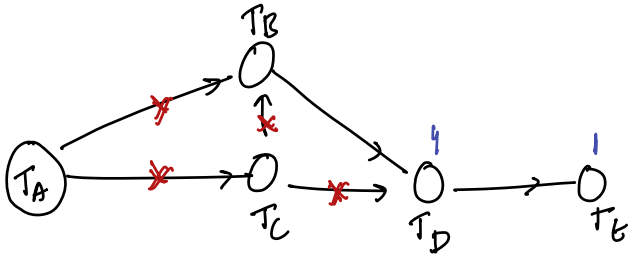
Today's Content:

→ Topological Sort → lexico
→ cycle detection

→ Dijkstra's Algo → [Brief Thursday + Code]

→ Sorting won't work.



T_A	T_B	T_C	T_D	T_E
0	2	1	2	1

Sort : T_A T_C T_E T_B T_D

We cannot resolve T_E

It fails

Topological Sort :

T_B depends on T_A

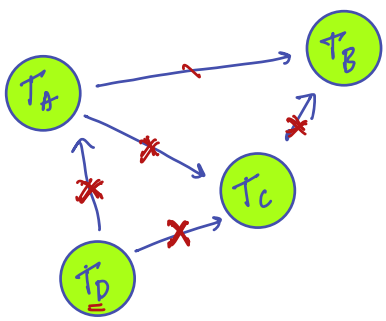
Recursion → dp : // $T_A \longrightarrow T_B$: First T_A should be done follows T_B

a) That means dp depends on Recursion

b) Before doing dp we need to study Recursion

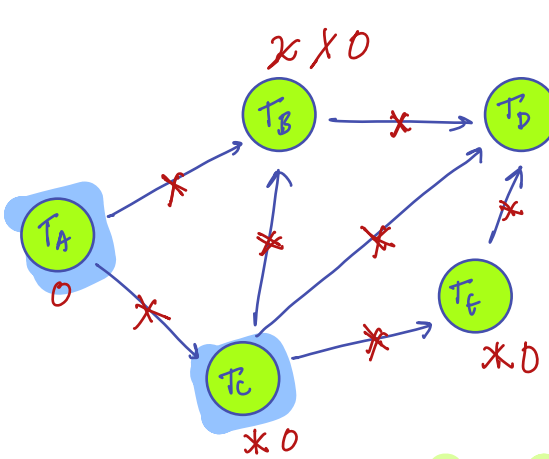
Note: If multiple orders are possible, pick lexicographically smaller

Eg1:



Task Order : T_D T_A T_C T_B

Eg2:



Task Order: T_A T_C T_B T_E T_D

Obs1: When there is no incoming edge it's dependencies are resolved

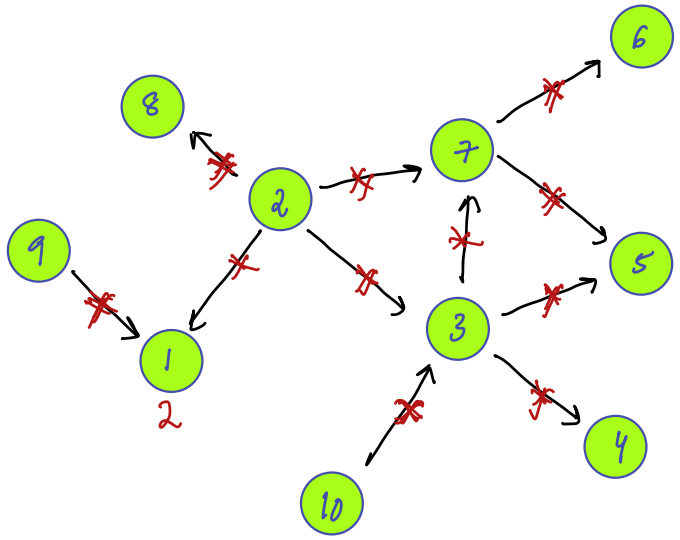
Count Incoming Edges :

T_A	T_B	T_C	T_D	T_E
0	2 → 1 → 0	1 → 0	3 → 2 → 1 → 0	1 → 0

Idea: Sorting worst work

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

Eg:



Adj:

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

Count Incoming Edges

int indeg[11] =

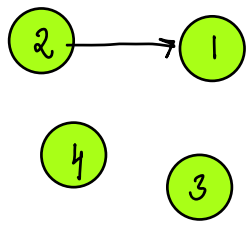
0	1	2	3	4	5	6	7	8	9	10
	x → x → 0	0	x → x → 0	1 → 0	2 → 1 → 0	1 → 0	2 → 1 → 0	1 → 0 =	0	0

one of the topological order →

2	9	10	8	1	3	5	7	4	6

↳ above order might not be lexicographically smallest

Eg:



lexicographically smallest order: 2 1 3 4

As per above approach:

2	3	4	1

→ order → 2 3 4 1

```
// Given graph as Input  N & M Edges → TC: O(N+E)
                                       SC: O(N+E)
                                        ↳ To store graph
List g[N+1]                               + indeg[]
int indeg[N+1]                            + Queue[]

for(i=1; i<=M; i++){
    // for edge will contains u,v . u → v
    indeg[v]++;
    g[u].add(v)
}
Queue<int> q;
for(i=1; i<=N; i++){        // ith node is already resolved
    if( indeg[i]==0) { q.push(i) }
}
while( q.size()>0){
    int u = q.front();
    q.remove()
    print(u) // indicating Task u is solved, clear its dependies
    for(i=0; i<g[u].size(); i++){
        int v = g[u][i] // u → v
        indeg[v]--;
        if(indeg[v]==0) { q.push(v) }
    }
}
```

// lexicographically smaller one :

→ out of all nodes which have in-degree we should always pick min (minheap) // We insert a node in minheap when it's indeg becomes =0

→ instead of Queue we minheap

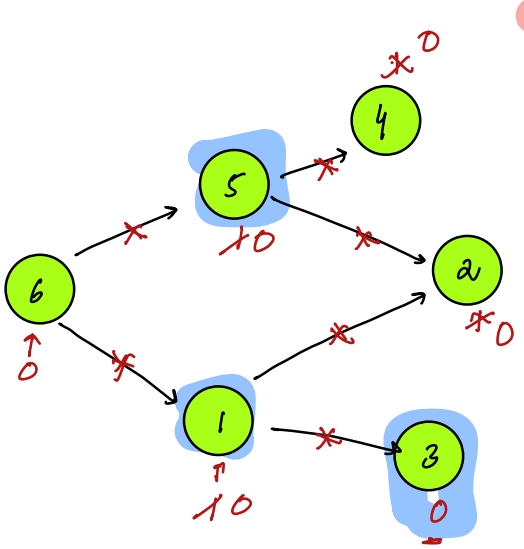
→ TC : ( ) → {TC → ___ }

min → O(1)

delete min → O(log N)

insert → O(log N)

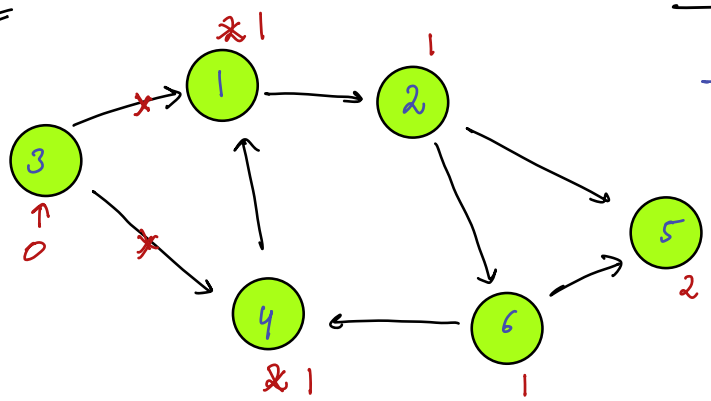
Minheap :→ Is Present in your language Lib



6, 1 3 5 2 4

lexicographically smaller Topological Order

Eg :



→ If there is a cycle in your graph, we cannot resolve all dependencies we cannot get topological sort

' Given graph as Input N & M Edges TC → O(N+G)

→ To check if Cycle in Directed graph SC → O(N+G)

```
list g[N+1]
int indeg[N+1]

for(i=1; i<=M; i++){
  // for edge will contains u,v . u → v
  indeg[v]++;
  g[u].add(v)
}

Queue<int> q;
for(i=1; i<=N; i++){        // ith node is already resolved
  if( indeg[i]==0) { q.push(i) }
}

int c=0
while( q.size()>0){
  int u = q.front();
  q.remove()
  print(u) // Indicating Task u is solved, Clear its dependies
  c++
  for(i=0; i<g[u].size(); i++){
    int v = g[u][i] // u → v
    indeg[v]--;
    if( indeg[v]==0) { q.push(v) }
  }
}
                    // Cycle is formed
if( c != N) { return True }
                  ↳ Indicate Cycle
```

Unweighted graph:

S → D : shortest path from S → D : graph1

Weighted graph

S → D : shortest path from S → D

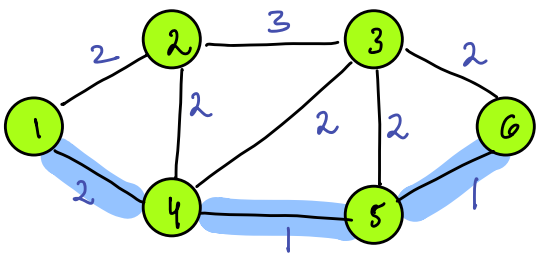
Undirected graph : Cycle detection : graph 2

Directed graph : Cycle detection / Cycle detection can be done using DFS

Break : 10:48 pm

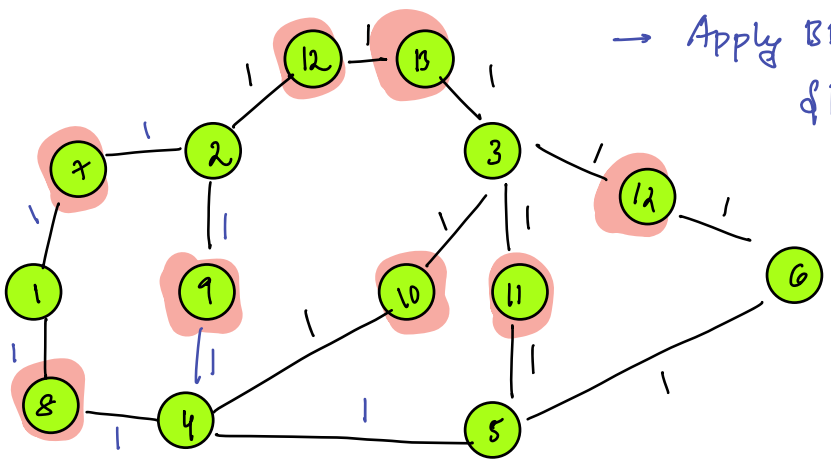
28) Given a weighted graph with +ve weights find the min cost to reach from S → D

S	D	val
1	6	4



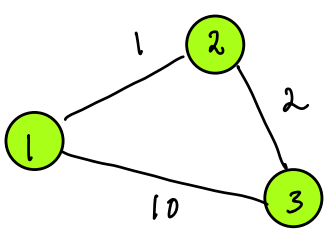
Eg1:

Ideal: → (All weights in graph we are making them → 1)

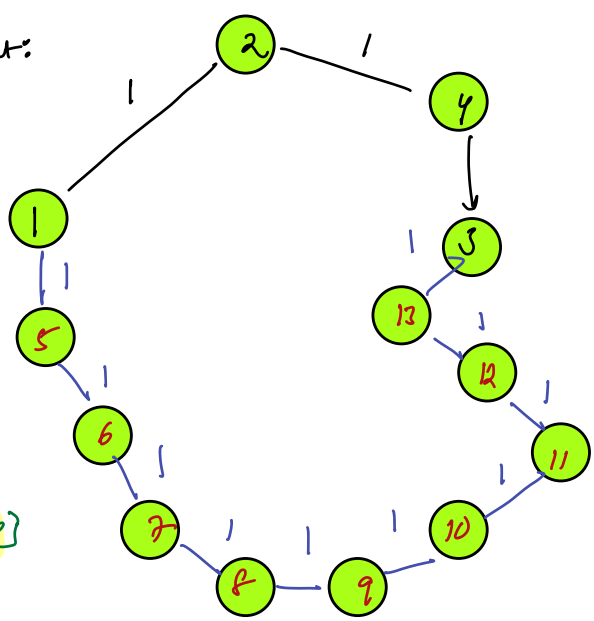


→ Apply BFS to get length of shortest path from S → D

→ Eg:



→ Convert:



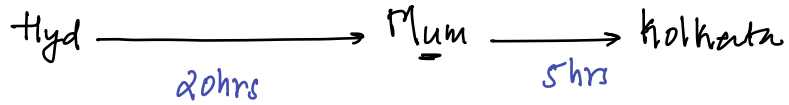
Issue Space Wastage:

→ Extra Nodes: → (Huge Space Wastage)

