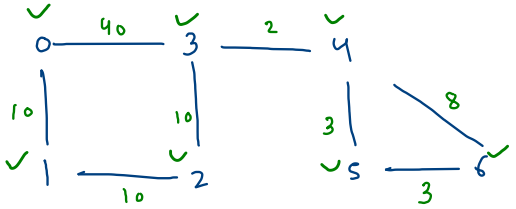
Shortest path in wt: single source to all dest

Shortest in single traversal.

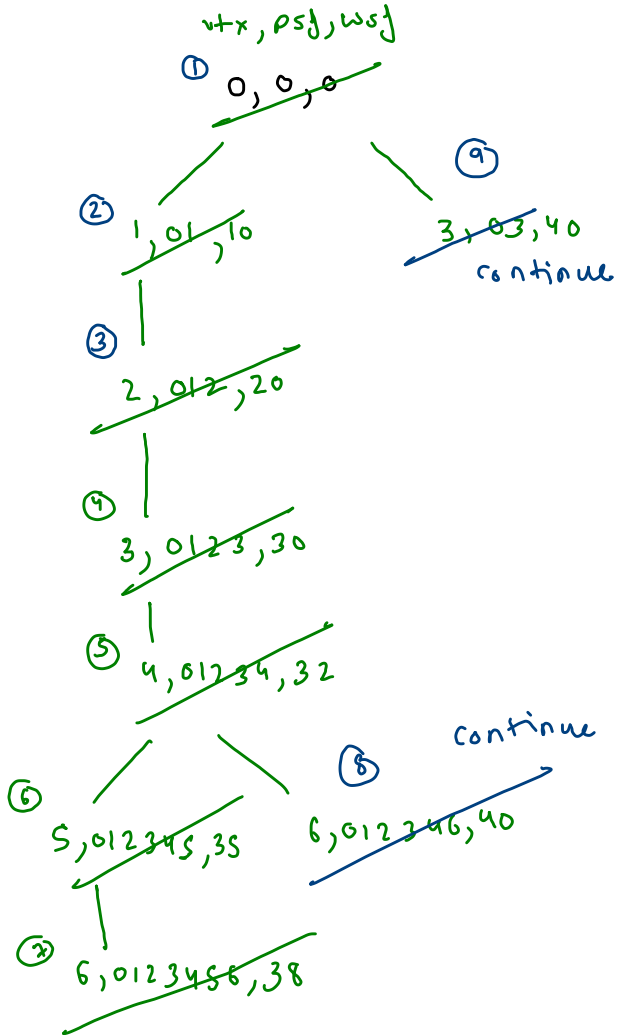


remove

mark*

work

add unvisited nbr



0 -> 0@0

1 -> 01@10 src=0

2 -> 012@20

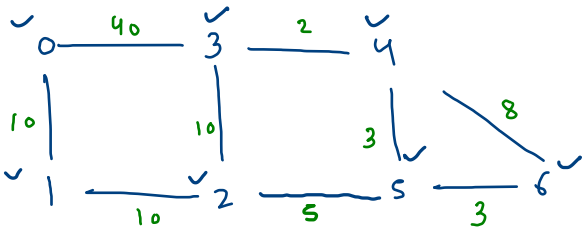
3 -> 0123@30

4 -> 01234@32

5 -> 012345@35

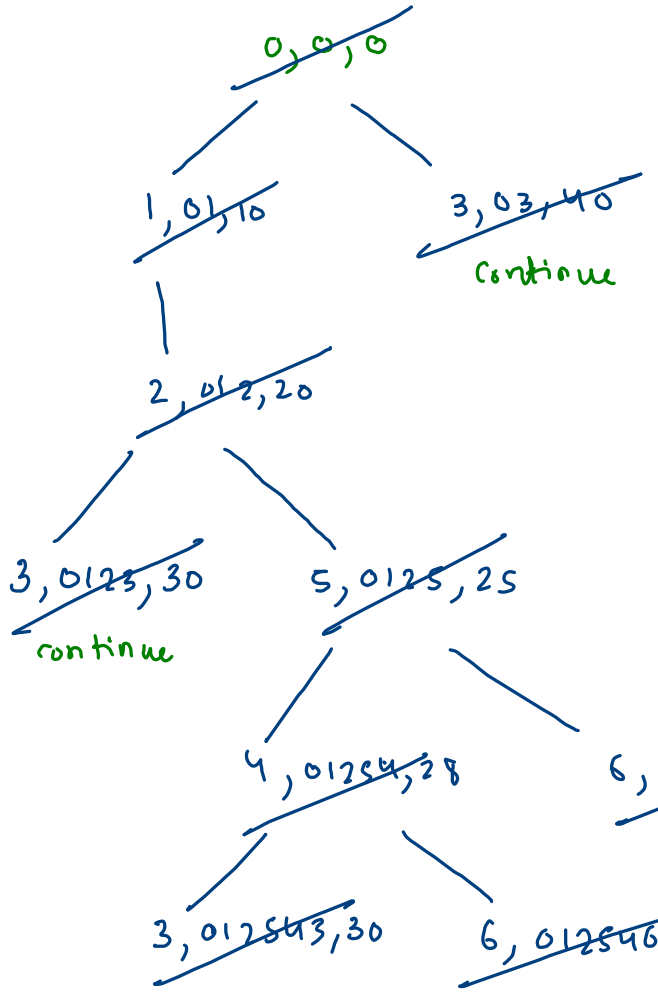
6 -> 0123456@38

BFS	Dijkstra
queue	pq
path is shortest (edges)	path is shortest (wt)




```java
public static void dijkstra(ArrayList<Edge>[]graph,int src) {
    PriorityQueue<Pair>pq = new PriorityQueue<>();
    boolean[]vis = new boolean[graph.length];

    pq.add(new Pair(src,src+"",0));

    while(pq.size() > 0) {
        //remove
        Pair rem = pq.remove();

        //mark
        if(vis[rem.vtx] == true) {
            continue;
        }
        vis[rem.vtx] = true;

        //work
        System.out.println(rem.vtx + " via " + rem.psf + " @ " + rem.wsf);

        //add unvisited nbr
        for(Edge ne : graph[rem.vtx]) {

            if(vis[ne.nbr] == false) {
                pq.add(new Pair(ne.nbr,rem.psf + ne.nbr,rem.wsf + ne.wt));
            }
        }
    }
}
```

dijkstra → single src to all dest.



src = 0

(vtx, psf, wsf)

