

Graphs level 1

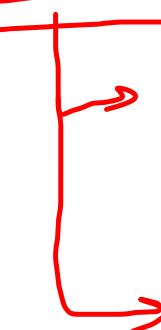
Data Structures

- Arrays & Strings
- ArrayList
- Stack
- Queue
- linked list
- Generic Tree
- Binary Tree
- Binary Search Tree

→ Hashmap

→ Heap

graph

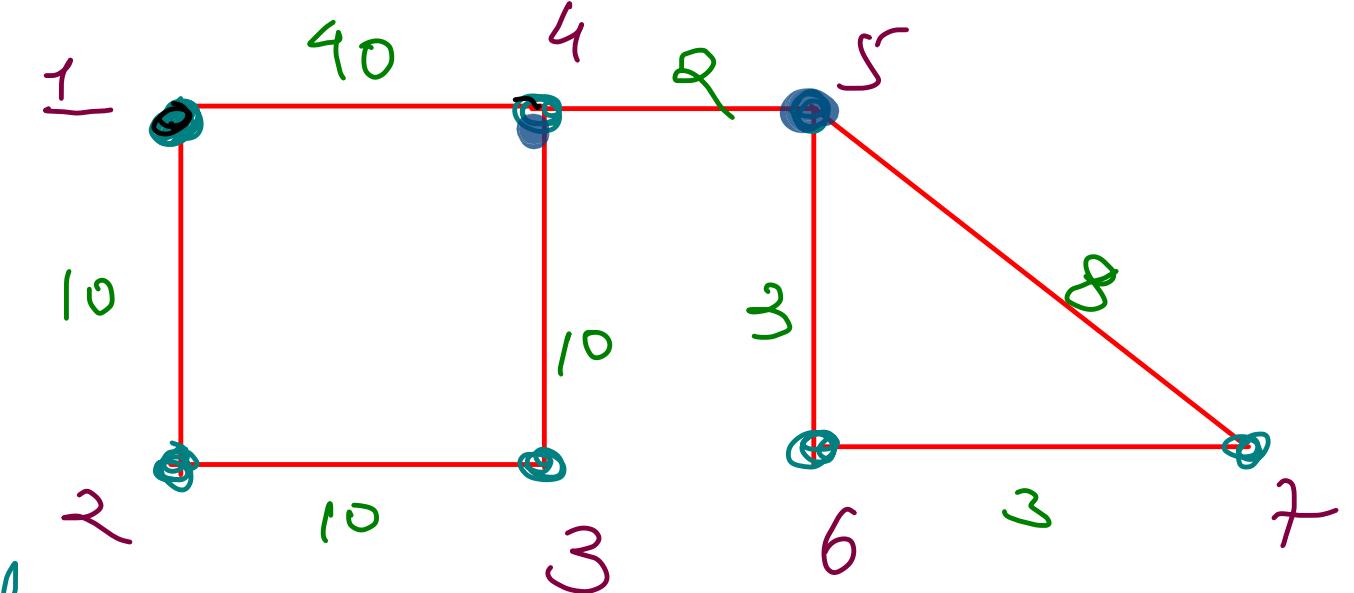


graph algorithm

graph application

Terminologies

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

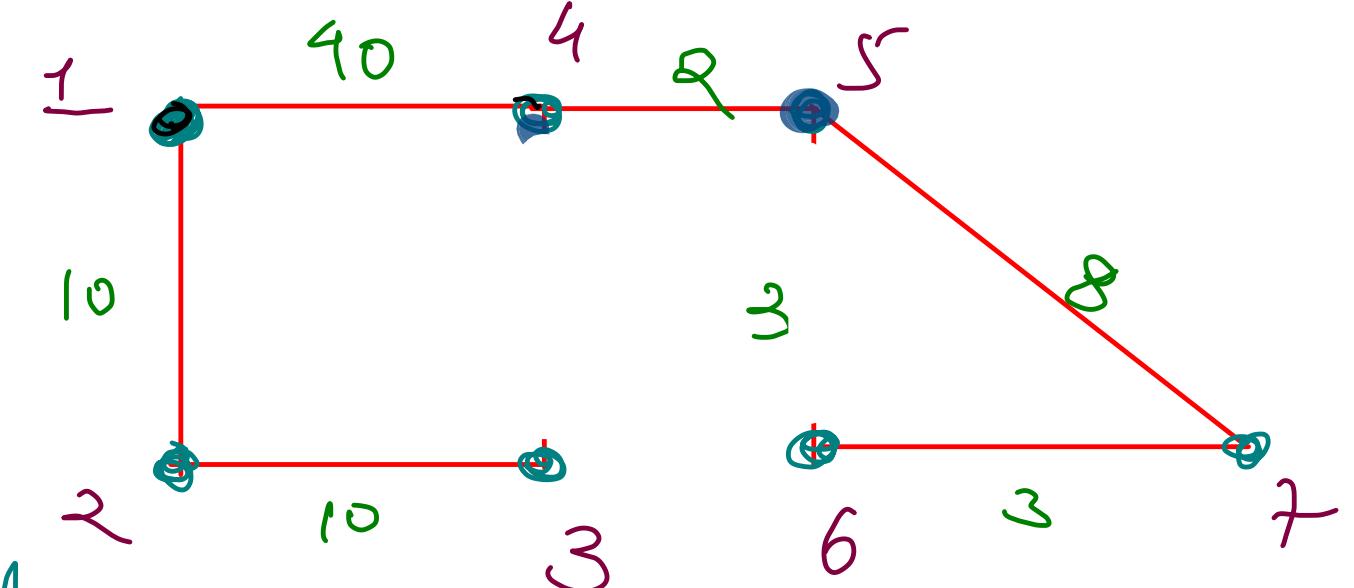


Applications

- ① Google maps
 - ② Social Media
 - ③ Airline Management
- Shortest distance* → Min cost to connect all cities

Terminologies

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



Applications

- ① Google maps → shortest distance
- ② Social Media → Min cost to connect all cities
- ③ Airline Management

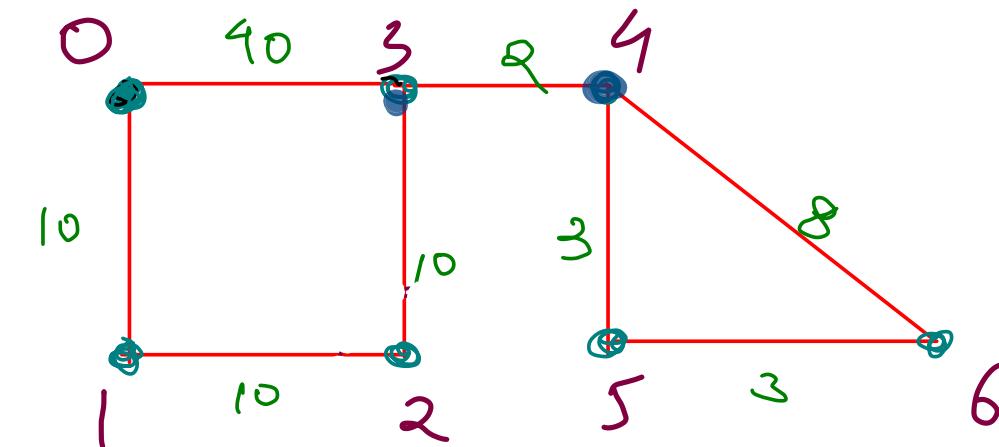
$O(n)$

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



Vertices $\rightarrow n$

Edges $\Rightarrow \frac{n^2}{2}$

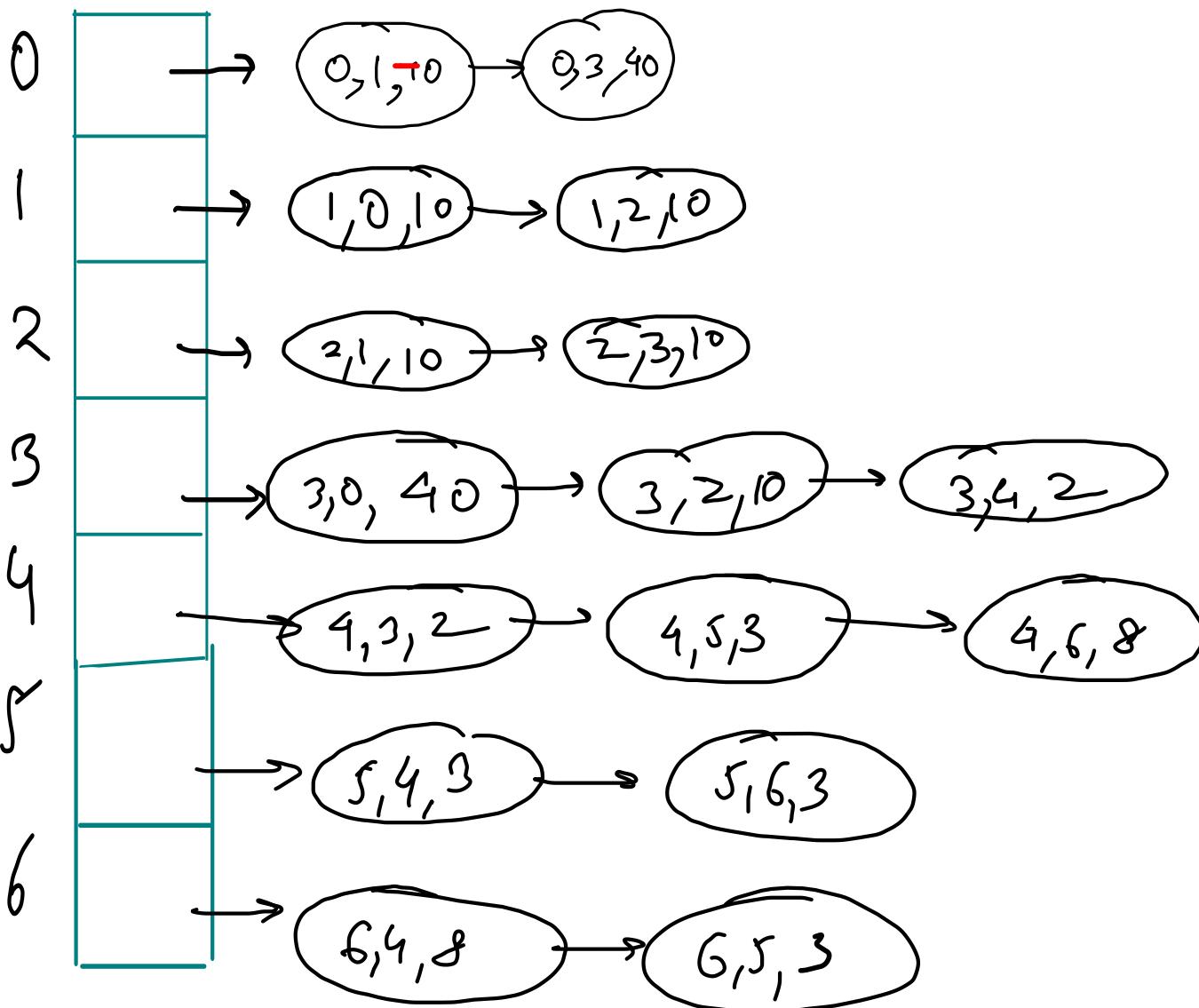
$$= \frac{n \cdot (n-1)}{2}$$

$O(n^2)$

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

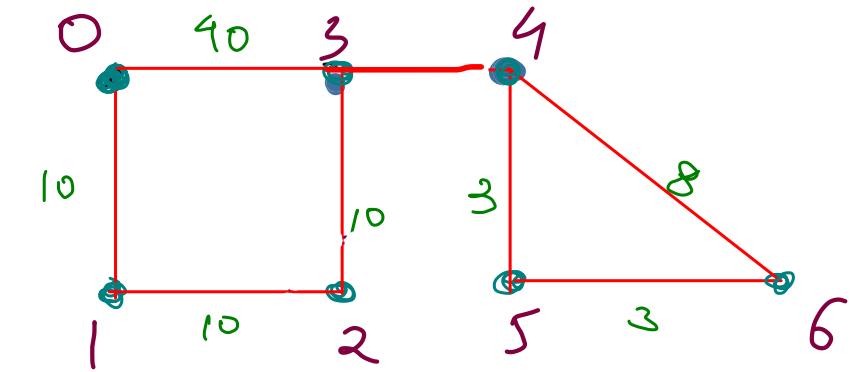


```

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



```

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

```

Edge class

```

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

```

Initialization

```

int edges = scn.nextInt();

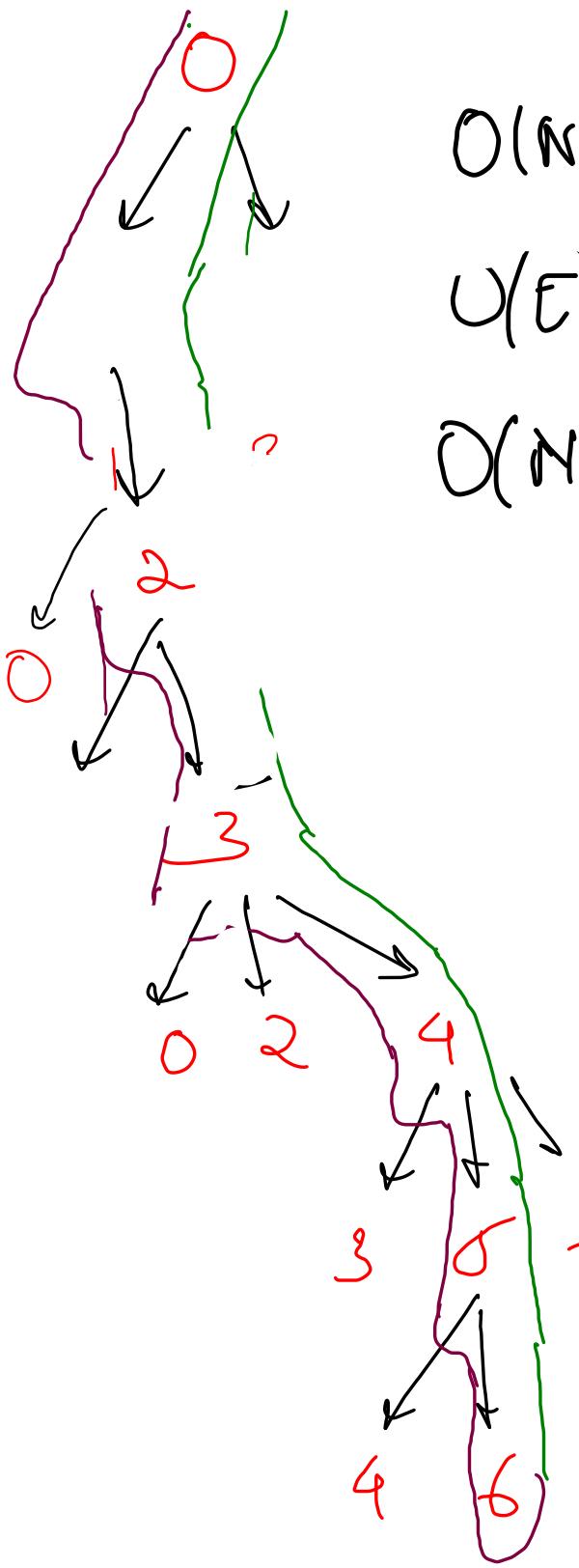
for(int i=0; i<edges; i++){
    int v1 = scn.nextInt();
    int v2 = scn.nextInt();
    int wt = scn.nextInt();

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

```

Construct

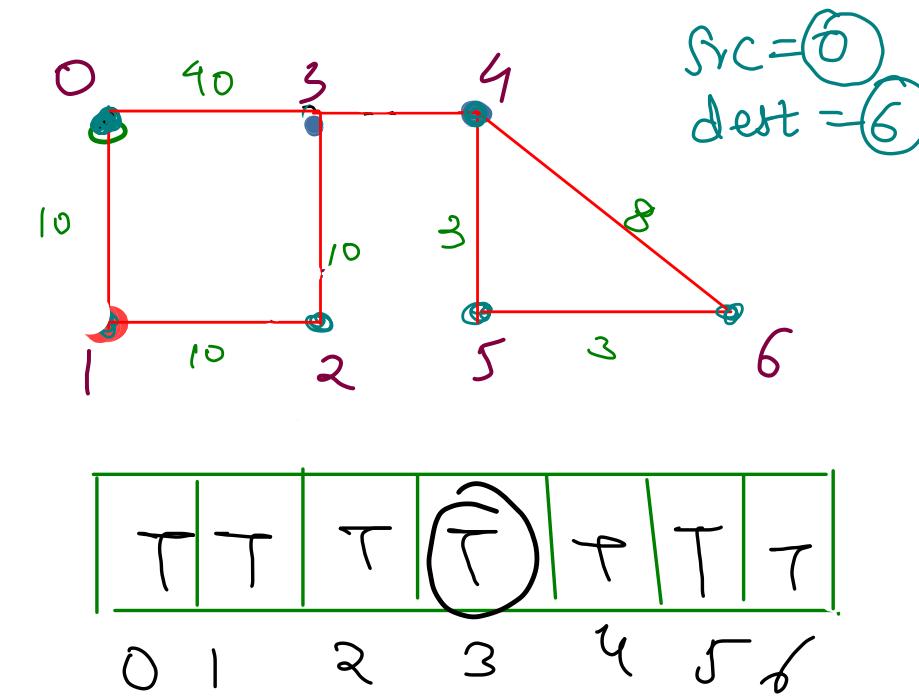
has Path



$O(N)$

$O(E)$

$O(N+E)$



2f 2f 3p

- - -

$\sim O(F)$

```
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) *

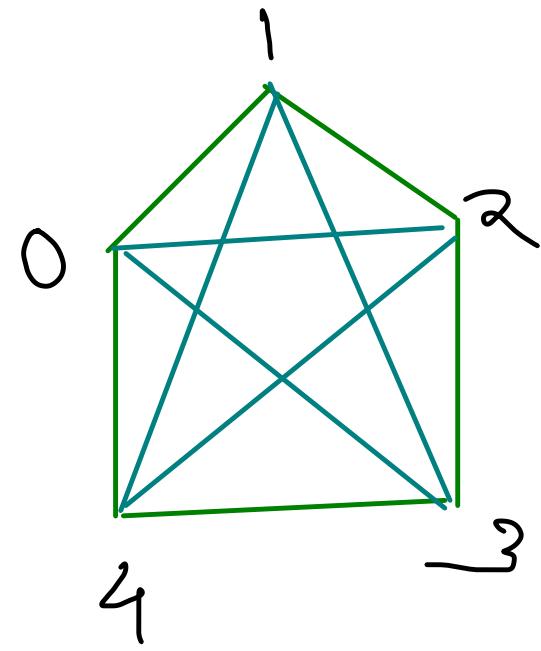
30/30 (100%) answered

- 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%

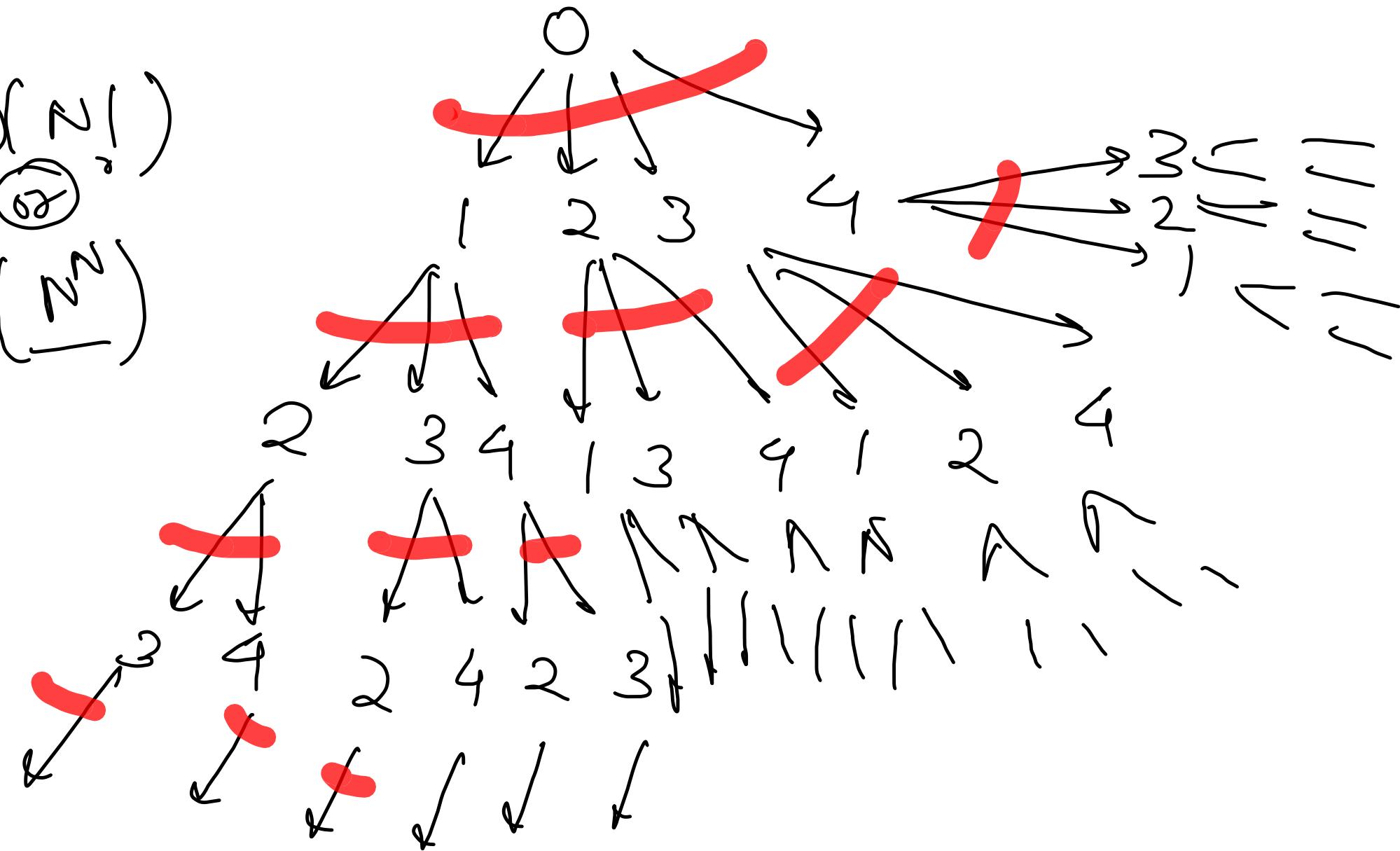
Backtracking \rightarrow ~~Poly~~ ~~non~~ ~~Exponential~~

Recursion

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

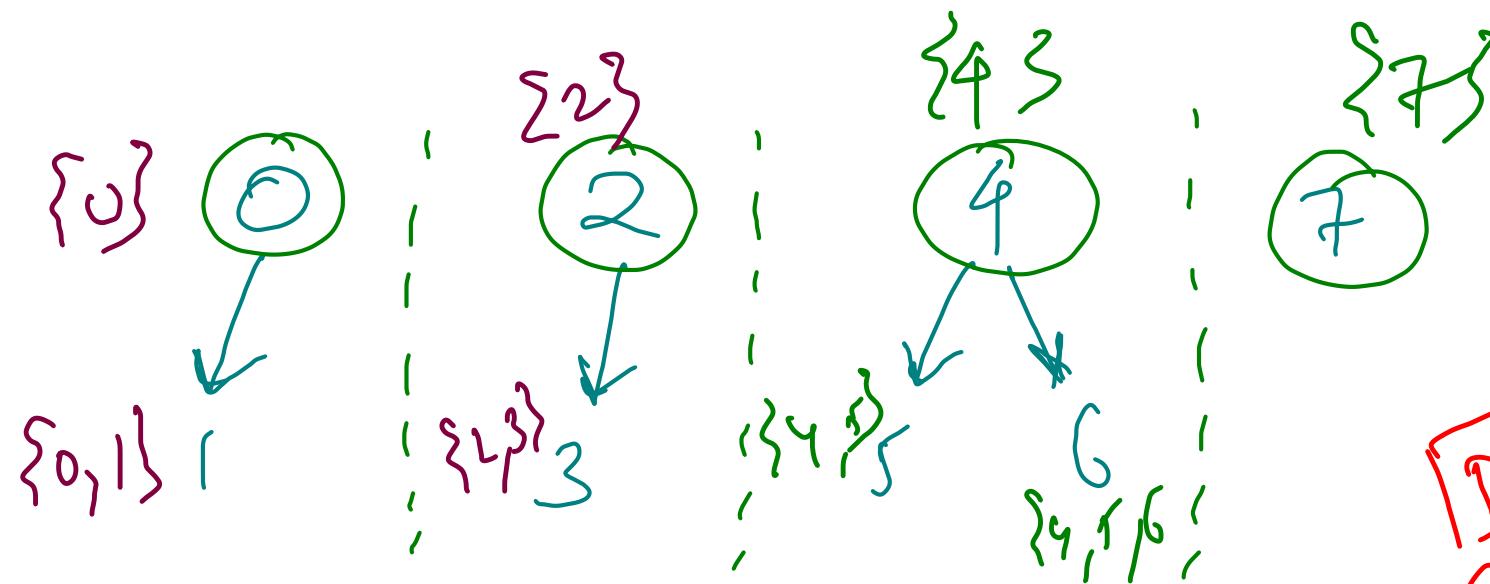


A hand-drawn mathematical expression consisting of two parts. The top part shows a large open parenthesis containing a large letter 'a' followed by a large letter 'n' with an exclamation mark. Below it, a smaller open parenthesis contains a large letter 'a' followed by a large letter 'n' with a question mark.