Total Sum of Weight - {No: of Edges}

→

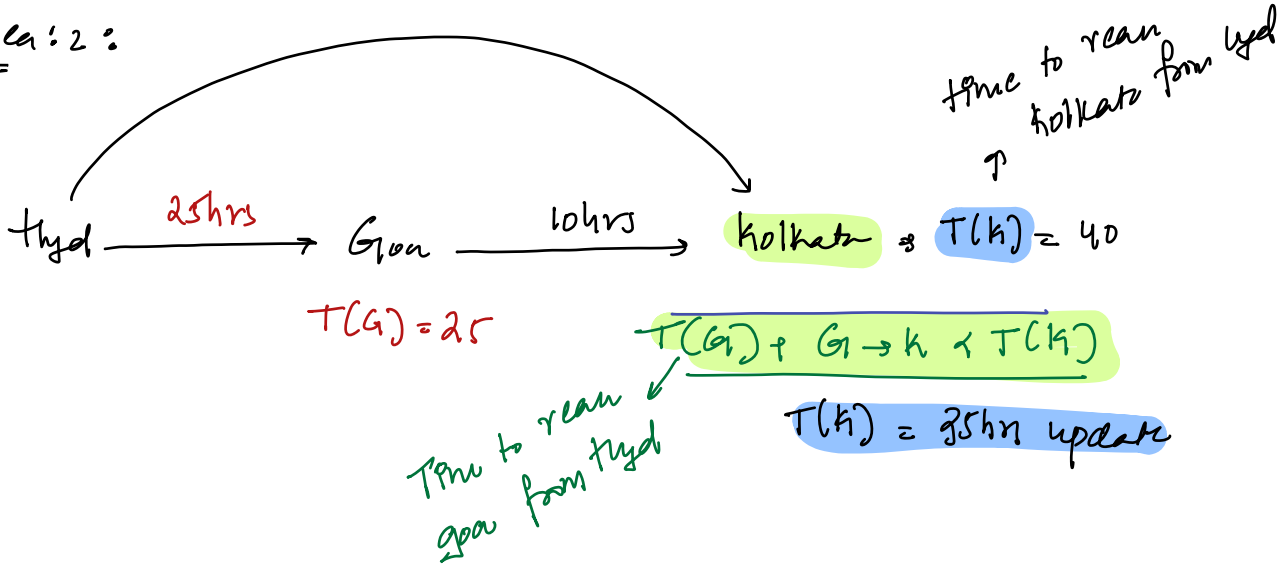
Basic Prerequisits :



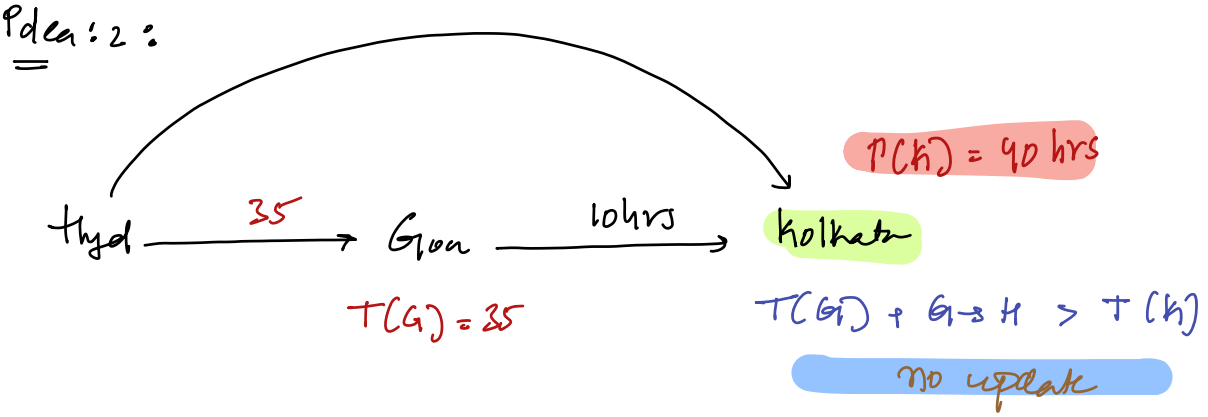
→ Time taken to reach from Hyd → Kolkata via Mum = 25 hrs

→ Dist (Mum) + (Mum → Kolkata) = 25

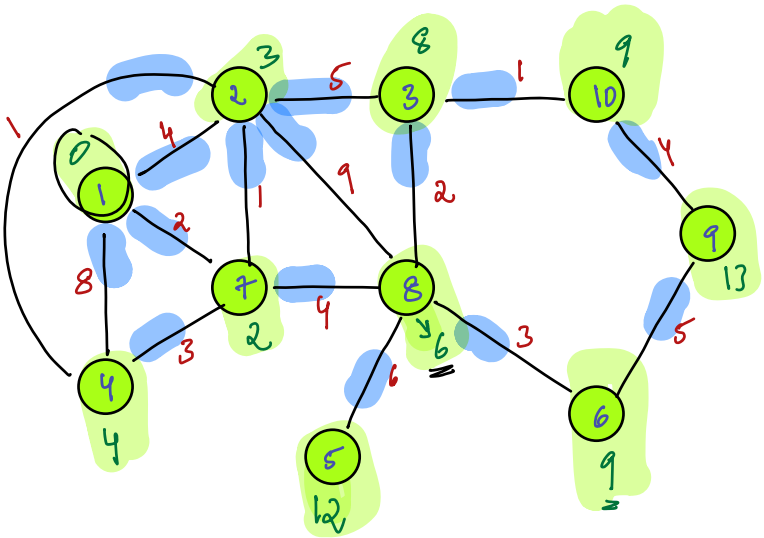
Idea : 2 :



