Graphs level 1

Data Structures

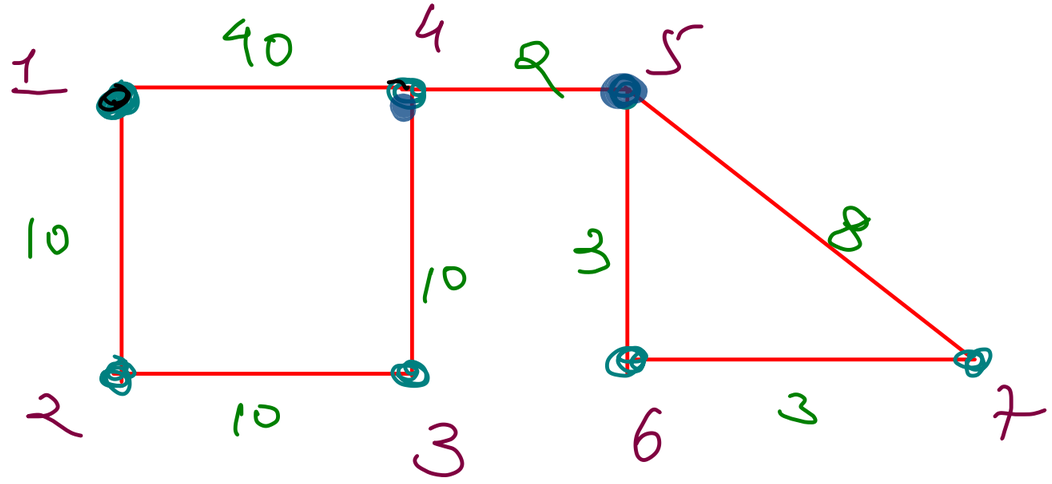
- → Arrays & Strings
- → Array list
- → Stack
- → Queue
- → Linked list
- → Generic Tree
- → Binary Tree
- → Binary Search Tree
- → Hashmap
- → Heap