connected components {undirected}

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



DFS
 $O(n + e)$

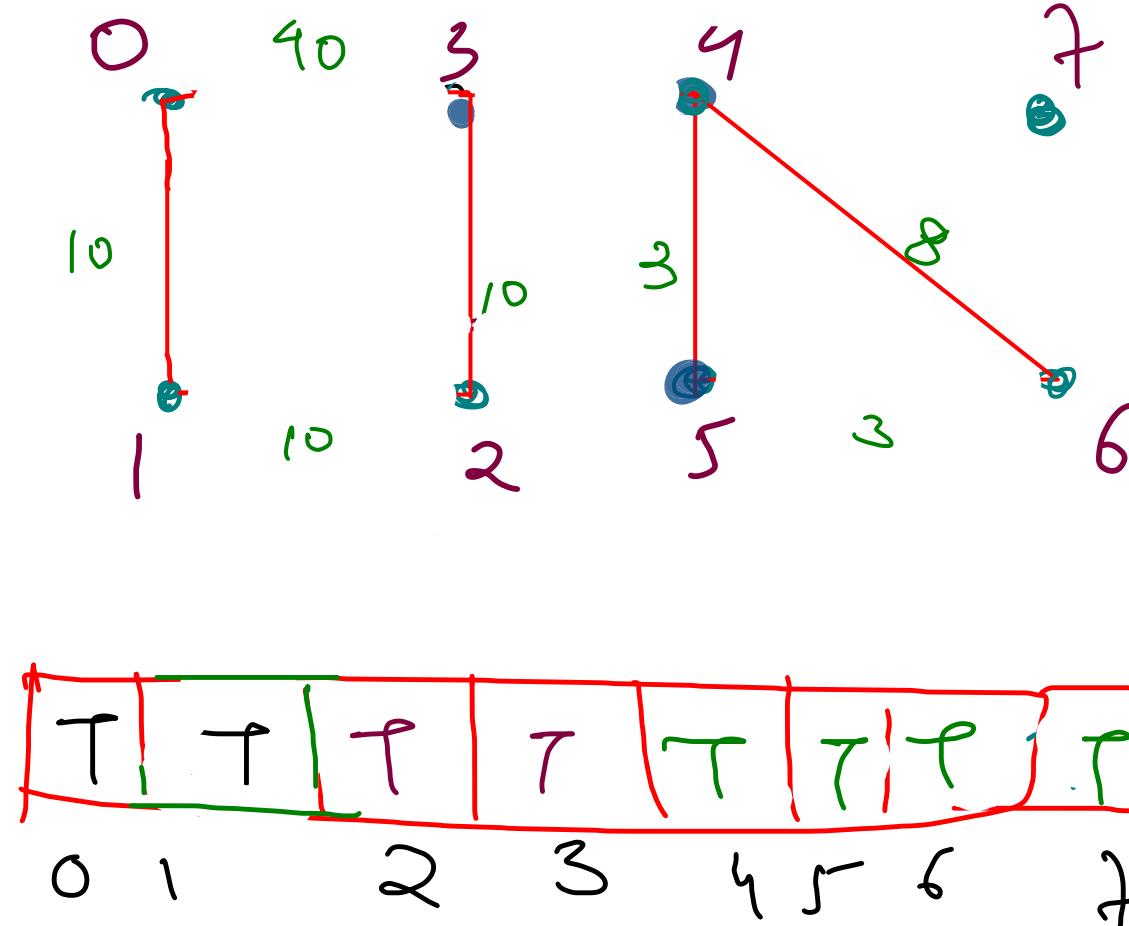
```

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);
        }
    }
}
  
```

```

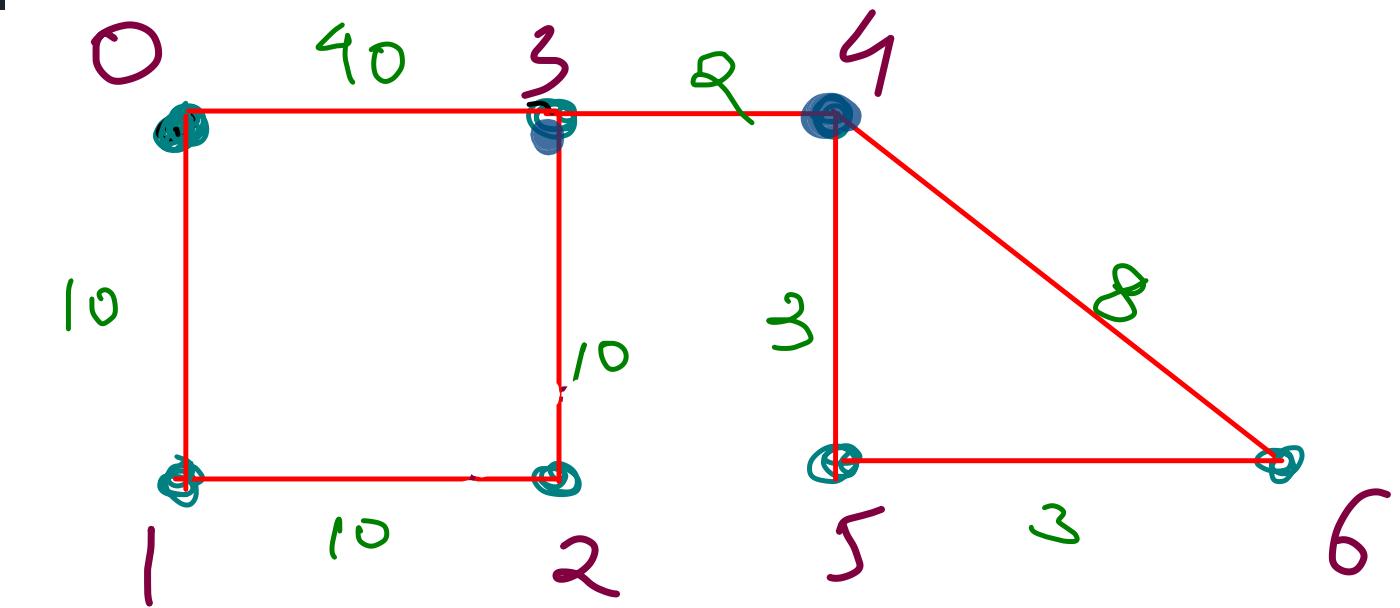
for(int i=0, i<vtces, i++){
    if(vis[i] == false){
        ArrayList<Integer> comp = new ArrayList<>();
        dfs(graph, i, comp, vis);
        comps.add(comp);
    }
}
  
```



Today's Questions

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

{ } *multisource
DFS*



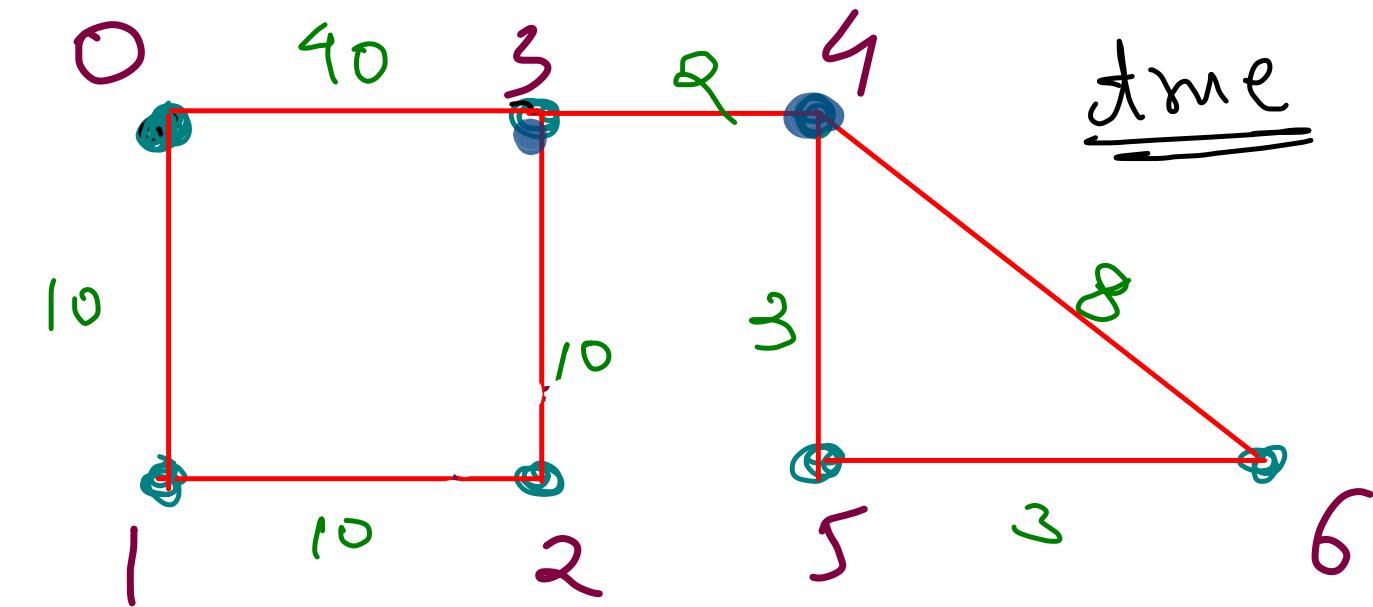
Is Graph connected

Component

= 1 : connected

> 1 : disconnected

= 0 component
{not possible}



{atleast one vertex should be present in graph}

Number of Islands

```
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);  
}
```

for each $\delta \times c$ cell

```
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);
```

no of vertices

Prerequisite :- floodfill, Connected components.

$O(\delta * c)$

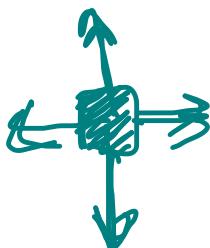
0	1	1	1	1	1	1
0	1	1	1	1	1	1
1	1	1	1	1	1	0
1	1	0	0	0	1	0
1	1	1	1	0	1	0
1	1	1	1	0	1	0
1	1	1	1	1	1	0
1	1	1	1	1	1	0
1	1	1	1	1	1	0

$\phi \times \psi^3$

$0 \Rightarrow$ island

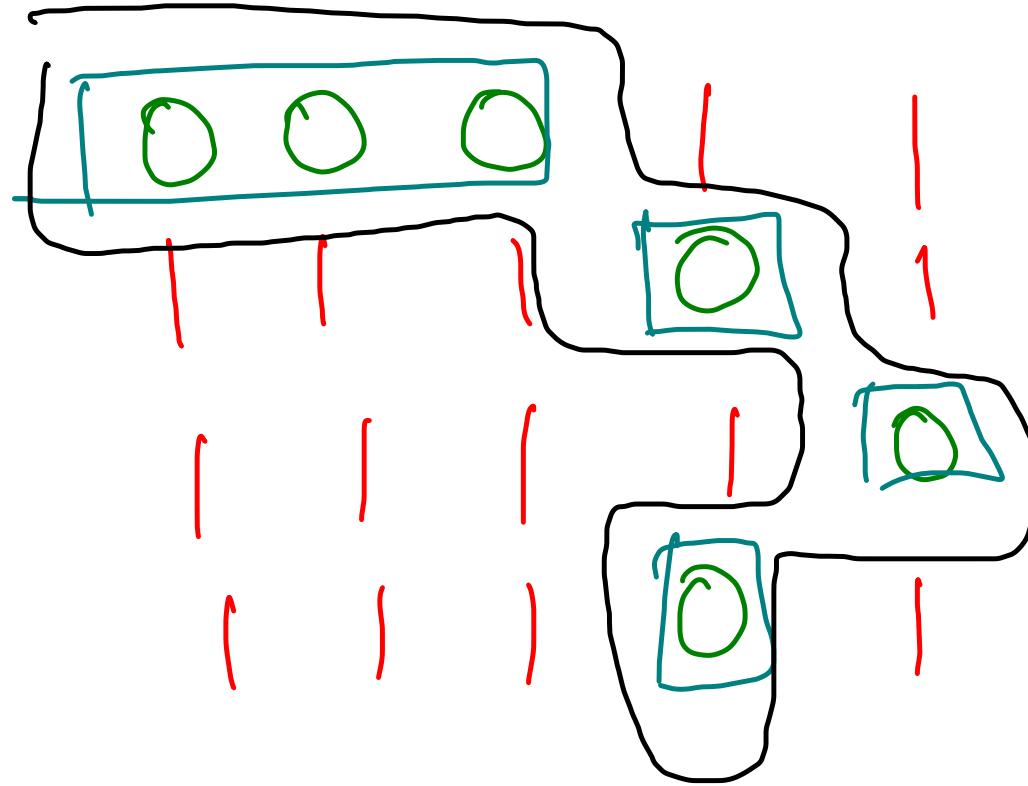
$1 \Rightarrow$ water

4 way
connected



Note

This matrix is
not
adjacency
matrix.



4 way connected $\stackrel{0}{\rightarrow} = 4$

8 way connected $\stackrel{0}{\rightarrow} = 1$

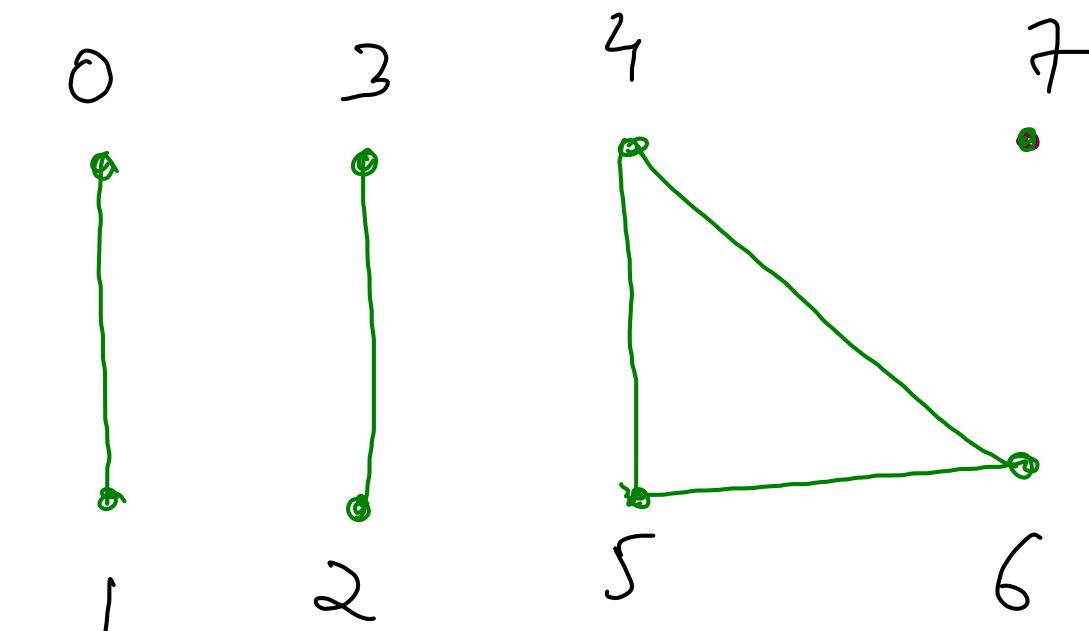
Perfect friends

	$v_1 = 0$	$v_1 = 1$
0	(0, 2)	(1, 2)
1	(0, 3)	(1, 3)
2	(0, 4)	(1, 4)
3	(0, 5)	(1, 5)
4	(0, 6)	(1, 6)
5	(0, 7)	(1, 7)

$v_1 = 2$	$v_1 = 3$
(2, 4)	(3, 4)
(2, 5)	(3, 5)
(2, 6)	(3, 6)
(2, 7)	(3, 7)

$v_1 = 4$	$v_1 = 5$	$v_1 = 6$
(4, 7)	(5, 7)	(6, 7)

$v_1 = 7$
no pair



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

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

$$= 12 + 8 + 3 + 0 = 23$$

~~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);
```

~~Approach 1~~

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

for(int size: comps){
    remVtces -= size;
    countWays += remVtces * size;
}

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

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

Src = 0, Dest = 6

val = 40

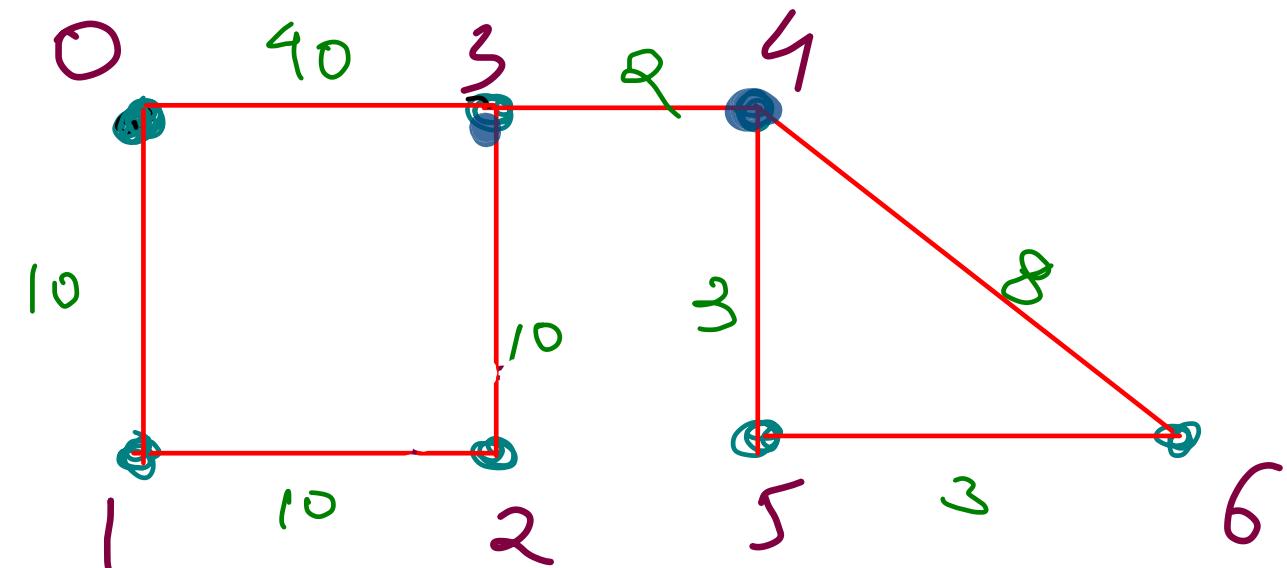
smallest :- 0123456 @ 38

longest :- 03456 @ 50

ceil(40) :- 03456 @ 40

floor(40) :- 0123456 @ 38

kthlargest :- 012346 @ 40
k = 3



0123456 :- 38

012346 :- 40

03456 :- 48

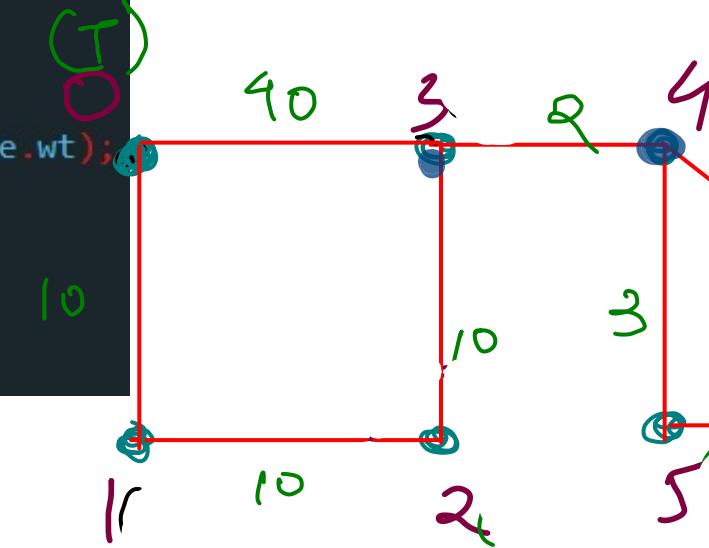
0346 :- 50

```

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;
}

```

$0, "0", 0$
 $1, "01", 10$
 $2, "012", 20$
 $3, "0123", 30 \rightarrow 4, "01234", 32$

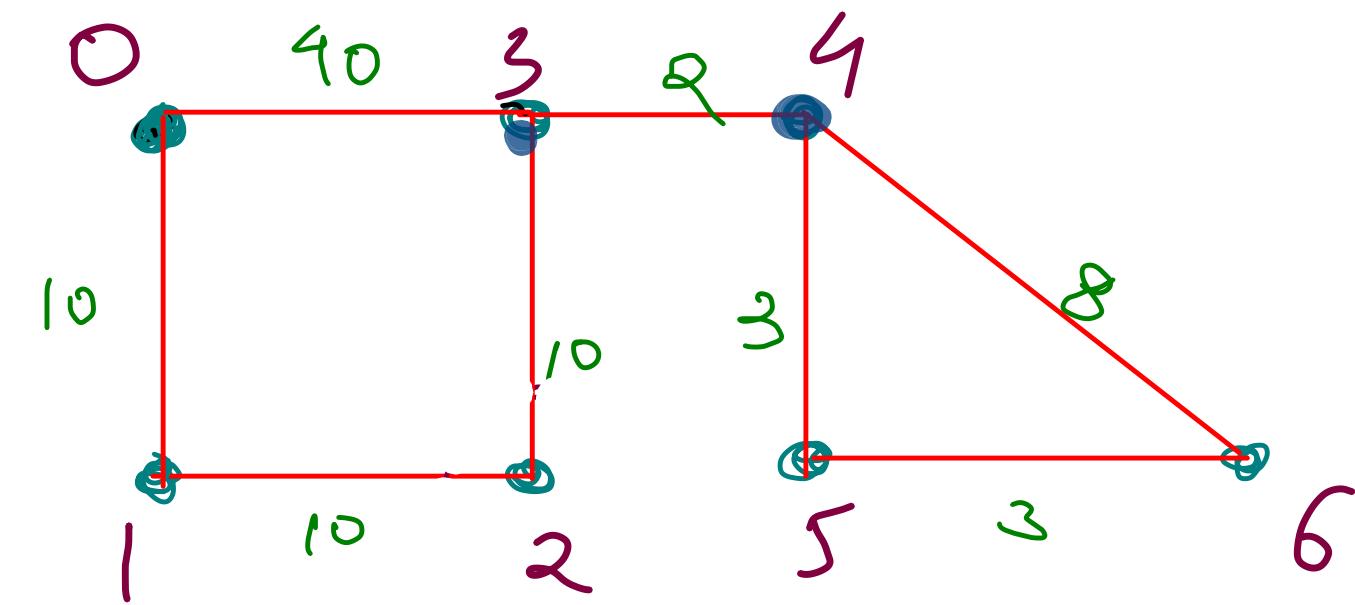


$src = 0, dest = 6$
 $val = 40$

$5, "012345", 35$
 $6, "012346", 40$
 $6, "0123456", 38$

Today's Target { Lecture -3 Graphs }

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



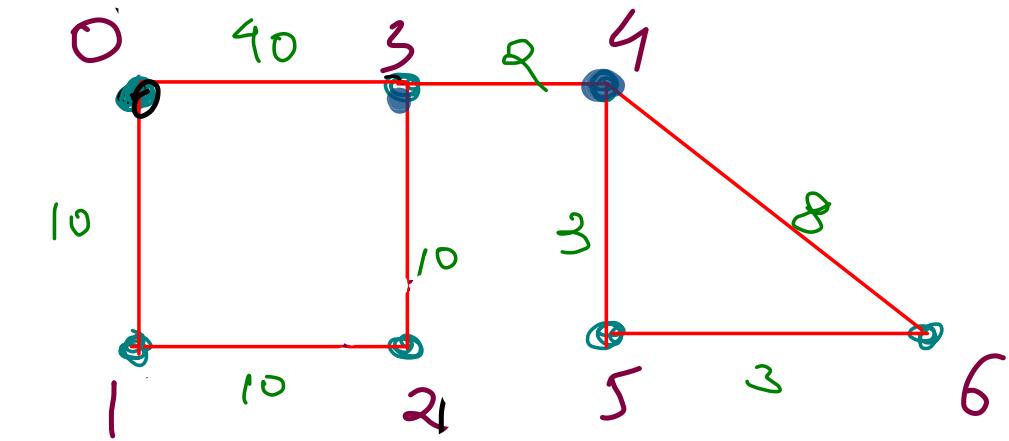
~~Worst case~~ { Exponential } $\rightarrow O(N^N)$
 $O(N!)$

