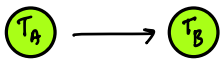
Todays Content:

- Topological Sort
- Dijtras Algorithm

Topological Sort:

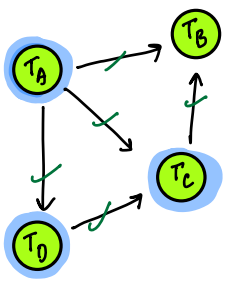
Recursion → Dynamic Programming



: → Compl T_A & goto T_B

→ T_B is depending on T_A

Ex1:

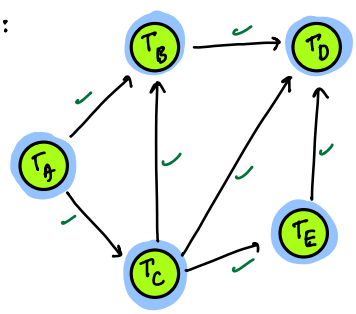


: Order of Execution of Tasks:

Note: Before doing a task its dependencies are needed to be resolved

order: T_A T_D T_C T_B

Ex2:



ord1: T_A T_C T_B T_E T_D

ord2: T_A T_C T_E T_B T_D

{Print any order}

{Print lexicographically Smallest}

T_A T_C T_B T_E T_D : lexicographically Smaller

Idea:

Nodes: T_A T_B T_C T_D T_E

Incoming edg: 0 | 2 1 0 | 1 0 | 3 2 1 0 | 1 0

order

T_A	T_C	T_B	T_E	T_D

T_A T_C T_B T_E T_D

Adj list:

T_A : T_B T_C →

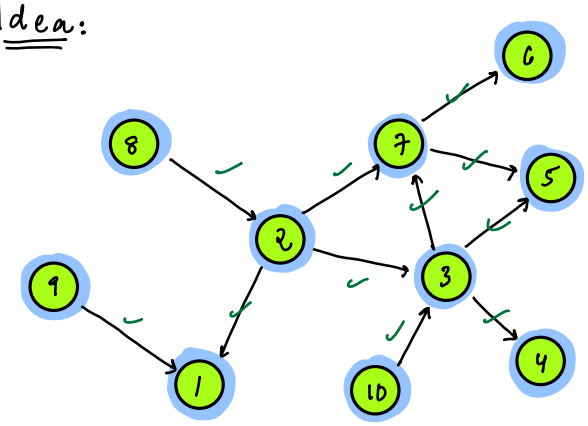
T_B : T_D →

T_C : T_B T_D T_E →

T_D : —

T_E : T_D →

Idea:



Create adj list:

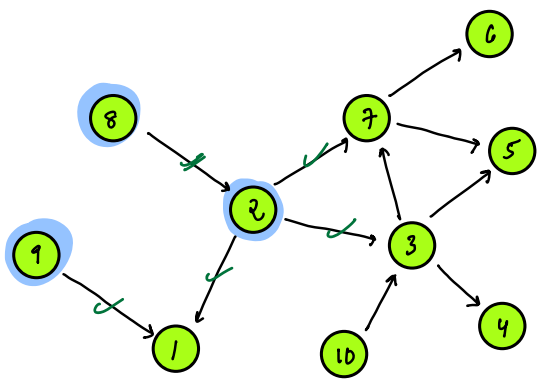
1 :
2 : 7 3 1
3 : 7 5 4
4 :
5 :
6 :
7 : 5 6
8 : 2
9 : 1
10 : 3

Create in[]: Store count of incoming edges:

	0	1	2	3	4	5	6	7	8	9	10
cIn[11] =	0	~~2~~	~~1~~	~~2~~	~~1~~	~~2~~	~~1~~	~~2~~	0	0	0
		~~1~~ 0	0	~~1~~ 0	0	~~1~~ 0	0	~~1~~ 0			

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

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



Create in[]: Store count of incoming edges:

	0	1	2	3	4	5	6	7	8	9	10
cIn[11] =	0	~~2~~	~~1~~	~~2~~	1	2	1	~~2~~	0	0	0
		~~1~~ 0	0	1				1			

DataStructure : [~~8~~ ~~9~~ 1 / 10 ~~2~~] 8 2 9 1 . . .

// Everytime we Pick min, So DataStructure = Minheap.