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

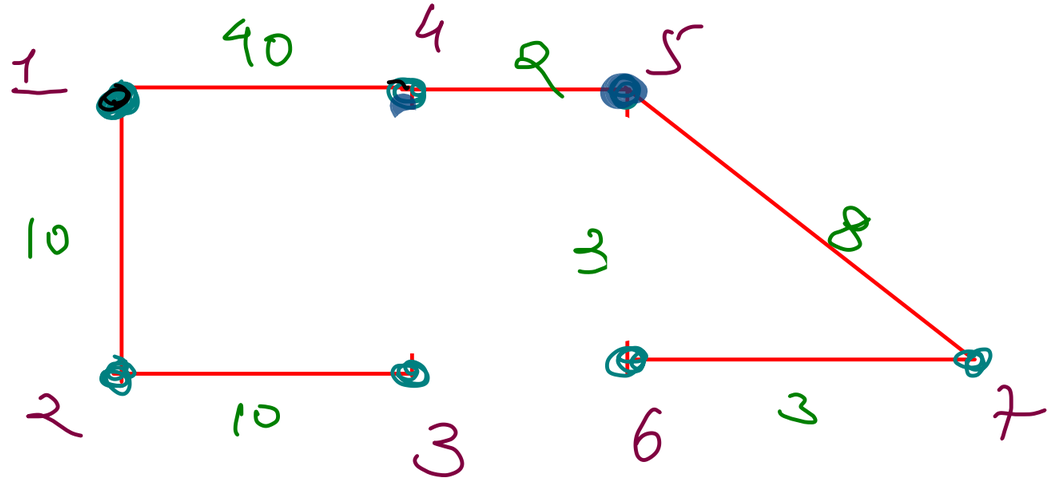


Applications

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



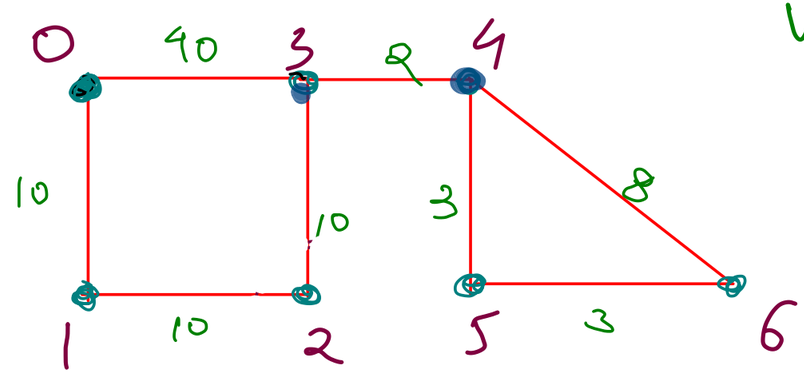
Applications

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 ⇒ $^nC_2$

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

