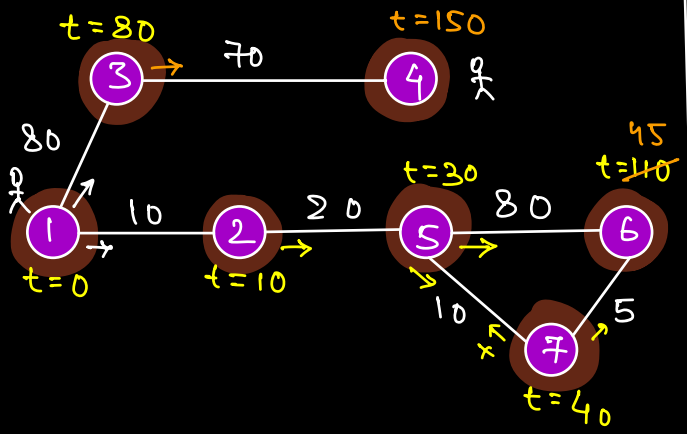
*Petrol Bunkers

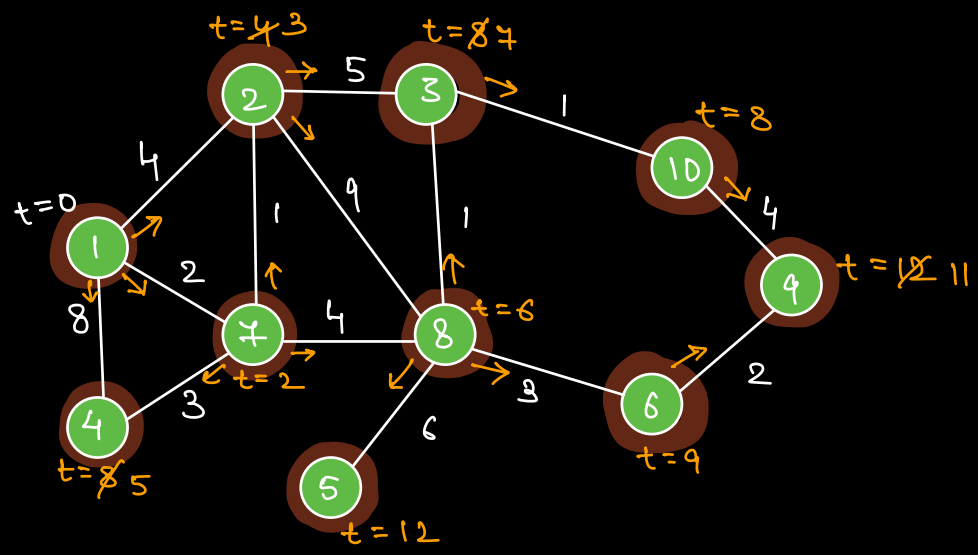


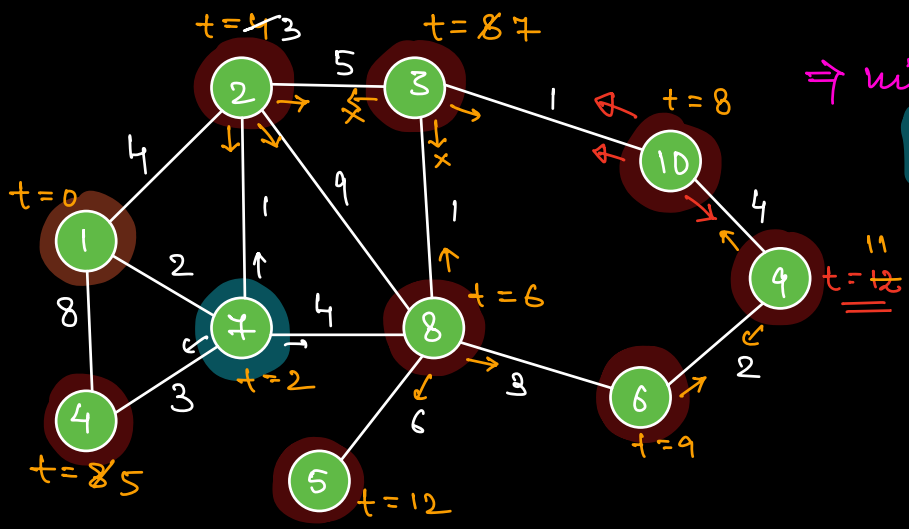
⇒ 150

1) Each node is a Petrol bunker.
2) Edge indicates connection b/w 2 bunkers & length is also given.
3) Initially bunker 1 will blast
4) Petrol burns at 1 km/min
5) Calculate the time at which all the bunkers will be blasted.