0 -> 0 @0

1 -> 01 @10

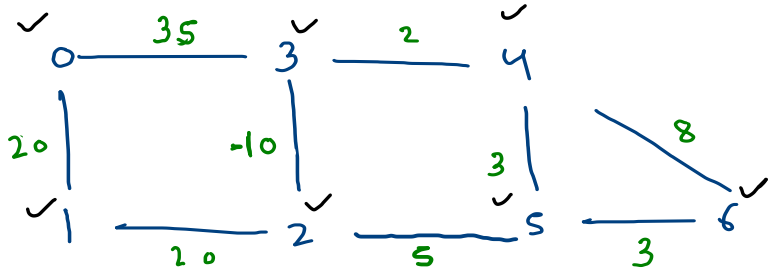
2 -> 012 @20

5 -> 0125 @25

4 -> 01254 @28

6 -> 01256 @28

3 -> 012543 @30



1 -> 01 @ 20

3 -> 03 @ 35

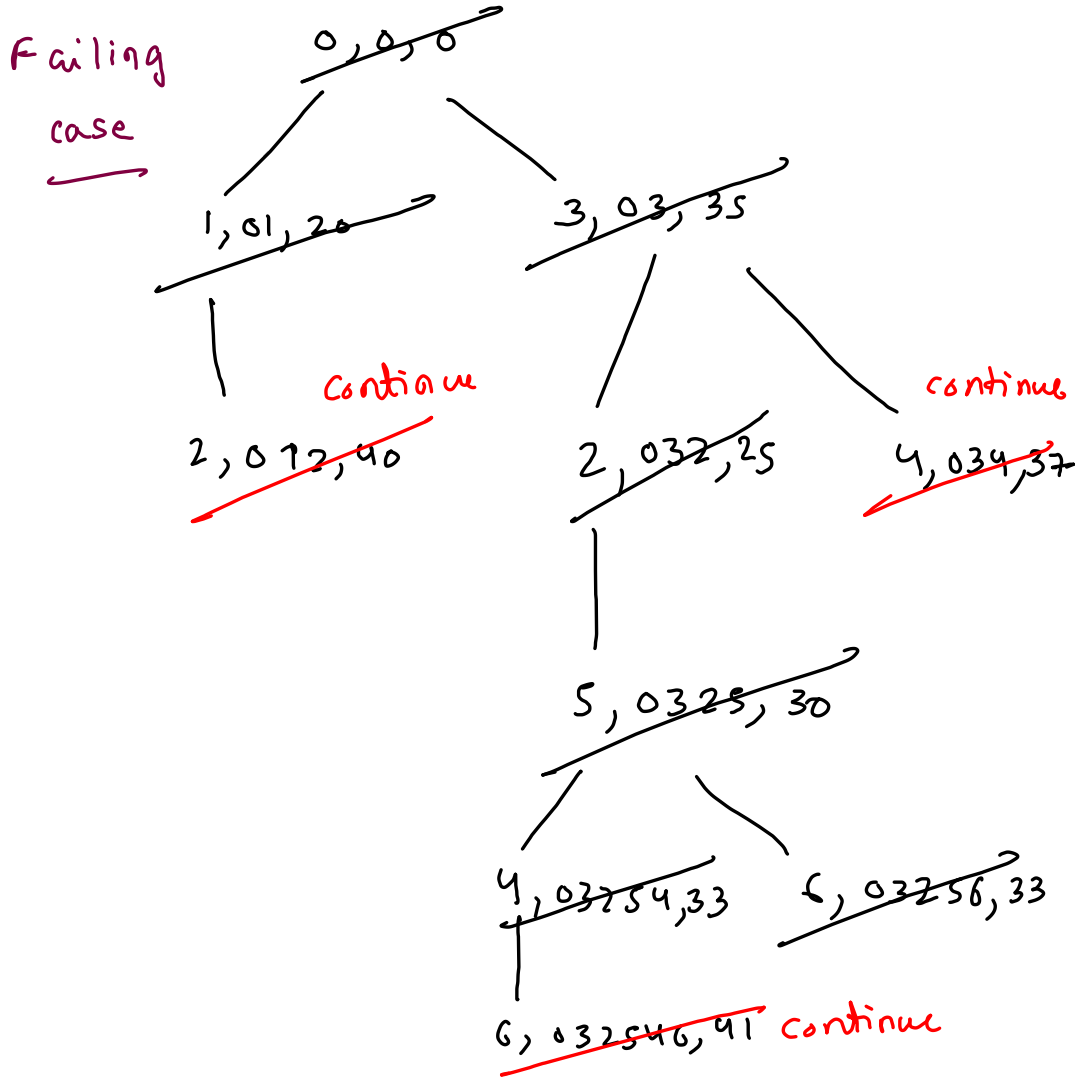
dijkstra : 2 -> 032 @ 25

fails on -ve edge wt 5 -> 0325 @ 30