$O(n^2)$

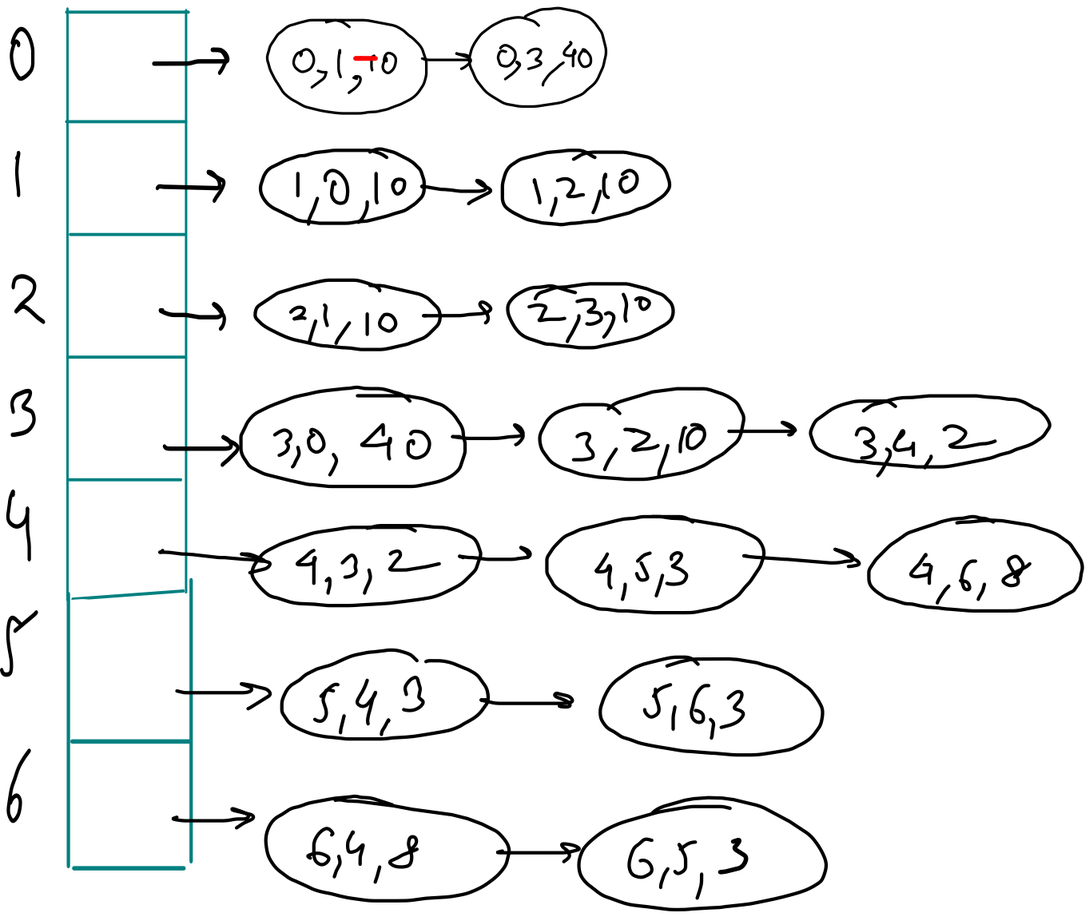
① Edge list

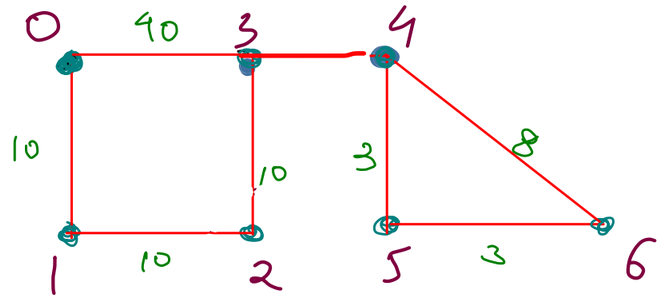
{ 0,3,40}, { 0,1,10}, { 1,2,10},

{ 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

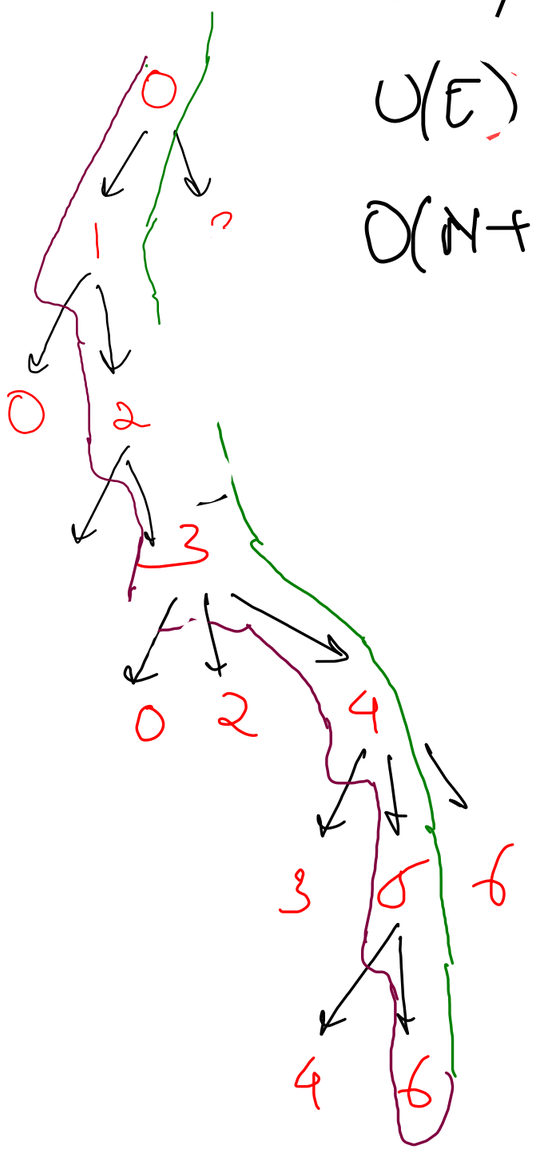
```java
int edges = scn.nextInt();

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));
}
```