⇒ DIJKSTRA's ALGORITHM
↳ Shortest path algorithm.

1) Node with the min time will blast first.
2) Once a node is blasted, fire will travel to all of its adjacent nodes.


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



⇒ minHeap

- getMin() → O(1)
- insert() → log N
- deleteMin() → log N
- search → N
- delete → N

int time[11]:

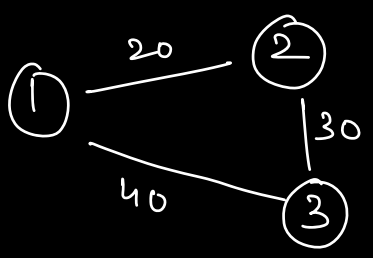
0	1	2	3	4	5	6	7	8	9	10
	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
		4 → 3	8 → 7	8 → 5	12	9	2	6	12 → 11	8

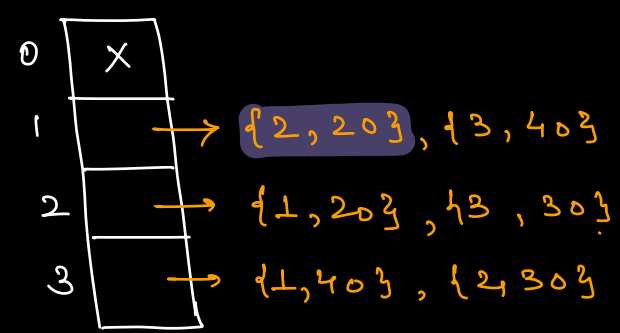
minHeap < pair < time, node > >

- ⟨4,2⟩ *
- ⟨8,4⟩ *
- ⟨2,7⟩ *
- ⟨3,2⟩ *
- ⟨5,4⟩ *
- ⟨6,8⟩ *
- ⟨8,3⟩ *
- ⟨7,3⟩ *
- ⟨12,5⟩ *
- ⟨9,6⟩ *
- ⟨8,10⟩ *
- ⟨12,9⟩ *
- ⟨11,9⟩ *

1) Find the node with min time & blast it.
2) Once a node is blasted, iterate over the adjacent node & update their blast time.
3) If a node is already blasted, skip it.

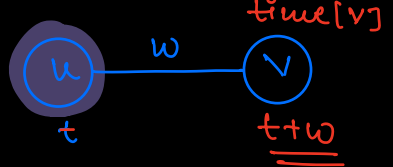
```
if (time in Heap > time in arr) {
    continue;
}
```

