Revision of DSA Topics

Agenda

1. Recursion
2. Dynamic Programming
   - → Memoisation
   - → Tabulation
3. Graphs
   - → DFS
   - → BFS

Hello Everyone
very special Good Evening
to All of you
We will start
from 9:06 PM

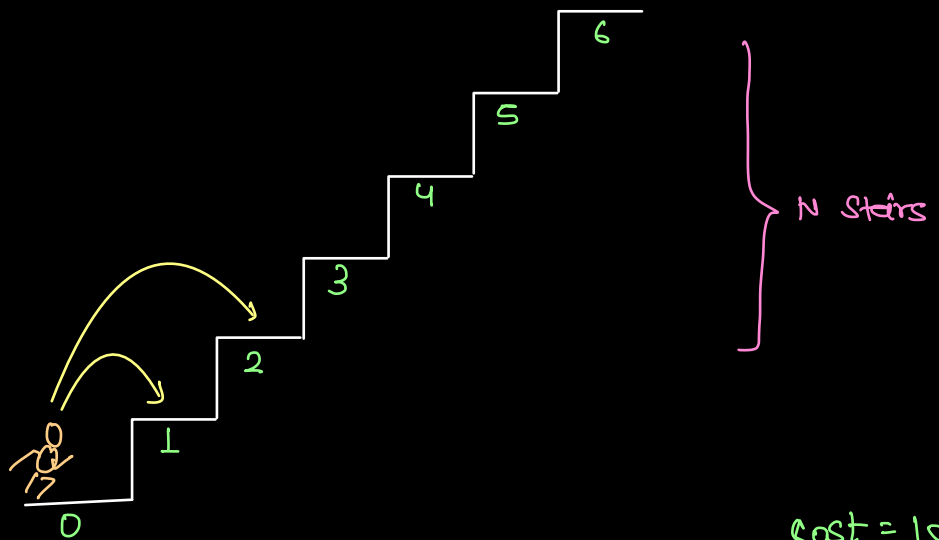
Content of DSA 4.2:

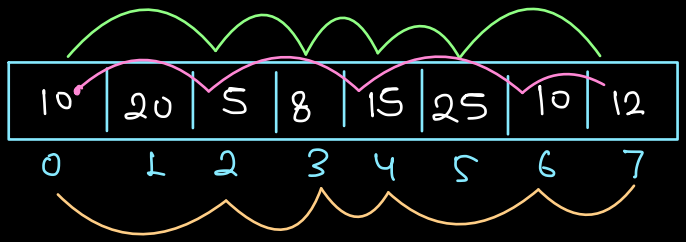
1. Revision of DSA topics ——— 1
2. Maths : inverse modulo and problems ——— 1
3. Backtracking ——— 1
4. Trie ——— 2
5. String Matching ——— 1
6. DP ——— 2
7. Graphs ——— 2
8. Contest ——— 1

Scaler Adventure Park :

Scaler has opened an Adventure Park and there is a staircase. Every staircase has a cost associated that you need to pay when you claims it.

You can either start from the 0th staircase or the 1st staircase.
* You can take 1 size jump or 2 size jump from any stair.
Goal: What is the least amount of money you can spend to reach the top.





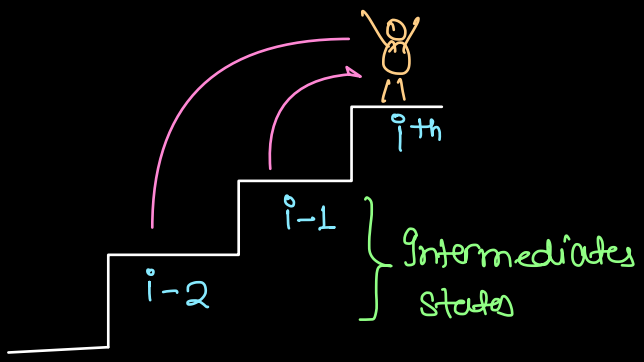
Cost = 10 + 5 + 8 + 15 + 25 + 12
= 75

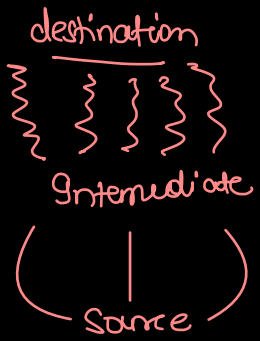
Cost = 10 + 5 + 8 + 15 + 10 + 12
= 60

Cost = 10 + 5 + 15 + 10 + 12
= 52

Ans = minCost(75, 60, 52 . . .)