Construction

```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

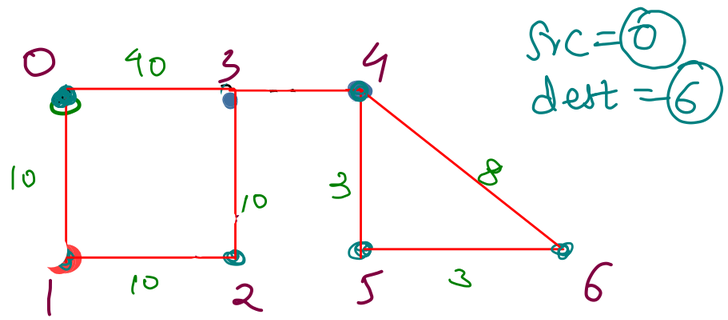
has Path

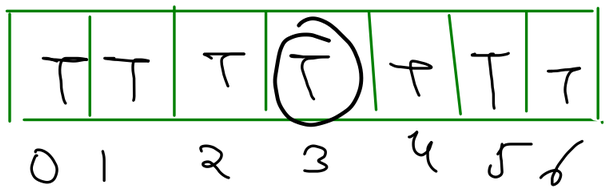


O(N)

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%

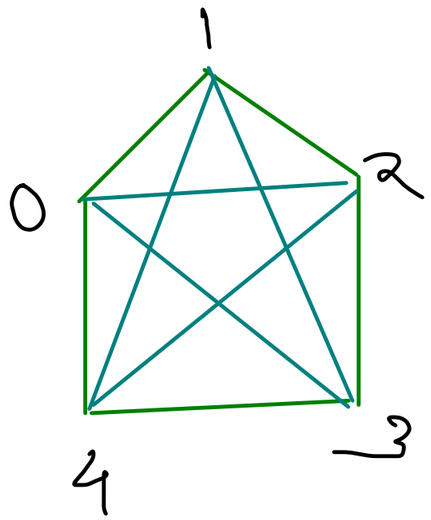
B Has path - O(N + E), Print All Paths - Exponential (14/30) 47%

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

Backtracking ⇒ ~~polynomial~~ Exponential

Recursion

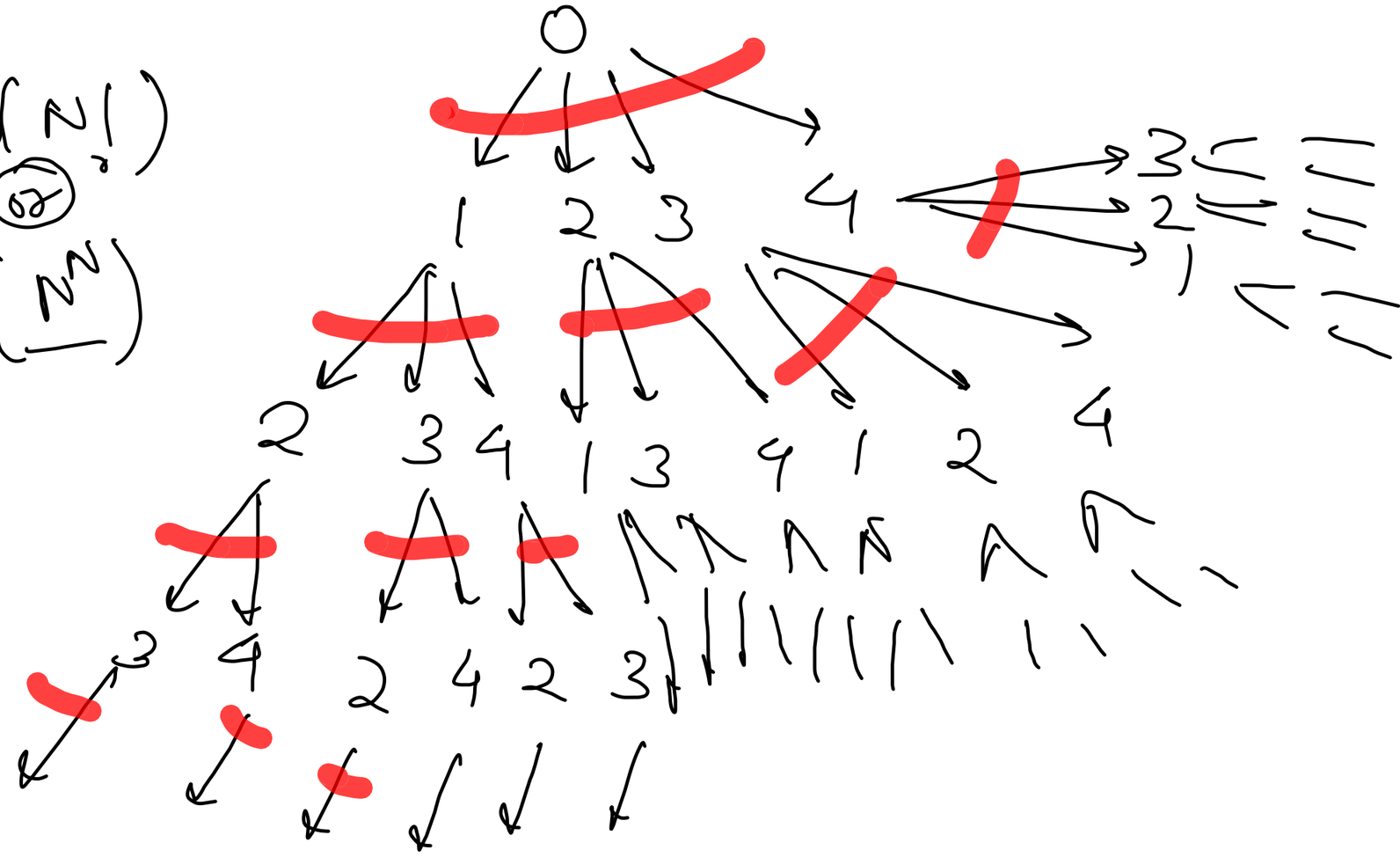
