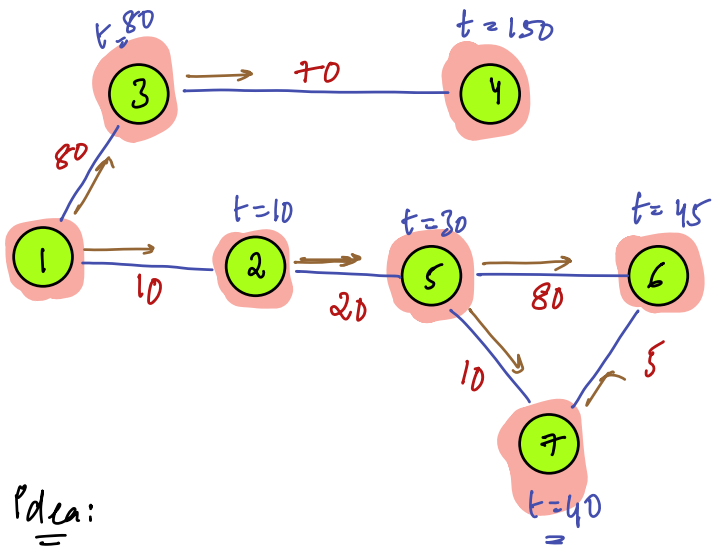
Today's Content :

Q: Fire / Petrol Bunk



Idea:

1) Out of all Unblasted bunks, Bunk with smallest time is blasted first

2) After blasting a bunk we updated time for it's adjacent bunks.

Be a Super Heroine / Hero

1) Nodes indicates petrol bunk

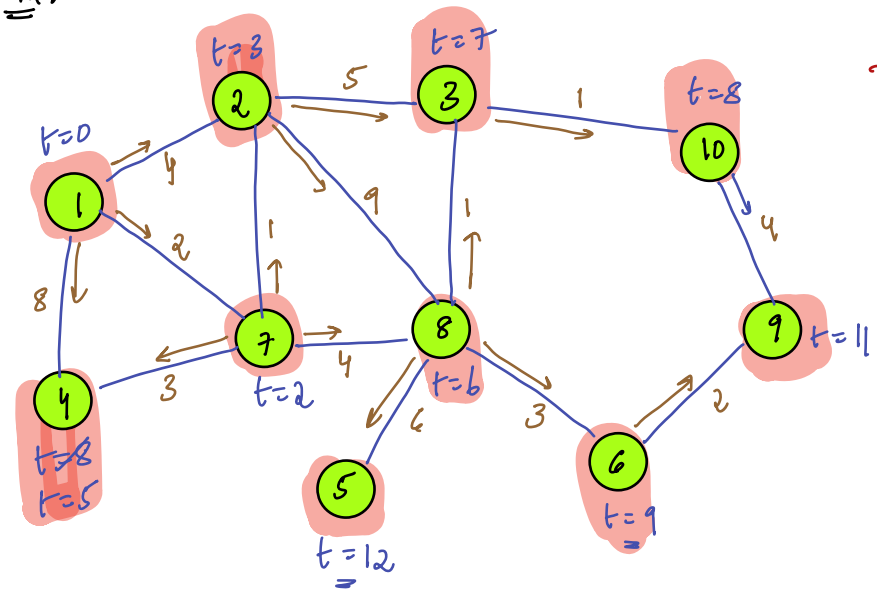
2) Edges indicates connection between 2 bunks & length of connection, bunks are connected via petrol pipes

3) Initially Say bunk 1 blasted

4) Petrol burns at 1km / 1min

5) Calculate time at which each bunk is blasted

Eg2:



Dijkstra's Algorithm:

From S, get shortest path to all other nodes

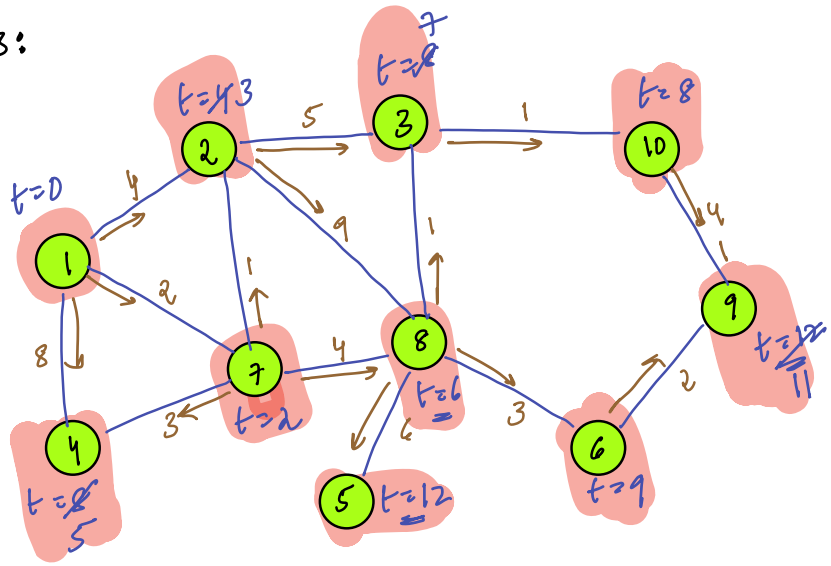
Step:

→ Unblasted, nodes min time ↳ blast pt

→ Update all pt's adjacent nodes

1	⟨2,4⟩ ⟨4,8⟩ ⟨7,2⟩
2	⟨1,4⟩ ⟨3,5⟩ ⟨7,1⟩ ⟨8,9⟩
3	⟨2,5⟩ ⟨8,2⟩ ⟨10,1⟩
4	⟨1,8⟩ ⟨2,1⟩ ⟨7,3⟩
5	⟨8,6⟩
6	⟨8,3⟩ ⟨9,5⟩
7	⟨1,2⟩ ⟨2,1⟩ ⟨4,3⟩ ⟨8,4⟩
8	⟨2,9⟩ ⟨3,2⟩ ⟨5,6⟩ ⟨6,3⟩ ⟨7,4⟩ (0 1 2 3 4)
9	⟨6,5⟩ ⟨10,4⟩
10	⟨3,1⟩ ⟨9,4⟩

Eg3:



int t[11] =

0	1	2	3	4	5	6	7	8	9	10
	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
	4	8	8	12	9	2	6	11		8
	3	7	5					11		

Min heap < int, int >

dist node

d n

<2, 7>

<4, 2>

<8, 4>

<6, 8>

<3, 2>

<5, 4>

<8, 10>

<11, 9>

d n

<8, 7> already done

skip, or no need to update again.

<7, 3>

<9, 6>

<12, 5>

<12, 9> already blated

Note: After getting min dist node from heap, check if it's already blasted or not

→ Dijkstras:

We can apply in both directed & undirected graphs

```
int Dijkmras( list<pair<int,int>> g[], int n, int s, int des)
    dist[n+1] = INT_MAX;  dist[s] = 0
    min_heap < pair<int, int> > mh;
    mh.insert({0, s})
    while( mh.size() > 0) {
                d   u
        Pair<int int> x = mh.getmin();
        mh.deletemin() // deleting our min
        d = x.first // to get 1st element in pair
        u = x.second // to get 2nd ele in pair
        if( d == dist[u]){ // blasting it for first time
            // Iterate in it's adjent nodes
                                  { if u == dest : return dist[u] }
            for( i=0; i < g[u].size(); i++){
                        node  weight
                Pair<int, int> y = g[u][i]
                v = y.first, w = y.second
                // from u --w--> v } dist[v]
                if( dist[u]+w < dist[v]){
                    dist[v] = dist[u]+w
                    mh.insert({dist[v], v})
            }
        }
    }
    return dist[des]
}
```

At max heap can contain how many elements ? ⇒ {E}

→ graph N Nodes & E Edges

→ Idea: A single edge can only update 1 node at max

↳ Heap size can at max go upto (E)

↳

operations : getMin → O(1) | deleteMin → log E | Insert → log E

Total TC : { (N + E) { log E } }

In general : E > N : Based on This

TC : O(N + E) log E

SC : O(N + E)