= 52 Ans





$$cost(i) = \min Cost(\underbrace{cost(i-1), cost(i-2)}_{\text{min from Intermediate}}) + \underbrace{A[i]}_{\text{Self cost}}$$

i ⟶ (i-1) , (i-2)

Base case
- ✓ 0 ⟶ -1 ✗ OR -2 ✗
- ✓ 1 ⟶ 0 ✓ OR -1 ✗

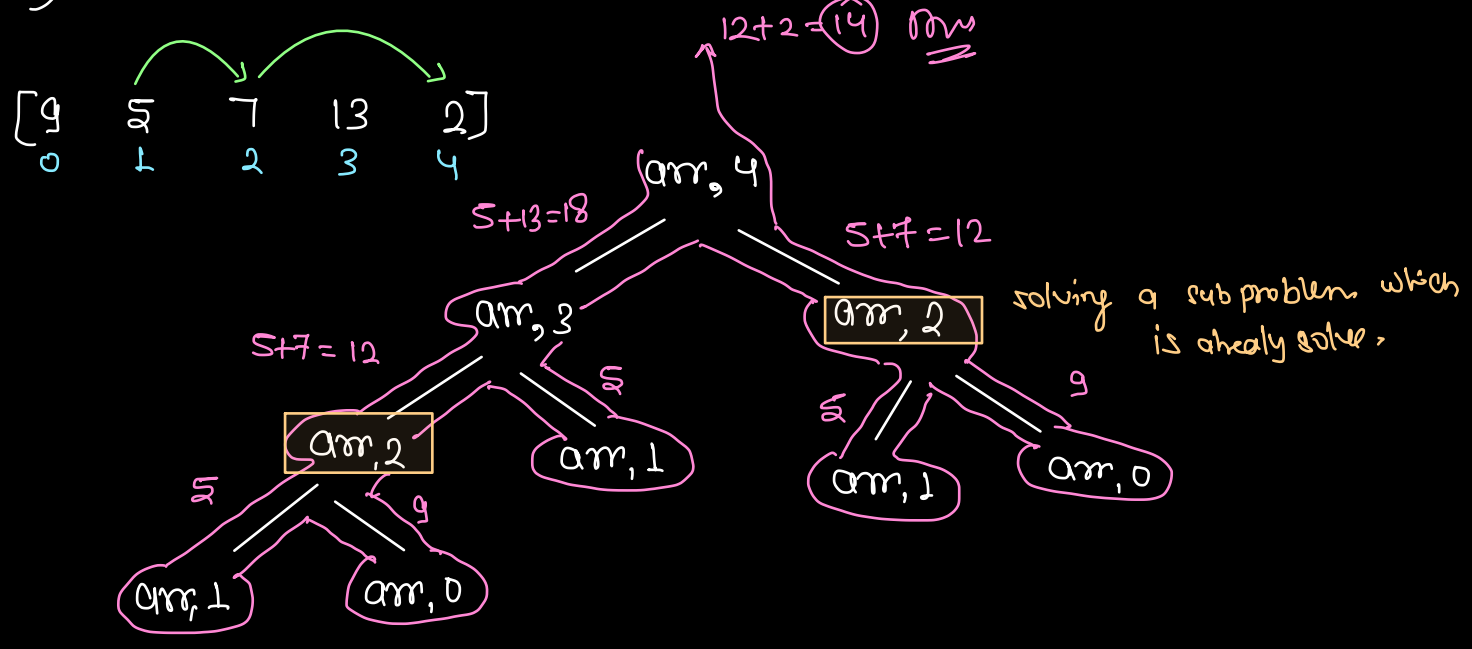
minCost(arr[], i) —
- minCost(arr[], i-1) P1
- minCost(arr[], i-2) P2