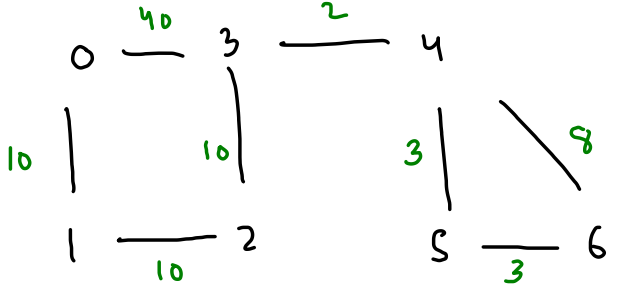
4 -> 03254 @ 33

6 -> 03256 @ 33

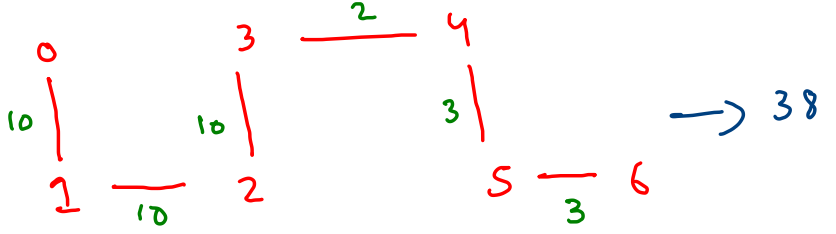
Failing case

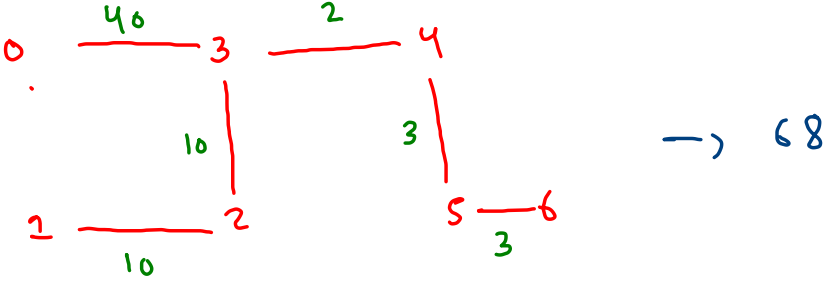


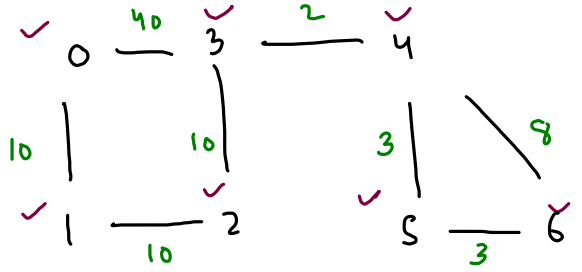
Prim's → MST (min spanning tree)

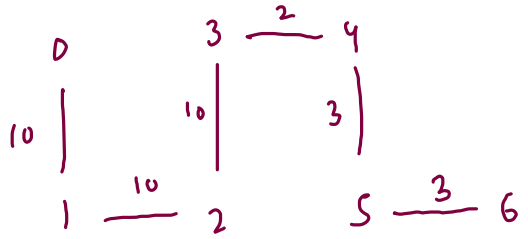


(mst)







