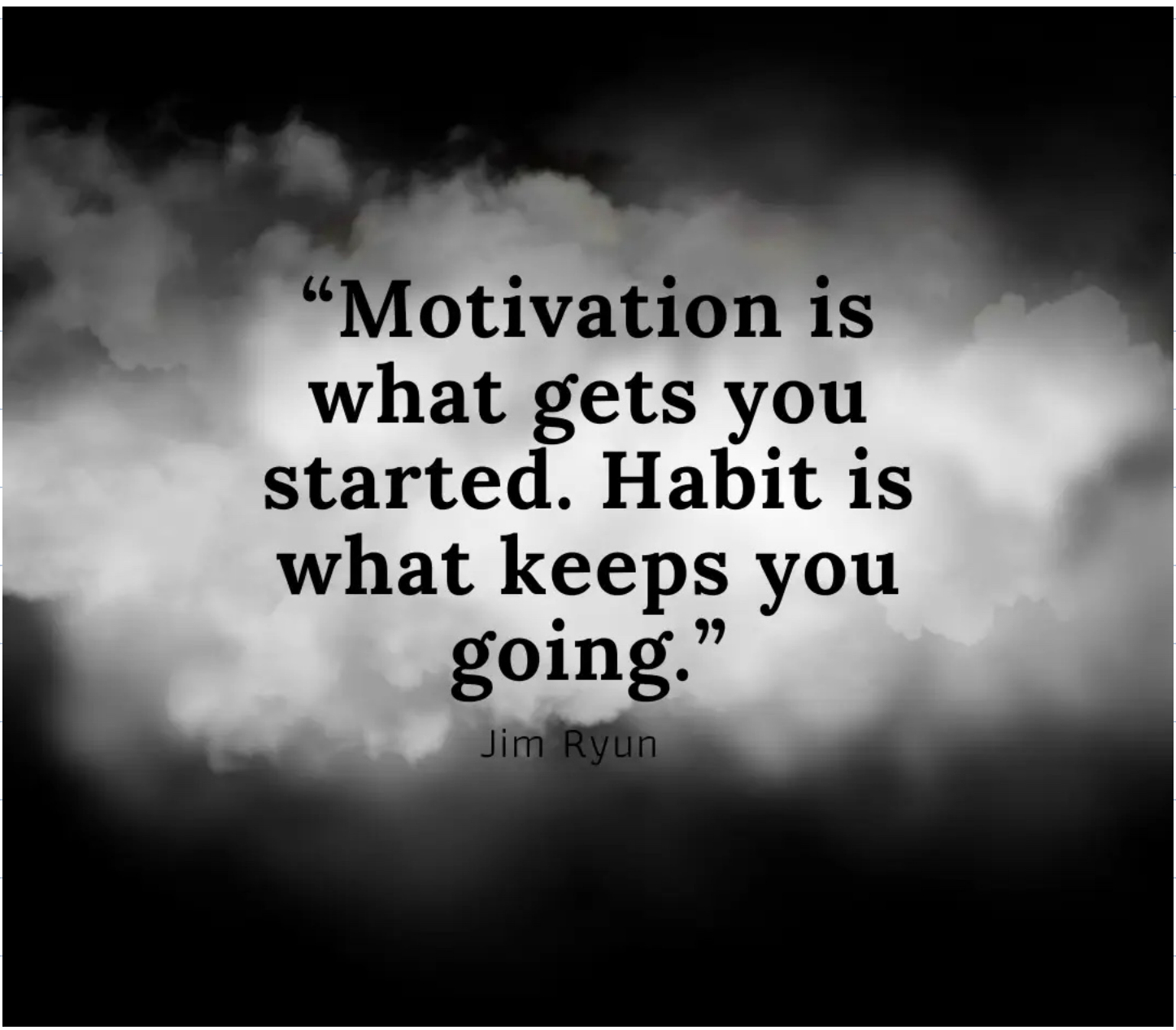
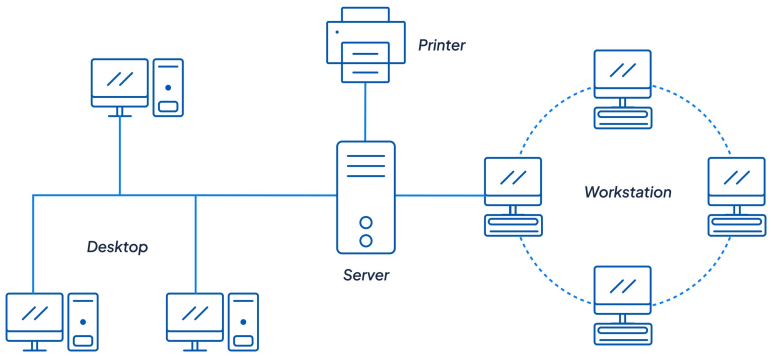
Graphs 1 : Introduction, DFS & Cycle Detection

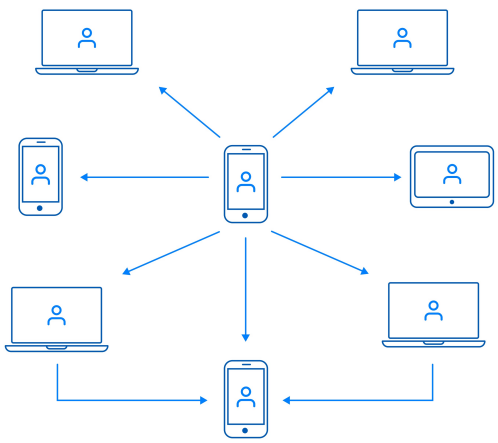
TABLE OF CONTENTS

1. Introduction to Graphs
2. Types of Graphs
3. DFS
4. Detect Cycle in directed graph



Graphs

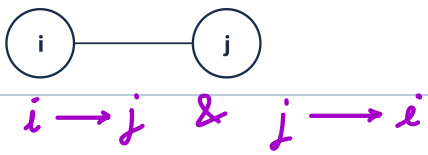




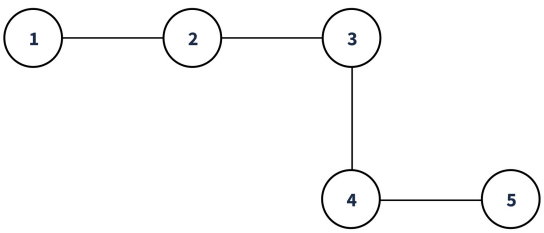
Properties / Types of Graphs

1. Directed

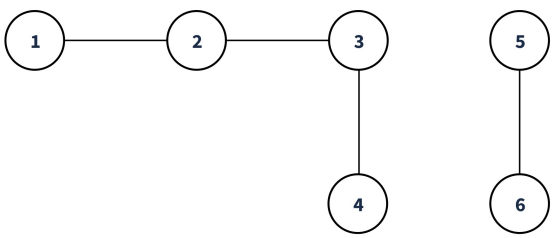
Un-directed graph



2. Connected

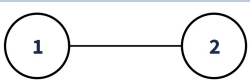


Disconnected

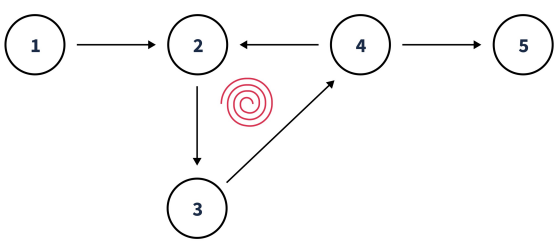


3. Weighted

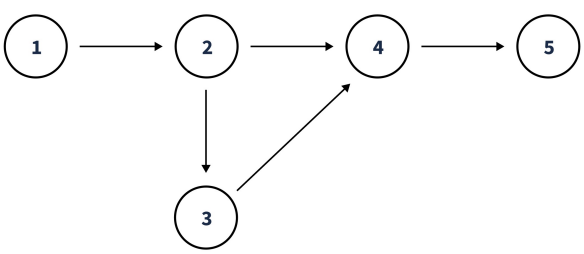
Unweighted

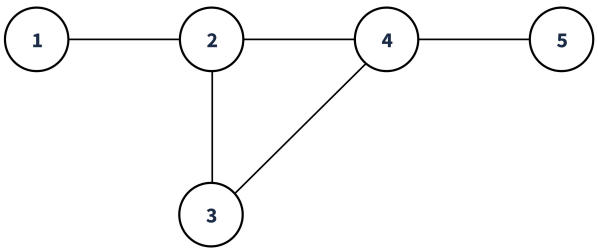


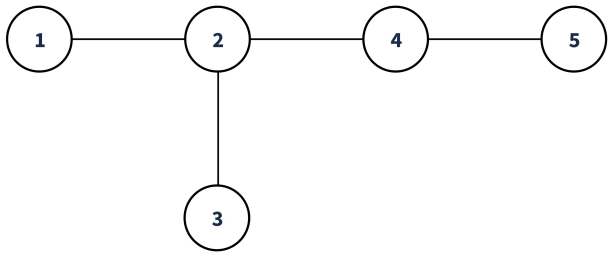
4. Cyclic



Acyclic

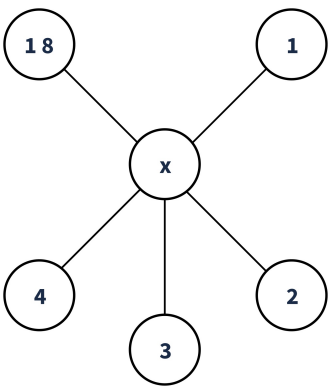






For undirected graph min 3 node cycle ⇒ cyclic graph.