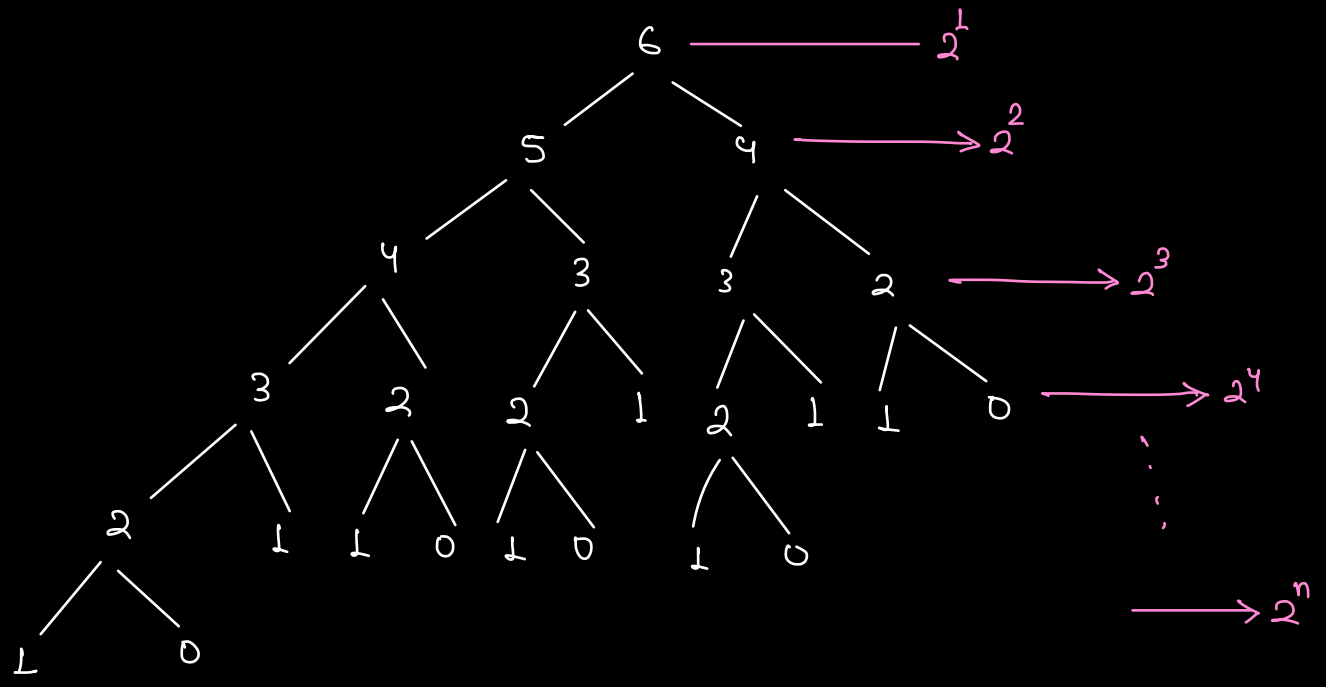
min(P1, P2) + arr[i]

pseudo code:

```
int minCost( arr[], int i) {
    if( i==0 || i==1) {
        return arr[i];
    }
    int val1 = minCost( arr, i-1);
    int val2 = minCost (arr, i-2);
    return min( val1, val2) + arr[i];
}
```

T.C: $O(2^n)$
S.C: $O(n)$





Total call $\Rightarrow$ $2^1+2^2+2^3+\cdots+2^n$

Sum of GP = ?

T.C : $O(2^n)$

$T(n) = T(n-1) + T(n-2) + k$ $\longrightarrow$ Recursive relation of fibbonacci

calculate $T(n)$?

$T(n-2)+T(n-2)+k \;<\; T(n) \;<\; T(n-1)+T(n-1)+k$

$\underbrace{2T(n-2)+k}_{f_1(n)} \;<\; T(n) \;<\; \underbrace{2T(n-1)+k}_{f_2(n)}$

$\downarrow$

$\underbrace{2^n}_{\text{upper bound}}$

$\Rightarrow \quad T(n) : O(2^n)$

Dynamic Programing:

1. memoisation : [Top down] : Recursive
2. Tabulation : [Bottom up] : Iterative

* Memoisation:

1. Decide storage
2. Store your answer in the storage before returning it
3. check if answer is pre-calculated.

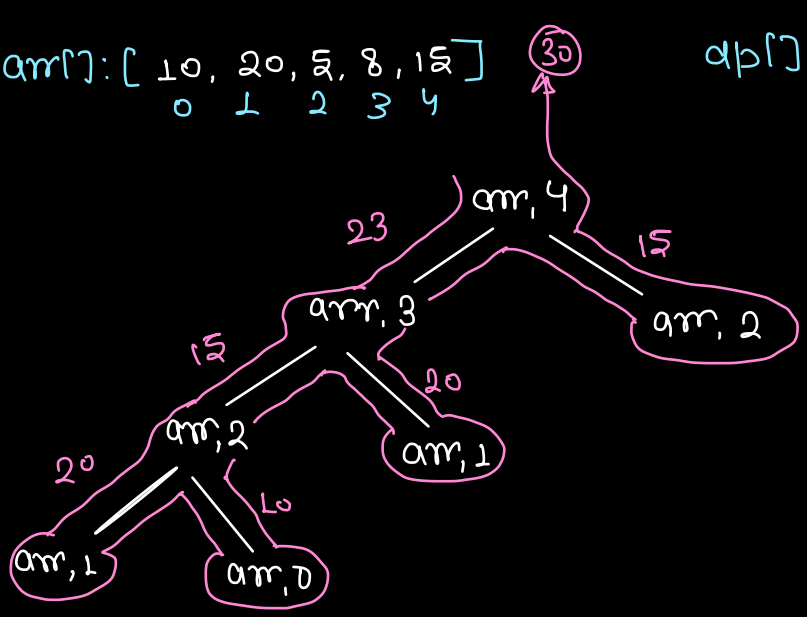
```
int[] dp[N]:  * initialised with -1

int minCost(arr[], int i) {
    if(i==0 || i==1) {
        return dp[i]=arr[i];
    }
    if(dp[i] != -1) {
        return dp[i];
    }
    int val1 = minCost(arr, i-1);
    int val2 = minCost(arr, i-2);

    return dp[i] = min(val1, val2) + arr[i];
}
```

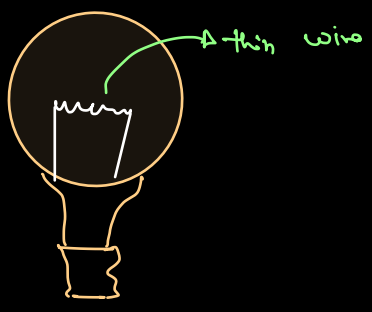
arr[]: [10, 20, 5, 8, 15]
0 1 2 3 4

dp[]: [~~-1~~, ~~-1~~, ~~-1~~, ~~-1~~, ~~-1~~]
0 1 2 3 4



T.C: O(n)

S.C: O(n)



Tabulation: (Bottom up)

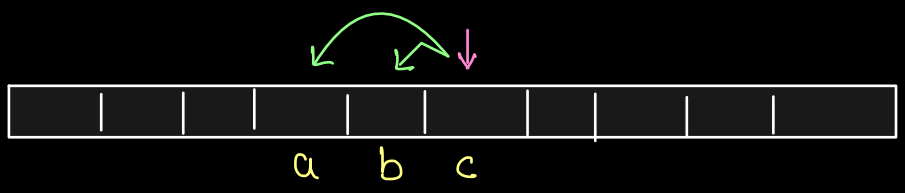
arr[]: [10, 20, 5, 8, 15]
0 1 2 3 4

dp: [10 20 15 23 30]
0 1 2 3 4

dp[i] → min cost to reach at ith stair

```
int[] dp[N];
dp[0] = arr[0];
dp[1] = arr[1];
for(int i=2; i<n; i++){
    dp[i] = min( dp[i-1] , dp[i-2]) + arr[i];
}
return dp[n-1];
```

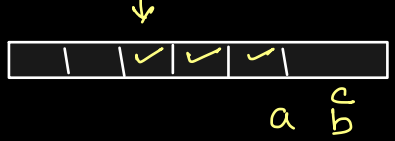
T.C: O(n)
S.C: O(n)



to prepare c → always using previous two variable i.e. a & b.

c ⇒ min(a,b) + arr[i]

Optimise space complexity:




```
int a= arr[0], b= arr[1];
for(int i=2; i<n; i++){
    int c = min(a,b)+arr[i];
    // prepare a & b for next iteration
    a=b;
    b=c;
}
return b;
```

Edge case
no. of stair = n
n=2 → ⑤ ?
think why?
→ TODO ?

~~~~~~~

Graphs :

~~~~~~~

* **Definition**: Nodes connected via edge: Graphs
* **Difference between Trees and Graphs**
* **Classification of Graphs**

Difference:

A graph is a tree ? ⟶ No

A tree is a graph ? ⟶ Yes

(1) Trees are heirarchical unlike graphs.

(2) no. of edge in a tree is exactly (n-1) for n-nodes.