```
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;

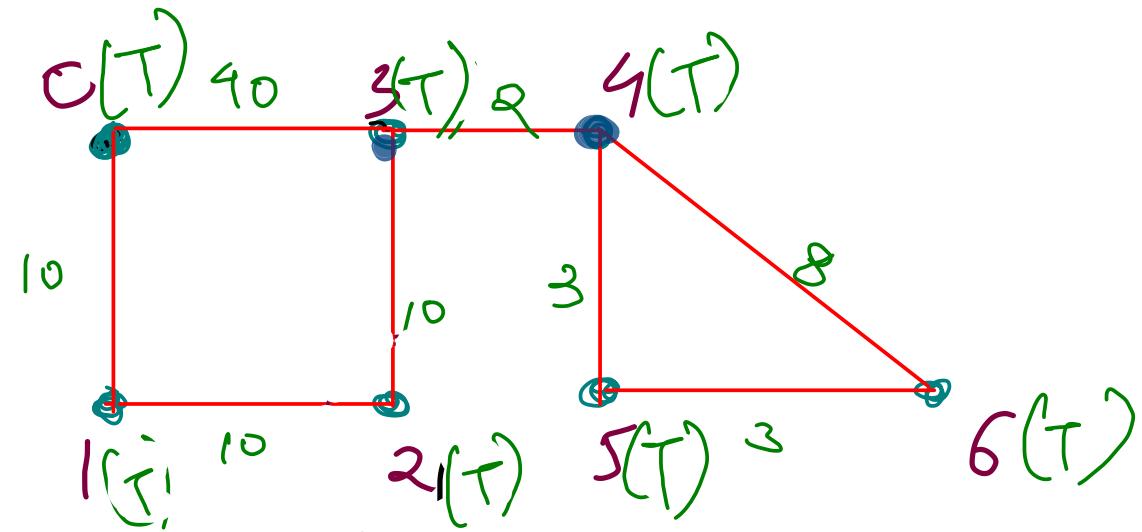
    for(Edge e: graph[node]){
        if(vis[e.nbr] == false){
            dfs(graph, e.nbr, visCount, vis, pathsofar + e.nbr);
        }
    }
    vis[node] = false;
}
```



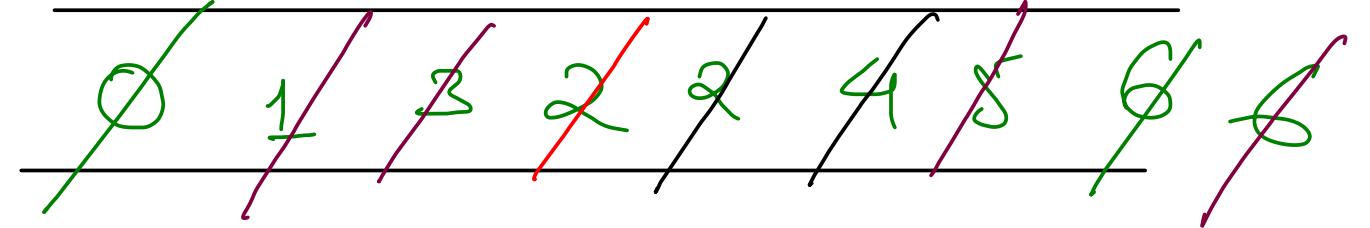
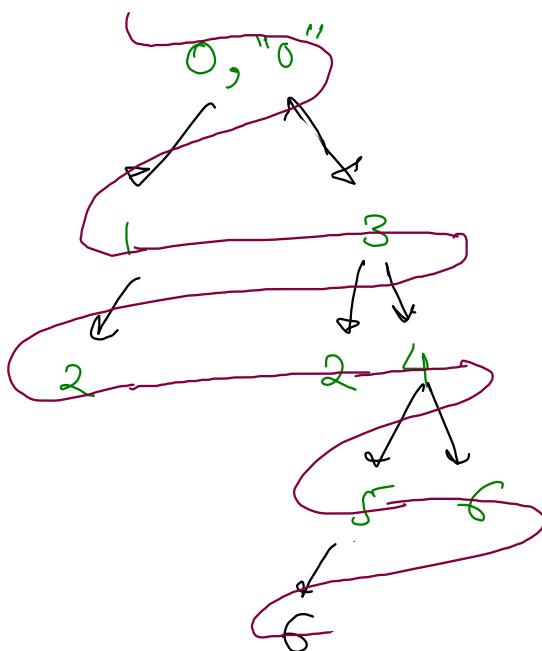
Hamiltonian Path & Cycle
 visits all vertices
 in its path
 without repeating
 any vertex

Edge b/w
 Starting &
 ending
 vertex

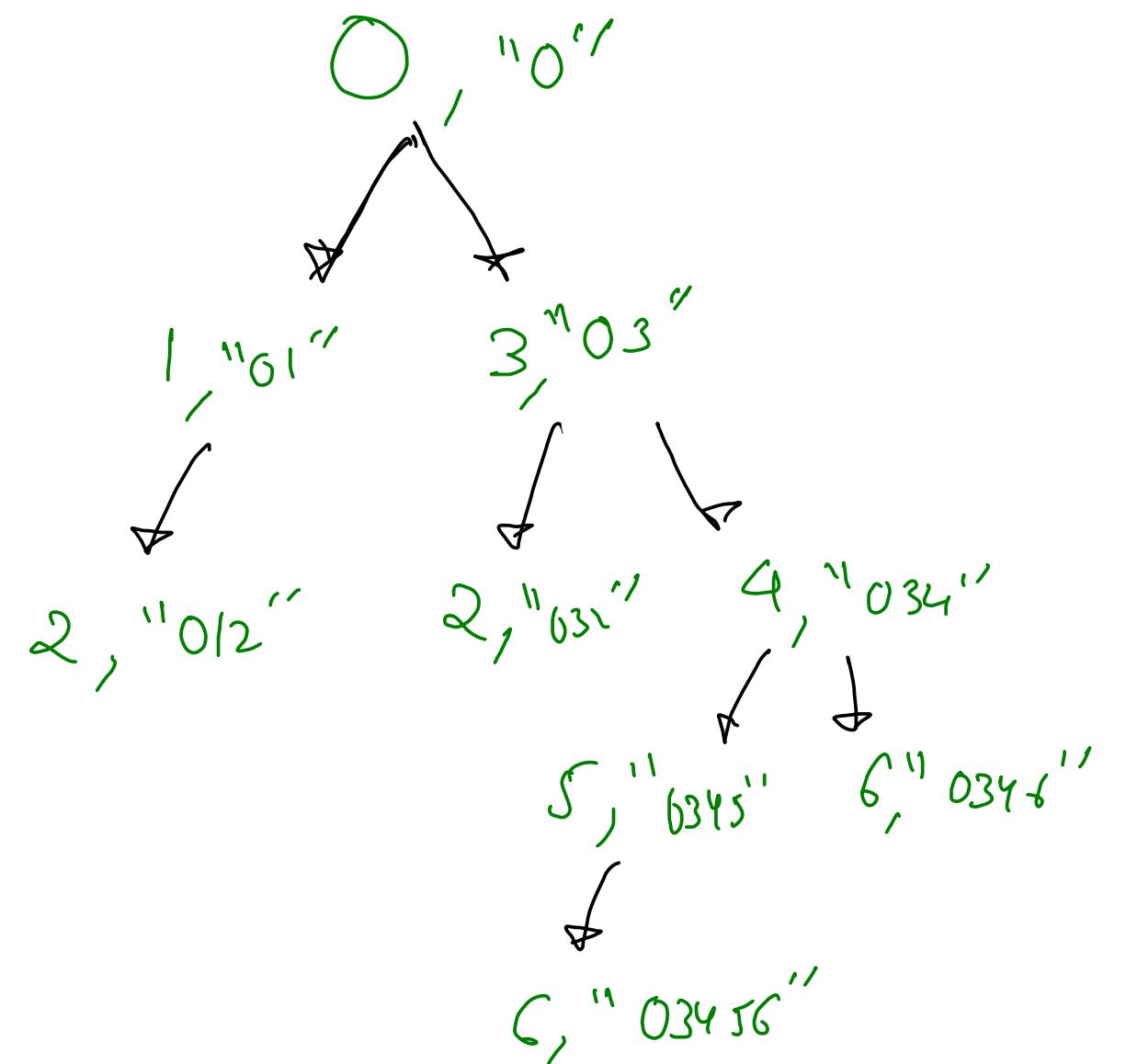
Breadth First Traversal



$0 @ 0, 1 @ 01, 2 @ 012, 3 @ 034$
 $4 @ 0345, 5 @ 0345, 6 @ 03456$

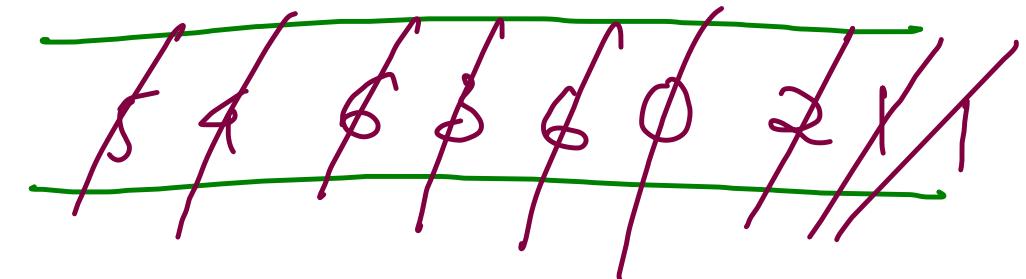


level order

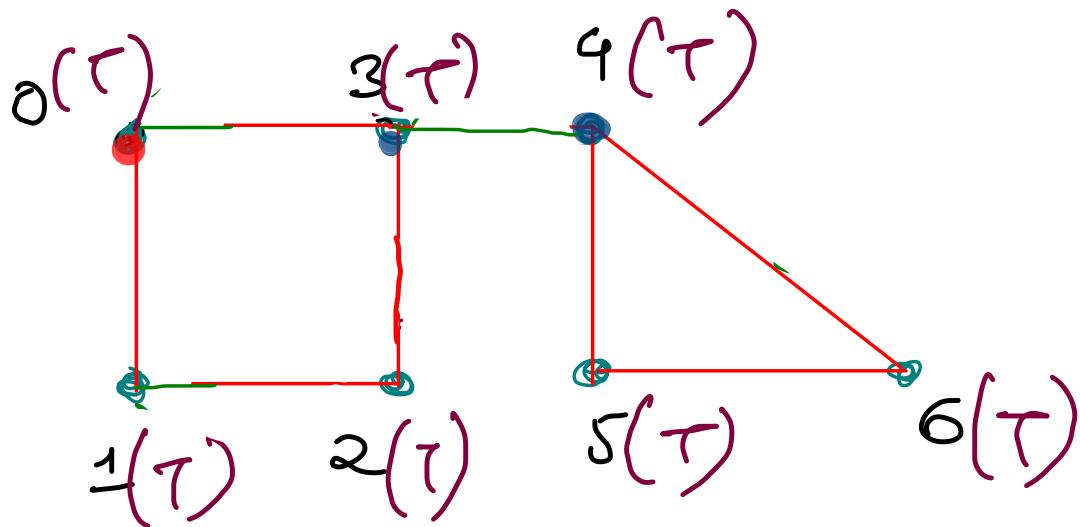


{
 TLE
 already
 marked
 node
 is worked
 upon
 { w, a *? }

$\text{src} = \textcircled{5}$



① Component BFS



Time
 $O(N+E)$

```

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

q.add(new Pair(src, "" + src));
while(q.size() > 0){
    // remove
    Pair curr = q.remove();

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

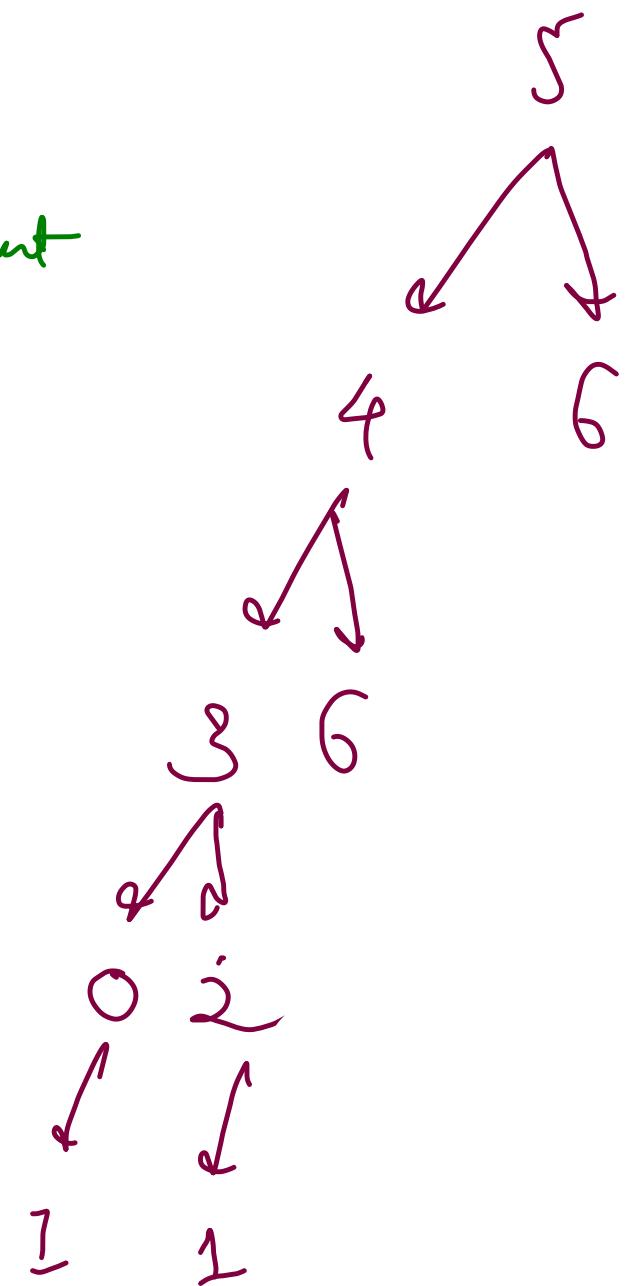
    // mark*
    vis[curr.node] = true;

    // 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 false;
Cycle not present

return true;
Cycle present

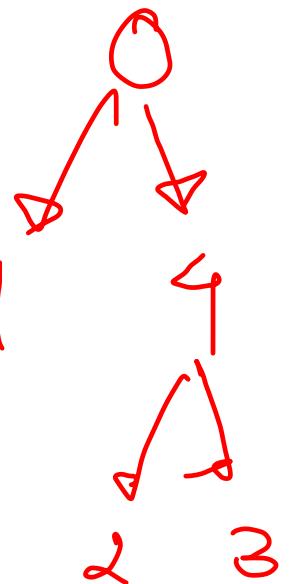
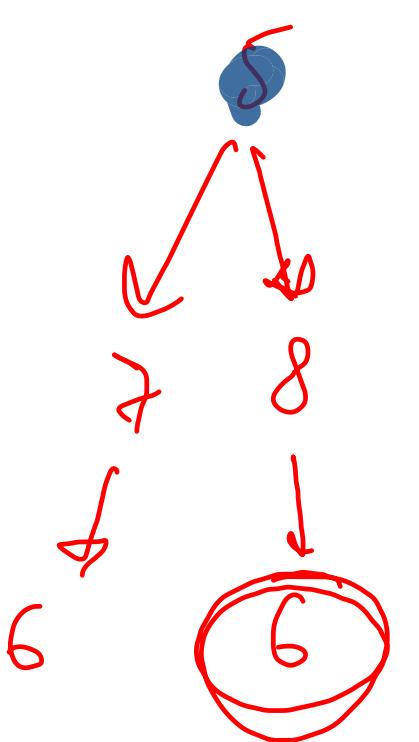
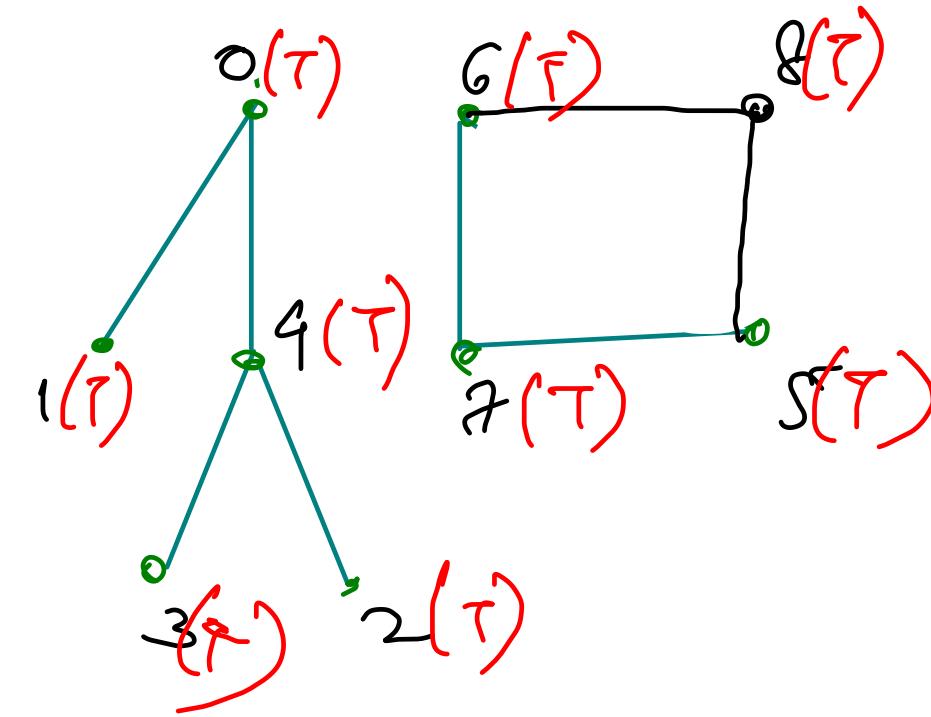


Is Cycle Present in Component

```
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

```
boolean[] vis = new boolean[vtes];  
for(int i=0; i<vtes; 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);
```



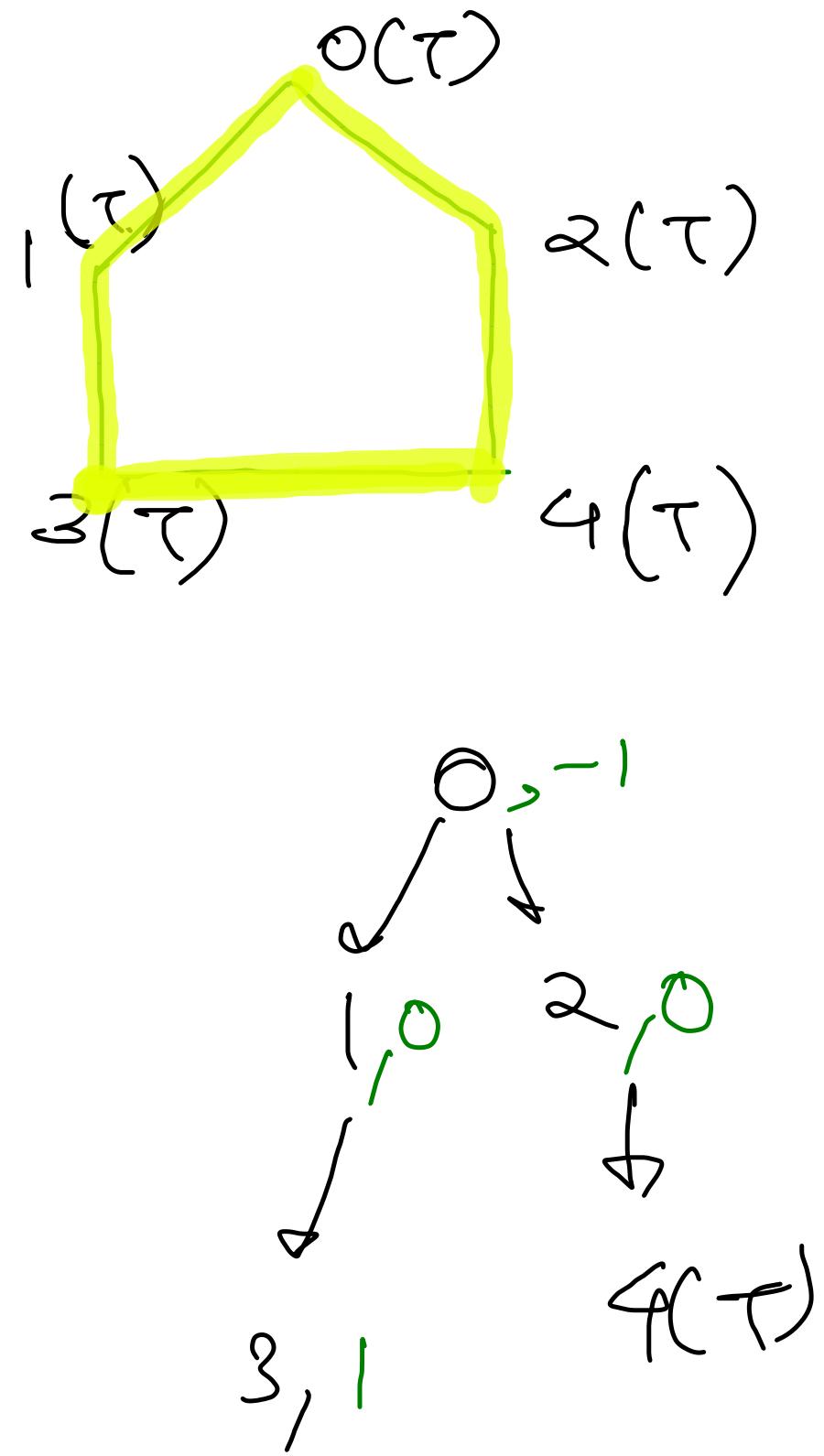
```

