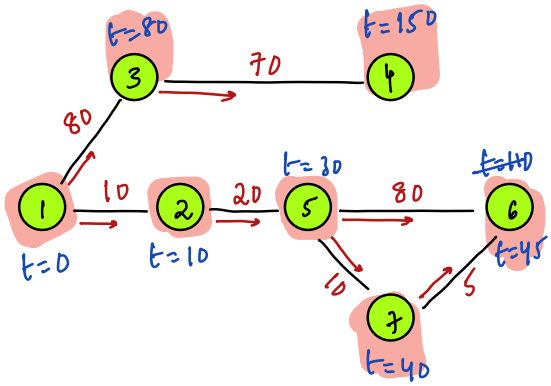
Todays Content: ⟶ Sunday 10:30 AM graphs optimal

- Dijkstras Algorithm
- Topological Sort

- cycle detection in undirected
- Bi-partite
- Chromatic number
- 0/1 BFS
  - Revise Edges

Q: Five / Petrol Bunk



Concept: Dijktra's :

1) Node with min time we blast

2) After a node is blasted

→ We updated blast time to adj node

Be a Super Heroine / Hero

a) Nodes indicates petrol bunk

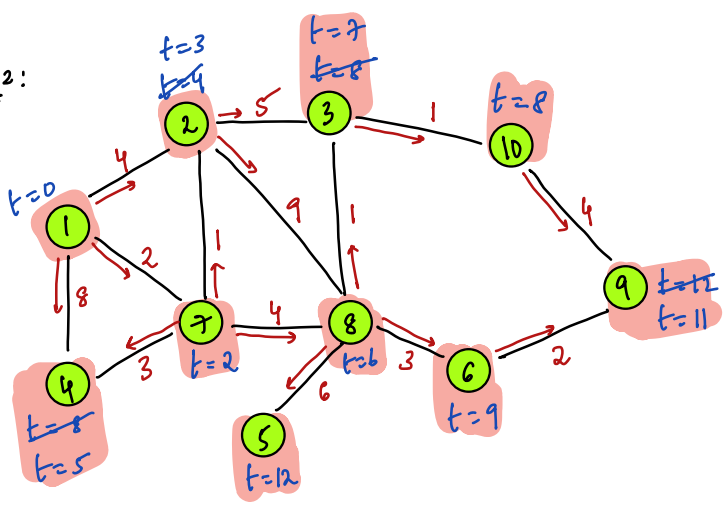
b) Edges indicates connection between 2 bunks & length of connection, bunks are petrol pipes

c) Initially say bunk 1 blasted

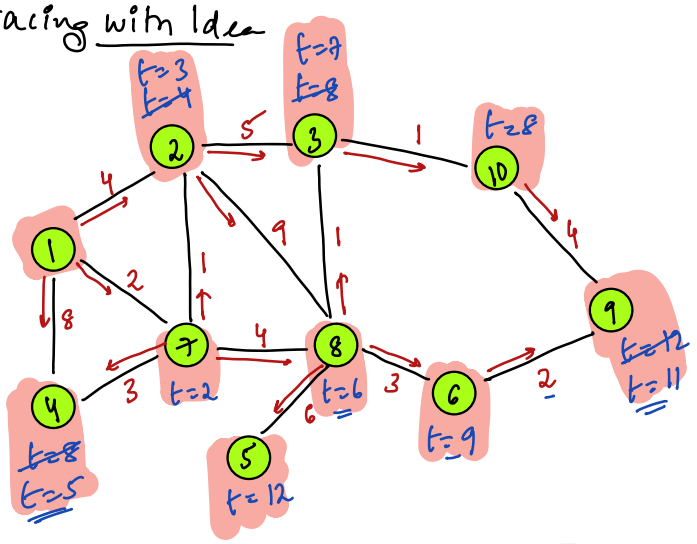
d) Petrol burns at 1km / 1min

e) Calculate time at which each bunk bunk is blasted

Ex2:



Tracing with Idea



int t[11] =

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

time, node

minheap < pair < int, int >> mh

<0, 1>*	<8, 3>*	<11, 9>*
<4, 2>*	<12, 5>*	
<8, 4>*	<9, 6>*	
<2, 7>*	<7, 3>*	
<6, 8>*	<8, 10>*	
<5, 4>*	<12, 9>*	
<3, 2>*		

Obs:

a) Find node with min time to blast

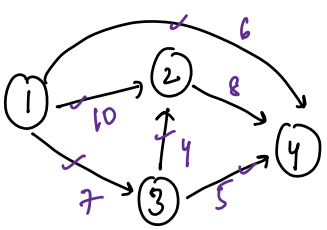
b) Iterate on neighbours of blasted & update their blast time

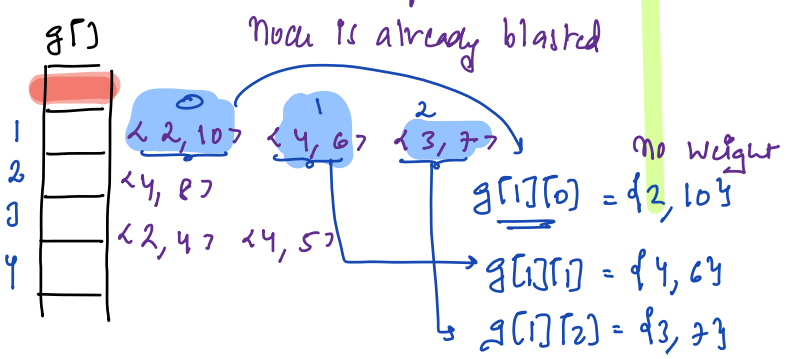
c) If is node already blasted skip it

Time in heap > Time in ans[]

Node is already blasted

Ex:



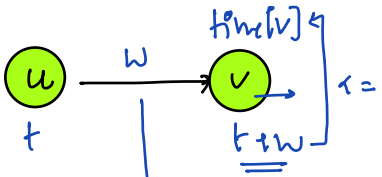


```
int blast( list<pair<int,int>> g[], int N, int s, int dest){
    int time[N+1] = {∞}, time[s] = 0
    minheap<pair<int,int>> mh, mh.insert({0,s})
    while(mh.size()>0){
        pair<int,int> data = mh.getMin();
        mh.deleteMin();
        int u = data.second
        int t = data.first
        if(t > time[u]){ // node u already blasted, continue
        // We are blasting node u, update adj nodes
        for(int i=0; i< g[u].size; i++){
            pair<int,int> ele = g[u][i]
            int v = ele.first
            int w = ele.second
            if( t+w < time[v]){
                time[v] = t+w
                mh.insert({time[v],v})
            }
        }
    }
    return time[dest]
}
```

→ Adj list of graph passed as parameter

t n

t u



If t+w < time[v]: we update blast time of node v

TC: for every edge we can update node
- TC: { E * [log E + log E + 1] }
- TC: E log E SC: O(E)

How many ins = E
- Heap size = E
- getMin : 1

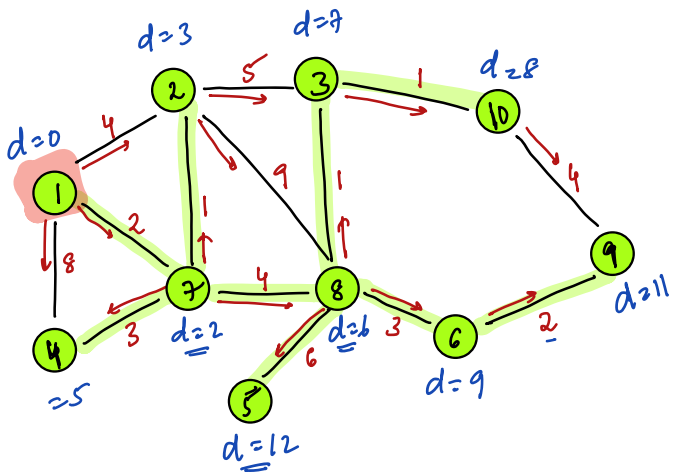
insert : O(log E)

delMin : O(log E)