Idea : 2 :



Idea2: → wt → {dist}



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⟩
9	⟨6,5⟩ ⟨10,4⟩
10	⟨3,1⟩ ⟨9,4⟩

S = 1

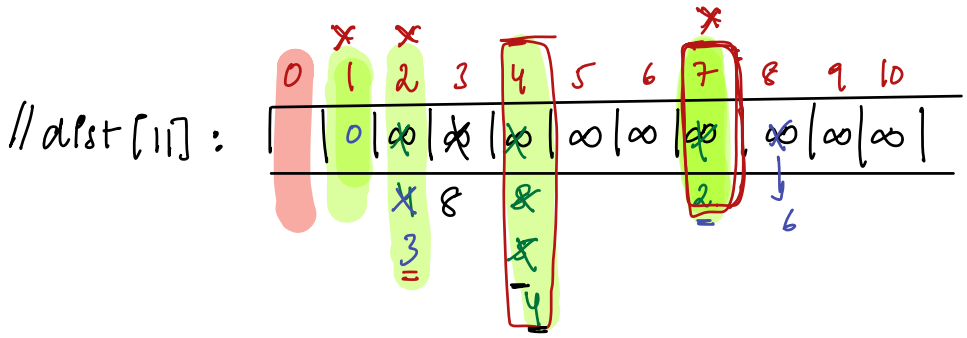
// dist[5] = dist to reach from S → 5

// dist[i] = dist to reach from S → i

// dist[11]:

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

⑦ →
- 2 : dist[2] = 4, dist[7] + 7 → 2 → 3 : dist[2] = 3
- 8 : dist[8] = ∞, dist[7] + 7 → 8 → 6 : dist[8] = 6
- 4 : dist[4] = 8, dist[7] + 7 → 4 → 5 : dist[4] = 5



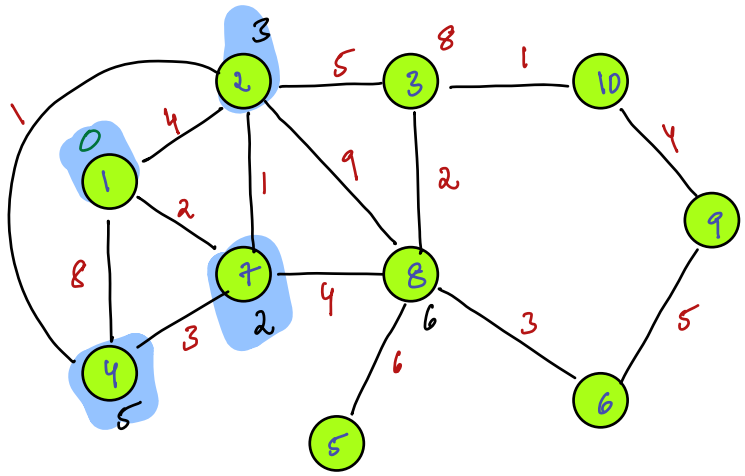
(2) →
- 3 : dist[3] = ∞, dist[2] + 2 → 3 ⇒ dist[3] = 8
- 4 : dist[4] = 5, dist[2] + 2 → 4 ⇒ dist[4] = 4
- 7 : dist[7] = 2 dist[2] + 2 → ∄ dist[7] = 2
- 8 : dist[8] = 6 dist[2] + 2 → 8 dist[8] = 6

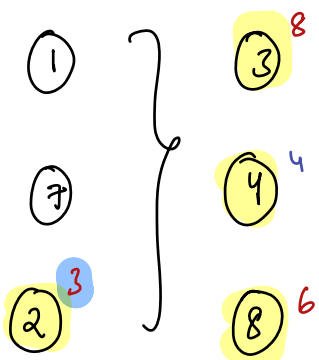
⇒ Using the above from given source we can get shortest distance to all other nodes (edges)

- → At every step node with min dist
- → With that node we are updating all its adjacent nodes

→ Thursday : {Brief + Code}

//


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


1
3
8
7
4
4
2
3
3
6

//