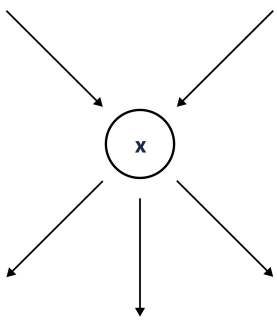
5. Degree



degree(x) = 5

edges connected

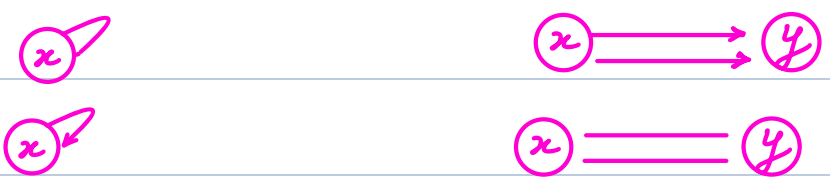
In-degree and Out-degree

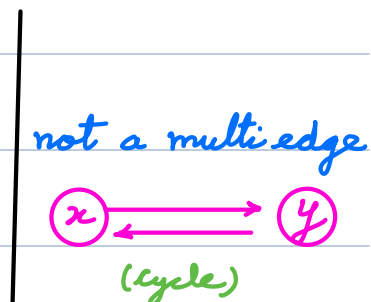


in-degree(x) = 2 incoming edges

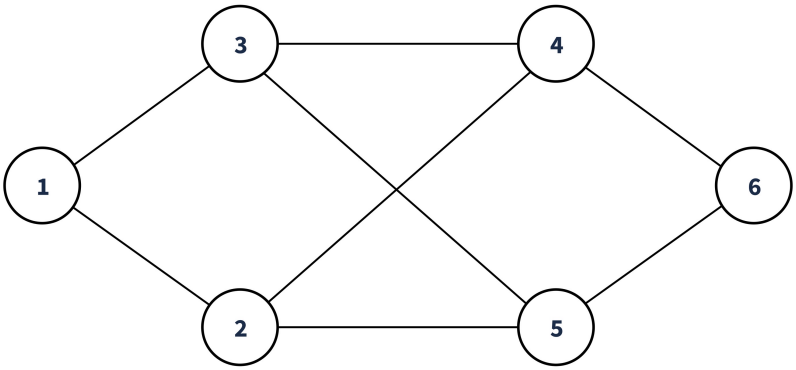
out-degree(x) = 3 outgoing edges

6. Simple Graph → A connected graph without self loops & multiedges.





How to store a graph?



nodes → N = 6

edges → E = 8

1. Adjacent Matrix

N * N matrix

Unweighted

A[i][j] → 1 → i → j

→ 0 → no edge

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

Weighted

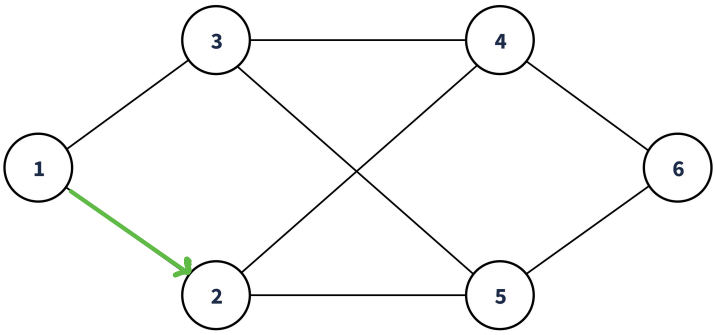
A[i][j] → wt → i —wt→ j

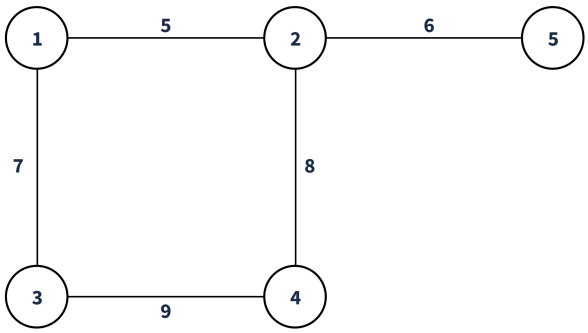
→ 0 → no edge

SC = O(N²)

2. Adjacent List

A[i] → list of nodes connected to i.





outgoing edges in case of directed graph.

0	
1	{2, 3}
2	{~~1~~, 4, 5}
3	{1, 4, 5}
4	{2, 3, 6}
5	...
6	...

0	
1	{(2,5), (3, 7)}
2	{(1,5), (4,8), (5,6)}
3	{(4,9), (1, 7)}
4	...
5	...

A[i] → list of pairs (j, w) i.e. $i \xrightarrow{w} j$

SC = O(N + E) — sum of length of all lists

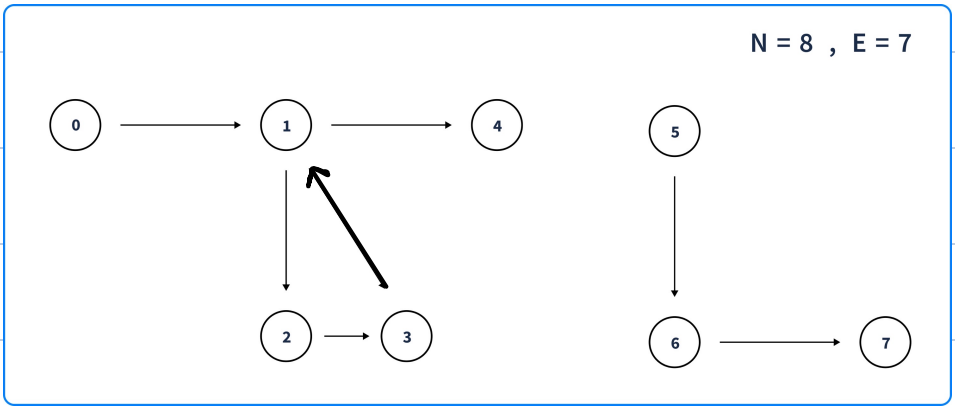
N <= 10^5 } use Adj. List

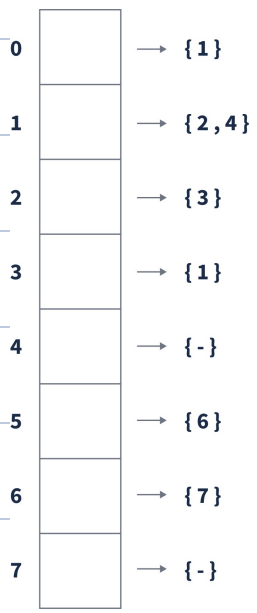
E <= 10^5

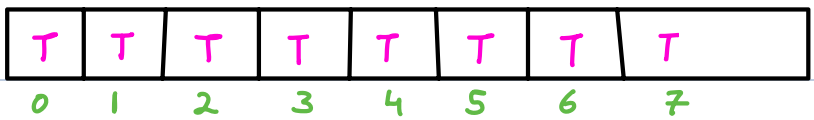
Traversal

Preorder / Inorder / Postorder

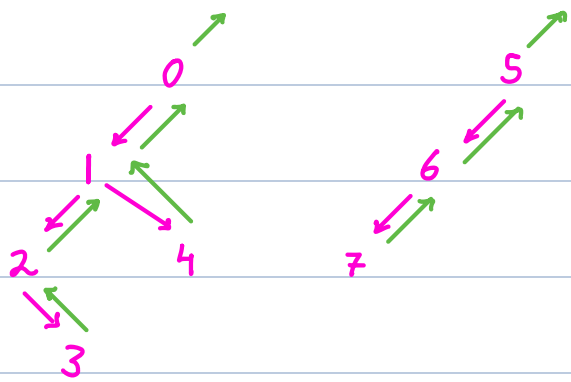
Depth First Traversal







Keep track of visited nodes.