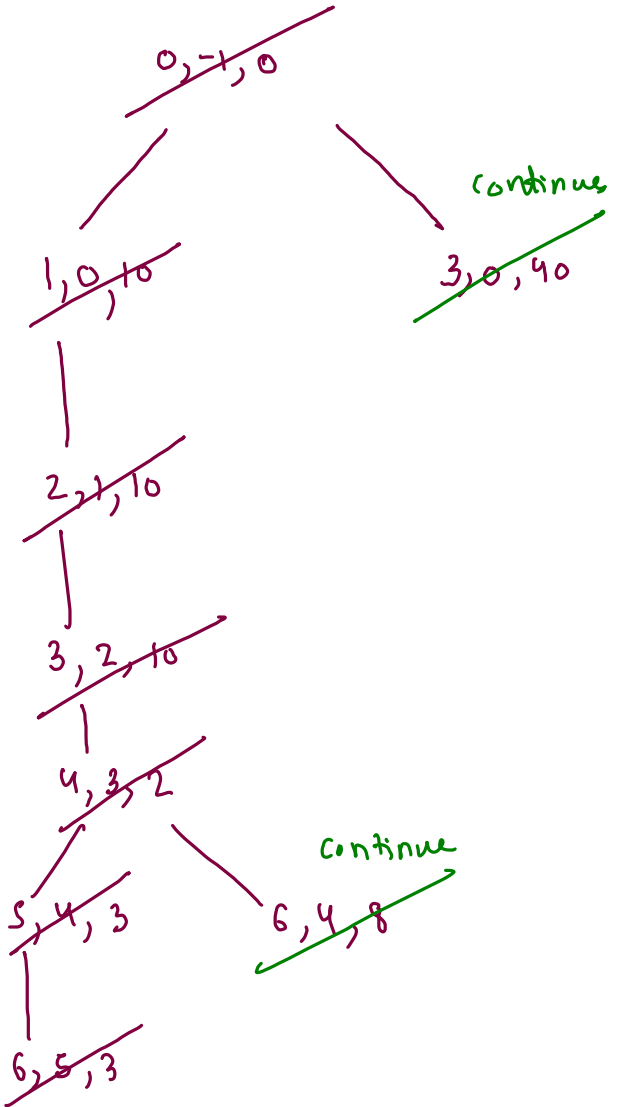


remove

mark*

work → add edge

add nbr



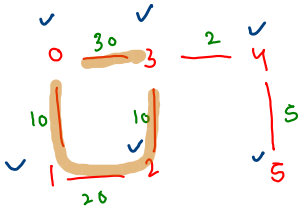
Pair :

Vtx, aq, wt.

Dijkstra

(i) shortest path (in terms of wt)

(ii) src -> single src to all dest shortest path.



0 -> 0 @0

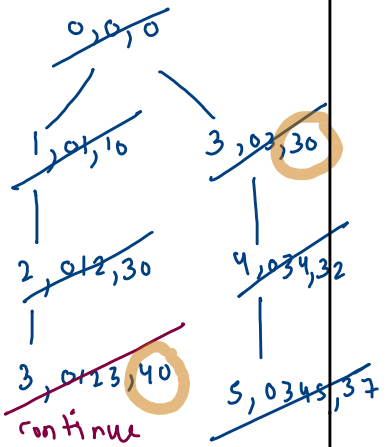
1 -> 01 @10

2 -> 012 @ 30

3 -> 03 @ 30

4 -> 034 @32

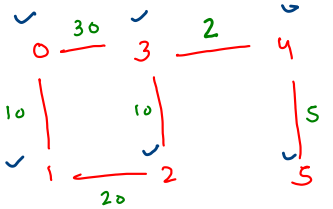
5 -> 0345 @ 37

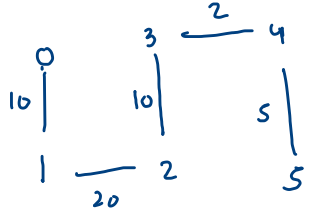


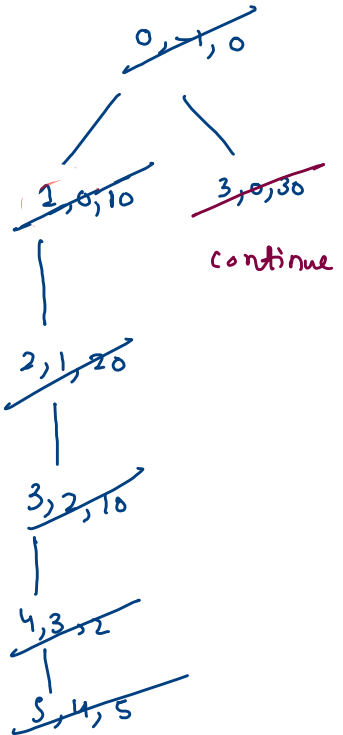
Prim

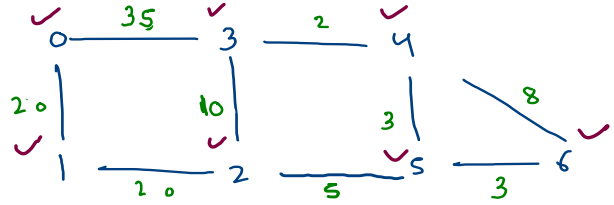
(i) MST (connect all vertices in min cost).

(ii) src : does not matter.