Classification:

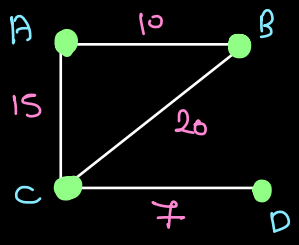
(1)

Undirected graph (facebook)

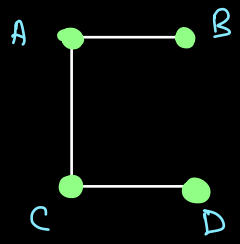
directed graph (Instagram)

(2)

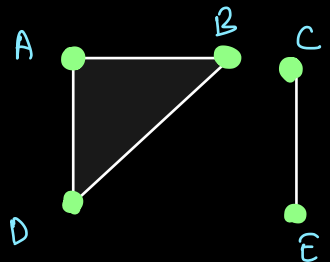
weighted

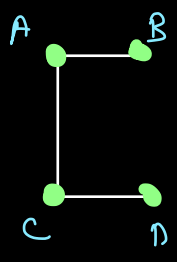


unweighted

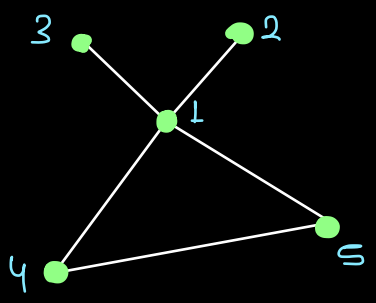


(3)



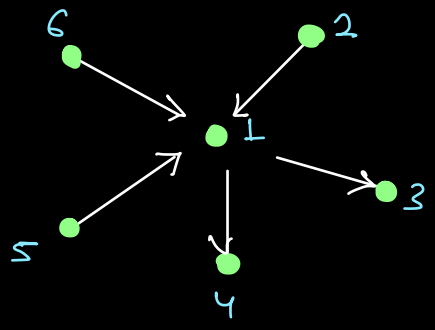


④ Degree



degree of 1 → 4
degree of 4 → 2

In / out - degree

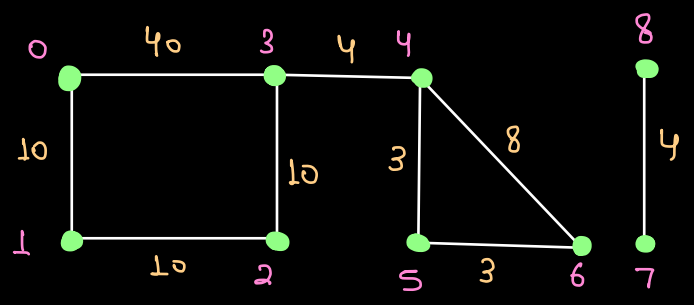


In degree of 1 → 3
outdegree of 1 → 2
In degree of 5 → 0

Break : 10:24 – 10:34 Am

Input and Storage of Graphs

Undirected weighted graph



u	v	wt
0	3	40
0	1	10
1	2	10
2	3	10
3	4	4
4	5	3
5	6	3
4	6	8
7	8	4

Nodes = 9 → 1 to 9
↳ 0 to 8

Edges = 9

Input: Edges and weight associated with it.

Store:

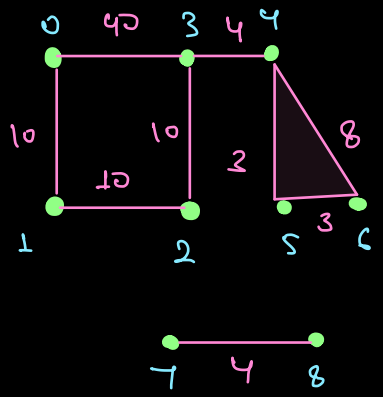
* Adjacency Matrix
* Adjacency List

Adjacency matrix:

Nodes = 9
Edges = 9

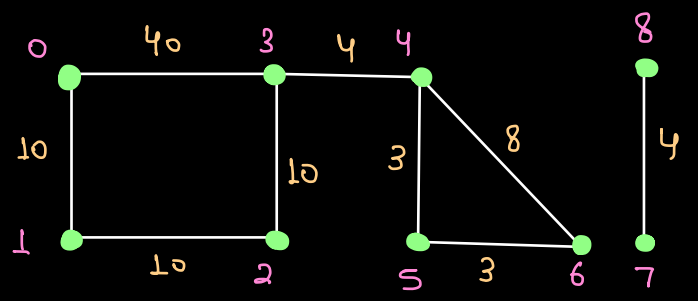
u	v	wt
0	3	40 ✓
0	1	10 ✓
1	2	10 ✓
2	3	10 ✓
3	4	4 ✓
4	5	3 ✓
5	6	3 ✓
4	6	8 ✓
7	8	4 ✓