Q) Weighted Graph: length of Shortest path from Source → To All Nodes

TC: E log E

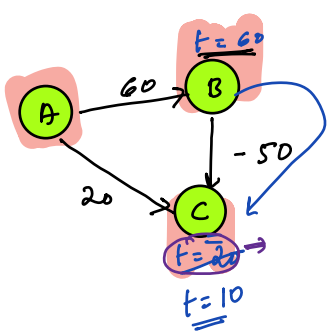
SC: O(E)



- Dijkstras Algorithm
  - Weighted graph
  - make sure that all edges weights the
  - In-ve :
    - a) Bellmn frd
    - b) Floyd Warshal
      - Monday Wednesday

→ Classification :
- Shortest unweighted : BFS
- Shortest weighted : Dijkstra's

→ Djkstras for -ve :



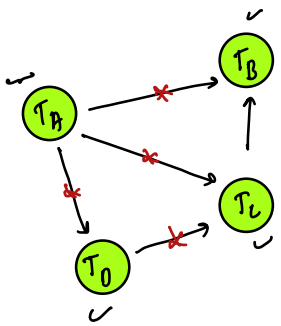
10:50 → 10:57pm

Topological Sort:

Recursion → DP → ↳ dp depends on Recursion
↳ First finish Recursion later dp

$T_A \rightarrow T_B$ → ↳ T_B depends on T_A
↳ First finish T_A & later T_B

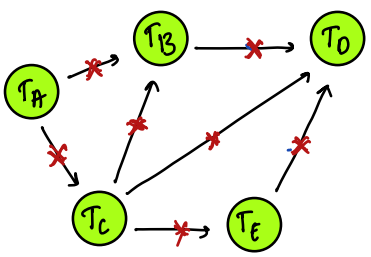
Ex1:



Order of Executing Tasks:

T_A T_D T_C T_B

Ex2:



any order of Executing Tasks

T_A T_C T_B T_E T_D

Idea:

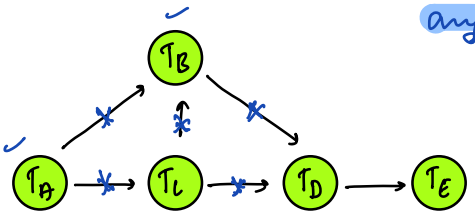
Incoming:

T_A	T_B	T_C	T_D	T_E
0	2 ✗ 0	✗ 0	3 2 ✗ 0	✗ 0

T_A T_C T_B T_E T_D

~~T_A~~ | ~~T_C~~ | ~~T_B~~ | ~~T_E~~ | ~~T_D~~

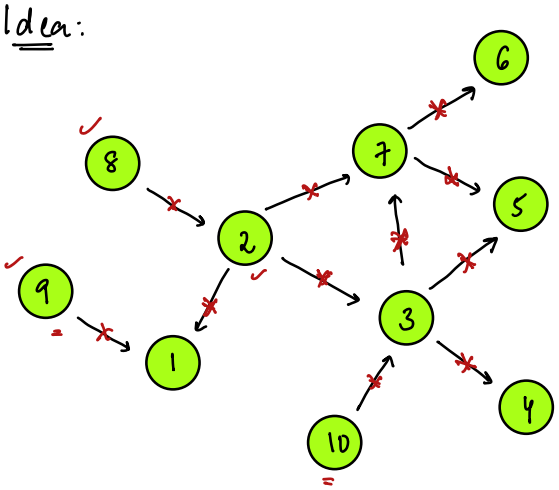
Ex3:



any order of Executing Tasks

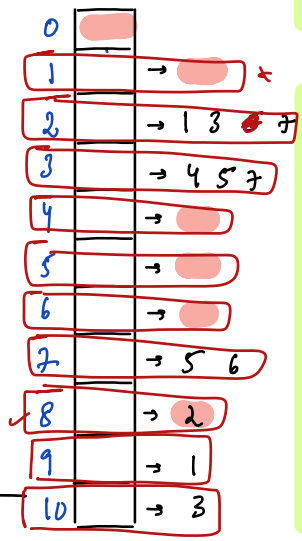
T_A T_C T_B T_D T_E

Idea:

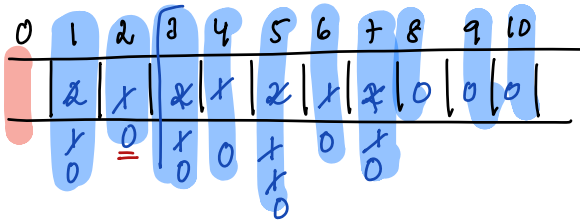


Adj list:

list<int> g[11]



int in[11] =



~~x x 0 x x x x x x x~~

Final order: 8 9 10 2 3 1 4 7 6 5

→ DataStru: Queue

Steps1 : Get count of incoming edge for all nodes

Steps2 : If count of incoming edges = 0

→ add in queue

→ resolve its dependies of it's adjacent nodes

↳ is present in adj list itself

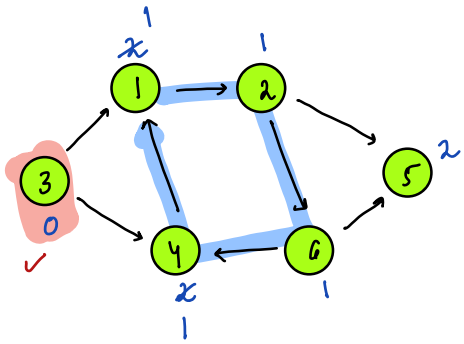
Pseudocode:

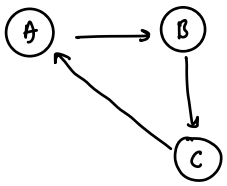
```
// given adj list of directed graph
                                  # No: of nodes
void TopoSort( list<int> g[], int N){

    int in[N+1] = {0}
    for(int i=1; i<=N; i++){                ---> TC: O(E)
        for(int j=0; j<g[i].size(); j++){
            int v= g[i][j];
            in[v]++; // increase incoming edge count
        }
    }
                                  -> in coming edges
    queue<int> q // insert all node with 0,
                                  -> dependency
    for(int i=1; i<=N; i++){ -> O(N)
        if( in[i] ==0){
            q.insert(i);
        }
    }
    while( q.size()>0){ -> O(E)
        int u = q.front() // get node u
        print(u)
        q.delete() // remove front node from que
        for(int i=0; i<g[u].size(); i++){
            int v= g[u][i]   // u ---> v
            in[v]-- // reduce incoming edge of v by 1
            if (in[v]==0){ q.insert(v)}
        }
    }

    TC: O(N+E+E) => O(N+2E) = O(E)   SC: O(E+N)
}
```

Detecting Cycle in Directed Graphs:





Obs: if we are not able to resolve all dependencies then

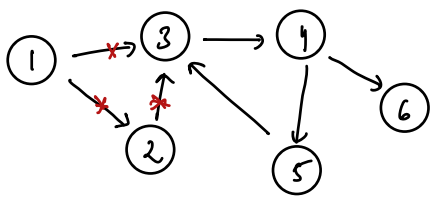
Is cycle in **Directed Graph**

count of incoming edges

Obs2: If there exists a non-zero value in in[], in that

can, then is a cycle in directed graph

Ex2:



in:

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

x	*	→ // Queue empty

In your in[], we have a non-zero data, **cycle present**