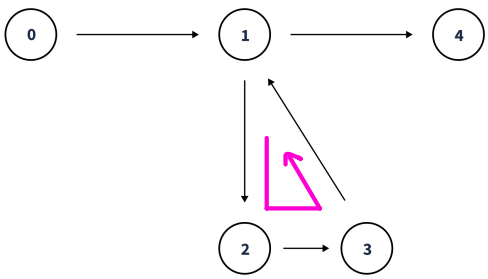
o/p → 0 1 2 3 4 5 6 7

```
∀i , vst[i] = false
for i → 0 to N {   // 1 → N
    if (!vst[i])   dfs(i)
}
```

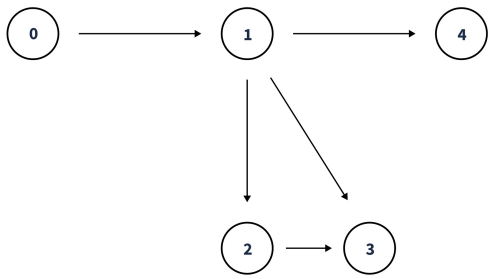
```
void dfs(i) {
    vst[i] = true
    print(i)
    for ( j : Adj[i] ) {   // i → j
        if (!vst[j])   dfs(j)
    }
}
```

TC = O(N+E) SC = O(N) recursion + vst[]

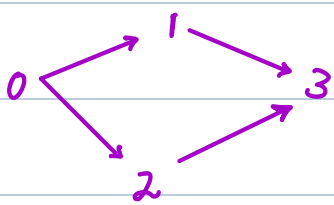
< *Question* > : Check if given directed graph has a cycle or not.



Ans = true



Ans = false



Ans = false

Cycle is present if we visit the same node again in the current path.

path[i] → True if node is in current path.

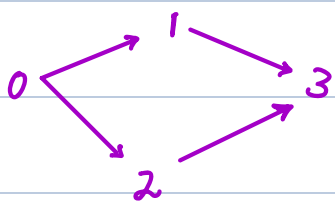
∀i, vst[i] = false ∀i, path[i] = false

```
for i → 0 to N {  // 1 → N
    if (!vst[i]) dfs(i)
}
```

```
boolean dfs(i) {
    vst[i] = true
    path[i] = true

    for ( j : Adj[i] ) {   // i → j
        if (path[j]) return true
        if (!vst[j] && dfs(j)) return true
    }

    path[i] = false
    return false
}
```

TC = O(N+E)

SC = O(N + N + N) = O(N)



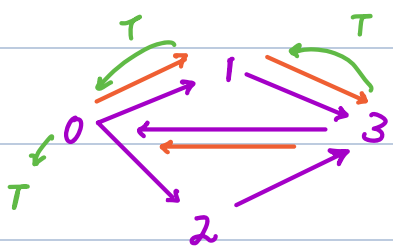
vst →

0	1	2	3
T	T	T	T

path →

0	1	2	3
F	F	F	F

Ans = false



vst →

0	1	2	3
T	T	F	T

path →

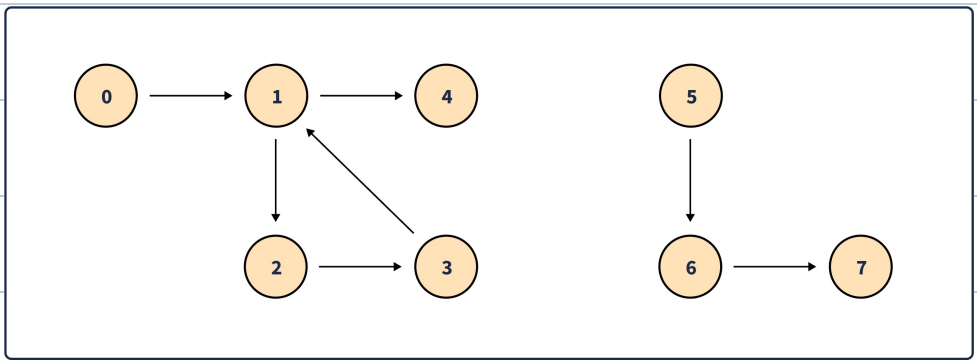
0	1	2	3
T	T	F	T

Ans = true

Agenda

1. BFS
2. Rotten Oranges
3. Number of Islands
4. Shortest Distance in a Maze

Breadth First Traversal (BFS)



0 → { 1 }

1 → { 2 , 4 }

2 → { 3 }

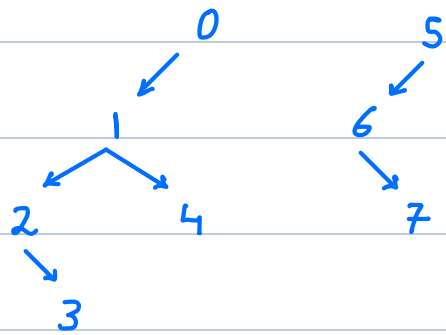
3 → { 1 }

4 → { - }

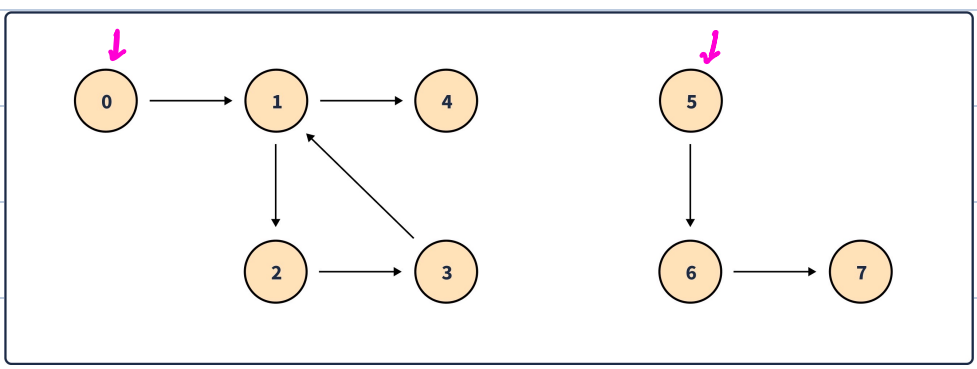
5 → { 6 }

6 → { 7 }

(Level Order Traversal) → Queues



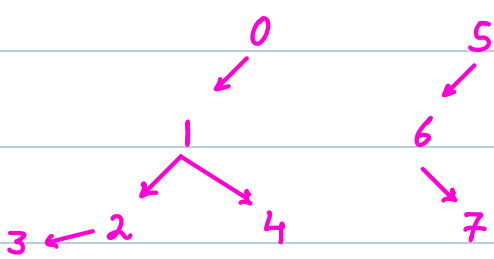
o/p → 0 1 2 4 3 5 6 7



0 1 2 3 4 5 6 7

vst = [T T T T T T T T]

Queue ↙ ~~0~~ ~~1~~ ~~2~~ ~~4~~ ~~3~~ ~~5~~ ~~6~~ ~~7~~



o/p → 0 1 2 4 3 5 6 7

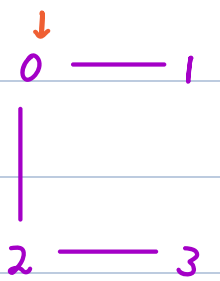
(while inserting or removing)

```
∀i, vst[i] = false
for i → 0 to (N-1) {
    if (!vst[i])  bfs(i)
}

void bfs(u) {
    vst[u] = true
    q.enqueue(u)
    while (!q.isEmpty()) {
        u = q.dequeue()
        print(u)
        for (v : Adj[u]) {   // u → v
            if (!vst[v]) {
                q.enqueue(v)
                vst[v] = true
            }
        }
    }
}
```

TC = O(N + E)

SC = O(N) Queue / vst[]

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



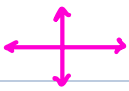
✓ 0 1 2 3 or

0 2 1 3

Rotten Oranges

There is given a matrix and there are 3 values where 0 means empty cell, 1 means fresh orange present and 2 means rotten orange present, we need to find the time when all oranges will become rotten.