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

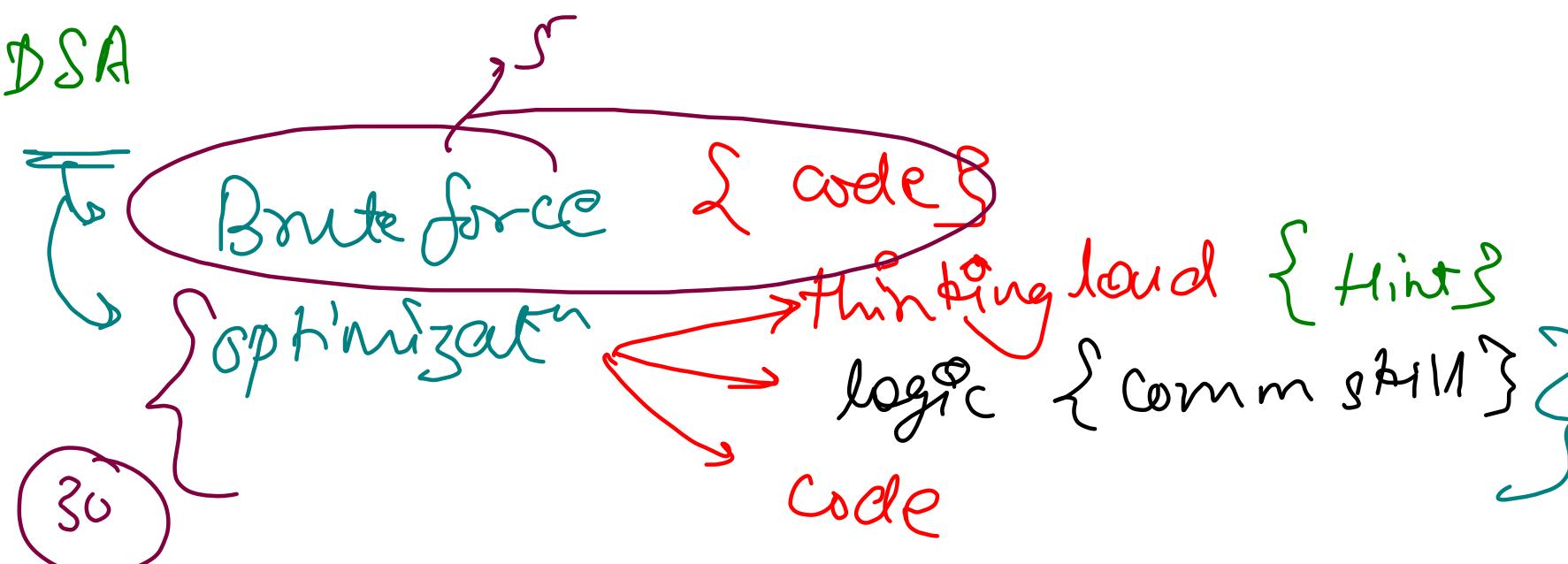
Test case

① Introduce Yourself

↳ 2-3 mins

5-10

② DSA



→ Preparing
→ Interview Bit
→ Ramp

Mock Interview

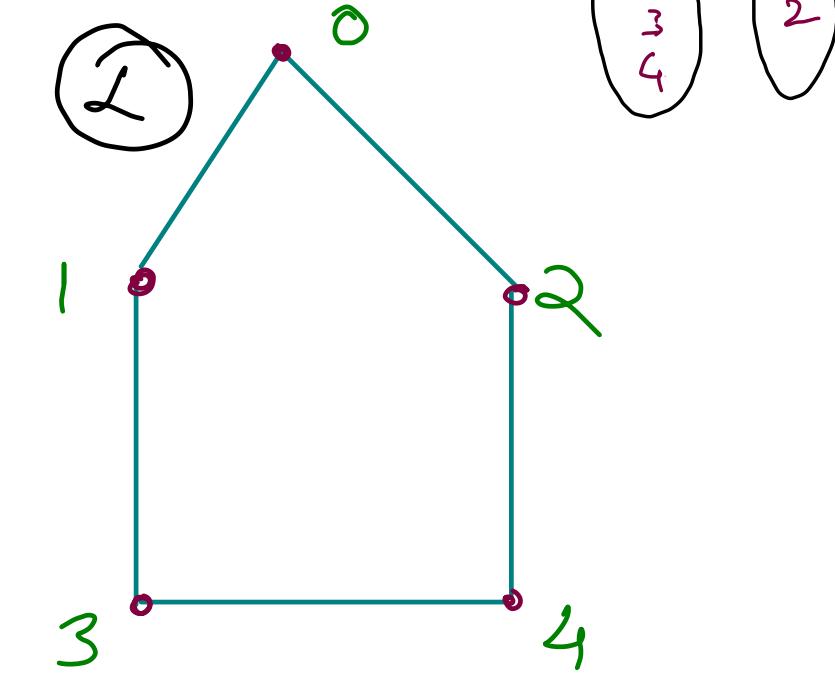
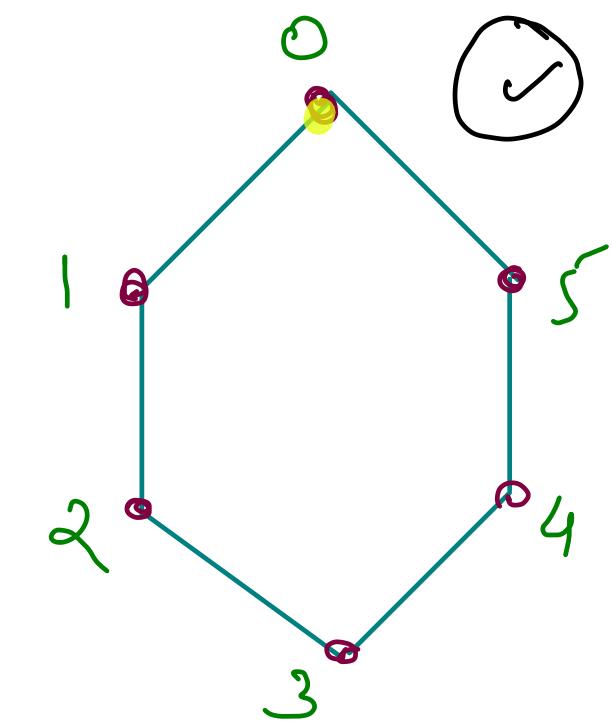
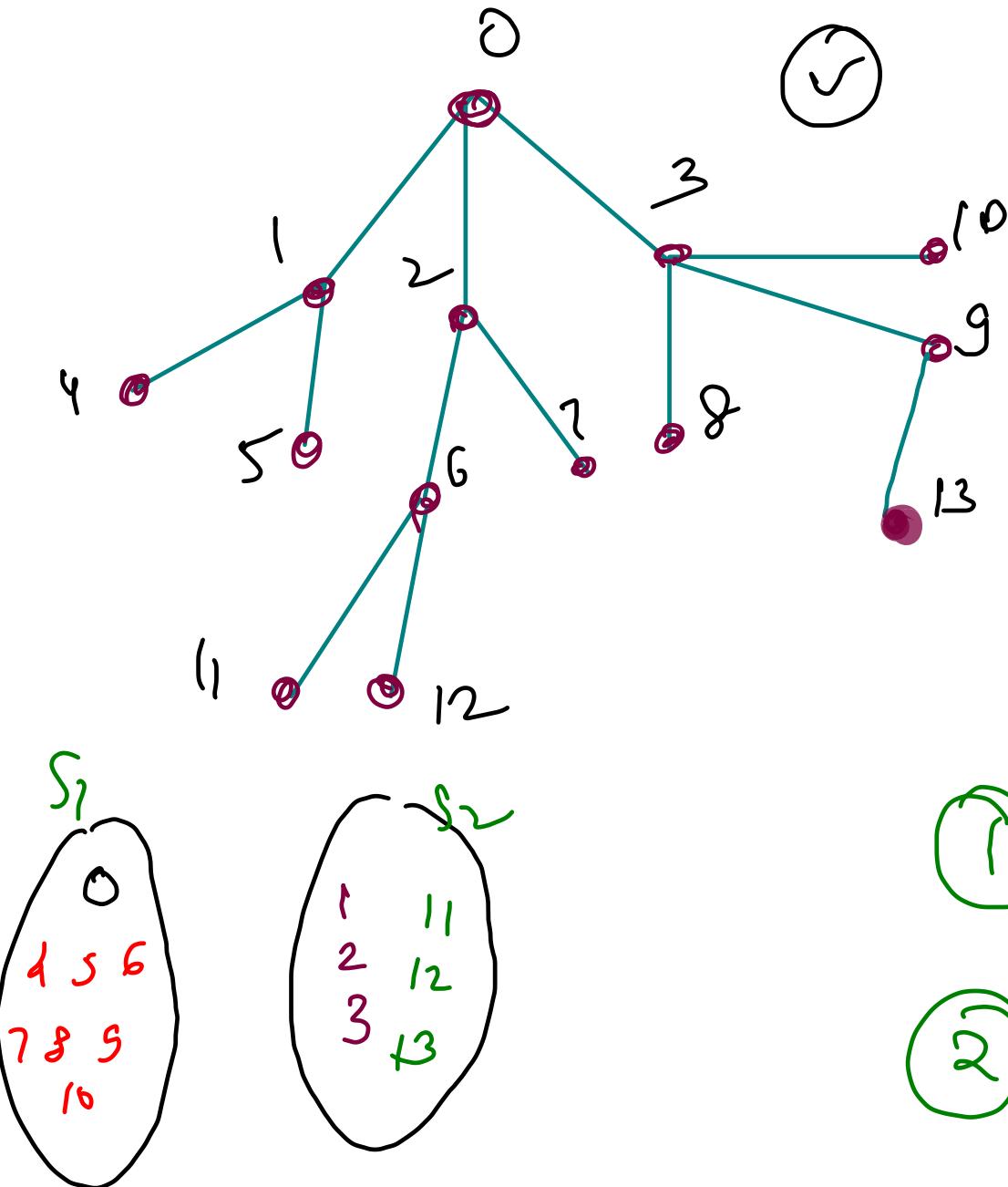
10 min

follow up

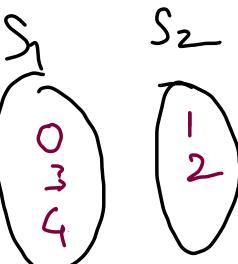
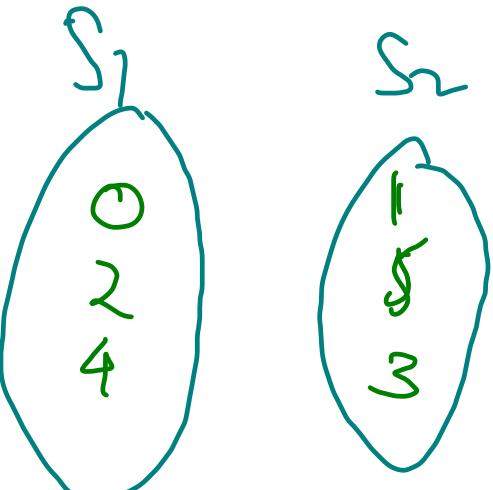
Graphs

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

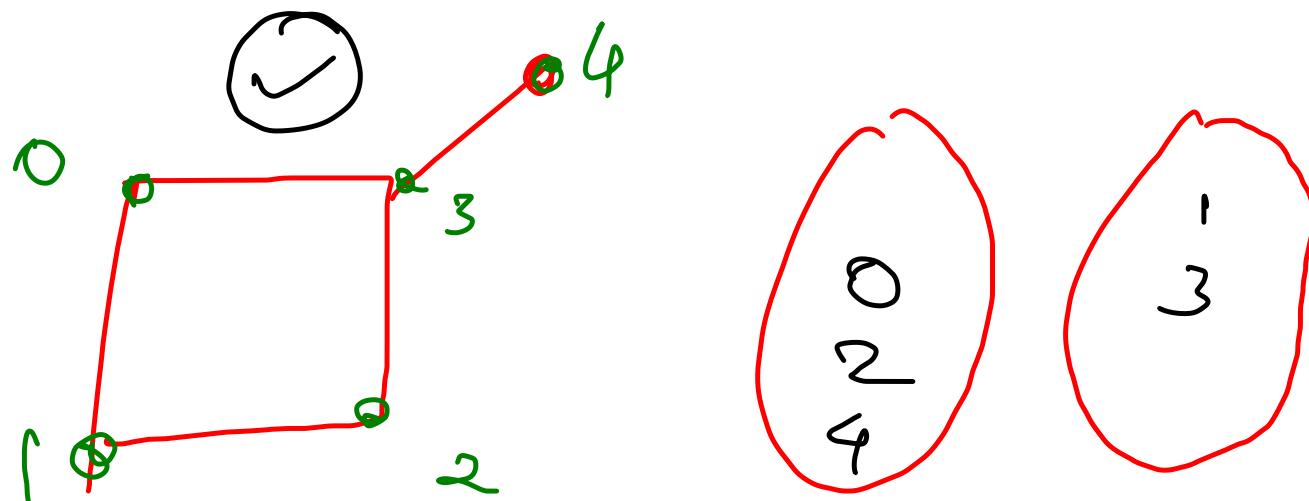
Bipartite Graph

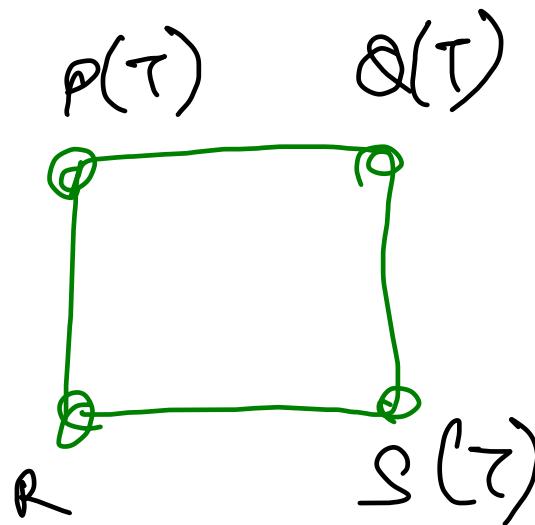
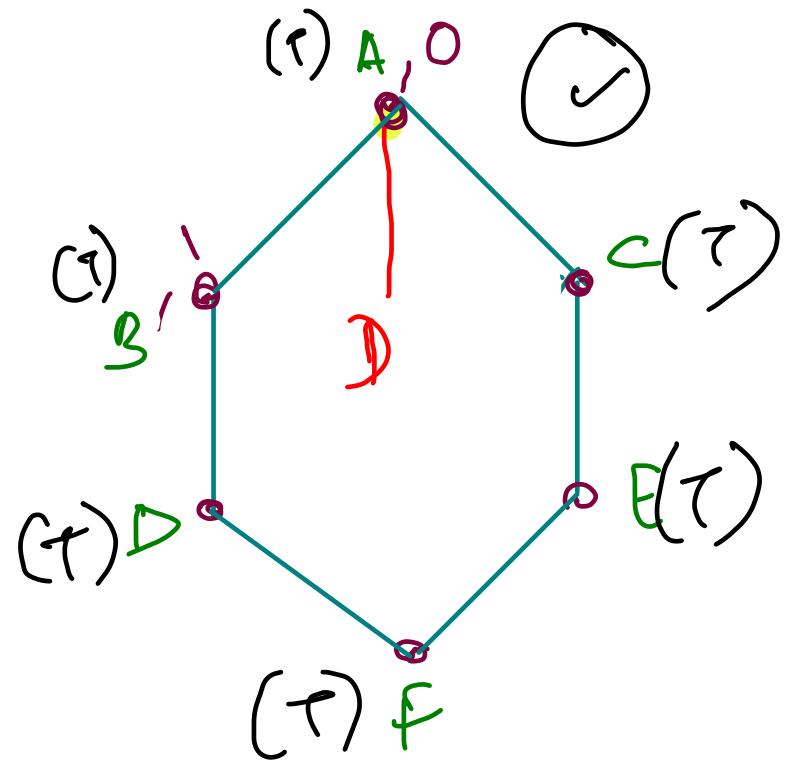


- Properties of Bipartite Graphs:
- ① mutually exclusive
 - ② exhaustive
 - Within set, any edge must not exist

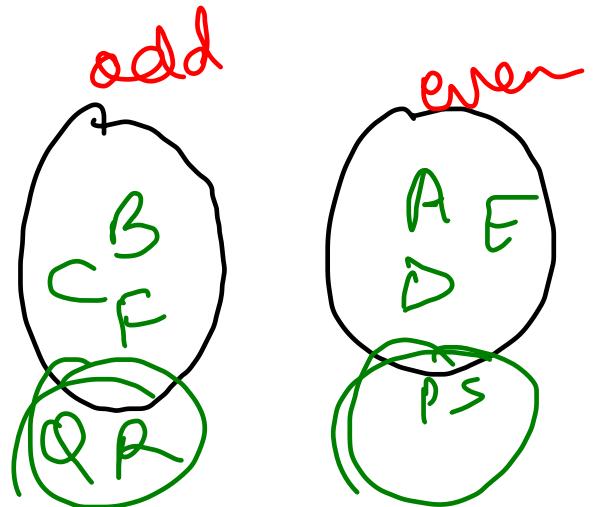


- ① Cycle does not exist \rightarrow Bipartite
- ② Cycle(s) are of even length \rightarrow Bipartite
- ③ Atleast 1 cycle in any component is of odd length
 \rightarrow Non-bipartite.

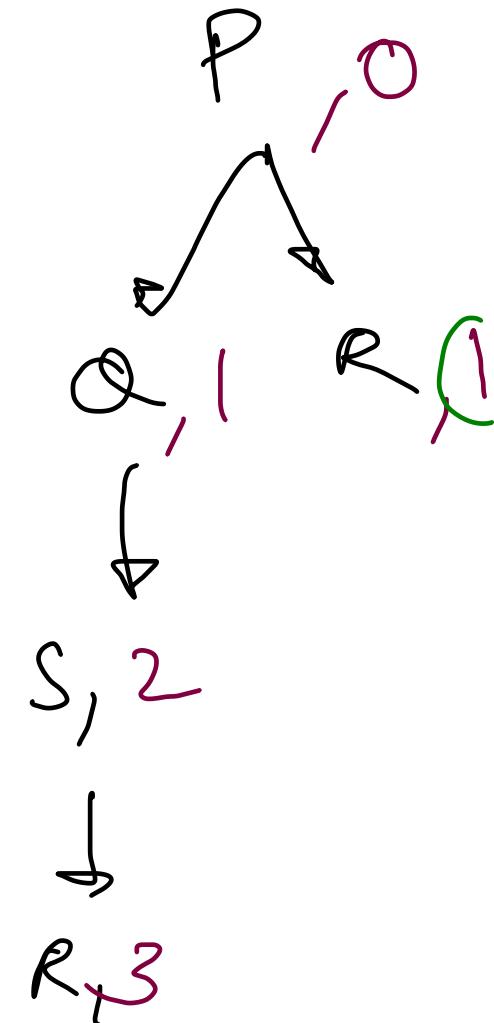
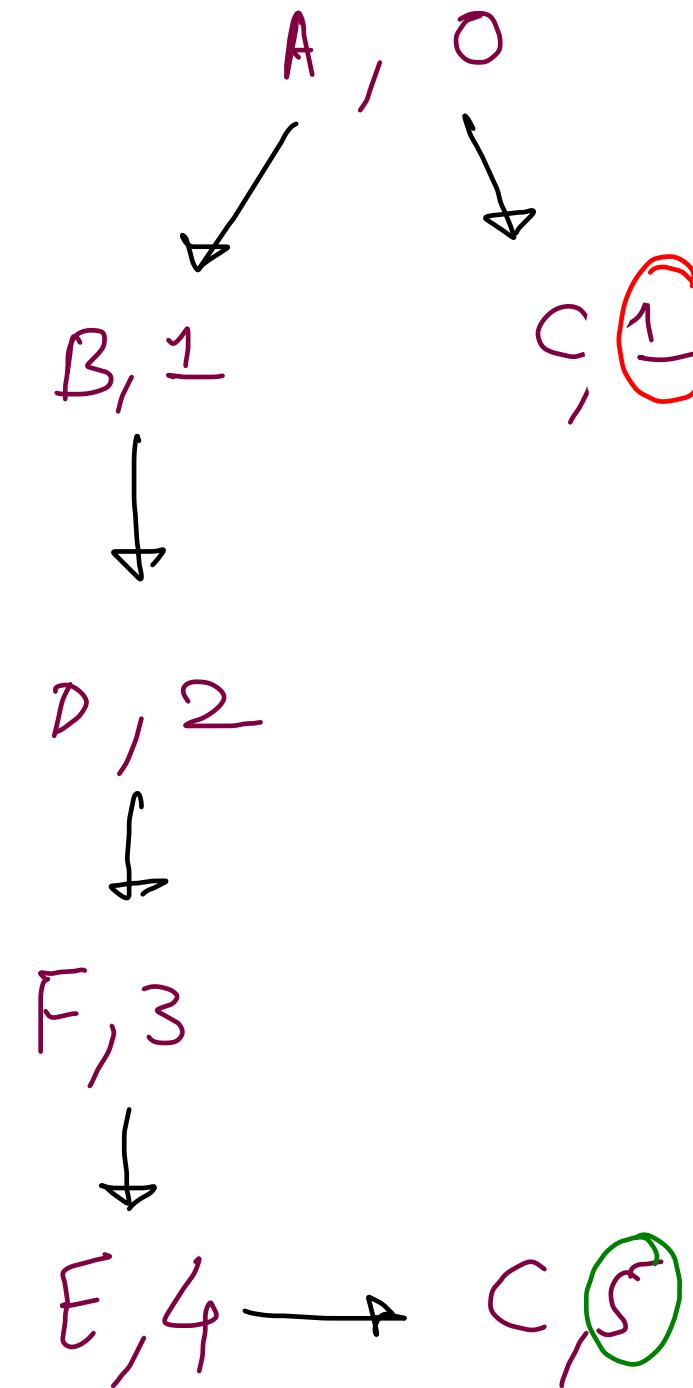


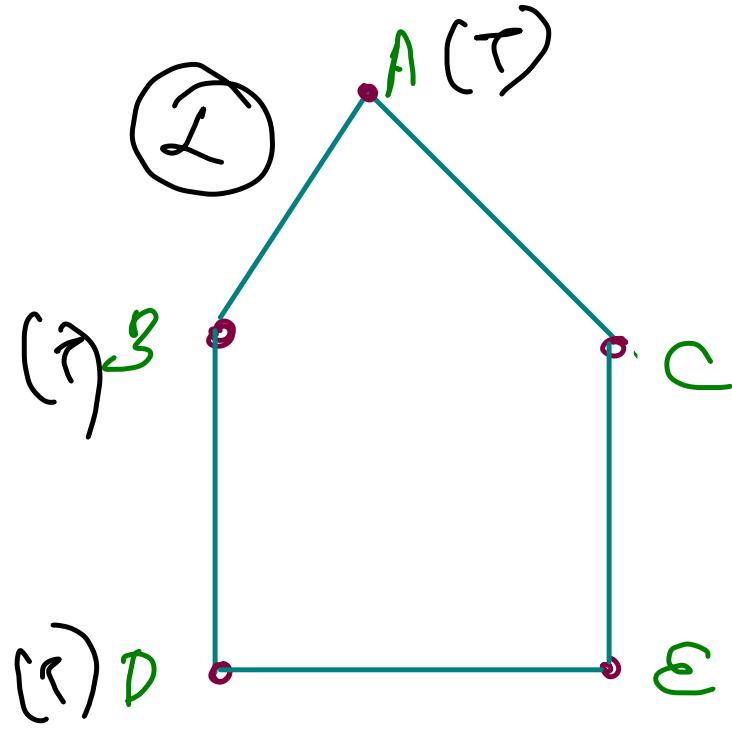


HashSet<int> odd, even



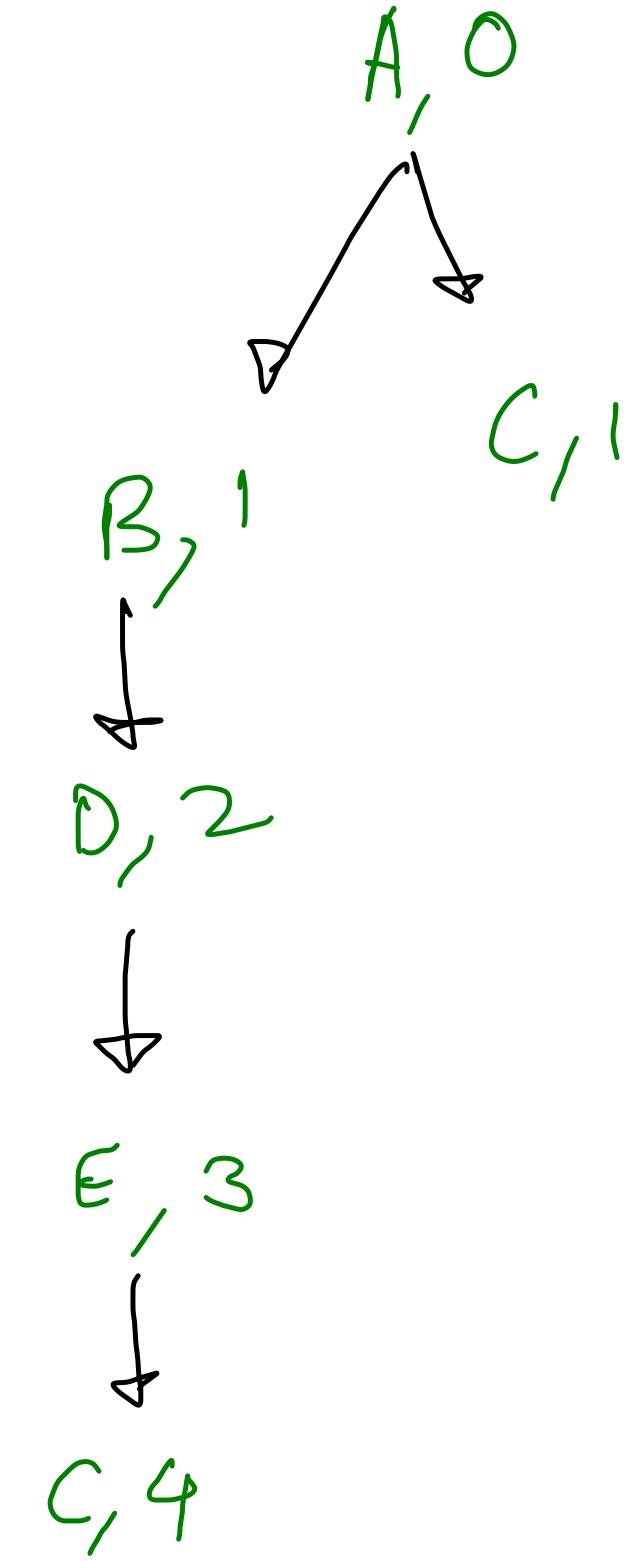
boolean
DFS



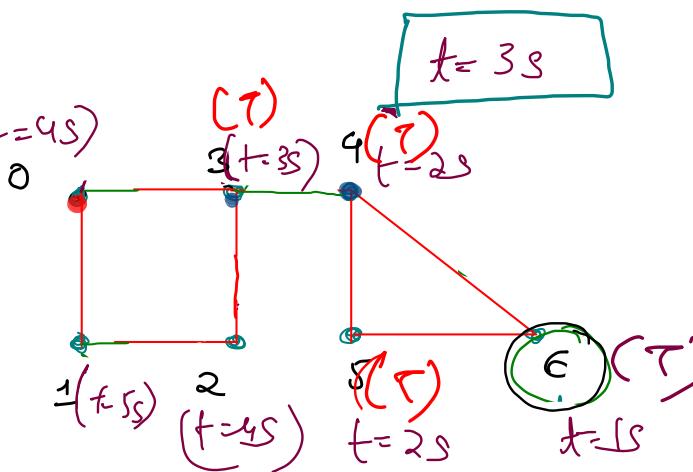
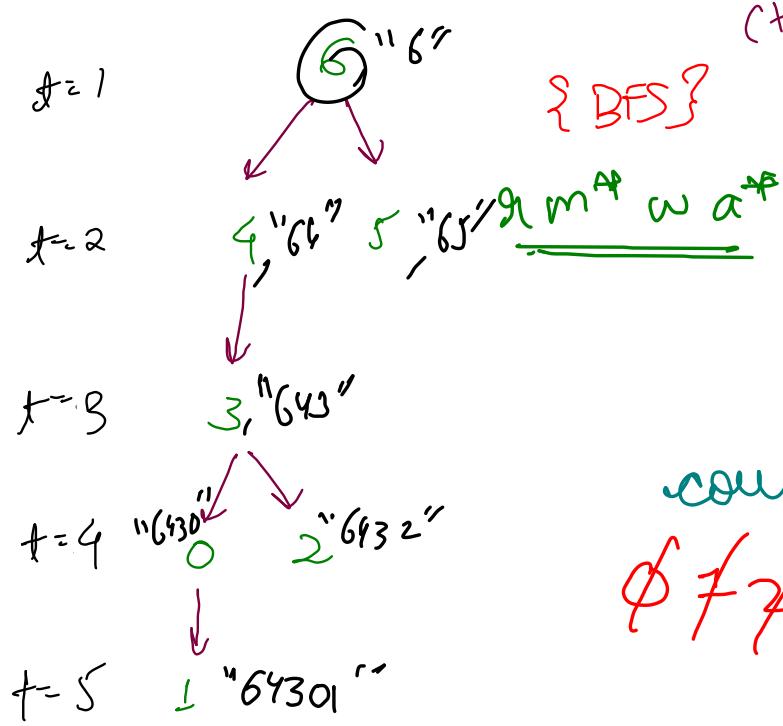