↓ ↓

dist Array | heap

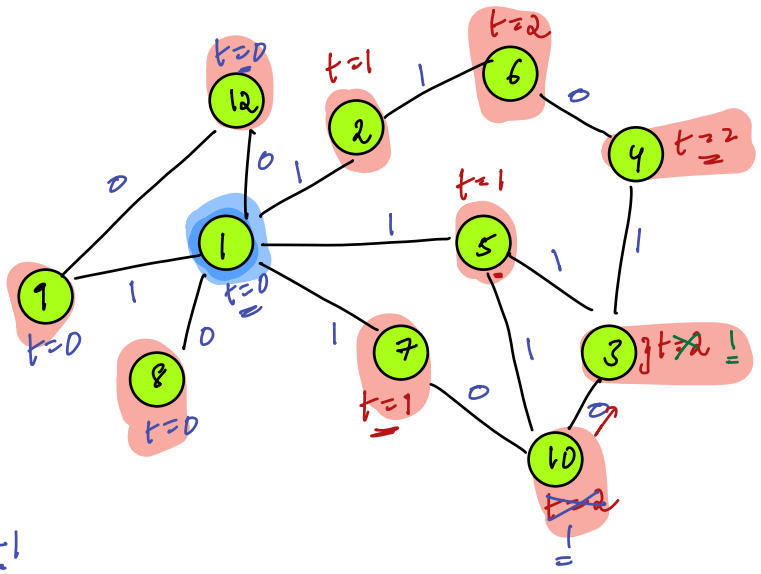
Adj List

TC : O(E log E)

SC : O(E)

→ 10:42 pm break

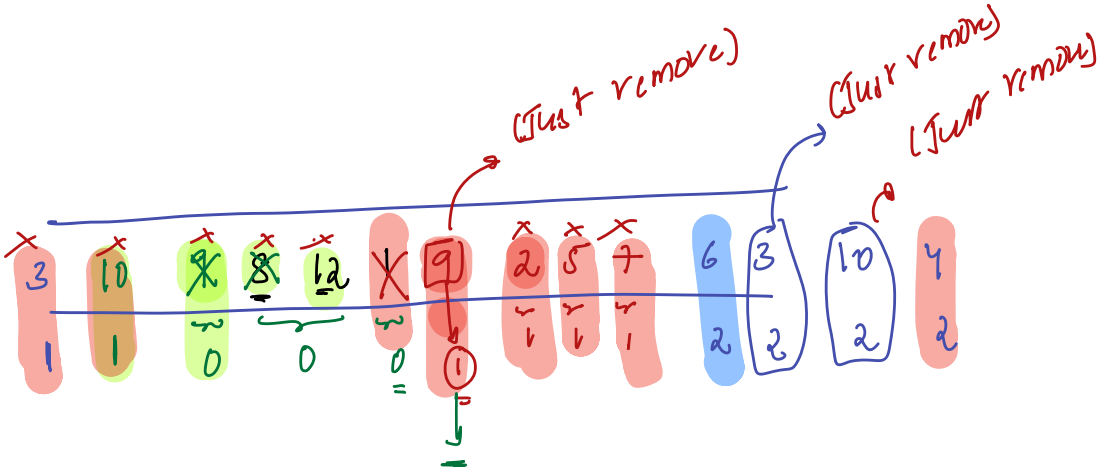
Q: N = 12, S = 1, D = 4 ⟶ { length of shortest path }



→ Dijkstras :
- TC : E log E
- SC : O(E)