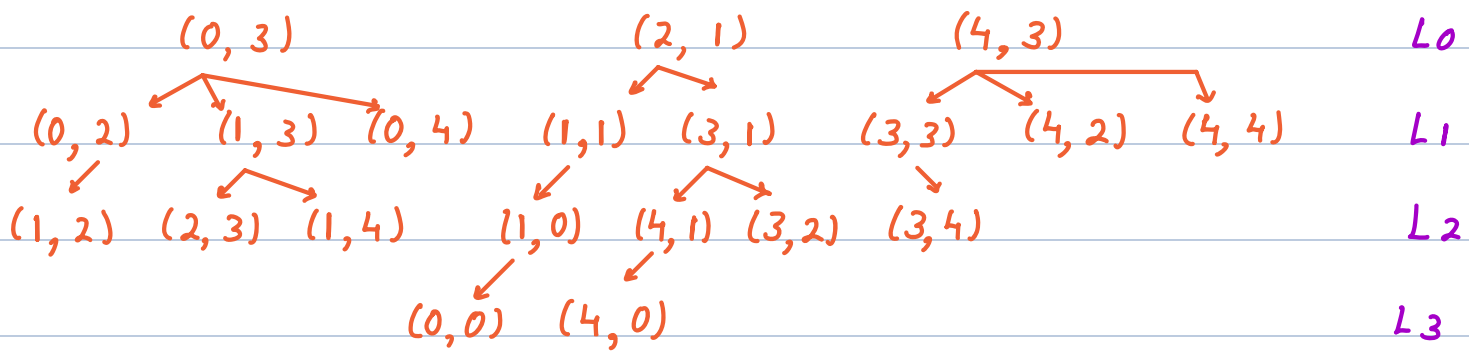
→ if that is not possible, return -1.



→ in every 1 minute, fresh orange adjacent to rotten orange becomes rotten.

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

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



Ans = 3 (∀ oranges, time >= 0)

↑ ← → ↓

dx = [-1 0 0 1]

dy = [0 -1 1 0]

∀i, j → T[i][j] = -1

```
for i → 0 to (N-1) {
    for j → 0 to (M-1) {
        if ( A[i][j] == 2 ) {   // Rotten orange
            T[i][j] = 0
            q.enqueue ( {i, j} )
        }
    }
}
```

```
while (! q.isEmpty ()) {
    r , c = q.dequeue ()
    for i → 0 to 3 {   ↔↕
        x = r + dx[i]        y = c + dy[i]
        if ( 0 <= x && x < N && 0 <= y && y < M
            && A[x][y] == 1 && T[x][y] == -1 ) {   // unvisited fresh orange
            T[x][y] = T[r][c] + 1
            q.enqueue (x, y)
        }
    }
}
```

```
ans = 0
for i → 0 to (N-1) {
    for j → 0 to (M-1) {
        if (A[i][j] == 1) {   // fresh orange
            if (T[i][j] == -1)  return -1  // unvisited
            else  ans = max(ans, T[i][j])
        }
    }
}
return ans
```

TC = O(N * M) SC = O(N * M)

Flipkart Delivery Optimisation

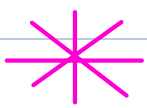
Flipkart Grocery has several warehouses spread across the country and in order to minimise the delivery cost, whenever an order is placed they try to deliver the order from the nearest warehouse.

Therefore, each Warehouse is responsible for a certain number of localities which are closest to it for deliveries, this minimises the overall cost for deliveries, effectively managing the distribution workload and minimising the overall delivery expenses.

Problem Statement

You are given a 2D matrix A of size NXM representing the map, where each cell is marked with either a 0 or a 1. Here, a 0 denotes a locality, and a 1 signifies a warehouse. The objective is to calculate a new 2D matrix of the same dimensions as A .

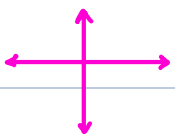
In this new matrix, the value of each cell will represent the minimum distance to the nearest warehouse. For the purpose of distance calculation, you are allowed to move to any of the eight adjacent cells directly surrounding a given cell.



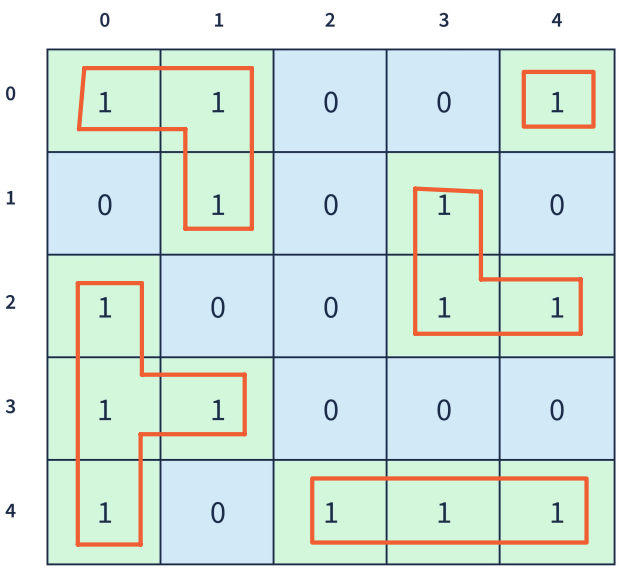
0 0 0 0 1 0 0 1 0
0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1

Number of Islands

You are given a 2D grid of '1's (land) and 'O's (water). Your task is to determine the number of islands in the grid. An island is formed by connecting adjacent (horizontally or vertically) land cells. Diagonal connections are not considered. Given here if the cell values has 1 then there is land and 0 if it is water, and you may assume all four edges of the grid are all surrounded by water.



arr[N][N]



Ans = 5

↑ ← → ↓

dx = [-1 0 0 1]

dy = [0 -1 1 0]

∀i, j → vst[i][j] = false

ans = 0

```
for i → 0 to (N-1) {
    for j → 0 to (M-1) {
        if ( A[i][j] == 1 && !vst[i][j]) { // unvisited land
```

```
            ans++
            dfs(i, j)
        }
    }
} return ans
```

```
void dfs(r, c) {
    vst[r][c] = true
    for i → 0 to 3 {
        x = r + dx[i]      y = c + dy[i]
        if (0 <= x && x < N && 0 <= y && y < M
            && A[x][y] == 1 && !vst[x][y])
                dfs(x)(y)
    }
}
```

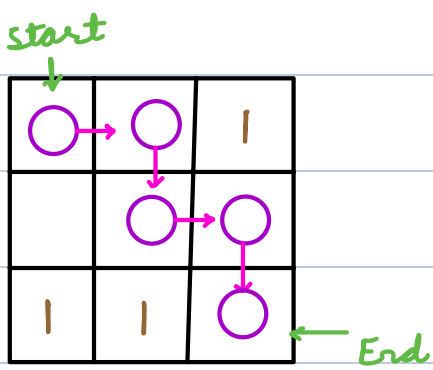
TC = $O(N * M)$ SC = $O(N * M)$

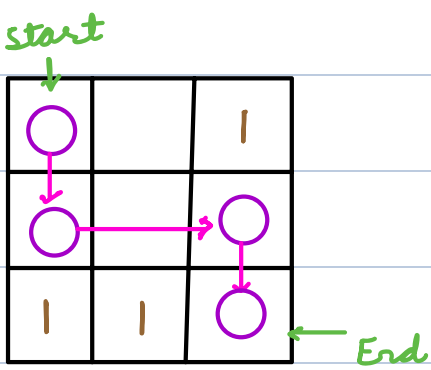
Shortest Distance in a maze

Given a matrix of integers A of size N x M describing a maze. The maze consists of empty locations and walls. 1 represents a wall in a matrix and 0 represents an empty location in a wall.