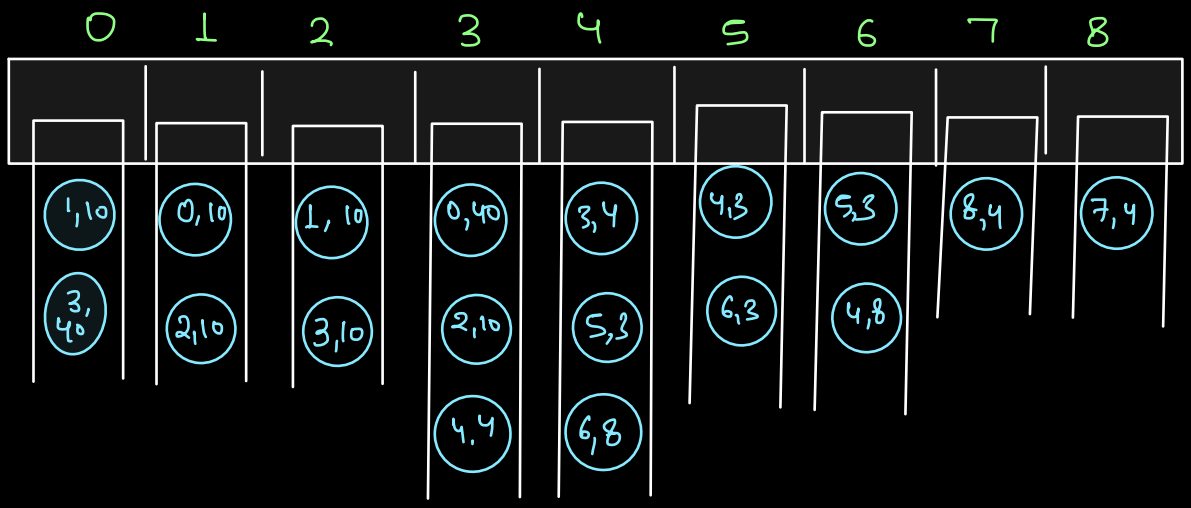
	0	1	2	3	4	5	6	7	8
0	-1	10	-1	40	-1	-1	-1	-1	-1
1	10	-1	10	-1	-1	-1	-1	-1	-1
2	-1	10	-1	10	-1	-1	-1	-1	-1
3	40	-1	10	-1	4	-1	-1	-1	-1
4	-1	-1	-1	4	-1	3	8	-1	-1
5	-1	-1	-1	-1	3	-1	3	-1	-1
6	-1	-1	-1	-1	8	3	-1	-1	-1
7	-1	-1	-1	-1	-1	-1	-1	-1	4
8	-1	-1	-1	-1	-1	-1	-1	4	-1



Wastage of space

Adjacency List:





List < List < Node > >

↳ Neighbour weight

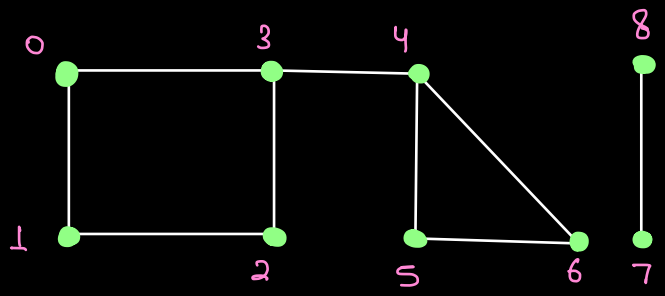
~~~~~~~~~~~~~~~~~~~~

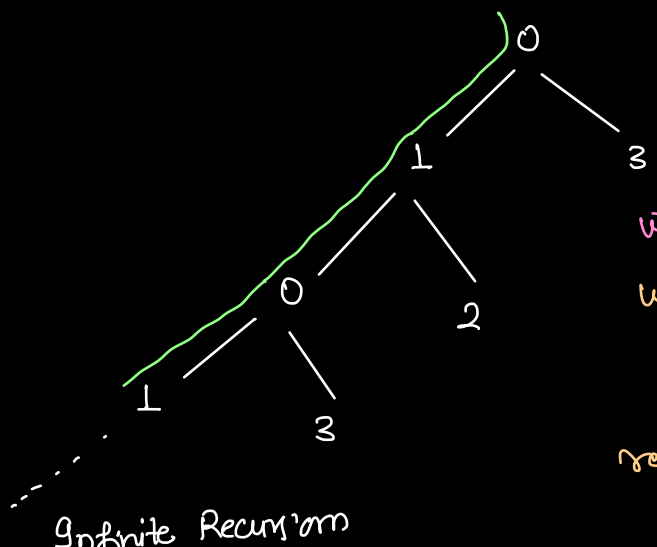
Traversal in Graphs