→ Every step : Take node with min value

0-1



//

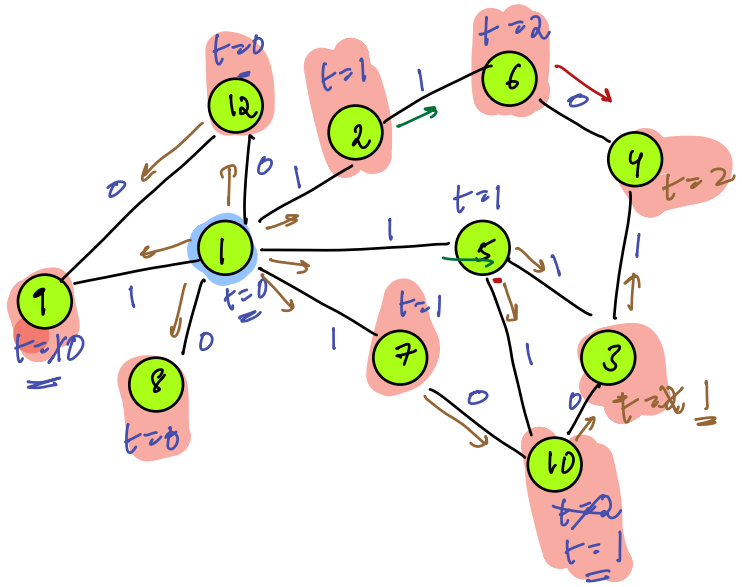
Idea

1) Pick from left side

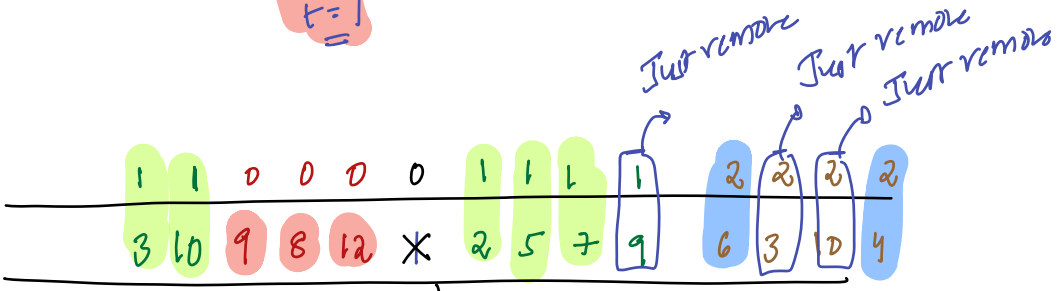
2) Inserting all it's adjacent node, value & insert

→ If edge weight : 0 we insert adjacent node in left

→ If edge weight : 1 we insert adjacent node in right



dist



Deletion order:

1 : 0
8 : 0
12 : 0
9 : 0

// All distance at 0, are done

2 : 1
6 : 1
7 : 1
10 : 1
3 : 1

All distance are 1

6 : 2
4 : 2

// All nodes at dist 2

insert at front()
insert at back()
→ deque()

weights should either be 0 or 1

remove at front()

→ Insert() } → O(1)
remove() }

0/1 BFS, 0/2, 0/3, 0/n, 1/2

→ TC: O(N+E) = O(E)

→ SC: O(N+E) = O(E)

dist, deque

Adj lem

Try at our at won't work

1 : left
2 : right

Trick in empty, we won't delete weights in ptr decrea

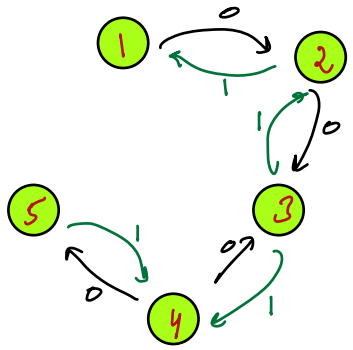
Google:

Reversing Edges: → {TODO}

Given a directed graph N & E Edges

Find the min no: of edges that needs to be reversed in order to reach Node N from Node 1

Ex1:

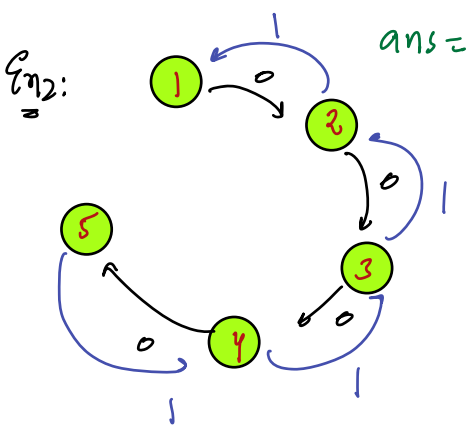


→ reversed edge: 3 → 4 : ans = 1

→ path: 1 → 2 → 3 → 4 → 5 → ans = 1

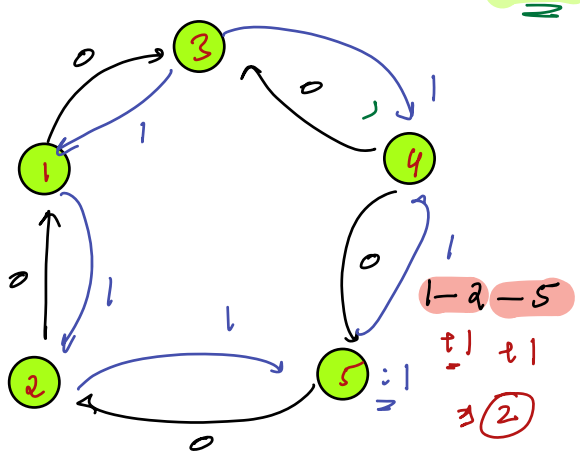
+1

Ex2:



ans = 0

Ex3:



reverse 3-4 ans = 1

1-2-5

+1 +1

→ 2

//

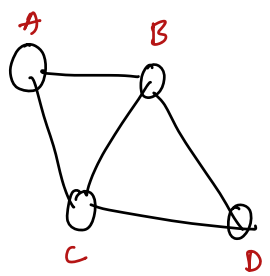
Idea: Assume there is no directed graph apply BFS & get path (Shortest path)

1) In path how many edges we need to reverse?