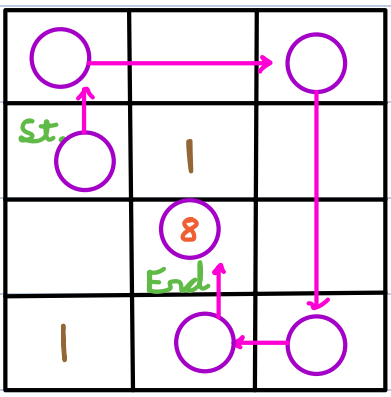
There is a ball trapped in a maze. The ball can go through empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall (maze boundary is also considered as a wall). When the ball stops, it could choose the next direction.

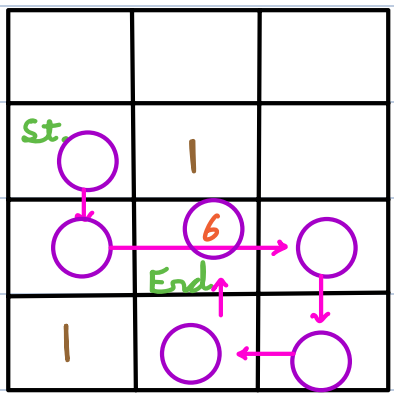
Given two array of integers of size B and C of size 2 denoting the starting and destination position of the ball. Find the shortest distance for the ball to stop at the destination. The distance is defined by the number of empty spaces traveled by the ball from the starting position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.



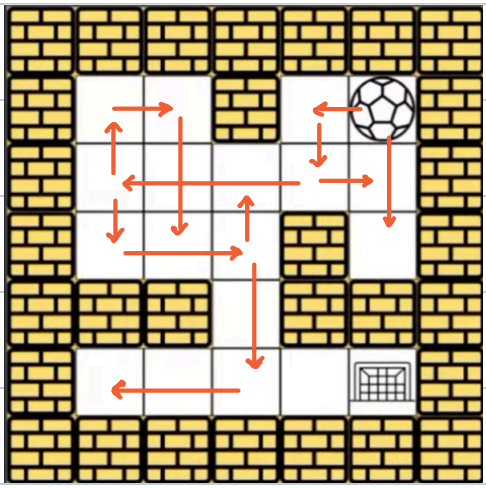


Ans = 4





Ans = 6



6	7		1	0 START
5	-1	9	2	3
6	9	8		2
		-1		
12	-1	10	-1	12 END

Distance

Ans = 12

Distance → BFS ✓

Path → DFS

↑ ← → ↓

dx = [-1 0 0 1]

dy = [0 -1 1 0]

∀ i, j → d[i][j] = -1 // Int Max

q.enqueue (start_x , start_y)

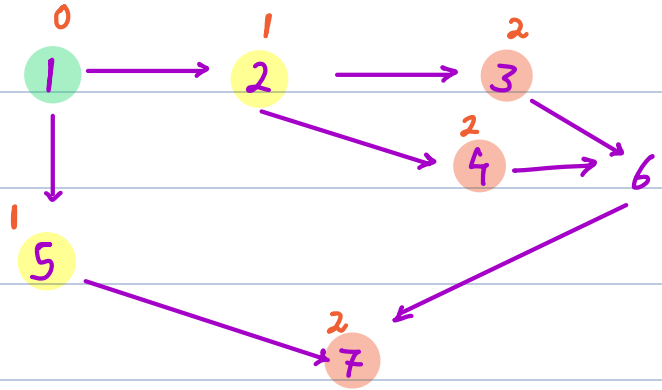
d [start_x] [start_y] = 0

```
while (! q.isEmpty ()) {
    r, c = q.dequeue ()
    for i → 0 to 3 {
        x = r   y = c   cnt = 0
        while ( 0 <= x && x < N && 0 <= y && y < M
                && A[x][y] == 0) { // Empty cell
```

```
            x += dx[i]    y += dy[i]
            crt++
        }
        x -= dx[i]   y -= dy[i]   crt--  // undo last step
        dist = d[r][c] + crt
        if (d[x][y] == -1) {  // unvisited
            d[x][y] = dist
            q.enqueue({x, y})
        }
    }
}
return d[end_x][end_y]
```

blocked

TC = O(N * M) SC = O(N * M)



Find min #edges to travel from 1 to 7.

⇒ if root = 1, find level of 7

value

weight

index

dp[index][wt] = value

find min wt to make every profit.

SCALER

Graphs 3

TABLE OF CONTENTS

1. M. S. T.
2. Prim's Algorithm
3. Dijkstra's Algorithm

Notes

Scenario:
Suppose Flipkart has N local distribution centers spread across a large metropolitan city. These centers need to be interconnected for efficient movement of goods. However, building and maintaining roads between these centers is costly. Flipkart's goal is to minimize these costs while ensuring every center is connected and operational.

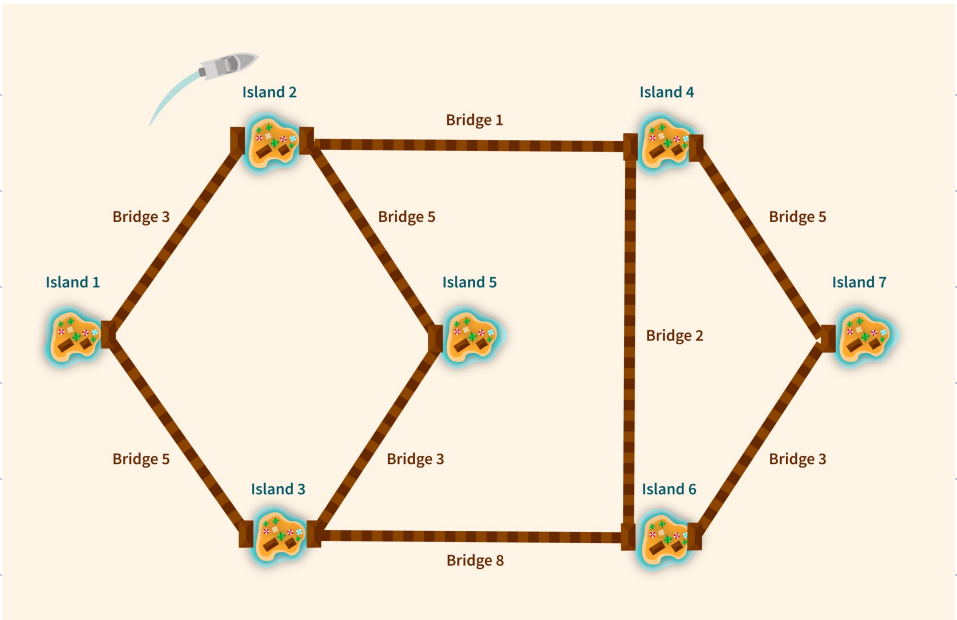
Goal: You are given number of centers and possible connections that can be made with their cost. Find minimum cost of constructing roads between centers such that it is possible to travel from one center to any other via roads.

< *Question* > : Given N islands and cost of construction of a bridge between multiple pair of islands. Find minimum cost of construction required such that it is possible to travel from one island to any other island via bridges. If not possible, return -1.

Example :

N= 7 E=9

1 <-3-> 2
1 <-5-> 3
2 <-1-> 4
2 <-5-> 5
3 <-3-> 5
4 <-2-> 6
3 <-8-> 6
4 <-5-> 7
6 <-3-> 7



N = 7 min # bridges = 6

min # bridges = N-1

Minimum Spanning Tree (MST)

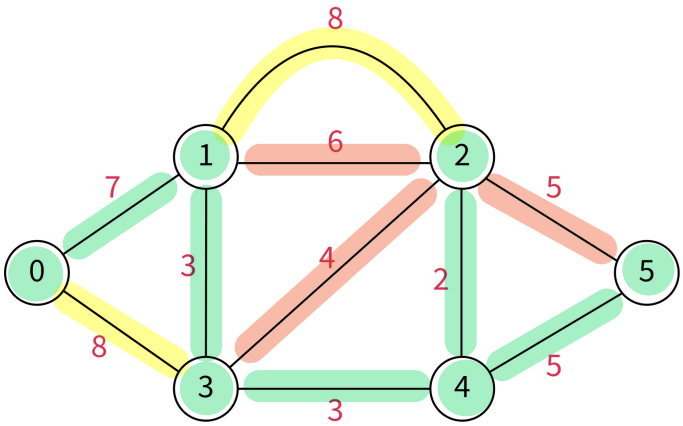
Tree constructed from a connected weighted graph s.t. the sum of weights of selected edges is min.