DFS

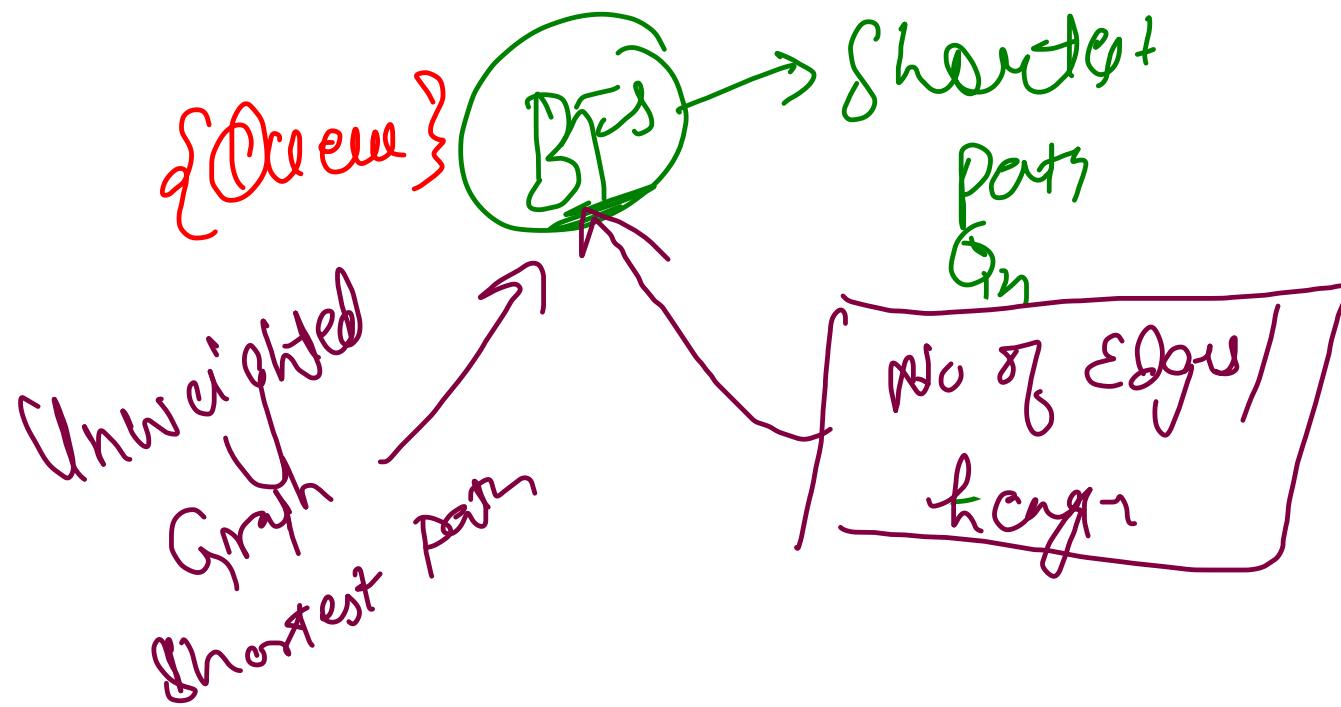
Two red-outlined ovals labeled "odd" and "even". The "odd" oval contains nodes B and E. The "even" oval contains nodes A and D. A small red cross is placed between the two ovals.



Spread of Infection



count \uparrow
 $\phi FFB4$



Rotten oranges

multi-source BFS

```
Queue<Pair> q = new LinkedList<>();
boolean[] vis = new boolean[vtes];
q.add(new Pair(src, 1));
int count = 0;

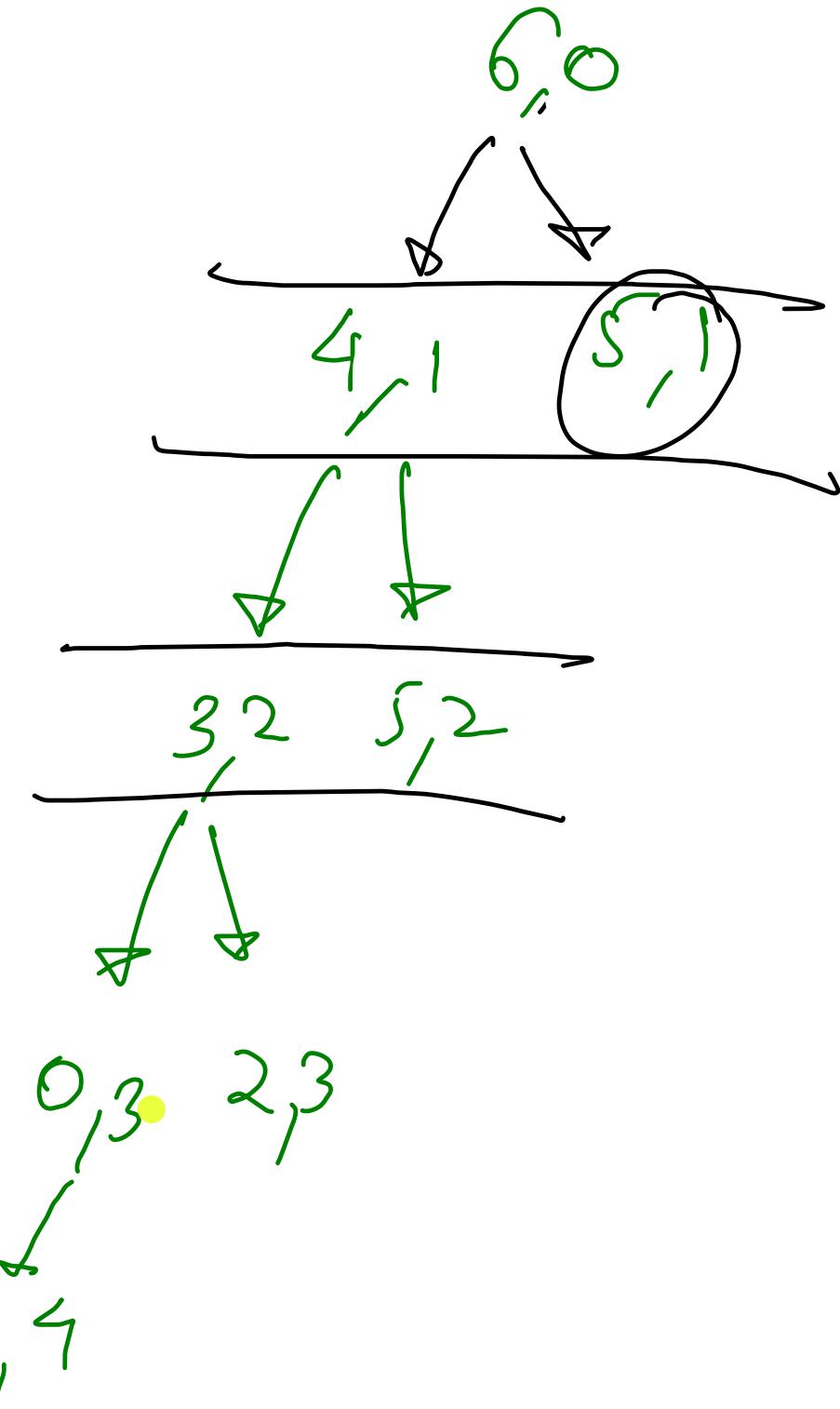
while(q.size() > 0){
    // remove
    Pair curr = q.remove();

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

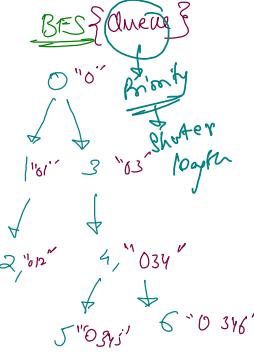
    // work
    if(curr.time > t) break;

    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
of shortest path in weight
single source



[0346 @ 58]
{0123456 @ 38} ▷
weight

$$TC = O(V + E \log V)$$

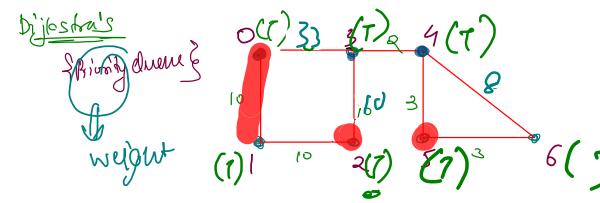
```
PriorityQueue<Pair> q = new PriorityQueue<>();
boolean[] vis = new boolean[vtes];
q.add(new Pair(src, "" + src, 0));
while(q.size() > 0){
    // remove
    Pair curr = q.remove();
    if(vis[curr.node] == true) continue;
    // mark
    vis[curr.node] = true;
    // work
    System.out.println(curr.node + " 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

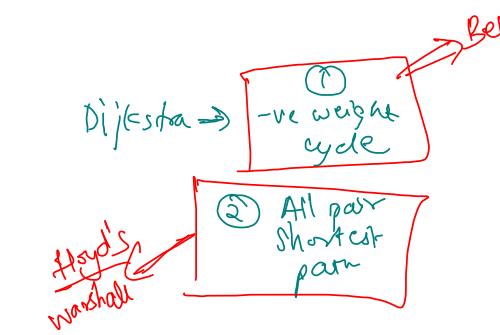
0, "0", 0
1, "01", 10
2, "02", 20
3, "03", 30
4, "04", 32
5, "05", 35
6, "06", 40

6, "0123456"

38



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



1

2

3

4

5

6

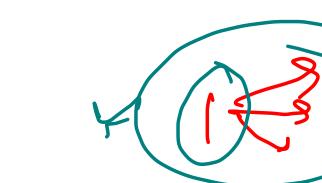
7

8

9

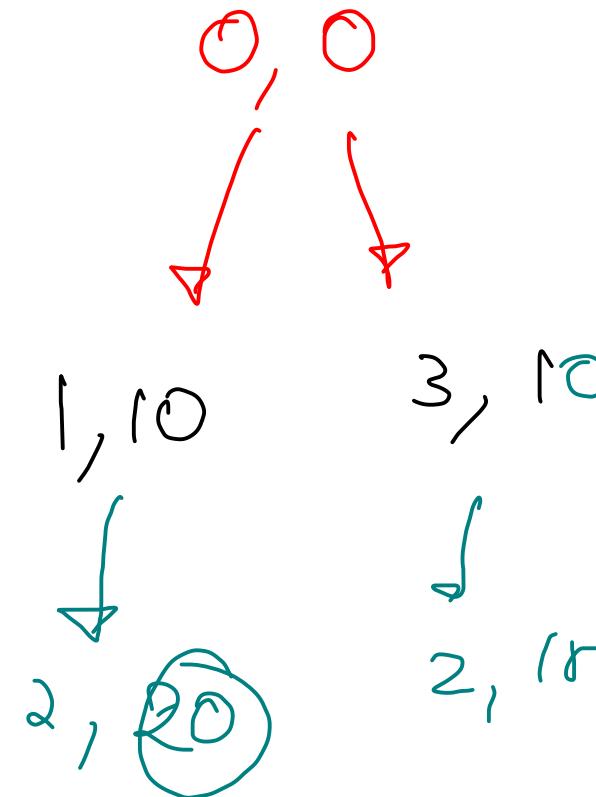
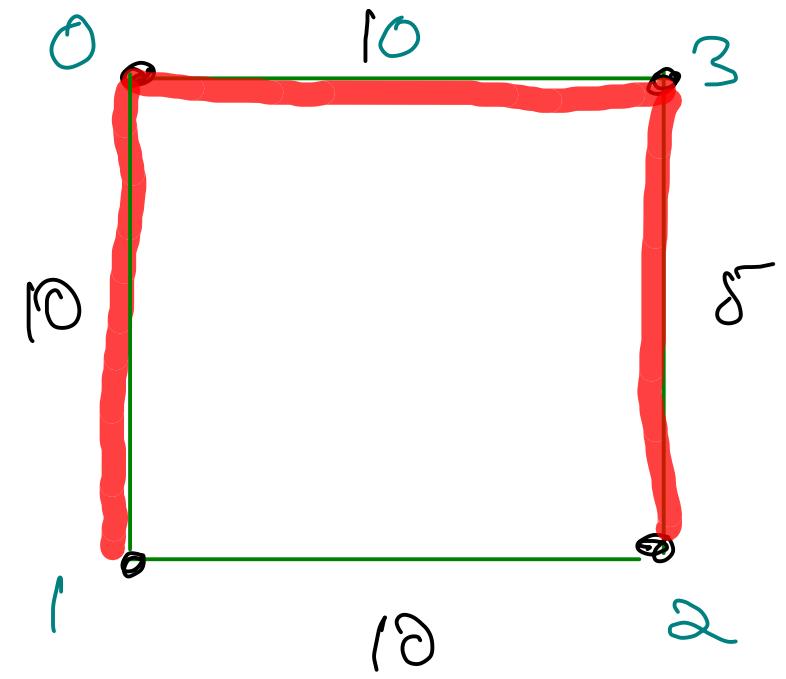
10

11

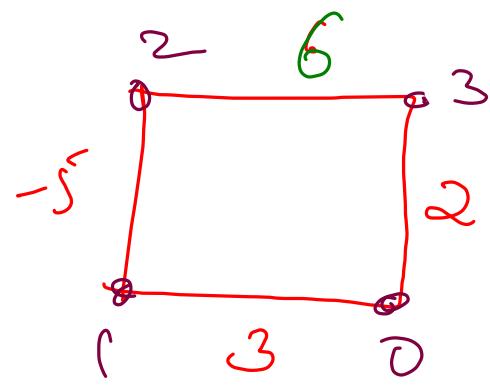


② Negative weight cycle

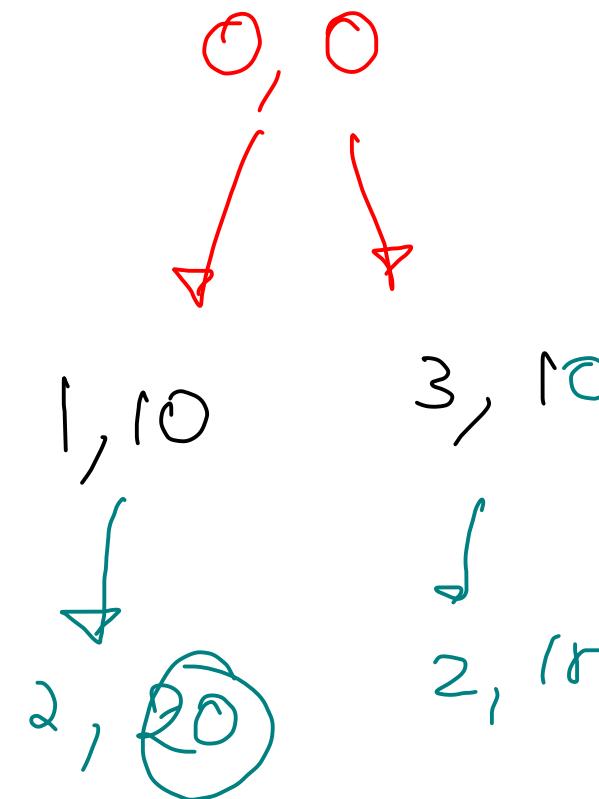
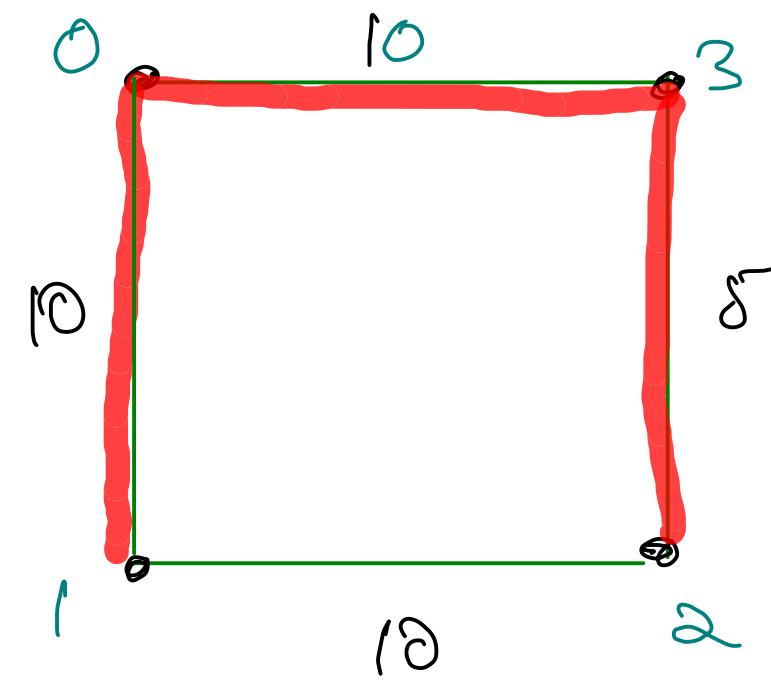
▷ Bellman Ford



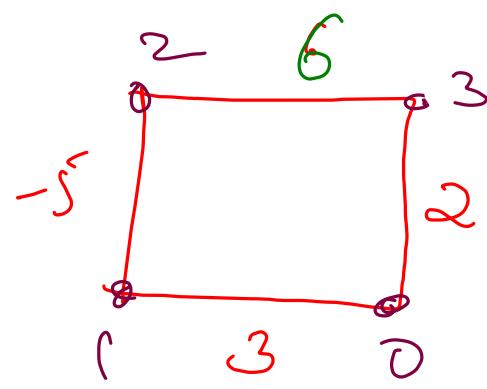
$$\begin{array}{c}
 \boxed{0 \ 2 \ 3 \ 4} \ 0 \ 1 \ 2 \ 3 \ 4 \ \boxed{} \ \boxed{} \ \boxed{} \\
 -6 + (-1) + (-6) + (-6)
 \end{array}$$



$$\begin{array}{l}
 0 \leftarrow 1(3) \rightarrow 2(-2) \\
 3(2) \rightarrow 2(8)
 \end{array}$$



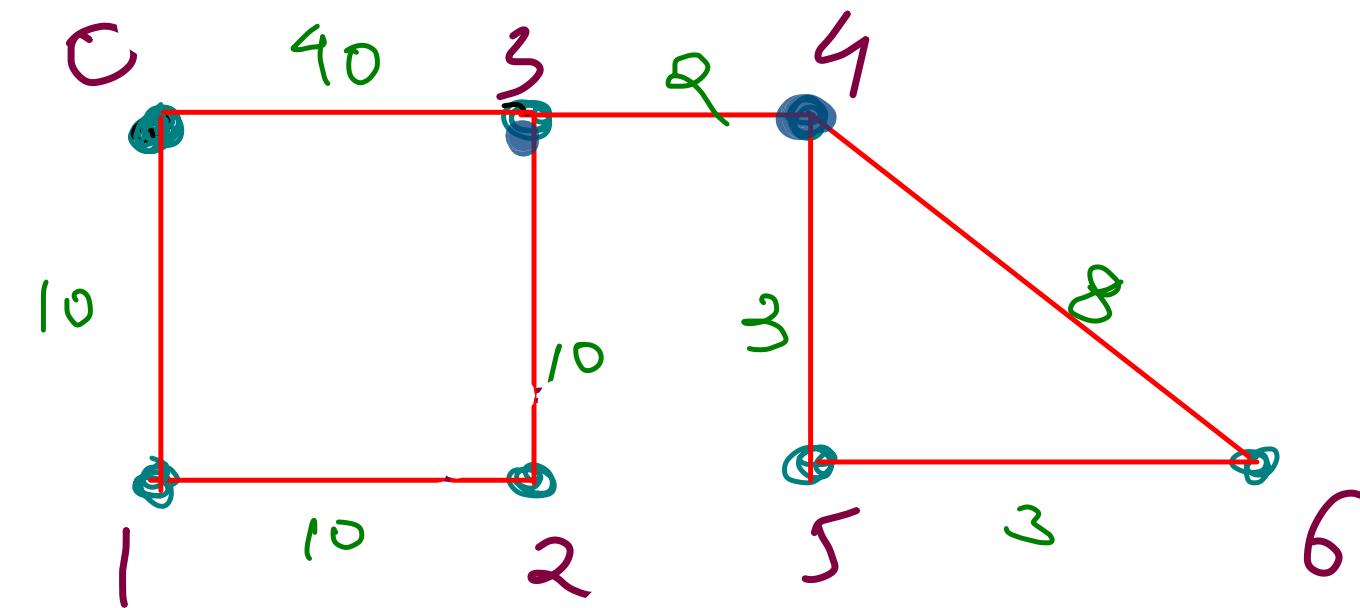
$$\begin{array}{c}
 \boxed{0 \ 2 \ 3 \ 4} \ 0 \ 1 \ 2 \ 3 \ 4 \ \boxed{\quad} \ \boxed{\quad} \ \boxed{\quad} \\
 -6 + (-1) + (-6) + (-6)
 \end{array}$$



$$\begin{array}{l}
 0 \leftarrow 1(3) \rightarrow 2(-2) \\
 3(2) \rightarrow 2(8)
 \end{array}$$