$(\text{calls})^{\text{height}} = \left(\frac{E}{N}\right)^{N} = \left(\frac{N^2}{N}\right)^{N} = O(N^N)$



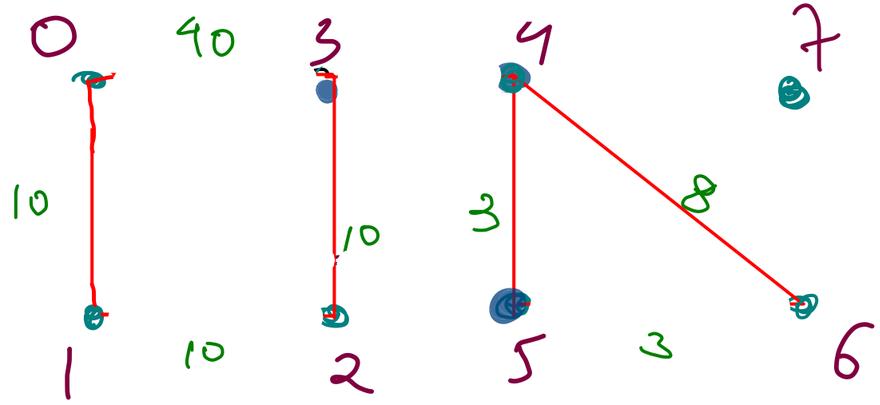
$O(N!)$

or

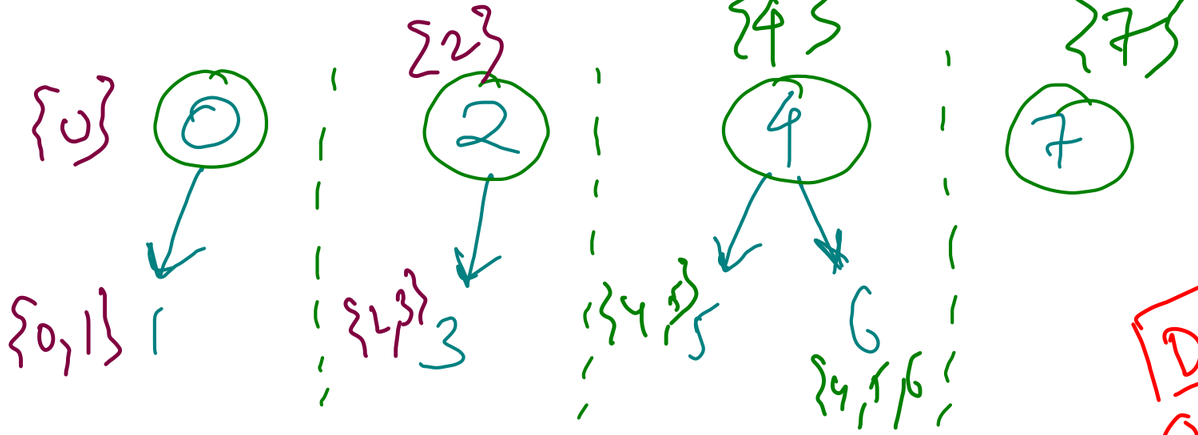
$O(N^N)$



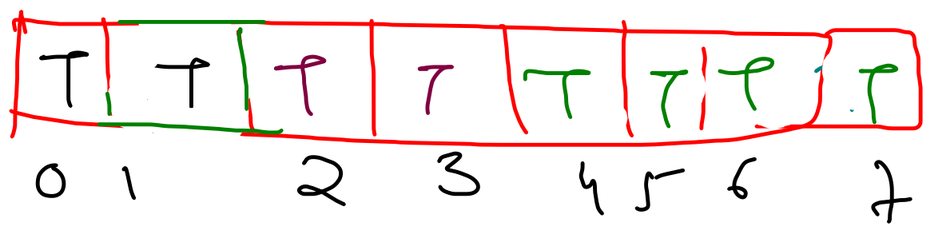
connected components {undirected}



{{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);
    }
}
```

</> Multisolver - Smallest, Longest, Ceil, Floor, Kthlargest Path

</> Is Graph Connected

</> Number Of Islands

</> Perfect Friends

</> Hamiltonian Path And Cycle