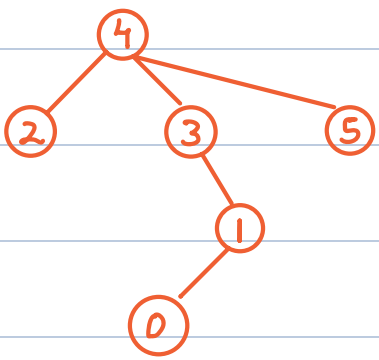
Algorithms → 1) Prim's Algo ✓

2) Kruskal's Algo

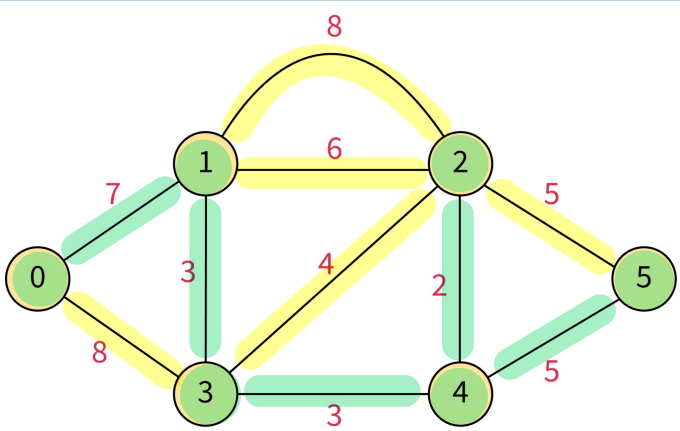
Prim's Algorithm

- Start with any node a the root node and keep on adding the other nodes with minimum weight.





sum of wt. = 20



Min Heap (weight, node)

~~(2, 2)~~ ~~(7, 0)~~

~~(5, 5)~~ ~~(3, 3)~~ (8, 1)

~~(6, 1)~~ ~~(4, 3)~~ ~~(5, 5)~~

~~(3, 1)~~ (8, 0)

	0	1	2	3	4	5
vst = [	T	T	T	T	T	T]

sum = 2 + 3 + 3 + 5 + 7 = 20 (Ans)

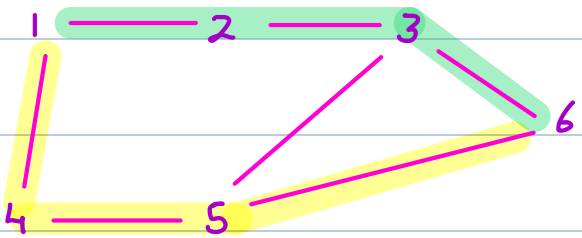
```
// min heap → h
∀i, vst[i] = false
vst[0] = true    // Any node can be root
for ( w, v : Adj[0]) {
    h.insert ({w, v})
}

sum = 0
while (! h.isEmpty ()) {
    w, v = h.getMin()        // Remove min wt edge
    if ( vst[v] )  continue
    vst[v] = true          sum += w
    for ( w, u : Adj[v]) {
        if ( !vst[u])  h.insert ({w, u})
    }
}

return sum
```

$TC = \underline{O(E * \log(E))}$ $\quad E >= (N-1)$

$SC = O(N + E) = \underline{O(E)}$

vst[] minHeap

Q → Given a simple graph find min # edges to travel from a given node u to v.



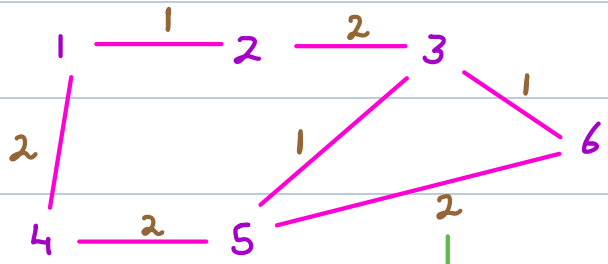
u = 1 v = 6

Ans = 3

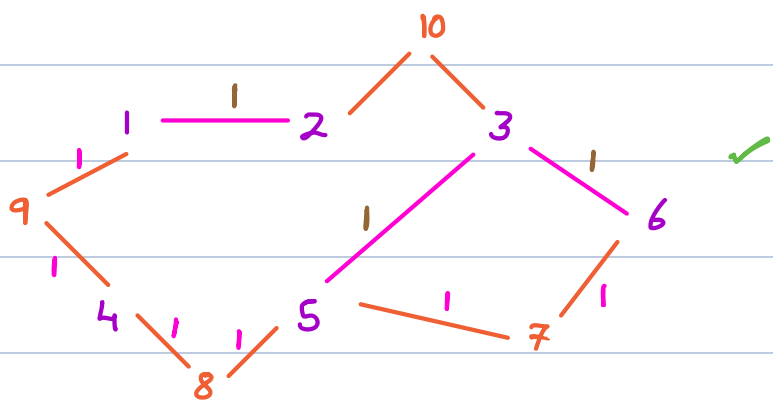
Sol → Strat BFS from u & Ans = level of v.

min distance in un-weighted graph

1 <= wt <= 2



add dummy nodes



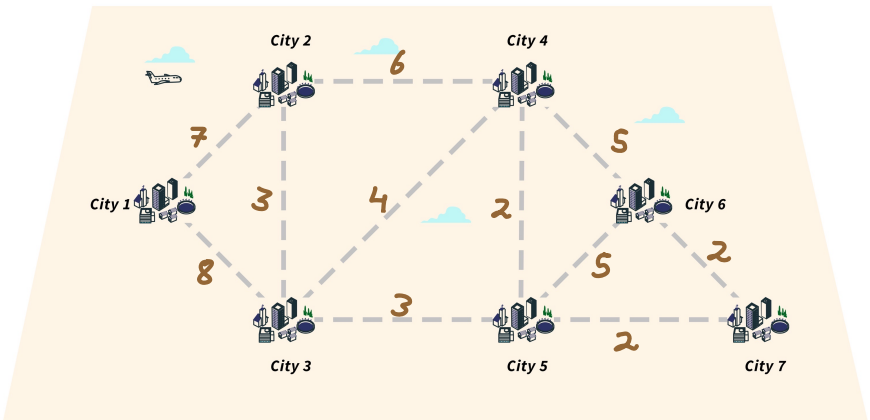
Dijkstra's Algorithm (Single Sourced Shortest Path)

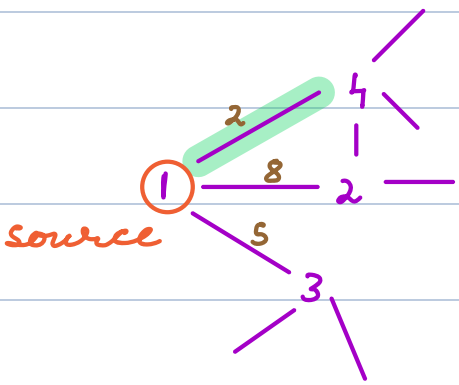
< *Question* > : There are N cities in a country, you are living in city 1. Find minimum distance to reach every other city from city 1. [edge wt > 0]

Expected Output ↓

dist →

0	7	8	12	11	16	13
1	2	3	4	5	6	7





Find the shortest distance to reach 4.

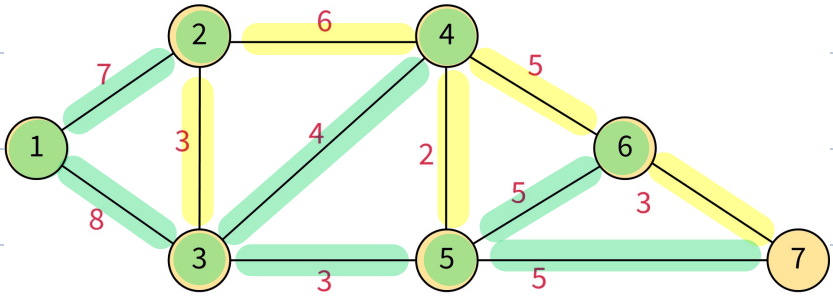
Ans = 2 ∵ its min wt edge from 1.

Find the shortest distance to reach 2.

Ans = 8 ✗

Direct edge b/w 2 nodes ≠ shortest distance b/w them.

