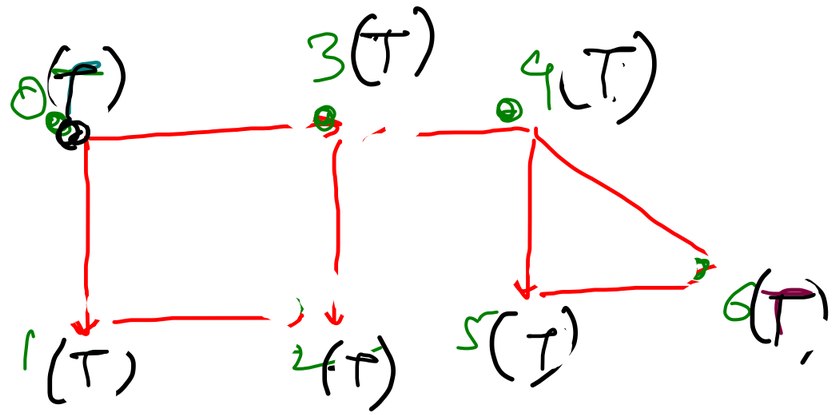



Iterative → Stack DFS

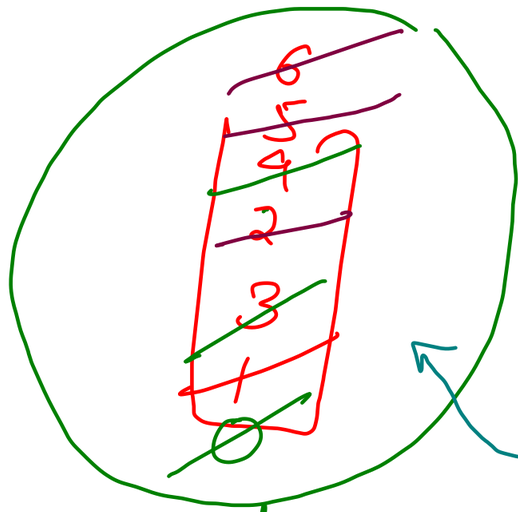
Recursion { Reverse Inorder }

O(V+E)




stack overflow




func^n call stack

10000 - 20000

Level ②

Backlogs → level ①

fjp-2

Foundat^n batch (level 0) {self prep}

3-4 referral {MS, AM} + 500-600

128. Oct 16 Resume Building

Flex 3 { off-campus }

level 2

Dev

interview prep {Mock interview}

Competitive programming

Development

1-2 decent projects

Project idea

Tech

Minimizing cash-flow ← DS-oriented items

① MERN

② Mobile → Kotlin, Cross-platform, ios

③ ML, AI

Master Course

DSA

level ①, level ②, level ③

B

love babbar sheet

short time

level (2) → leetcode Premiums

(1) Recursion & Backtracking

(2)

(IP) → Arrays (1D, 2D)

→ Strings

→ Sorting

Searching {BS}

2 pointer

Greedy algo

Sorting → OS MS

leetcode → virtual contest

JSP-4

FJP-2

(3) Basics → S&Q, LL, Trees

(4) Advanced DS → Hashmap, Heap, Graphs

Advanced DS → {1 week} 21st Nov

(5) Algorithms → DP

Bit Manipula$^{n}$

Trie

Ideal Revision

L1 L2 L3 L4 L5 L6

1. video
2. code-submit
3. short important pbs
4. codeforces & leetcode

#leetcode

↳ ranks

PBC

SDE

DSA  Dev

6 - 12 {LPA}

Startups , Amazon Aws, ----

DSA {Hiring}

Hackathon / Interviews → DEV

21st Nov

level 2

level 1

C++

# System Design

SDE - 1

HLD

LLD

Java { Design Patterns }