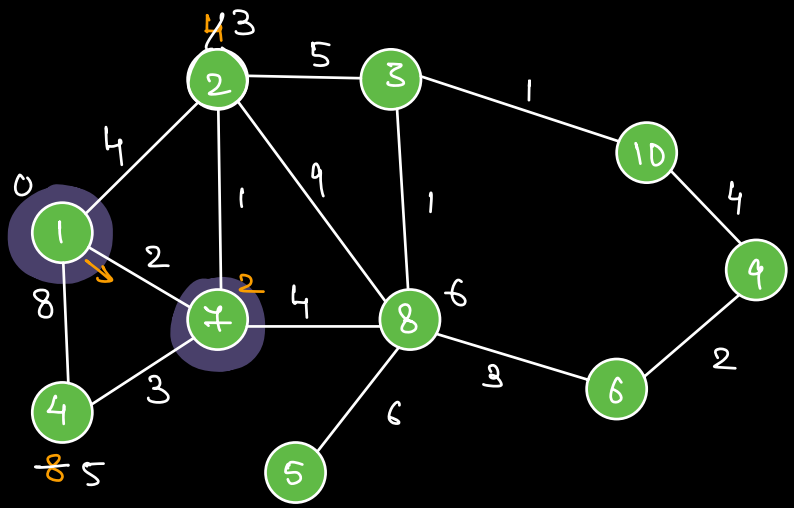


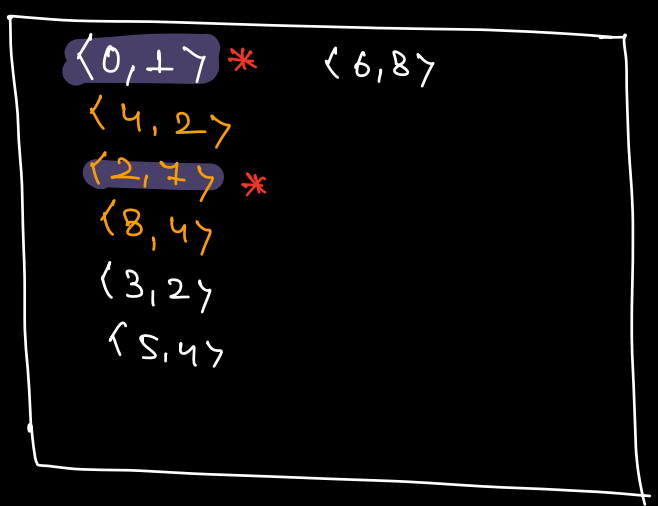



```
List<pair<int, int>> g[N+1];
```

```
int blast(list<pair<int,int>> g[], N, E, src, dest){
    int time[N+1] = {∞};  time[src] = 0;
    min_heap<pair<int, int>> mh;
                  time   node
    mh.insert({0, src});
    while(mh.size() > 0){
        pair<int,int> data = mh.getMin();
        mh.deleteMin();
        int t = data.first;
        int u = data.second;
        if(t > time[u]){ // Node is already blasted.
            continue;
        }
        for(i=0; i<g[u].size(); i++){
            pair<int,int> p = g[u][i];
            v = p.first;
            w = p.second;
            if(t+w < time[v]){
                time[v] = t+w;
                mh.insert({t+w, v});
            }
        }
    }
    return time[dest];
}
```







TC: $E * (1 + \log E + \log E)$

$\Rightarrow O(E \log E)$

Heap size : (E)
↳ getMin() → O(1)
deleteMin() → O(log E)
insert() → O(log E)

SC: $O(E + N + E) \rightarrow O(E)$

↑ Adj List ↑ time ↑ Heap

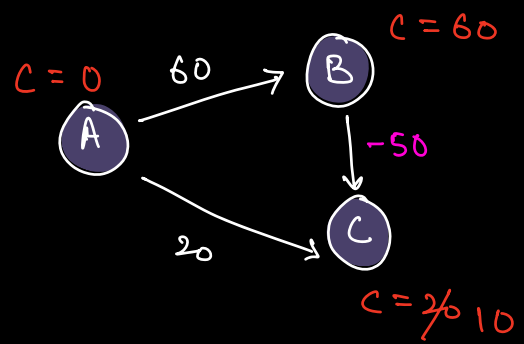
Weighted Graph

↳ Dijkstra's Algo : Shortest path from a source node to any node.

⇒ A* Algorithm ⇒ Google Maps.

Unweighted graph ⇒ BFS to find Shortest Path.

weighted graph ⇒ Dijkstra's to find Shortest Path.



-ve Edge Weights :

↳ Dijkstra's doesn't works

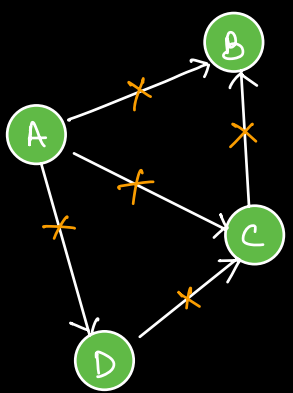
[Bellman ford] [Floyd Warshal.]

Topological Sorting

Recursion ⟶ DP

⇒ T_A T_B

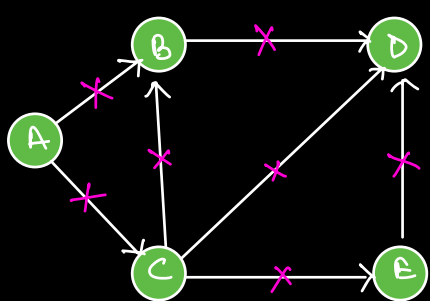
T_B is dependent on T_A.



Order of execution of task

T_A T_D T_C T_B.

↑ No incoming edge.



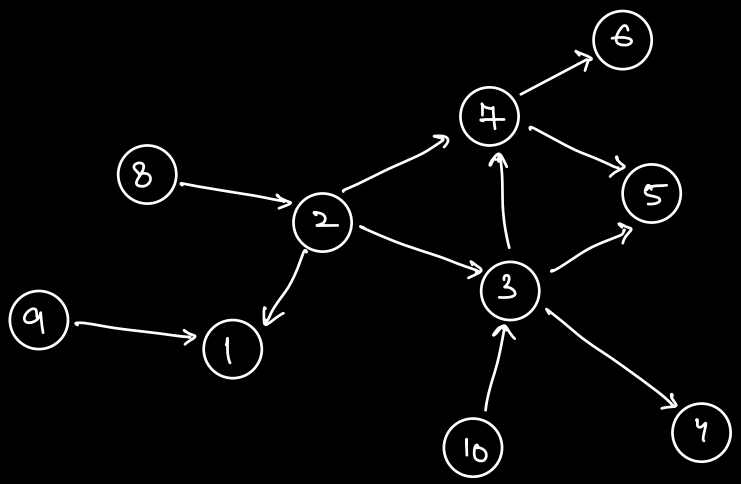
T_A T_C T_B T_E T_D.