Src - 1



dist[source] = 0

	1	2	3	4	5	6	7
dist →	0	7	8	12	11	16	16

p → [-1 1 1 3 3 5 5]

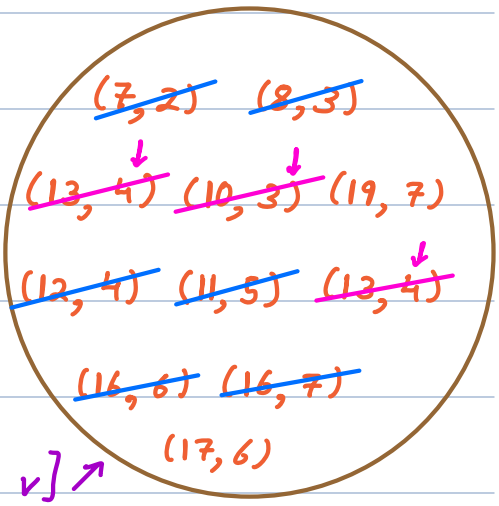
MinHeap

~~wt~~, node

(distance from source)

Find path to 7.

7 ← 5 ← 3 ← 1



[(d[u] + wt(u,v)), v]

(, u, v)

```
// min heap → h
∀i, dist[i] = INT_MAX
dist[s] = 0      // source node → s
for ( w, v : Adj[s]) {
    h.insert ({w, v})
}
```

```
while (! h.isEmpty ()) {
    d, v = h.getMin()        // Remove min dist
    if (dist[v] != INT_MAX) continue
    dist[v] = d
    for ( w, u : Adj[v]) {
        if (dist[u] == INT_MAX) h.insert (d+w, u)
    }
}
return dist
```

$TC = \underline{O(E * \log(E))}$ $\boxed{E >= (N-1)}$

$SC = O(N + E) = \underline{O(E)}$

dist[] minHeap

< *Question* > : Given N courses with pre-requisite of every course.

Check if it is possible to finish all the courses.

N = 5

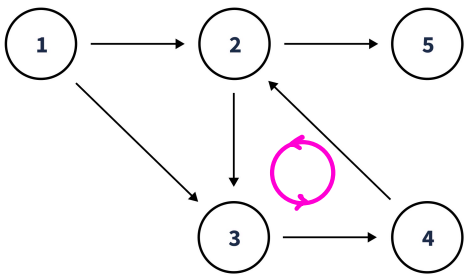
x is a pre-requisite of y $x \rightarrow y$

$1 \rightarrow 2, 3$

$2 \rightarrow 3, 5$

$3 \rightarrow 4$

$4 \rightarrow 2$

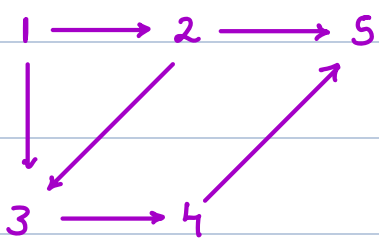


Ans = false

Sol → If cycle is present ⇒ false

else ⇒ true .

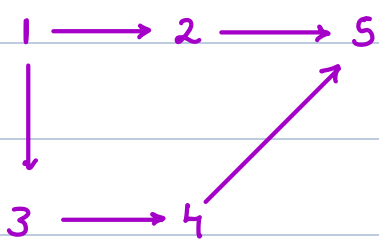
x is a pre-requisite of y $x \rightarrow y$



Ans = true

1 2 3 4 5

1 → 2 → 3 → 4 → 5



1 2 3 4 5 } multiple order

1 3 4 2 5 possible .

Topological Sort / Order

Linear ordering of the nodes such that if there is an edge from i to j, then i should be present on L.H.S of j

$i \rightarrow j$

DAG [Directed Acyclic Graph]

Find topological order →

N = 7

$0 \rightarrow \{1, 3\}$

$1 \rightarrow \{2, 3\}$

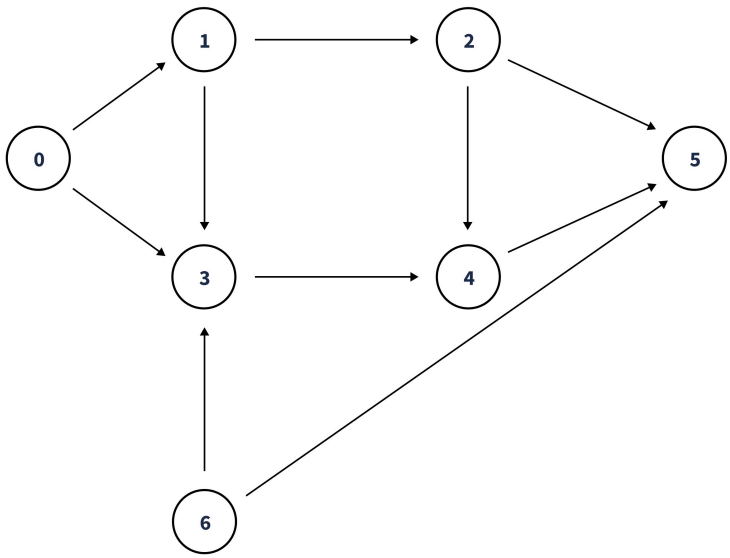
$2 \rightarrow \{4, 5\}$

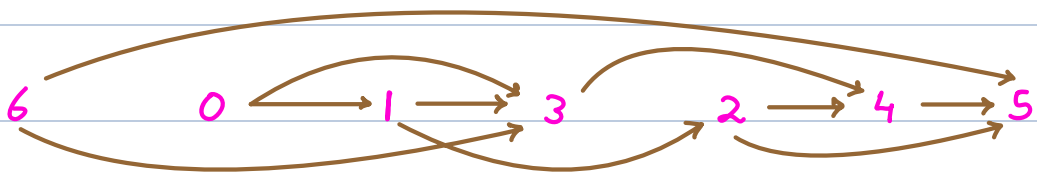
$3 \rightarrow \{4\}$

$4 \rightarrow \{5\}$

$5 \rightarrow \{-\}$

$6 \rightarrow \{3, 5\}$





Left to Right

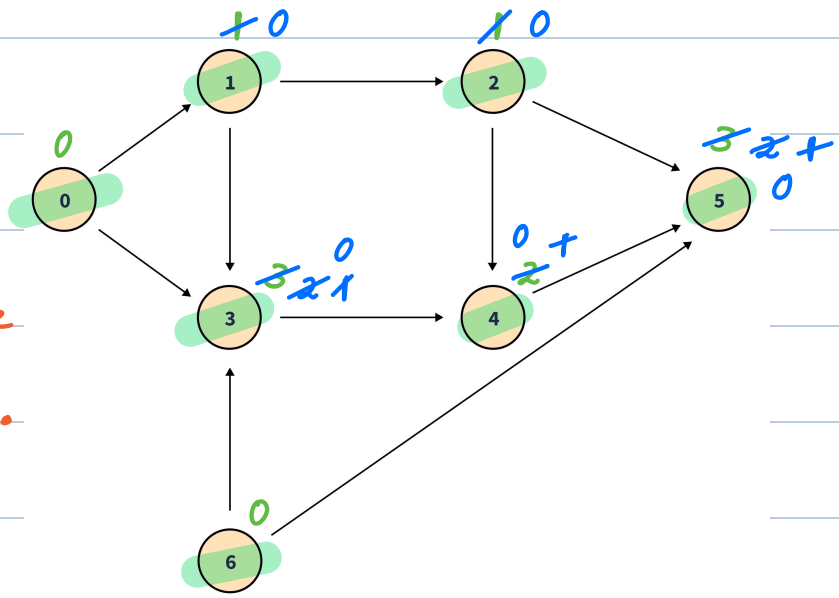
How to decide first node ?

Node with indegree = 0.

Find indegree ∀ nodes ↴

```
∀i, in[i] = 0
for u → 0 to (N-1) {
    for (v : Adj[u]) {   // u → v
        in[v]++
    }
}
```

TC = O(N + E)



1) Select all nodes with indegree 0 & store in a array / set / queue.