Idea1: Assume there is no direction & Get all Paths from S → D

↳ For every path: Calculate number of Edges we need to reverse

TC: → // factorial // → {TODO}

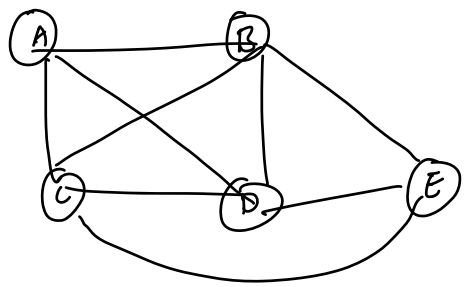


A B D

A C D

A B C D

A C B D



A B E

A C E

A D E

A B D E

A B C E

A D B E

.

.

Idea2: // Adj List change

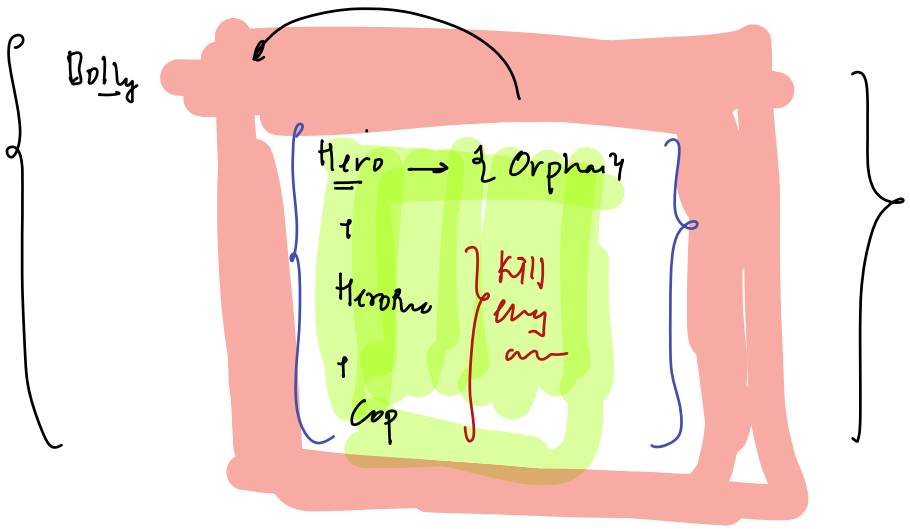
1) For all given Edges : 0 (w)

2) For all given Edges, we need to consider its reverse as well : 1

3) Apply 0-1 BFS from S to D & get length of Shortest Path

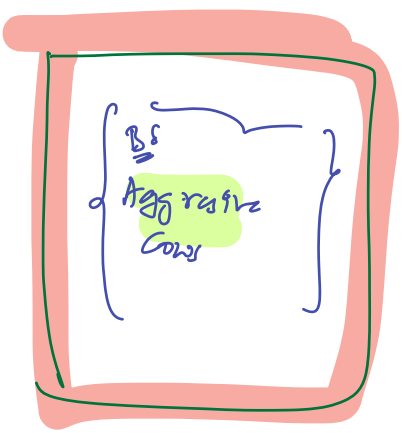
{ Number of Edges we need to reverse }

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


Bolly
Hero → 2 Orphans
Heroine
Kill any one
Cop

// SOE:-

Ques:


BS
Aggressive
Cows

Question : → {BFS + ↩}

N data points : {2 people} → log N

Type 1 : a) P_1 P_2 : After P_1 is died P_2 is born

Type 2 : b) P_1 P_2 : There is a common time when both of lived together

c) Validate the data → Is data correct or wrong

Eg: N = 4

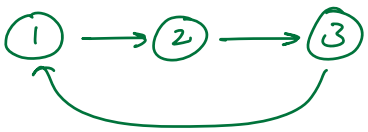
Type 1 : 1 2

Type 2 : 3 4

Type 1 : 2 3

Type 2 : 4 1

Type 1 : 3 1



→ There might some relevance

1) Given String

2) Find the palindromic permutation close to original string

{ couldn't find about in way }

Eg: Str : a c d a d c

a c d d c a

(3)