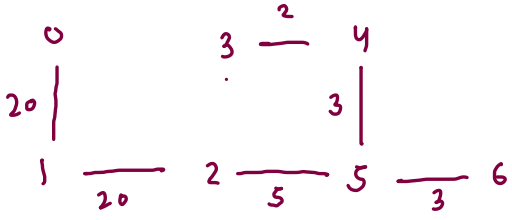




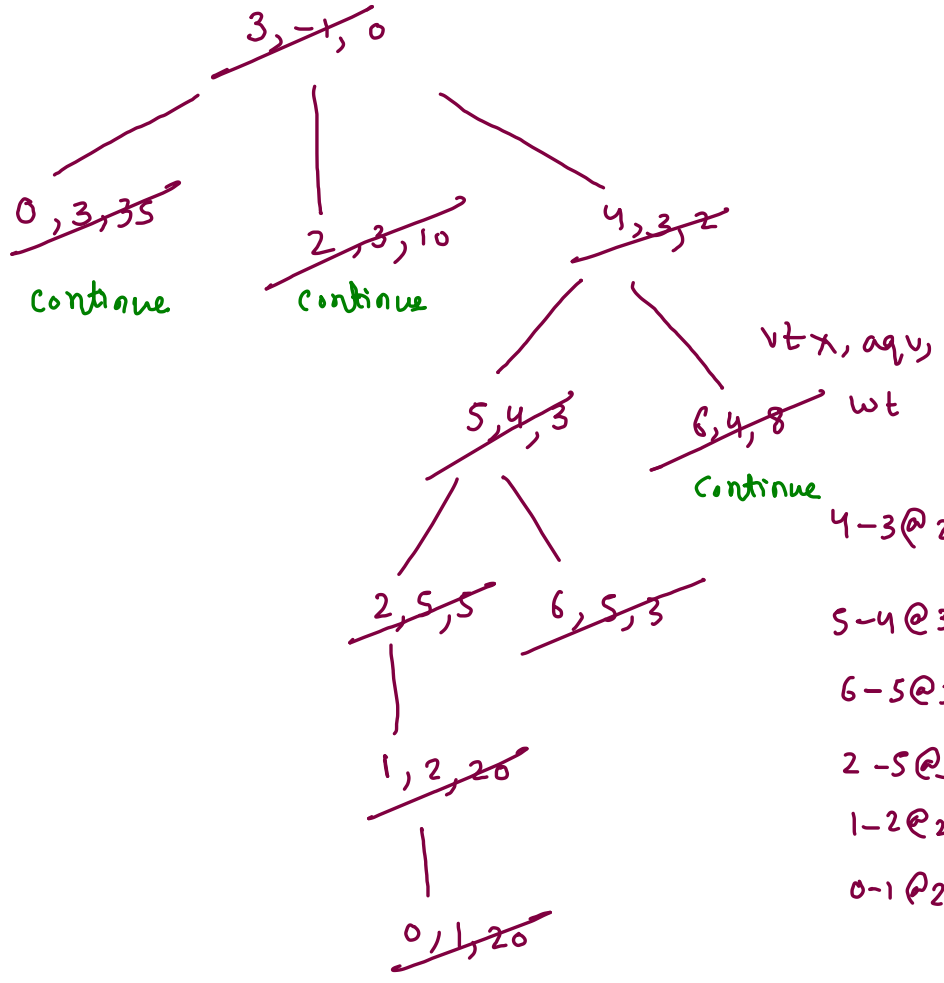


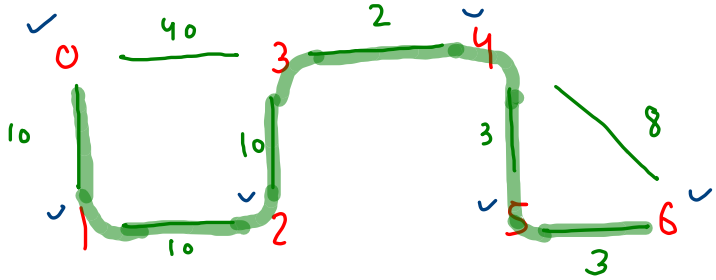


MST

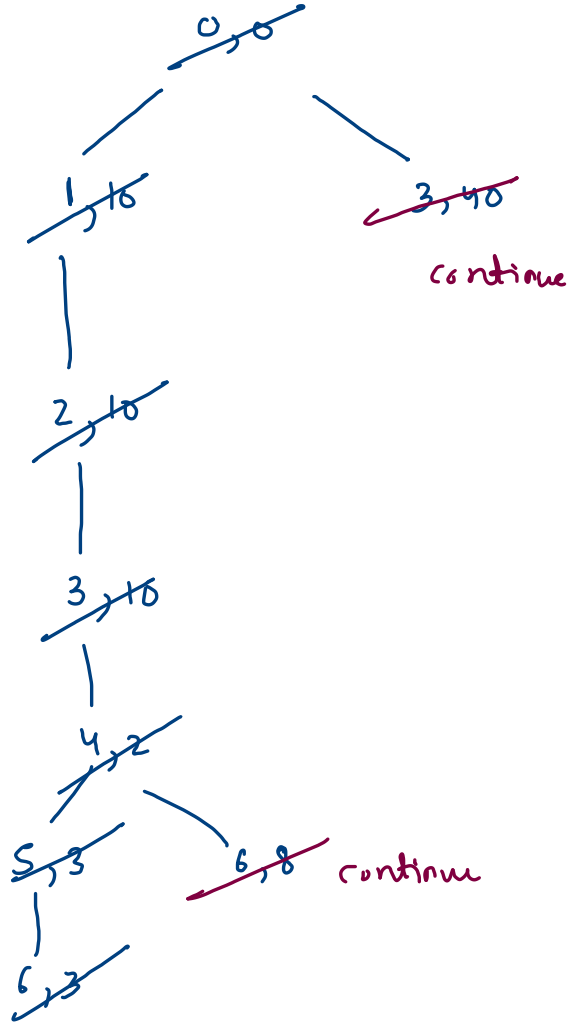



```
while(pq.size() > 0) {
    //remove
    Pair rem = pq.remove();

    //mark*
    if(vis[rem.vtx] == true) {
        continue;
    }
    vis[rem.vtx] = true;

    //work
    if(rem.aqv != -1) {
        System.out.println("[" +rem.vtx + "-" + rem.aqv + "@" + rem.wt + "]");
    }

    //add nbr
    for(Edge ne : graph[rem.vtx]) {
        if(vis[ne.nbr] == false) {
            pq.add(new Pair(ne.nbr,rem.vtx,ne.wt));
        }
    }