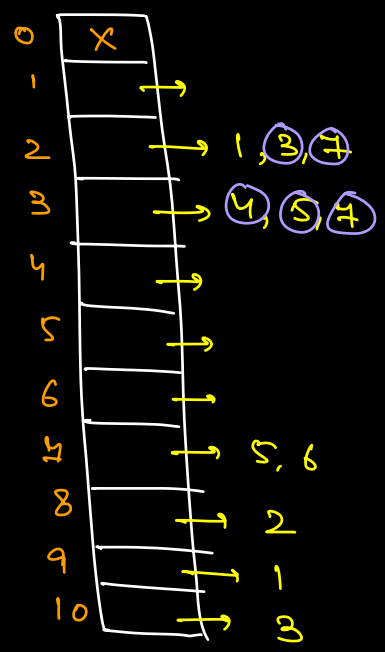
← ~~T_A~~ ~~T_C~~ ~~T_B~~ ~~T_E~~ ~~T_D~~ ←

⇒ Queue.

incoming edges.

A	B	C	D	E
0	~~2~~ 1 0	~~1~~ 0	~~3~~ ~~2~~ 1 0	~~1~~ 0





in[11]:

0	1	2	3	4	5	6	7	8	9	10
~~2~~	+	~~2~~	+	~~2~~	~~2~~	~~2~~	~~2~~	0	0	0
~~1~~ 0	0	~~1~~ 0	0	~~1~~ 0	0	~~1~~ 0				

Queue

~~8~~, ~~9~~, ~~10~~, ~~2~~, ~~1~~, ~~3~~, ~~4~~, ~~7~~, ~~5~~, ~~6~~

Order : 8 9 10 2 1 3 4 7 5 6

Approach :

1. Create incoming edges array.
2. If count of incoming edges for any node = 0
   → insert in the Queue
   → resolve the dependency of its neighbours.

Topological Sorting Code

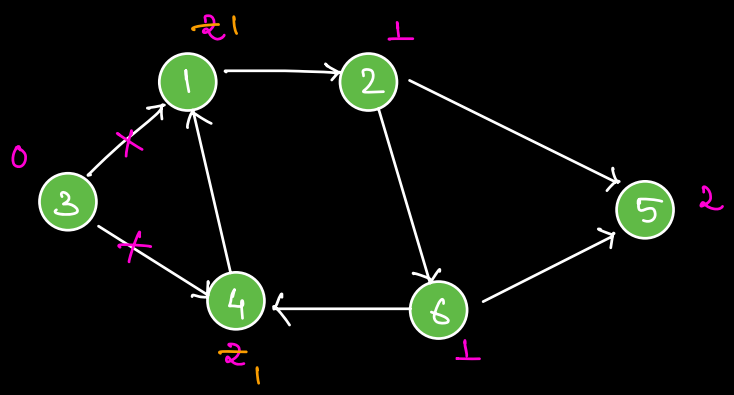
```
void topological( list<int> g[], int N){
    int in[N+1];
    for(i=1; i<=N; i++){ ⇒ O(E)
        for(j=0; j<g[i].size(); j++){
            int v = g[i][j];
            in[v]++;
        }
    }
    queue<int> q; //insert all nodes with 0
                  //dependency.
    for(i=1; i<=N; i++){ ⇒ O(N)
        if(in[i]==0) q.insert(i);
    }
    while(q.size()>0){ ⇒ O(E)
        int u = q.front();
        print(u);
        q.dequeue();
        for(i=0; i<g[u].size(); i++){
            v = g[u][i]
            in[v]--;
            if(in[v]==0) q.insert(v);
        }
    }
}
```

TC: $O(N+E+E) = O(2E+N)$

$= O(E)$

SC: $O(N+E)$

⇒



3

NOTE: If we are not able to resolve dependencies at any point of time, it means there's a Cycle in Directed Graph.

⇒ If we don't have any node with 0 dependency in the graph ⇒ Cycle.