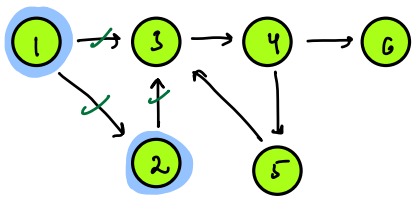
Topological Sort Pseudo Code: Any Correct order of Execution $u \longrightarrow v$

```
void TopoSort( int N, int E, int u[], int v[]){
    list<int> g[n+1]
    int inc[n+1] = 0
    Step 1: Adj list & incoming edge count
                                  // dependency
    for(i=0; i<E; i++){ TC: O(E)
        // edge from u[i] → v[i]
        g[u[i]].add(v[i])
        inc[v[i]]++
    }
    Step 2: Nodes with 0 incoming edge execute first
    Queue<int> q; TC: O(N)
    for(i=1; i<=n; i++){
        if(inc[i]==0){
            q.insert(i)
        }
    }
    Step 3: Order of Execution:
    while(q.size()>0){ TC: O(N+E)
        int u = q.front()
        q.delete() // deleting front node
        print(u) // Task u is Completed
        // Resolve dependies of u?
        for(i=0; i<g[u].size(); i++){
            int v = g[u][i]
            inc[v]--;
            if(inc[v]==0){ // for task v dependies resolved
                q.insert(v)
            }
        }
    }
}
```

Detecting Cycle in Directed Graph

Topological Order

Eg1:



	0	1	2	3	4	5	6
inc[ ]:	0	0	~~1~~ 0	~~2~~ 1	1	1	1

Queue:

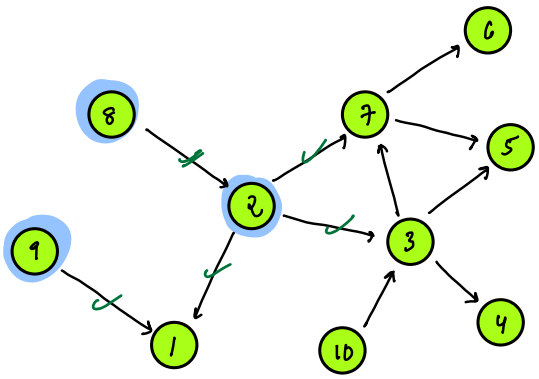
~~1~~	~~2~~

1. {We deleted 2 nodes & Queue is Empty}
2. Given 6 Tasks, resolved 2 Tasks?

Obs: In your directed Graph if we have a Cycle we cannot ensure all tasks.

```
void Cycle detection ( int N, int E, int u[], int v[]){
    list<int> g[n+1]
    int inc[n+1] = 0
    Step1: Adj list & incoming edge count  // dependency
    i=0; i<E; i++){ TC: O(E)
        // edge from u[i] → v[i]
        g[u[i]].add(v[i])
        inc[v[i]]++
    }
    Step2: Nodes with 0 incoming edge execute first
    Queue<int> q; TC: O(N)
    i=1; i<=n; i++){
        if(inc[i]==0){
            q.insert(i)
        }
    }
    Step3: Order of Execution:
    int c=0
    while(q.size()>0){ TC: O(N+E)
        int u = q.front()
        q.delete() // deleting front node
        print(u) // Task u is Completed
        c=c+1 // increase no: of tasks resolved
        // Resolve dependies of u?
        i=0; i<g[u].size(); i++){
            int v= g[u][i]
            inc[v]--;
            if(inc[v]==0){// for task v dependies resolved
                q.insert(v)
            }
        }
    }
    if(c==n){return no cycle} else{return cycle}
}
```

lexicographical topological Order



Create in[]: Store count of incoming Edges:

	0	1	2	3	4	5	6	7	8	9	10
cin[11] =	0	2	1	2	1	2	1	2	0	0	0
		1	0	1				1			
		0									

DataStructure: [8 1 / 9 2] 8 2 9 1 . . .

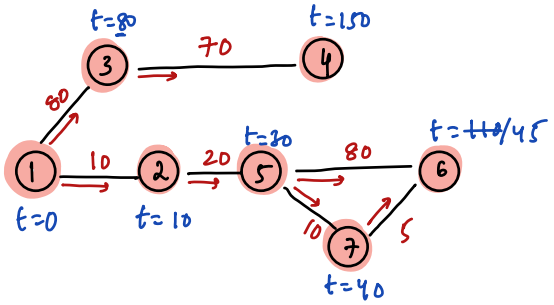
// Everytime we Pick min, so DataStructure = Minheap.

// In above Code replace queue with minheap.