}
```






```
while(pq.size() > 0) {
    //remove
    Edge rem = pq.remove();

    //mark*
    if(vis[rem.v] == true) {
        continue;
    }
    vis[rem.v] = true;

    //work
    cost += rem.wt;

    //add nbr
    for(Edge ne : graph.get(rem.v)) {
        int nbr = ne.v;
        if(vis[nbr] == false) {
            pq.add(new Edge(nbr,ne.wt));
        }
    }
}
```



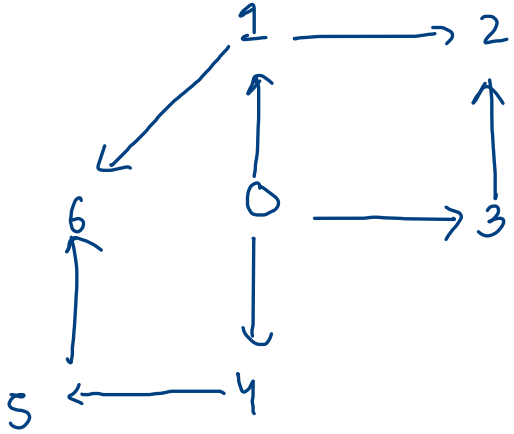
Cost = 10+10
+10+2
+3+3

= 38

Topological sort -> A permutation of vertices for a directed acyclic graph is called topological sort if for all directed edges uv, u appears before v in the graph