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}

2 Prim's Algorithm

subgraph

tree

minimum edges

Ayclic

Connected

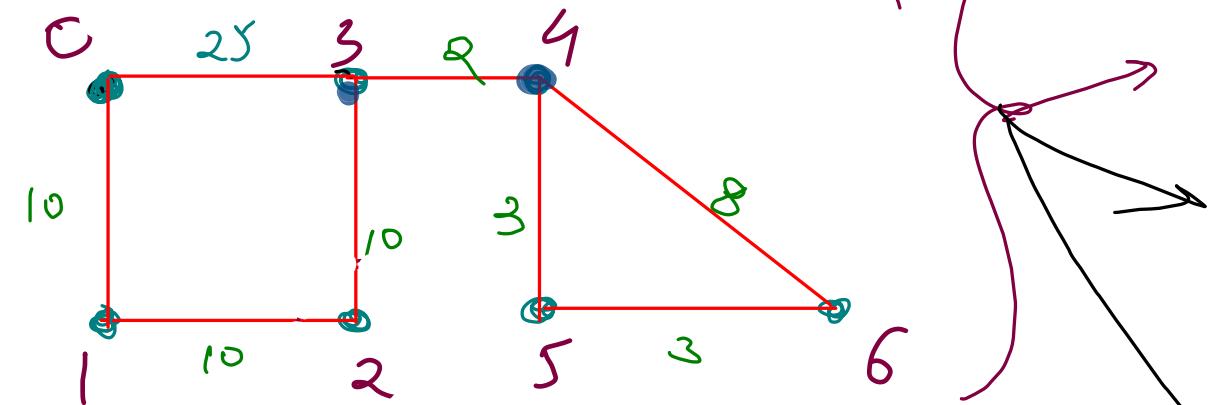
Spanning

all vertices
will be there

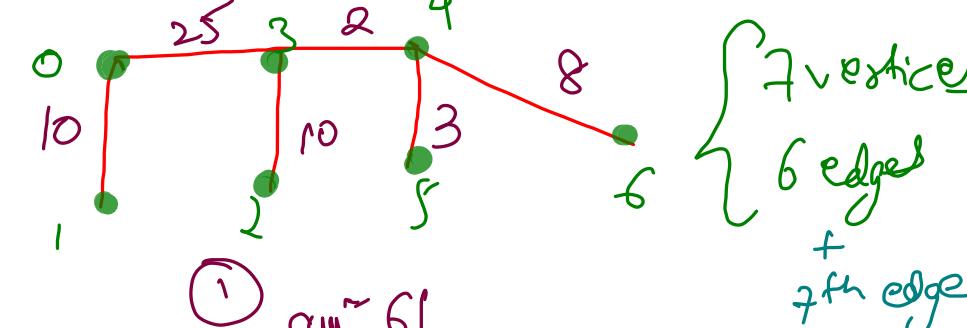
minimum

Spanning tree

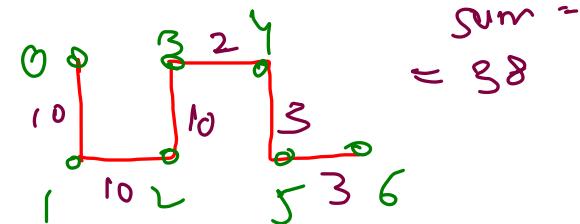
edges' sum should be
minimum



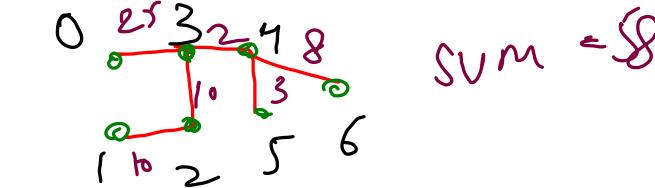
① sum ~ 61



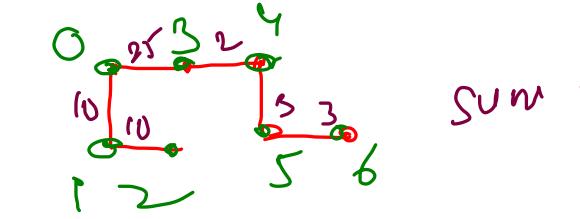
7 vertices
6 edges
+ 7th edge



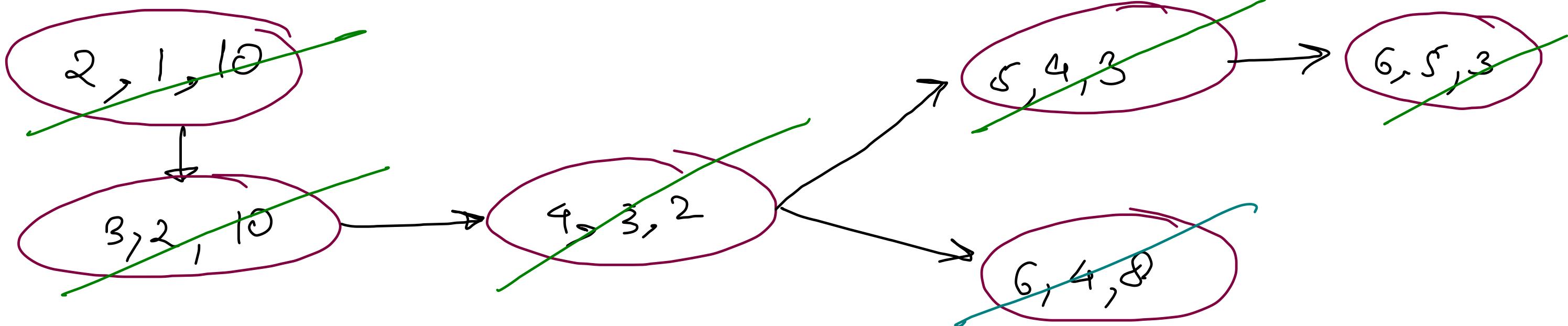
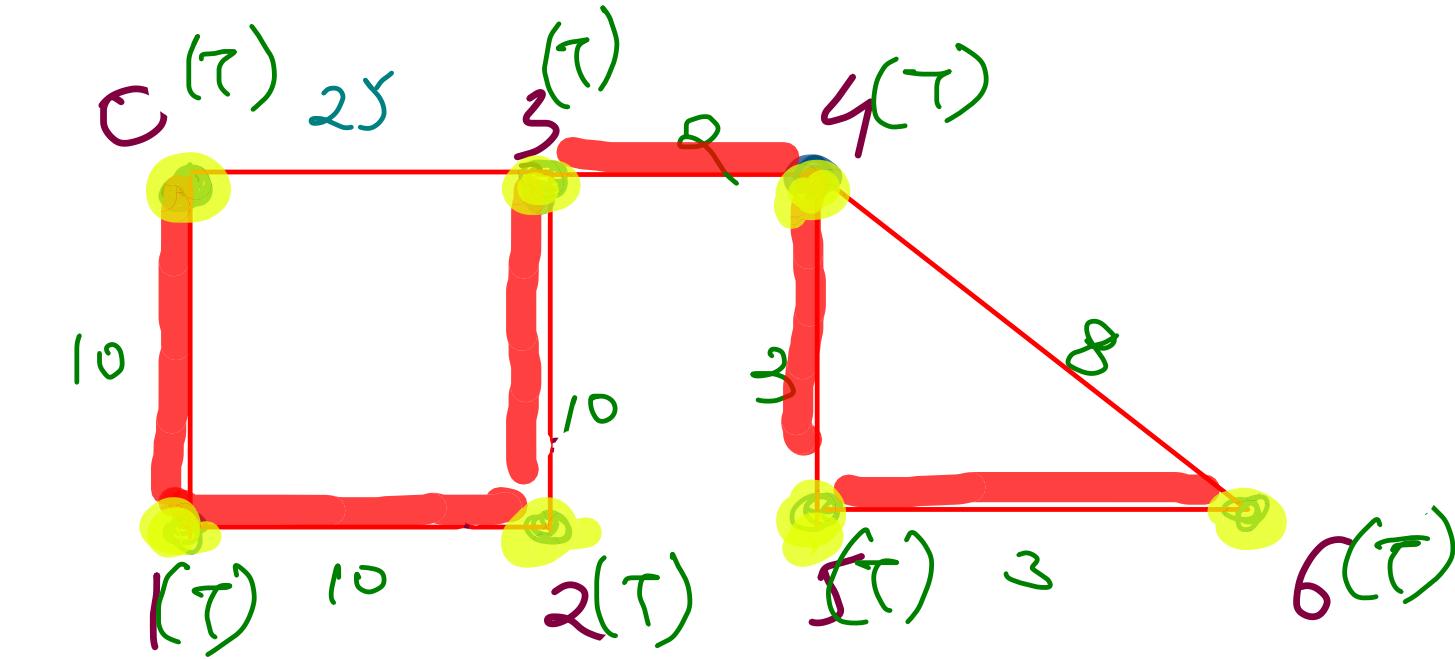
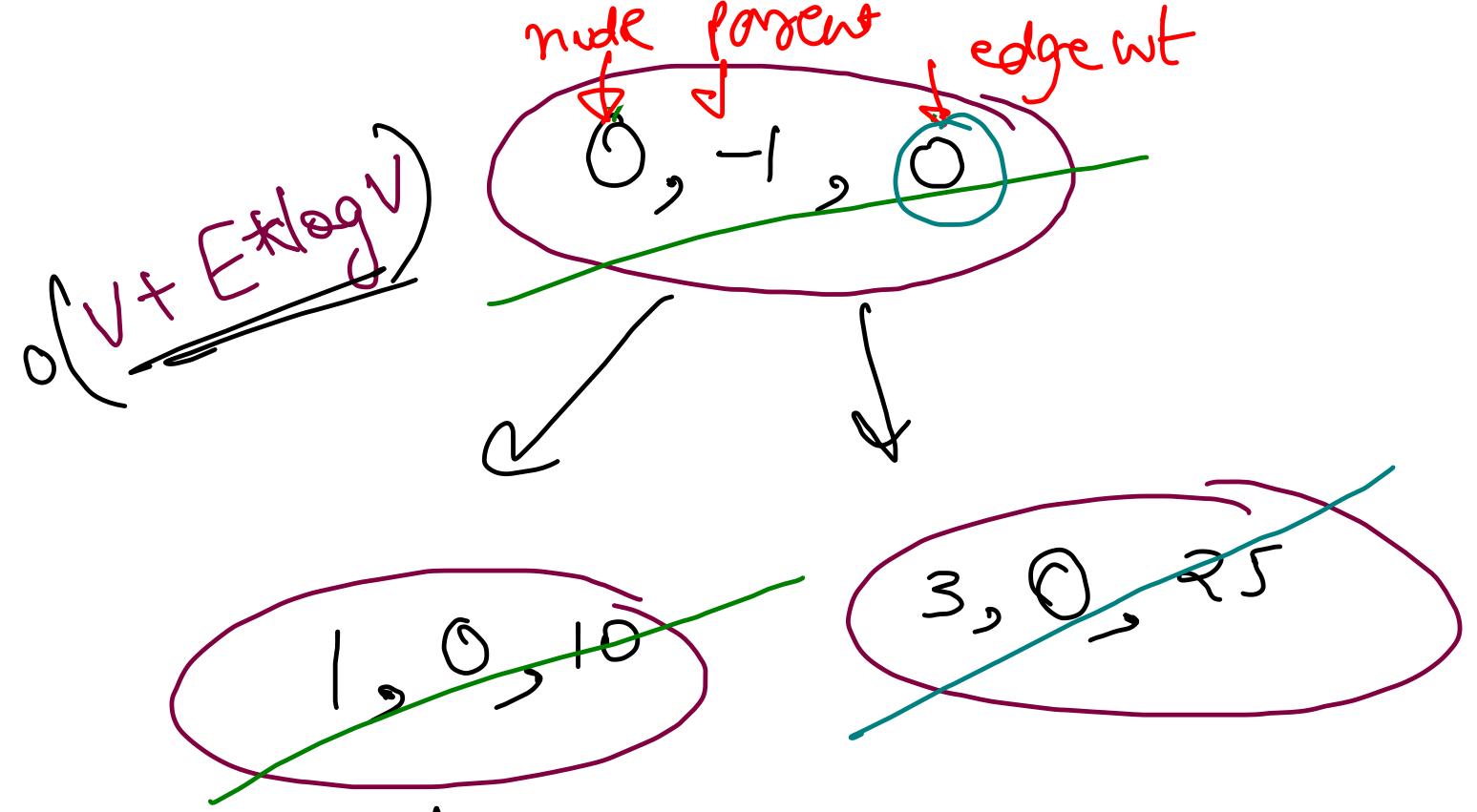
sum = 58



sum = 58



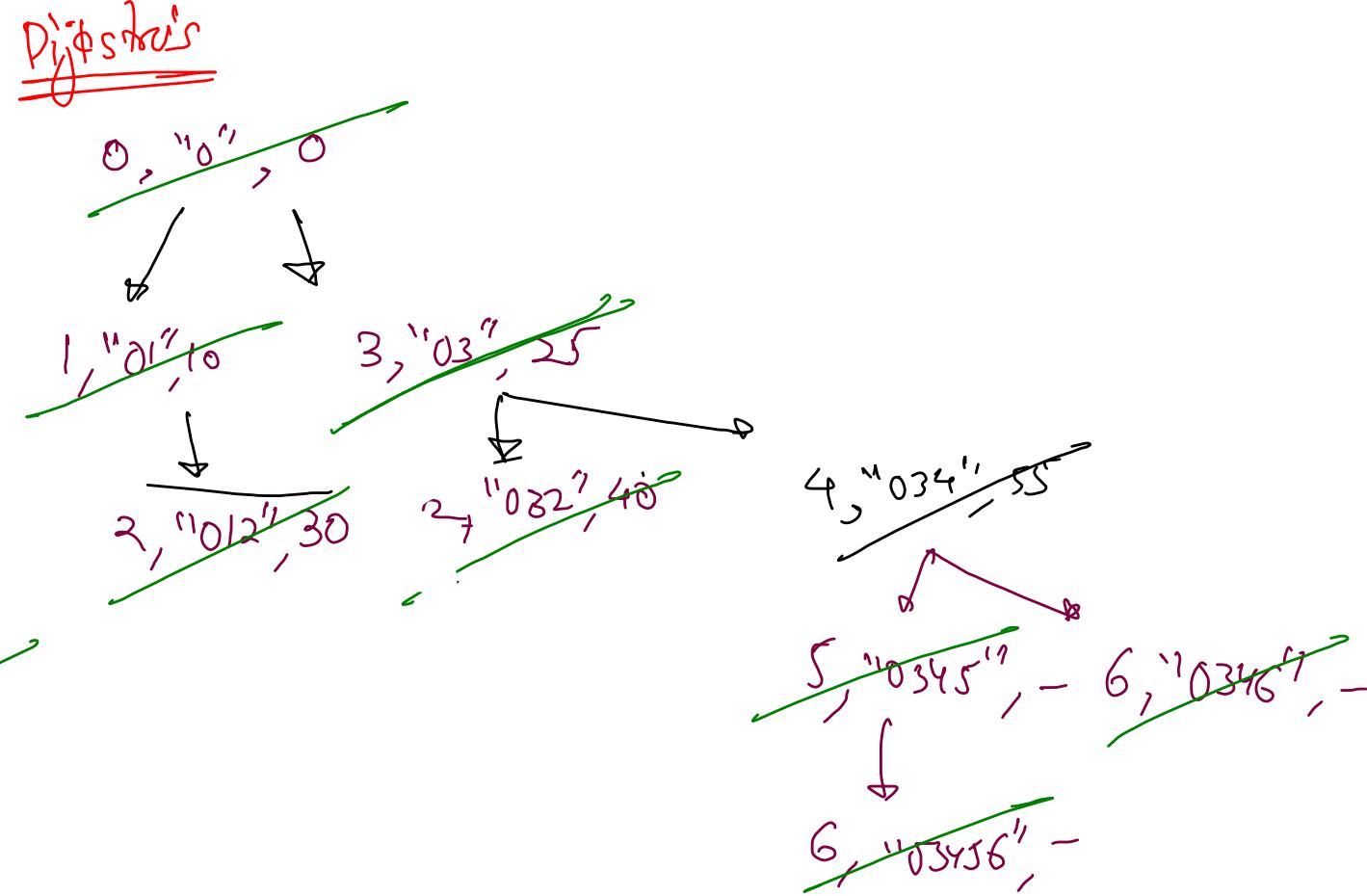
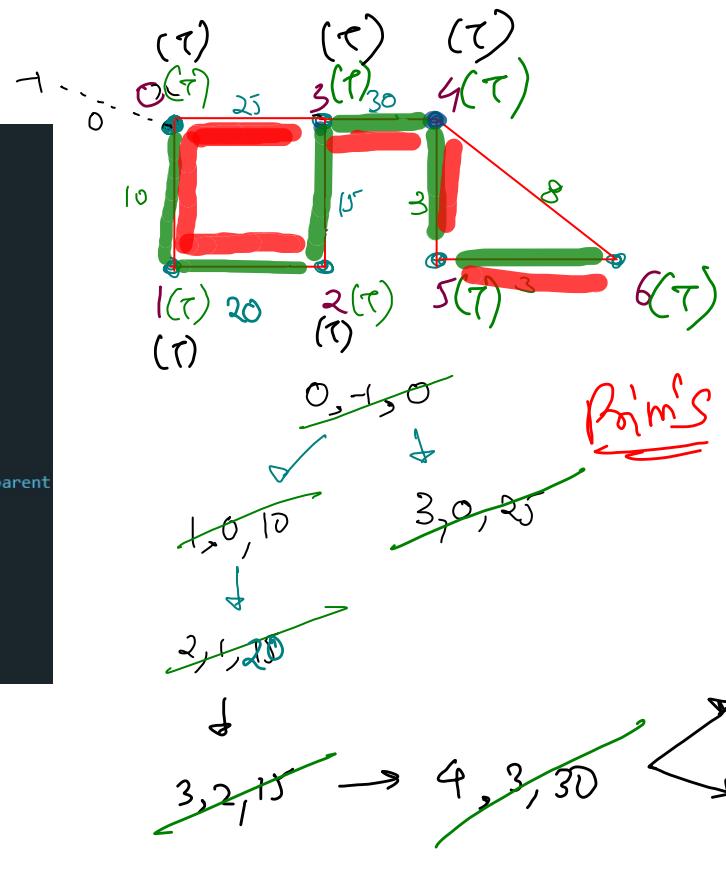
Wf → replace
edge weight



```

PriorityQueue<Pair> q = new PriorityQueue<>();
boolean[] vis = new boolean[vtes];
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.

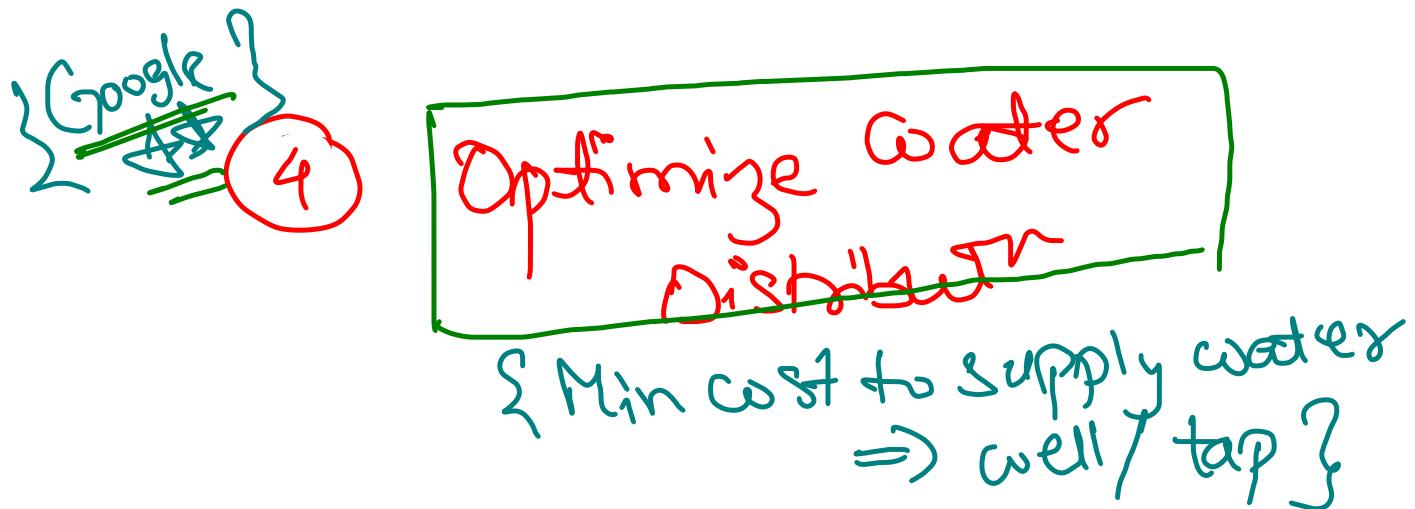
- telephone, electrical, hydraulic, TV cable, computer, road

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.

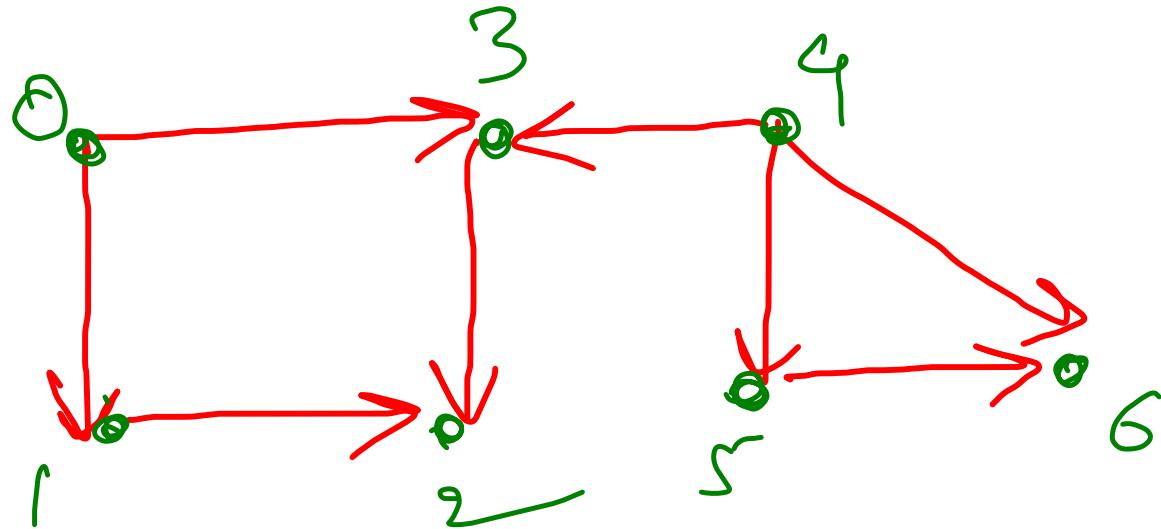
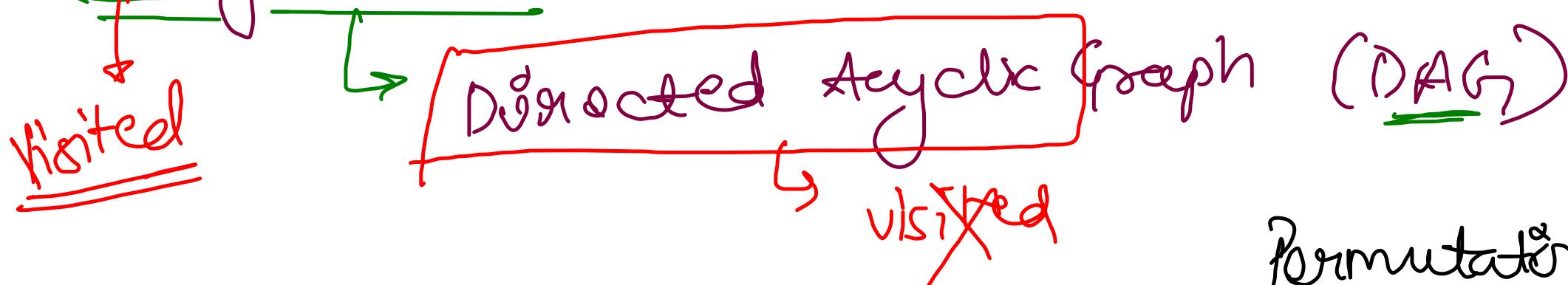
① Min cost of wire
connect PC's

② Min cost of roads
to connect all
cities

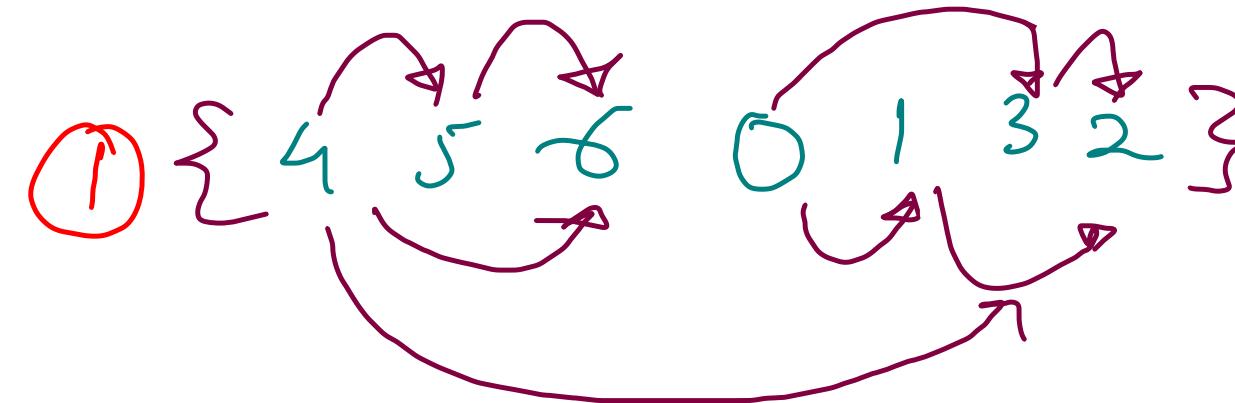
③ Min cost to connect
all 2D points

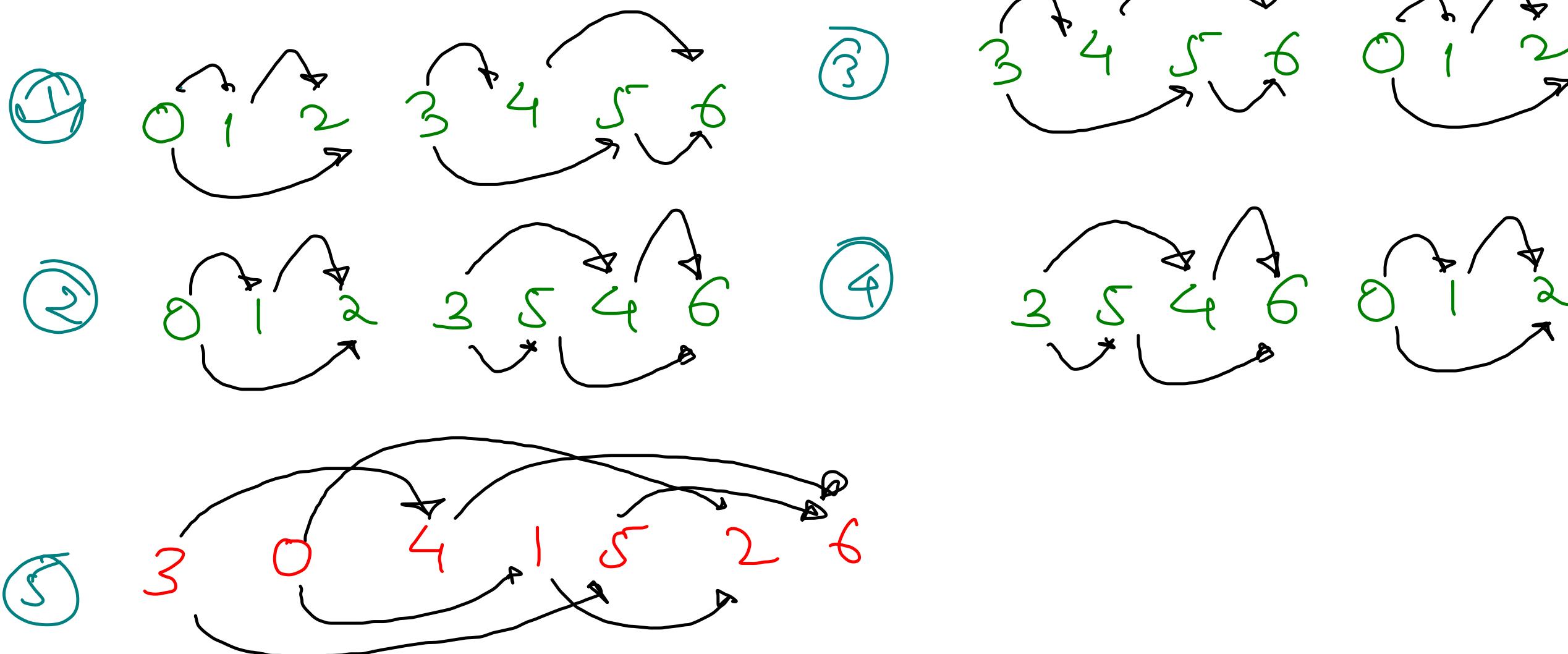
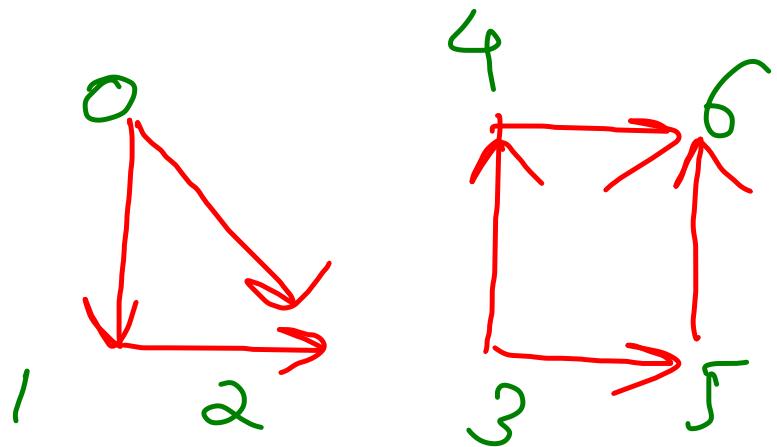


Topological Sort { Order of Compilation }



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





Applications of Topological Sorting

① Prequisites / Course Schedule {Medium}

~~FAANG~~
② Alien Dictionary {Hard}

③ Minimum Time Job Scheduling {Hard}

④ Mother Vertex

⑤ Reconstruct Itinerary

1. Finding cycle in a graph
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.

Cycle Detection

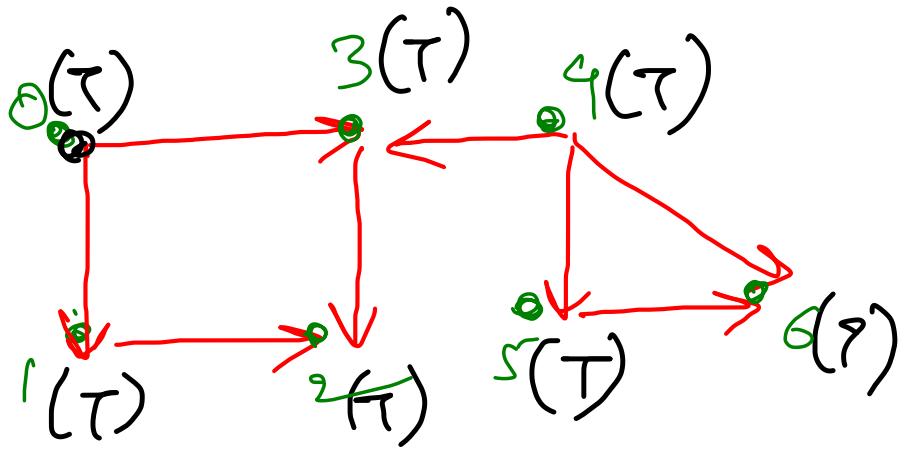
5

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);
        }
    }

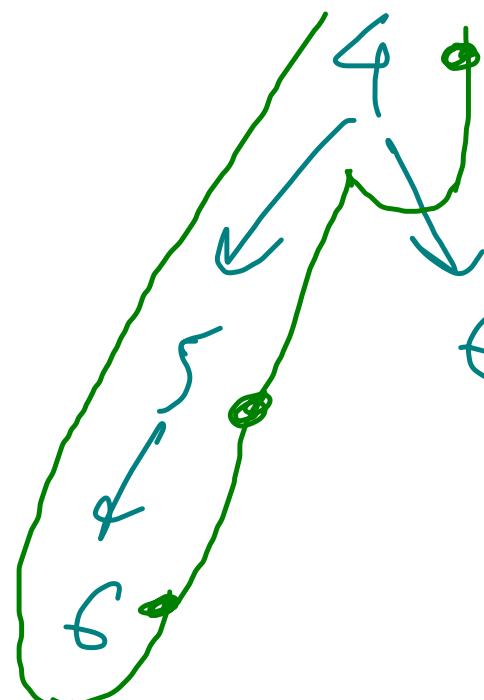
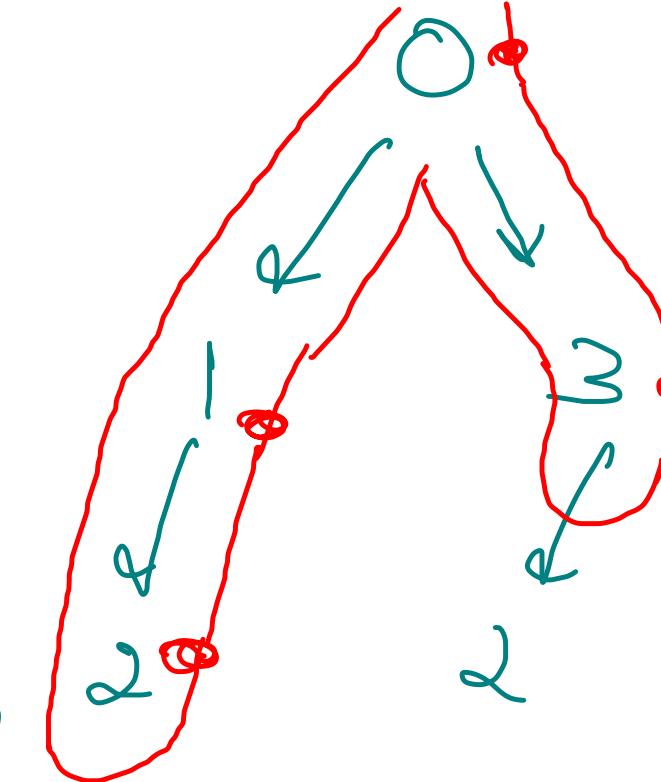
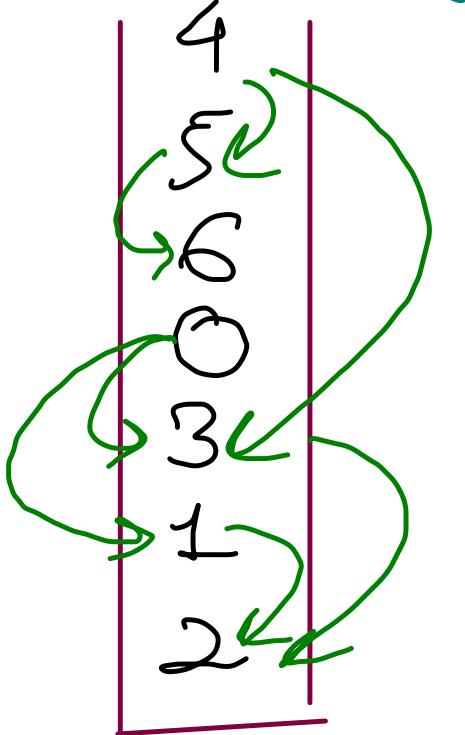
    topoSort.push(src);
}
```