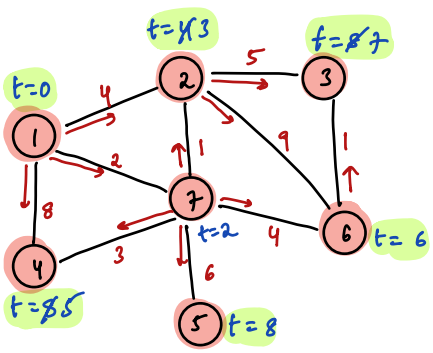
Fire at Petrol Bunk:

- → Nodes indicates petrol bunk
- → Edges indicates connection between 2 bunks & length of connection. pipes also contain petrol
- → Initially Say bunk 1 blasted / Petrol burns at 1km/1min
- → Calculate time at which each bunk is blasted

Ex:



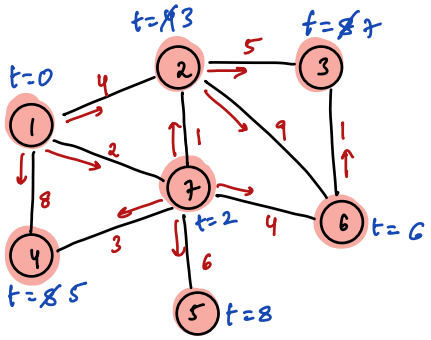
Ex2:

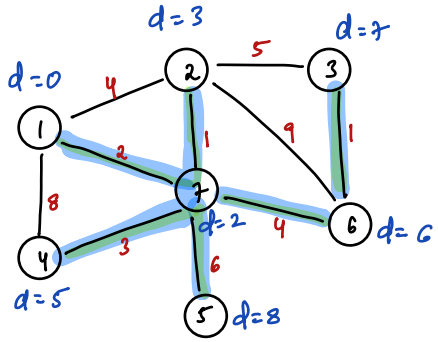


Idea:

1. At every Step blast bunk with min time
2. Once a bunk is blasted
   a) Fire is propaged towards it's adjacent nodes/ bunks
   b) We update blast time of Adjacent nodes to a smaller value.

Code:





int time[8] =

0	1	2	3	4	5	6	7
∞	0	∞ 4 3	∞ 8 7	∞ 8 5	∞ 8	∞ 6	∞ 2

time ← → node

Minheap< pair<int, int> >

Note: If time of node is updated insert in the minheap

t n

<4, 2> skip
<2, 7> deleted
<3, 2> del
<8, 4> skip
<5, 4> del
<6, 6>
<8, 5>
<8, 3>
<7, 3>

node: 2 time_heap: 4

In time[] node: 2 time[2] : 3

if(time_heap > time[]):

node already blasted Skip.

Notes: After blasting node need to go to its adj nodes hence we need adj list

Final col: Min dist path Src node to all nodes

Algo: Dijkstras

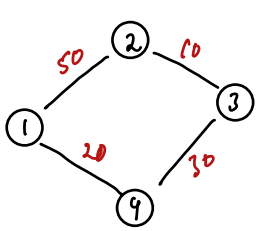
Weighted Shortest path from S → D, we use Dijkstras

google maps →

Weighted Graph:

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

g[5]



	n w	n w
0		
1	<2 50>	<4 20>
2	<1, 50>	<3, 10>
3	<2, 10>	<4, 30>
4	<3, 30>	<1, 20>

```
Blast(int N, int E, int u[], int v[], int w[], int s){

Step1: Create adj list
        list<pair<int,int>> g[N+1]
        i=0; i<E; i++){
            // u[i] --w[i]-- v[i]
            g[u[i]].add(pair{v[i], w[i]})
            g[v[i]].add(pair{u[i], w[i]})
        }

Step2: Getting Blast Times
        int time[n+1] = {∞}, time[s] = 0
                    time ↔ node
        minheap<pair<int, int>> mh; mh.insert({0, s})
        while(mh.size()>0){
                    t  node
            pair<int, int> data = mh.getMin();
            mh.deleteMin();
            int t = data.first, u = data.second
            if(t > time[u]){ // u is already blasted
                continue
            }
            // u : blasted 1st time, iterate on adj nodes
            i=0; i<g[u].size(); i++){
                        node weight
                pair<int, int> data2 = g[u][i]
                int v = data2.first  w = data2.second
                t[u]                    t[v]   // Time taken for fire to reach v via u
                // u ------ w ------> v        time[u]+w
                if(time[v] > time[u]+w){
                    time[v] = time[u]+w
                    mh.add(time[v], v)
                }
            }
        }
    return time[].
}
```

TODO TC:? $N+E+E\log E$

SC: $N+E$