2) Select any one node from array, print it (o/p). 6 0 1 2 3 4 5

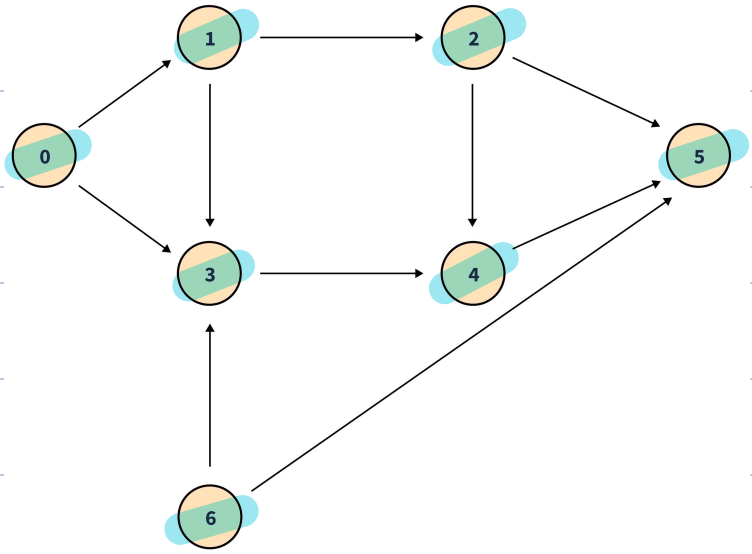
3) Decrease the indegree of adjacent nodes by 1, if updated indegree = 0, insert in array / set, & repeat from step 2 till all nodes are travelled.

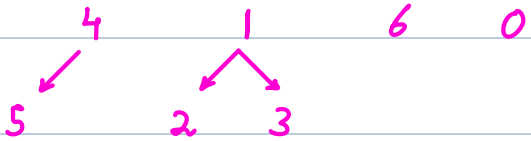
TC = O(N + E)

SC = O(N)

Right to Left

Start DFS from any node (till all nodes are visited).





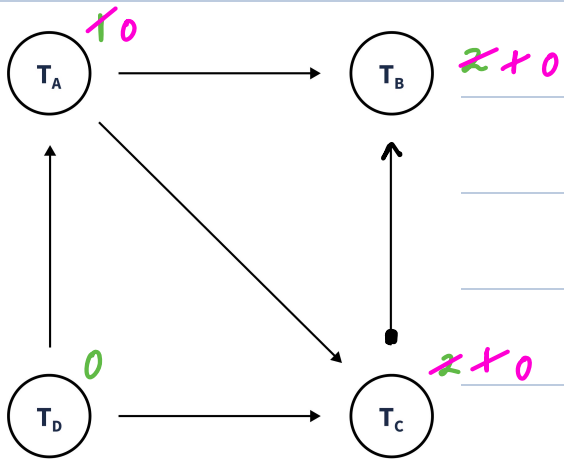
o/p → 5 4 2 3 1 6 0

```
∀ i, vst[i] = false
for i → 0 to (N-1) {
    if (!vst[i]) dfs(i)
}
```

```
void dfs(u) {
    vst[u] = true
    for (v : Adj[u]) {   u → v
        if (!vst[v]) dfs(v)
    }
    print(u)
}
```

TC = O(N + E) SC = O(N)

Quiz :



T_D T_A T_C T_B

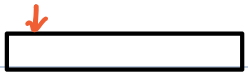
a) T_D, T_A, T_C, T_B ✓

b) T_A, T_D, T_C, T_B

c) T_C, T_A, T_D, T_B

Minimum jumps to reach end

You are given a 0-indexed array of integers arr of length n. You are initially positioned at arr[0].



Each element arr[i] represents the maximum length of a forward jump from index i. In other words, if you are at arr[i], you can jump to any arr[i + j] where:

* 0 <= j <= arr[i]
* i + j < n

Return the minimum number of jumps to reach arr[n - 1]. The test cases are generated such that you can reach arr[n - 1].

Example 1

Input: arr = [2,3,1,1,4]

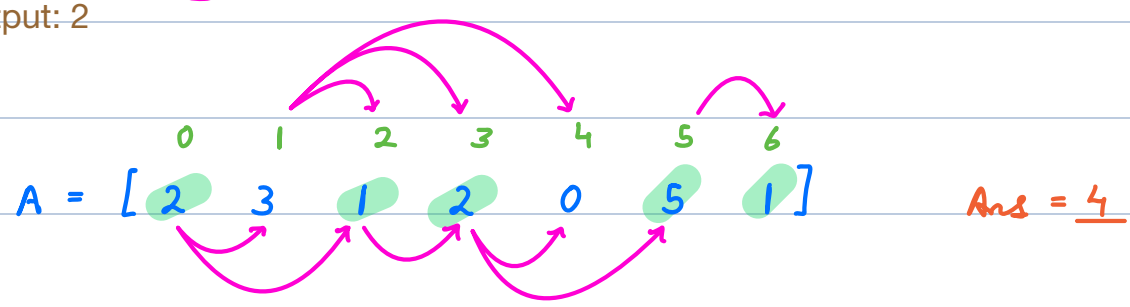
Output: 2

Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2

Input: arr = [2,3,0,1,4]

Output: 2



Next element from A[i] ⇒ A[i] <= element <= A[i + A[i]]

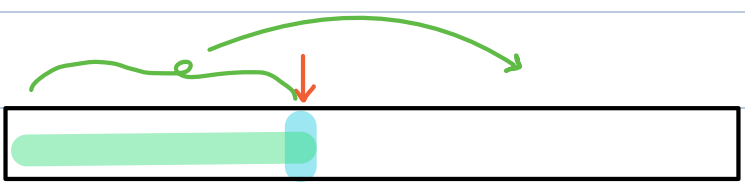
Sol. 1 → ∀ nodes, find min jumps to reach the node.

	0	1	2	3	4	5	6
A = [	2	3	1	2	0	5	1]
jump →	0	1	1	2	2	3	4

Ans = 4

```
if (A[0] == 0) return INT_MAX
∀ i, jump[i] = INT_MAX
jump[0] = 0
for i → 1 to (N-1) {
    for j → 0 to (i-1) {
        if ( j + A[j] >= i && jump[j] != INT_MAX )
            jump[i] = min (jump[i], jump[j]+1)
    }
} return jump[N-1]
```

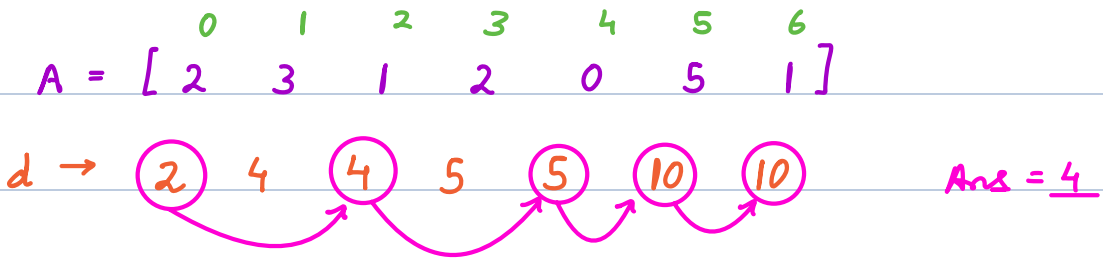
TC = $O(N^2)$ SC = $O(N)$



Sol. 2 → ∀ i, find max node we can reach considering nodes from 0 to i.

$i \longrightarrow i + A[i]$

```
d[0] = A[0]
for i → 1 to (N-1) {
    d[i] = max ( i + A[i], d[i-1] )
}
```




```
ans = 0
i = 0
while ( i < N ) {
    ans ++
    i = i + d[i]
}
return ans
```

TC = O(N) ✓

SC = O(N) → O(1)

d[] → A[]

Max profit from stock prices

Given an array A where the i-th element represents the price of a stock on day i, the objective is to find the maximum profit. We're allowed to complete as many transactions as desired, but engaging in multiple transactions simultaneously is not allowed.

b s b s ✓ b b s s ✗

A = [2 3 1 2 0 5 1] (0 1 2 3 4 5 6)

min on left → 2 2 1 1 0 0 0

Only 1 transaction → Ans = max (A[i] − min on left) ∀i

Multiple transactions

A = [2 3 1 2 0 5 1] (0 1 2 3 4 5 6)

1 + 1 + 5 = 7 (Ans)

A = [5 2 3 5 7 6 2 4 1 0] (0 1 2 3 4 5 6 7 8 9)

5 + 2 = 7 (Ans)

A = [5 ✗ 2 ✓ 3 ✓ 5 ✓ 7 ✗ 6 ✗ 2 ✓ 4 ✗ 1 ✗ 0] (0 1 2 3 4 5 6 7 8 9)

1 + 2 + 2 = 5 2

```
profit = 0
for i → 1 to (N-1) {
    if (A[i-1] < A[i])    profit += A[i] - A[i-1]
}
return profit
```

TC = O(N) SC = O(1)