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

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

DFS order does not matter
 { Any node can be source }

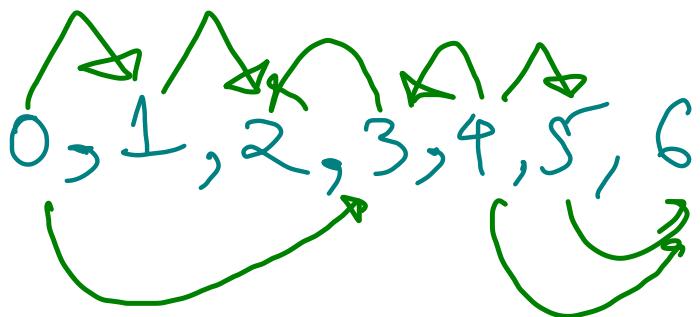
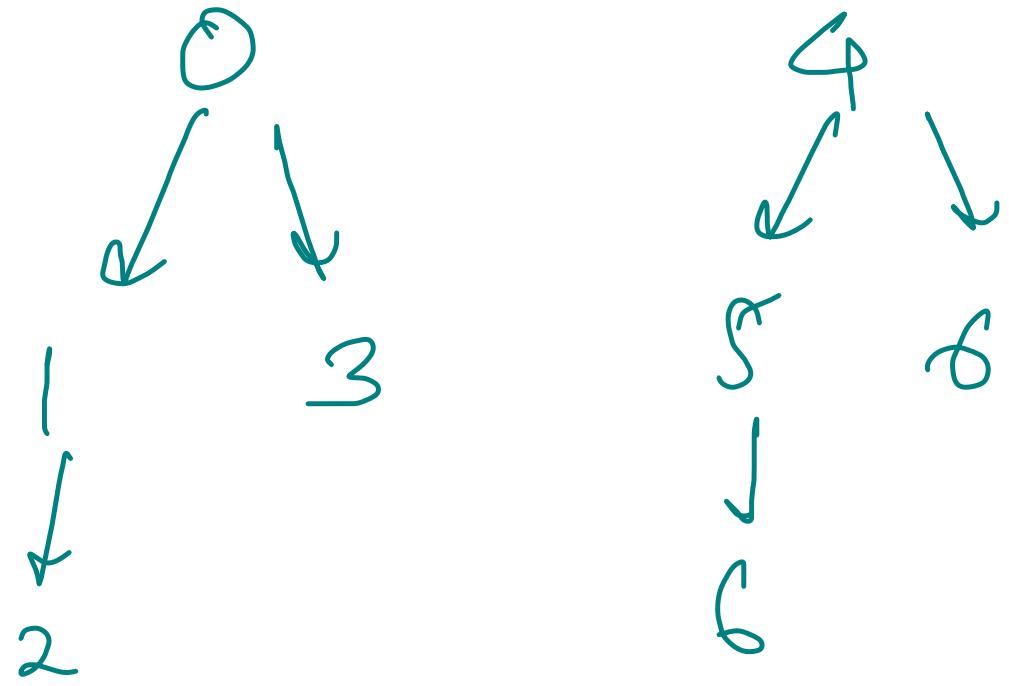
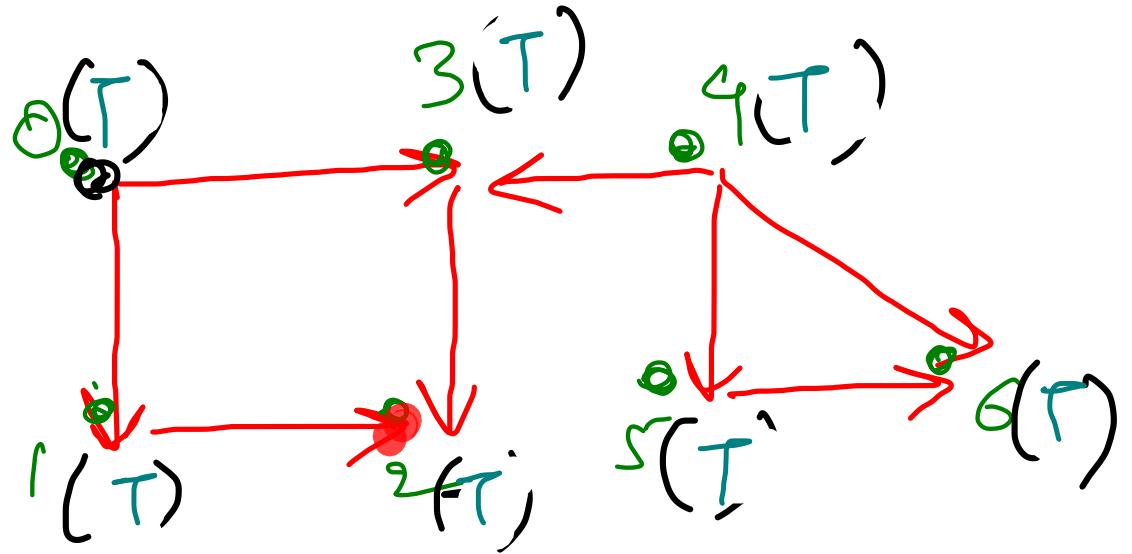


Pre → Point
 Post → Point
 Stack → Pre
 Stack → Post

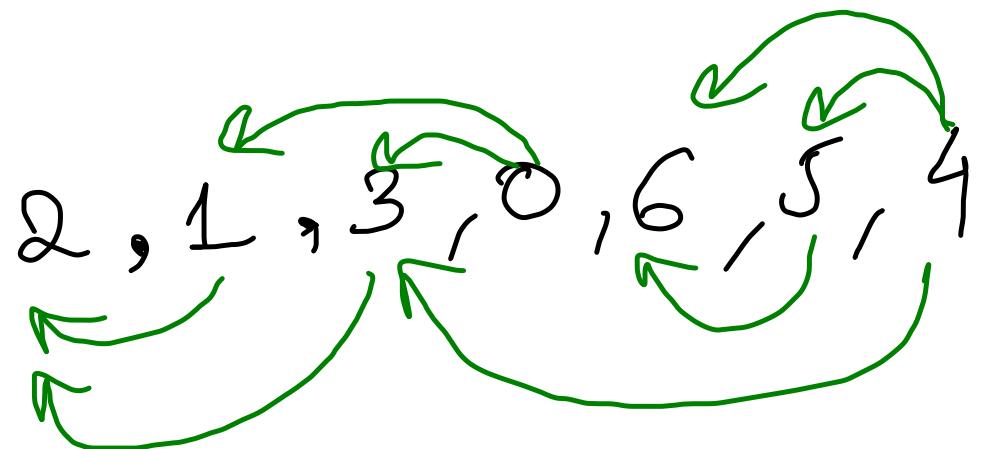
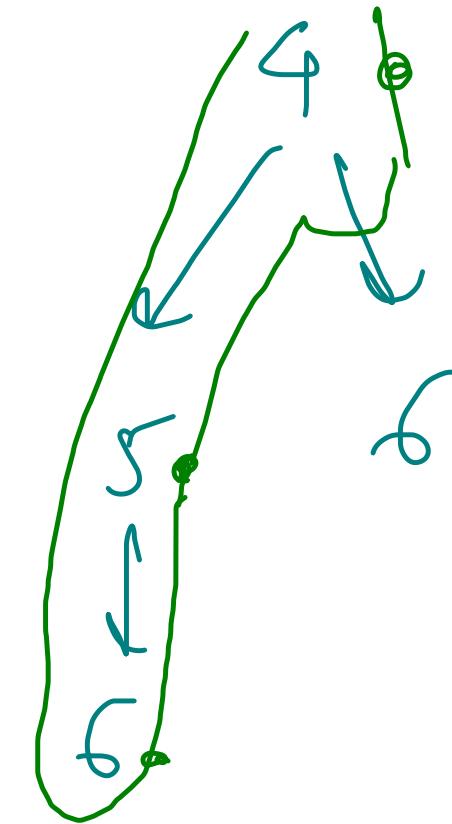
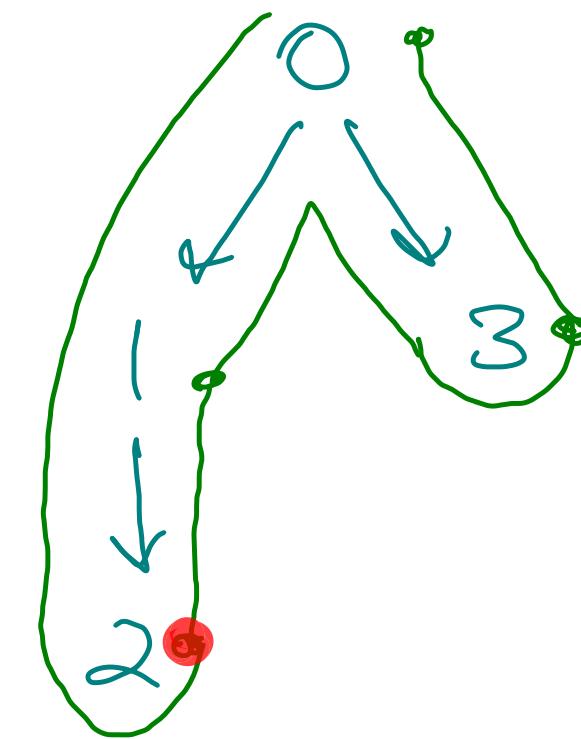
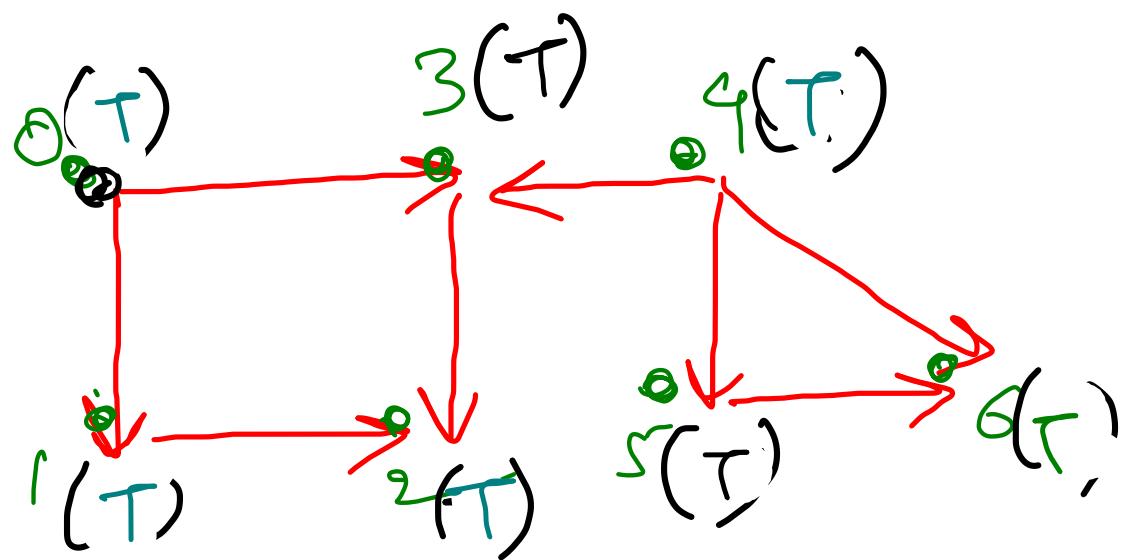
hum jinpar
 dependent
 honge
 wo already
 stack me
 honge