~~~~~~~~~~~~~~~~~~~~

1. DFS : Depth First Search
2. BFS : Breadth First Search

Depth first Search:



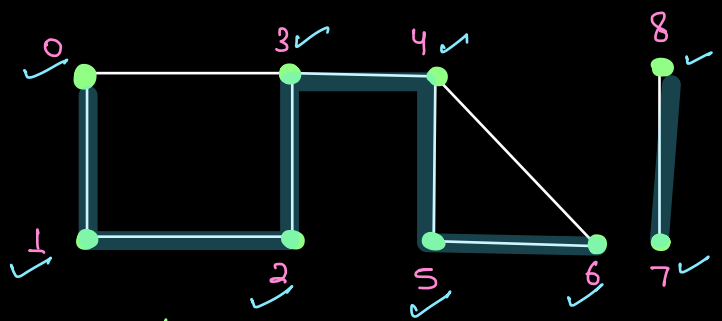


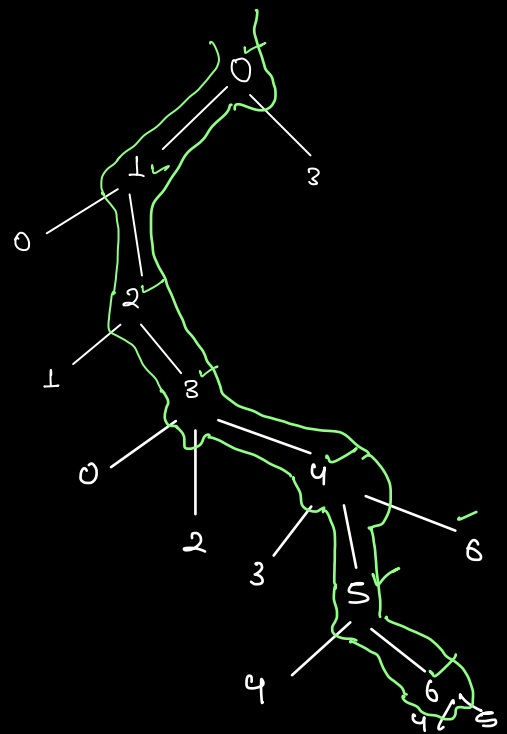
Why ?
we are visiting the node which is already visited.

resolution: track visited nodes
→ boolean visited array

vis:

T	T	T	T	T	T	T	T	T
F	F	F	F	F	F	F	F	F
0	1	2	3	4	5	6	7	8





assumption: List<List<Int>> graph : (given)

```
boolean[] vis = new boolean[n];
// initially all cells are false.

for(int v=0; v<n; v++){
    if(vis[v] == false){
        dfs(graph, v, vis);
    }
}

dfs(graph, int v, boolean[] vis){
    print(v);
    vis[v] = true;

    for(int nbr : grap[v]){
        if(vis[nbr] == false){
            dfs(graph, nbr, vis);
        }
    }
}
```

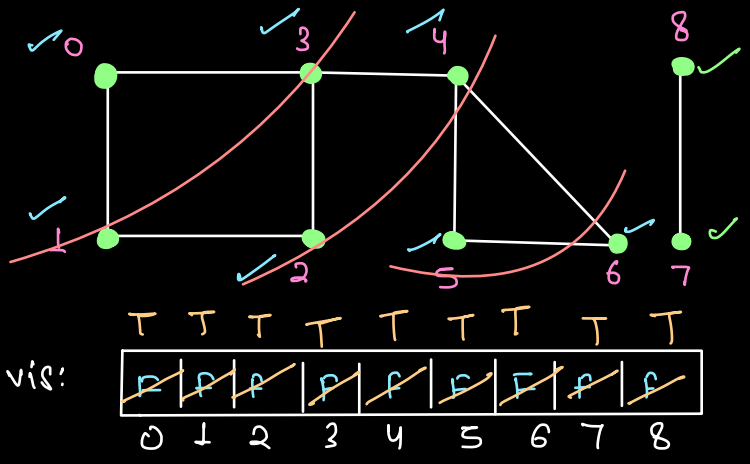
T.C: O(V+E)

S.C: O(V) ⟶ TODO why?

