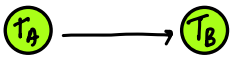
Todays Content:

1. Topological Sort
2. Dijtras Algorithm

Topological Sort:

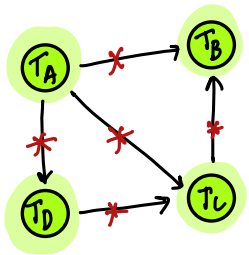
1. Recursion ⟶ Dynamic Programming



: Finish T_A & goto T_B

T_B is depending on T_A

Ex1:

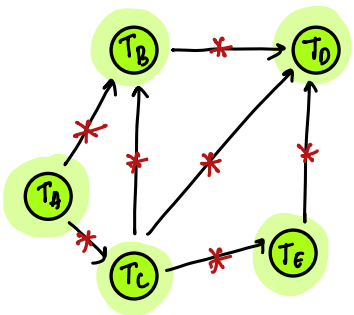


: Order of Execution of Tasks

Note: Before we execute a task, resolve all its dependencies.

Order: T_A T_D T_C T_B

Ex2:



Order1: T_A T_C T_E T_B T_D → random correct order

Order2: T_A T_C T_B T_E T_D → lenicographically correct order

Obs: If no: of incoming edges = 0, no dependencies

Idea: Keep a track of incoming edges for all nodes

Idea:	T_A	T_B	T_C	T_D	T_E
Incoming Edge:	0	~~2~~	~~1~~	~~3~~	~~1~~
		~~1~~	0	~~2~~	0
		0		~~1~~ : 0	

Queue:

Datastructure

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

Order : T_A T_C T_B T_E T_D → correct order to execute.

Adj list:

T_A : T_B T_C

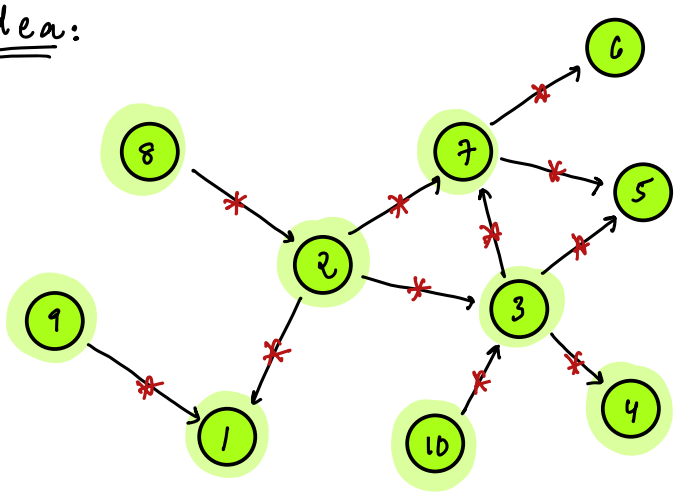
T_B : T_D

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:

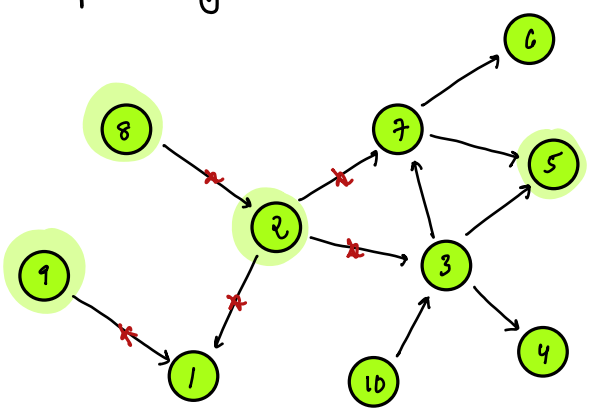
	0	1	2	3	4	5	6	7	8	9	10
cin[i] =	0	~~x~~	~~1~~	~~2~~	~~1~~	~~2~~	~~1~~	~~2~~	0	0	0
		~~x~~	0	~~x~~	0	~~x~~	0	~~x~~			
			0		0		0				

1-based indexing

Queue: ~~8~~ ~~9~~ ~~10~~ | ~~x~~ ~~x~~ ~~3~~ | ~~7~~ ~~4~~ ~~5~~ ~~6~~

order: 8 9 10 2 1 3 7 4 5 6 → correct order to execute.

lexicographically smallest:



Create in[]: Store count of incoming edges:

	0	1	2	3	4	5	6	7	8	9	10
cin[i] =	0	~~x~~	~~1~~	~~2~~	1	2	1	~~2~~	0	0	0
		~~x~~	0	1				1			
		0									

Datastructure:

Priority Que: Min
min, deletemin, insert { ~~8~~ ~~9~~ 10 ~~2~~ ~~1~~ }

Order: 8 2 9 1 .... → lexicographically correct order.