FAITH

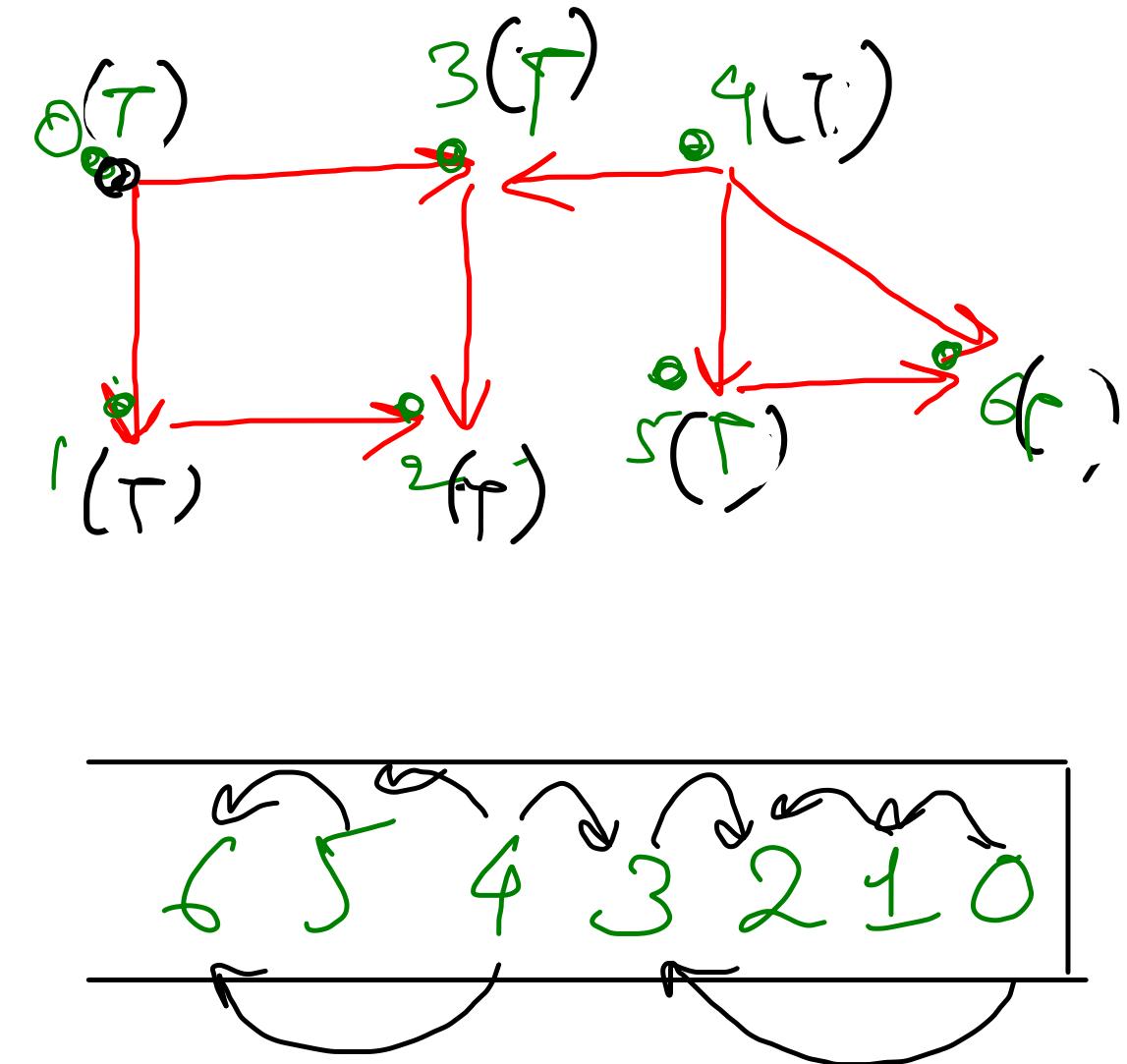
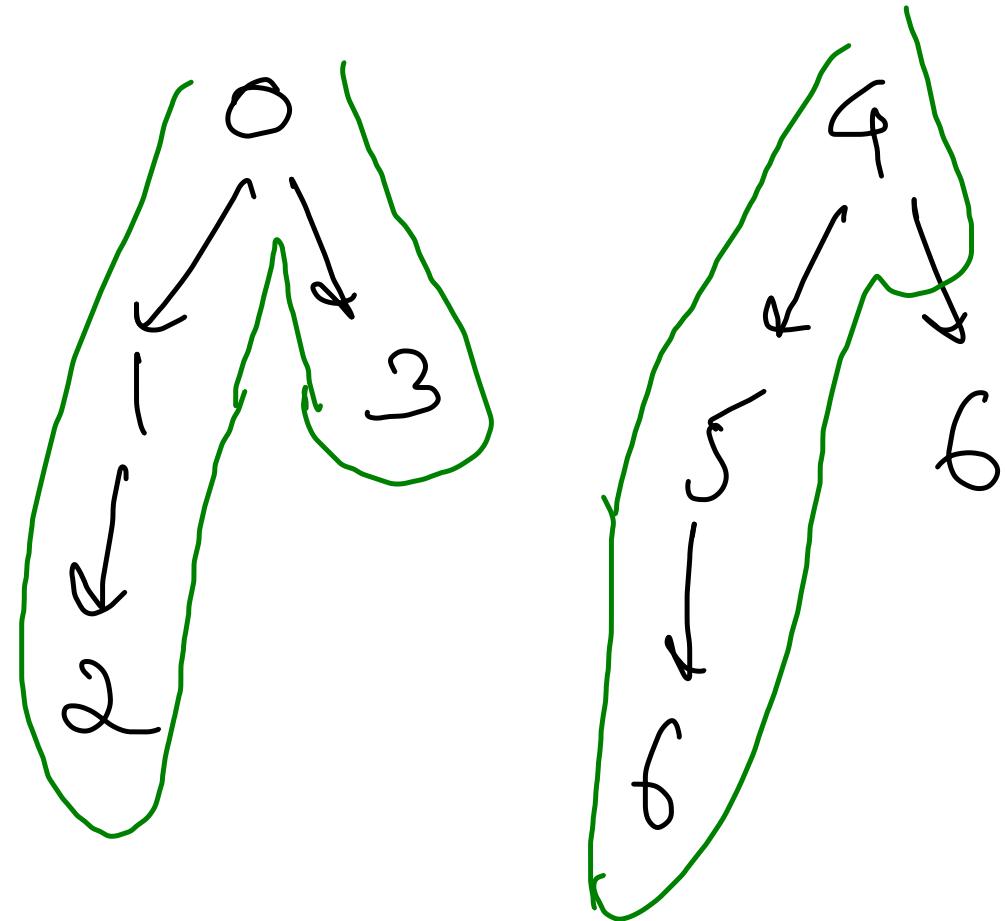
Preorder print

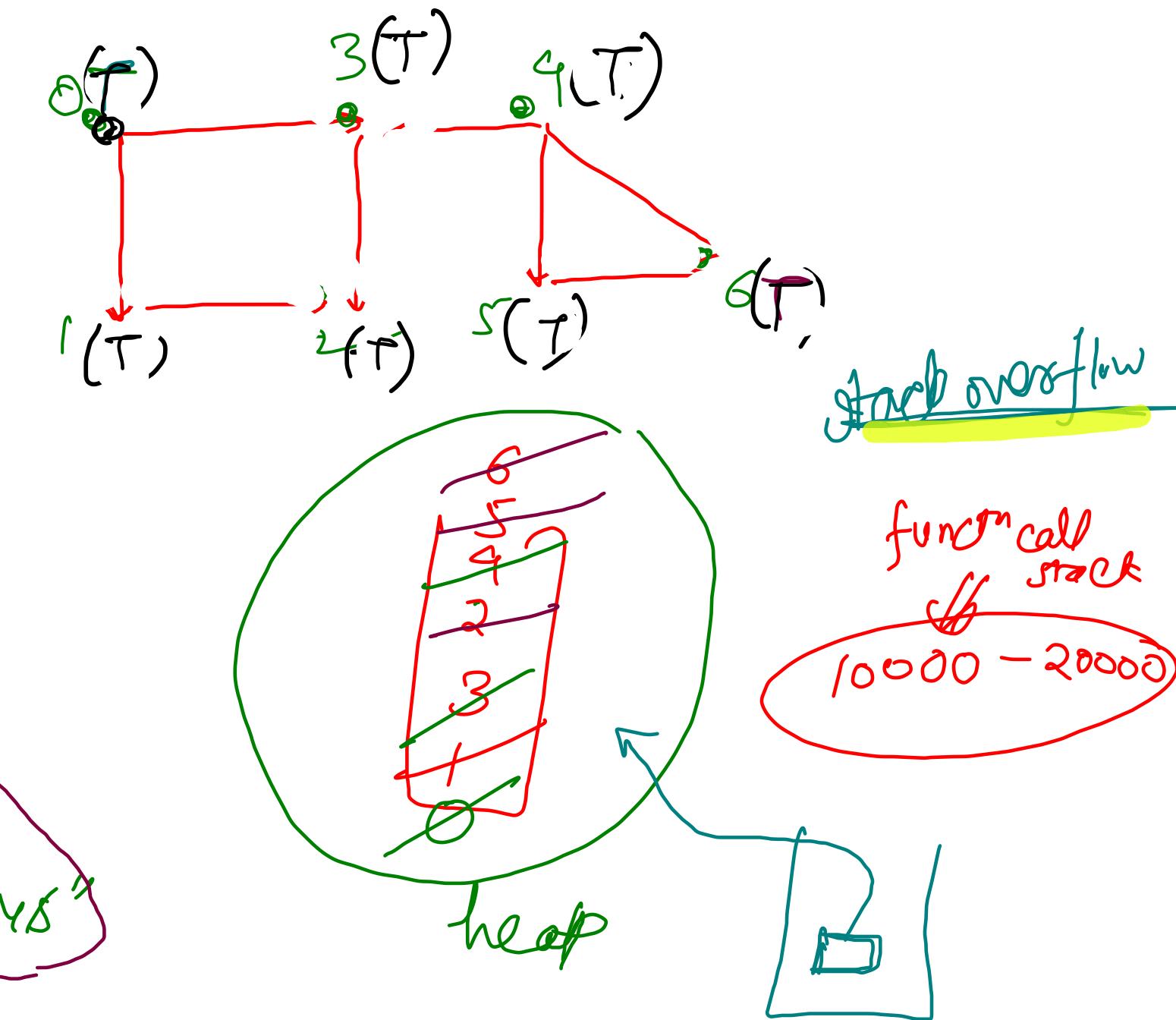
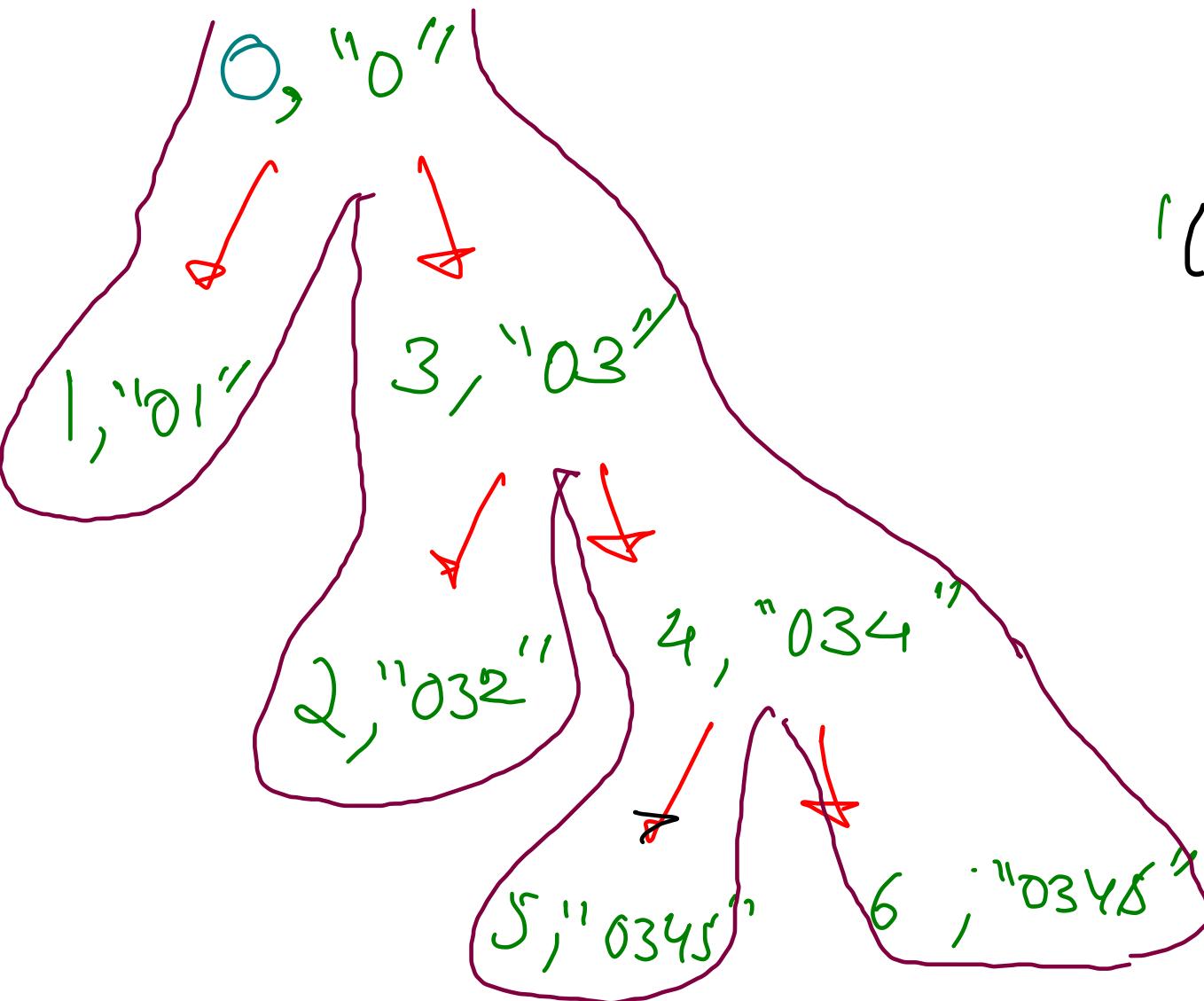
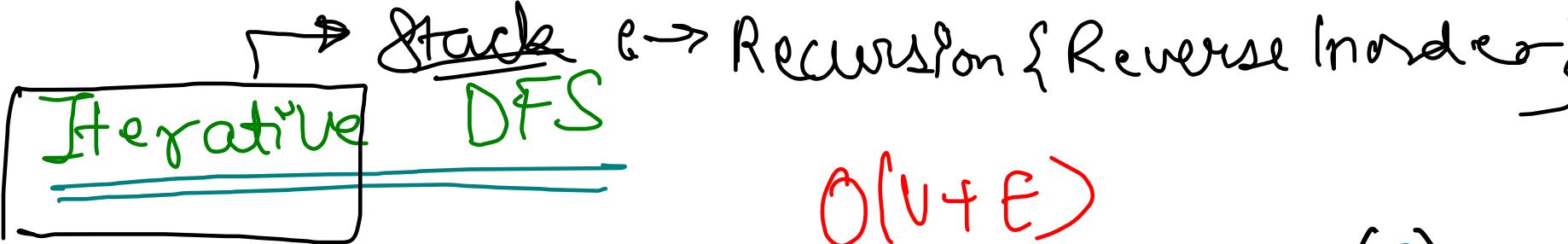


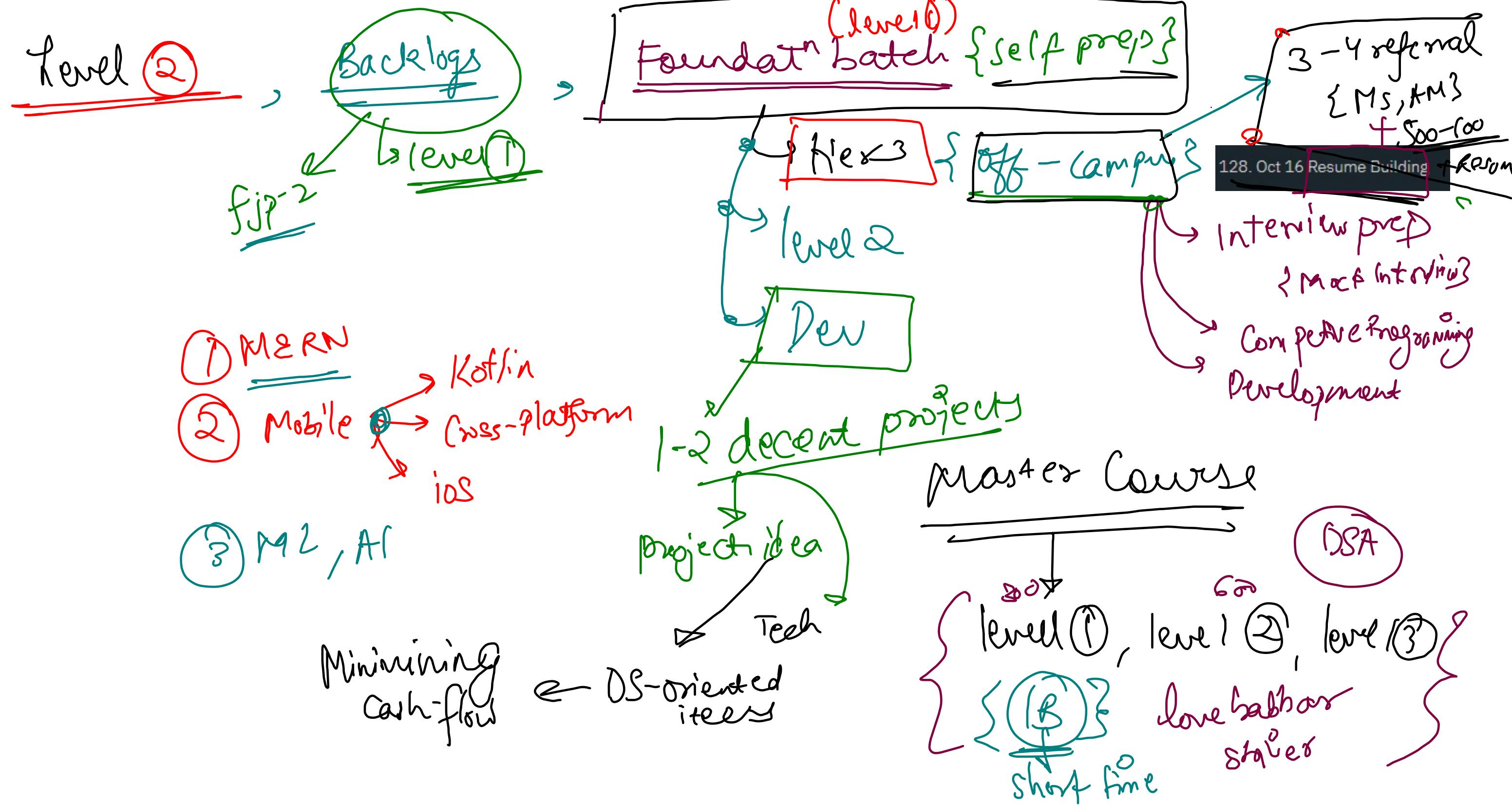
Postorder point

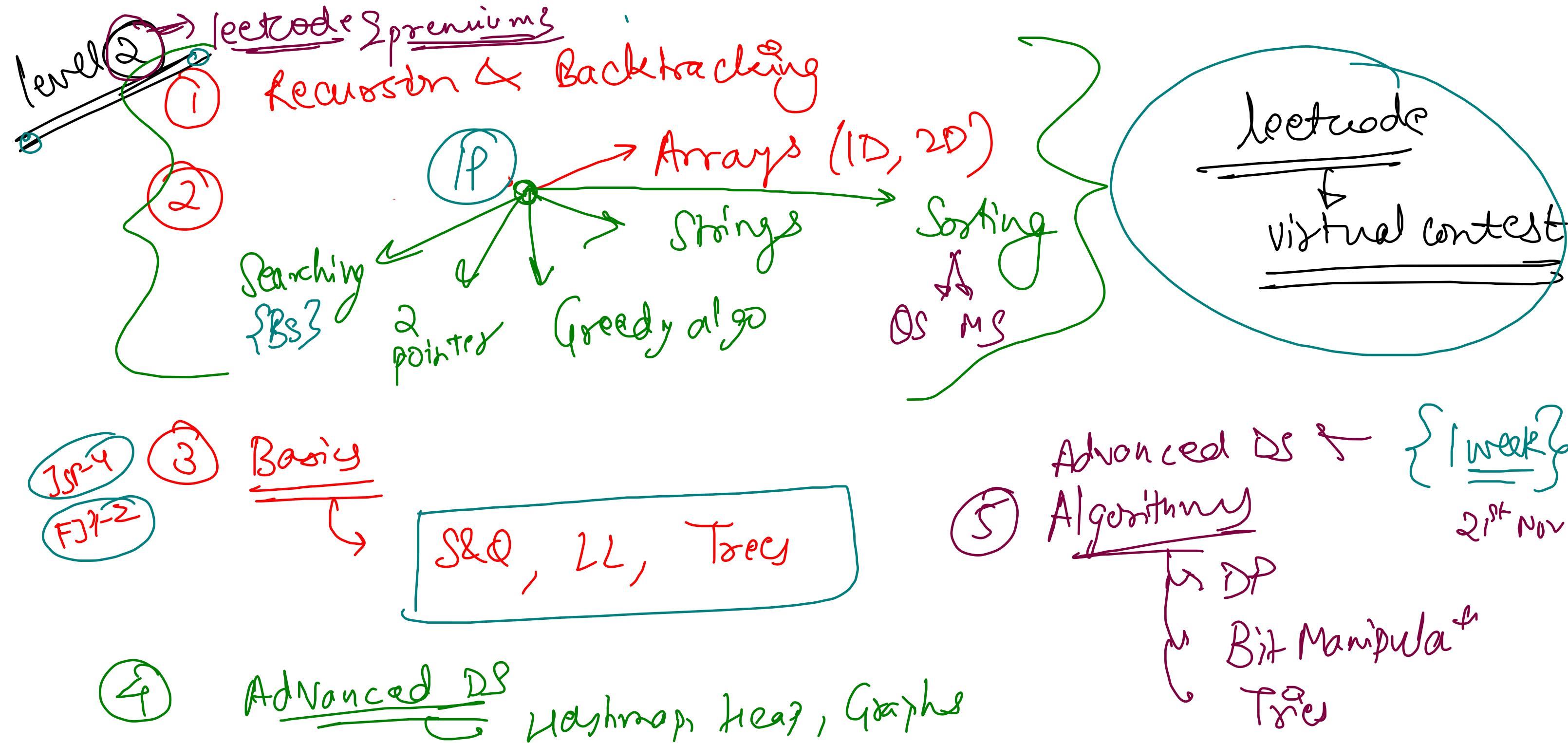


③ Stack push on preorder









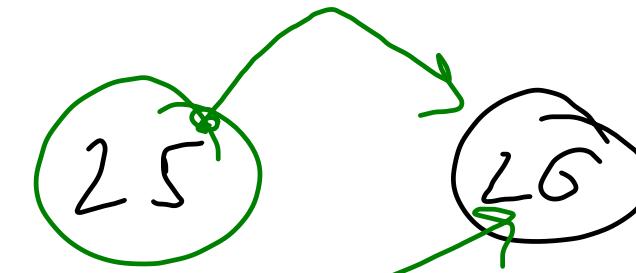


~~leetcode~~

rank

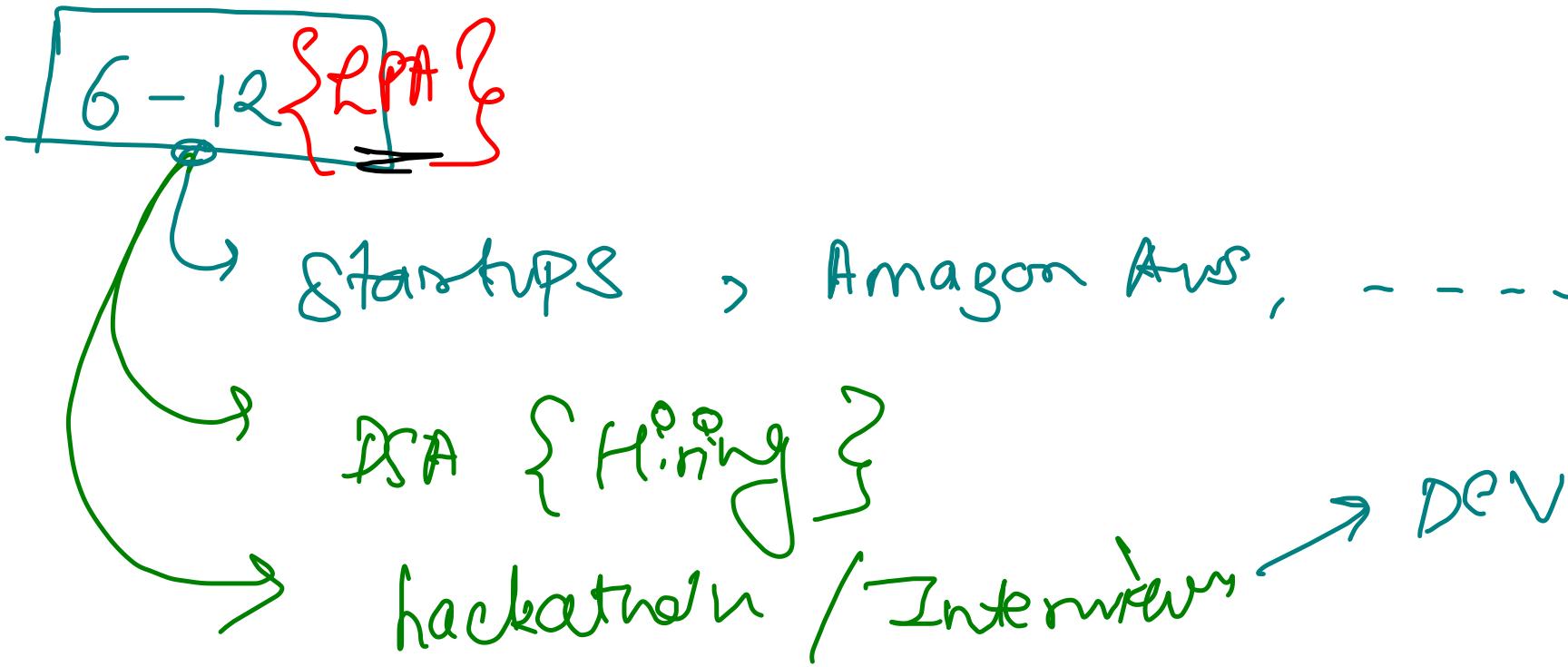
L2
L3

L4



- ① video
- ② code-submit
- ③ shoot important plots
- ④ Codeforces & leetcode

PBC
I
SDE
DSA Dev



21st Nov
↳ Level 2