Check the level of Recursion.

⟶ Explain: think about why?

Breadth First Search: → Similar to level order traversal



vis:

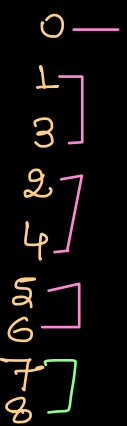
T	T	T	T	T	T	T	T	T
~~F~~	~~F~~	~~F~~	~~F~~	~~F~~	~~F~~	~~F~~	~~F~~	~~F~~
0	1	2	3	4	5	6	7	8

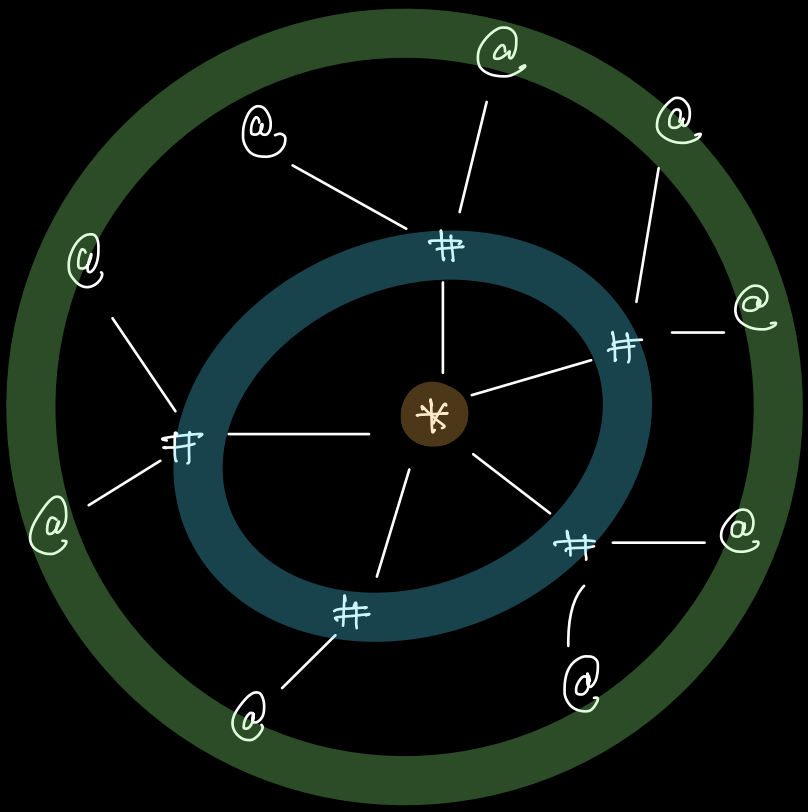
Steps of BFS

* Rem
* mark * → if already marked continue
* Print
* Add unvisited nbr

Data Structure: Queue

~~0, 1, 3, 2, 2*, 4, 5, 6, 6*, 7, 8~~





Expansion: Radial movement OR Radial expansion

→ Breadth First Search

pseudocode:

assumption: graph is given List<List<Int>> graph

```
boolean[] vis = new boolean[n];

for(int v=0; v<n; v++) {
    if(vis[v] == false) {
        bfs( graph, v, vis);
    }
}

void bfs (graph, int src, boolean[] vis) {
    Queue<int> que;
    que.add(src);

    while(que.size() > 0) {
        // 1. remove
        int rem = que.remove();

        // 2. mark *
        if(vis[rem] == true) continue;
        vis[rem] = True;

        // 3. print
        print(rem);

        // 4. add unvisited nbr
        for(int nbr: graph[rem]){
            if( vis[nbr] == false){
                que.add(nbr);
            }
        }
    }
}
```

T.C: O(V+E)

S.C: O(V)

Think why ??