Topological Sort

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

```
void TopoSort( int N, int E, int u[], int v[]){   // u[i] → v[i]
```

TC: O(N+E)

SC: O(N+E)

Step0 : Create Adj List TODO

```
ArrayList<ArrayList<Integer>> g = new ArrayList<>();
```

Step1: Adj list & incoming edge count

```
int inc[N+1] = 0; // 1 based indexing:
for(int i=0; i<E; i++){
    // Edge u[i] → v[i]
    inc[v[i]]++;
}
```

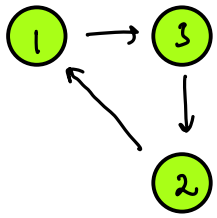
Step2: We Insert nodes, with 0 dependcies in Que

```
Queue<int> q;
for(int i = 1; i <= N; i++){
    if( inc[i] == 0) { q.add(i)}
}
```

Step3: Get Order of Execution.

```
while( q.size() > 0){
    int u = q.front();
    q.delete(); // remove node from front
    print(u); // resolve all dependcies from u.
    for(int i=0; i < g.get(u).size(); i++){
        int v = g.get(u).get(i);
        inc[v]--;
    }  if[ inc[v] == 0) { q.add(v)}
}
}
```

Ex:

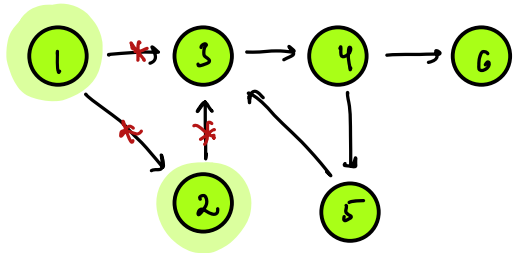


Note: We cannot execute tasks, because

Detect cycle in Directed Graph:

Obs: If we have a cycle in a directed, we cannot execute all tasks when we apply topological sort.

Ex2:



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

0 ~~2~~

2

Queue: ~~2~~ ~~2~~

2 2

Note: Our queue is already empty, loops stops.
We didn't resolve all nodes, we have cycle

```
boolean Cycledetection( int N, int E, int u[], int v[]){   TC: O(N+E)
                                                           SC: O(N+E)
  Step0 : Create Adj List TODO
  ArrayList<ArrayList<Integer>> g = new ArrayList<>();

  Step1: Adj list & incoming edge count
  int inc[N+1] = 0; // 1 based indexing:
  for(int i=0; i<E; i++){
  |  // Edge u[i] → v[i]
  |    inc[v[i]]++;
  }

  Step2: We Insert nodes, with 0 dependcies in Que
  Queue<int> q;
  for(int i = 1; i<= N; i++){
  }  if( inc[i] == 0) { q.add(i)}

  int c=0;
  while( q.size() >0) {
  |    int u = q.front();
  |    q.delete(); // remove node from front
  |    print(u); // resolve all dependcies from u.
  |    c = c+1;
  |    for(int i=0; i< g.get(u).size(); i++){
  |    |    int v = g.get(u).get(i);
  |    |    inc[v]--;
  |    }  if( inc[v] == 0) { q.add(v)}
  }
  if( c < N) { return true} → cycle exist.
  else {return false}
}
```

Q: Fire = Petrol Bunk

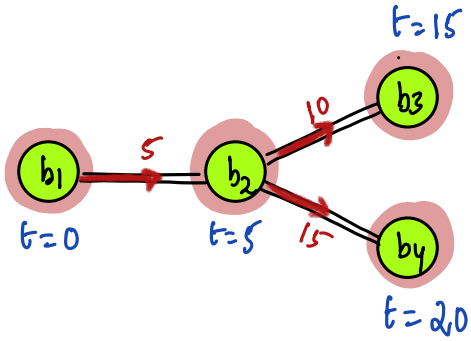
● → Representing petrol bunk ↗ filled with petrol

a. ——— line indicates lengths of petrol pipe between 2 bunks

b. Initially say bunk 1 blasted c. Petrol burns at 1km/min.

d. Calculate time at which each bunk is blasted

Ex1:



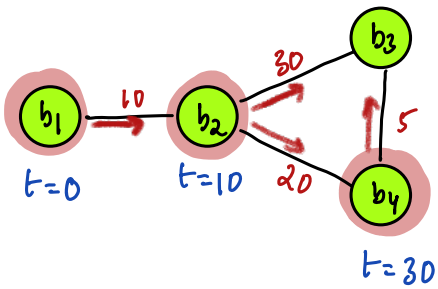
bunks : b_1 b_2 b_3 b_4

Time : 0 5 15 20

t = ~~40m~~, via = b_2

t = 35m, via = b_4

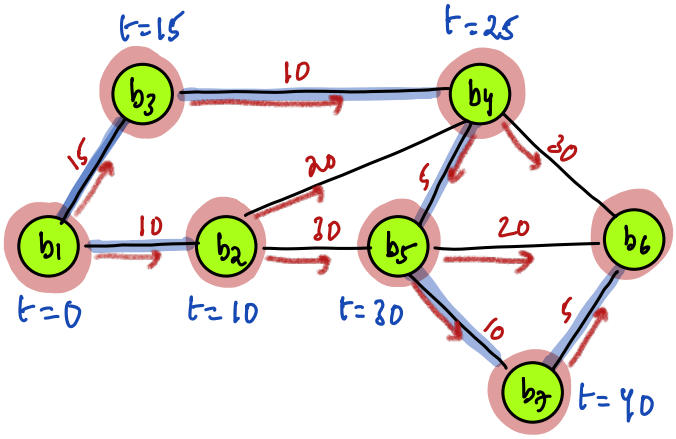
Ex2:



bunks : b_1 b_2 b_3 b_4

Time : 0 10 35 30

Ex3:



bunks: b_1 b_2 b_3 b_4 b_5 b_6 b_7

Time : 0 10 15 25 30 ~~55~~ ~~50~~ 45 40

↳ Inf: length of shortest path, from b_1 to all other bunks

Steps: At each step:

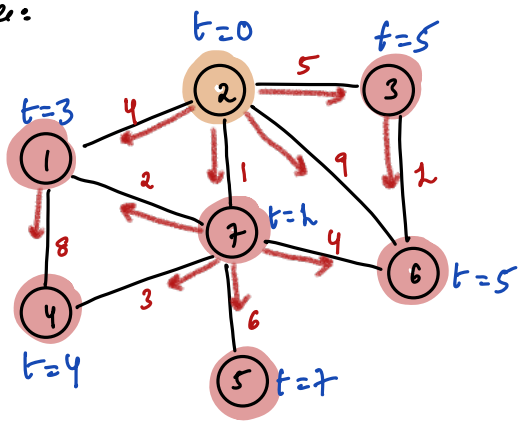
a) We blast the bunk with least time

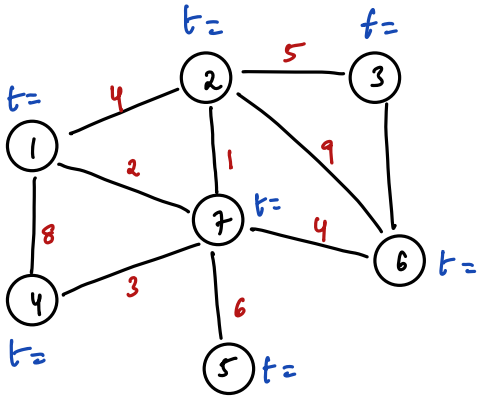
b) Fire propagates to: it's adjacent neighbours, We update blast time of bunks

↳ Dijkstra's Algo: Single Source → Shortest Paths:

↳ Used: Google maps, ...

Code:





	0	1	2	3	4	5	6	7
int time[8] =	∞	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~
	~~4~~	0	5	4	7	~~9~~	2	
	3					5		

PriorityQueue<Pair> pq = new PriorityQueue<>();

<node time>

<2:0>

<1:4> <1:3>

<7:2>

<6:9> <6:5>

<3:5>

<4:4>

<5:7>

Note: Define Priority based on time

Note2: Every time we update time for a node, we push in heap.

If node is already blasted:

1. if (node value in pq > node value in time[]){
   | already blasted, leave it.
   }

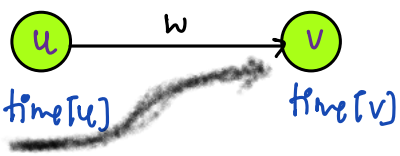
Note3: If priority queue is empty stop it, return time[].

```
class pair{
    int N;
    int d;
    pair(int a, int b){
        N=a, d=b;
    }
}
```

```
int[] Dijkstras(int u[], int v[], int w[], int s, int N, int E){
    //Step1: Adj list
    ArrayList<ArrayList<Pair>> g = new ArrayList<>();
    for(int i=0; i<=N; i++){
        ArrayList<Pair> al = new ArrayList<>();
    } g.add(al);
    //Step2: Filling array list
    for(int i=0; i<E; i++){ // u[i] ---w[i]--- v[i]
        g.get(u[i]).add(new pair(v[i], w[i]))
    } g.get(v[i]).add(new pair(u[i], w[i]))
    //Step3: Blasting nodes
    int dist[N+1] = INT_MAX; dist[s] = 0;
    PriorityQueue<Pair> pq = new PriorityQueue<>();
    pq.insert(new pair(s,0));
    while(pq.size()>0){
        Pair ele = pq.getMin(); // Pair ele : { u  d }
        pq.deleteMin();
        int u = ele.N;
        int d = ele.d;
        if(d > time[u]){ //already deleted; continue}
        for(int i=0; i<g.get(u).size(); i++){ // getting blasted 1st time
            Pair data = g.get(u).get(i);
            int v = data.N;                  // Pair data: { v  w }
            int w = data.d;
            if(time[v] > time[u]+w){
                time[v] = time[u]+w;
            } } pq.insert(new Pair(v, time[v]));
    }
    return time;
}
```

TC: TODO
↳ O(ElogE)
SC: O(N+E)

Comperator TODO



Cycle detection in Undirected Graph.

1. Calculate no: of components in graph
2. if (Edges == Nodes - Components) {
   } // No cycle
   else {
   } // Cycle Exist

Revison:

2. Revise all techniques :

   For every technique :

   1. Revise notes
   2. In mind solve assignment Problems
   3. If not able to solve a Problem.
      a. In a docu: make a note a Problem time which you wish revise this docu every week.