DAG

Directed acyclic graph



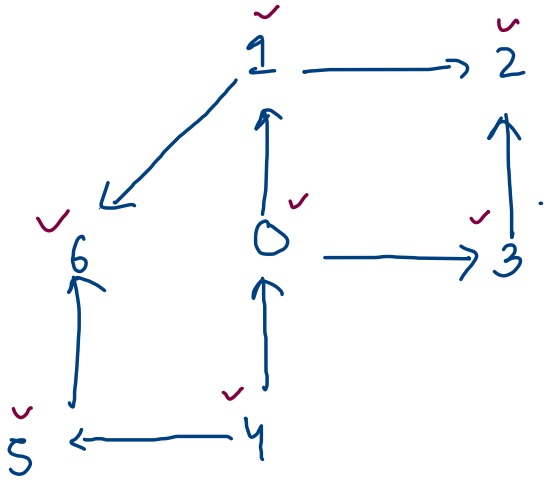
u → v

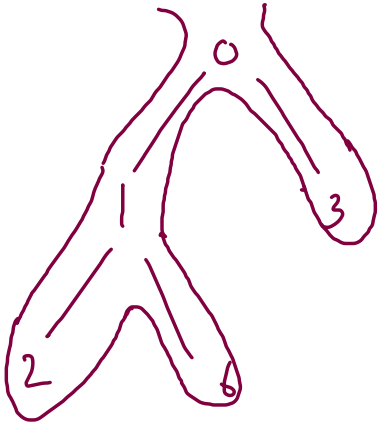
u should come before v

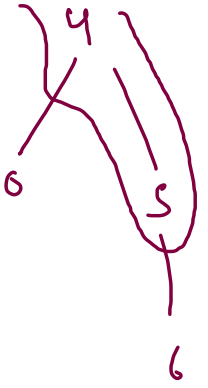
u is dependent on v

0 1 3 2 4 5 6 (topological sort)

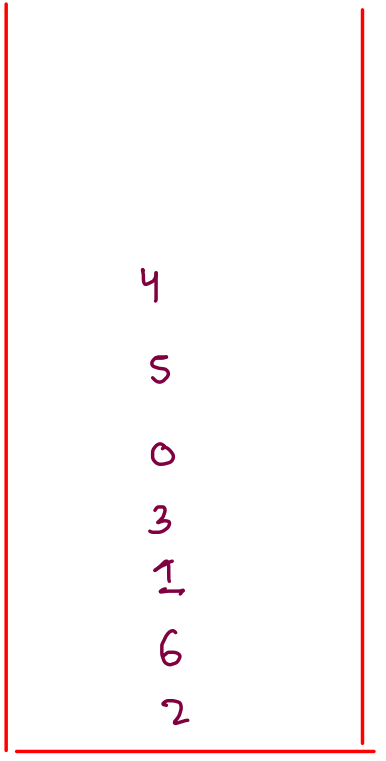
order of compilation → (topological sort)



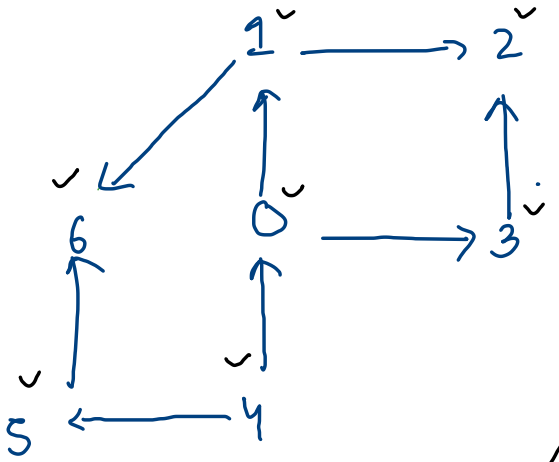


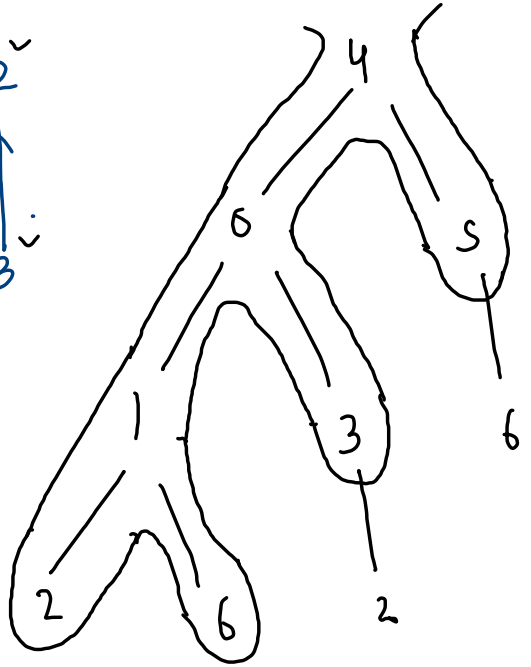


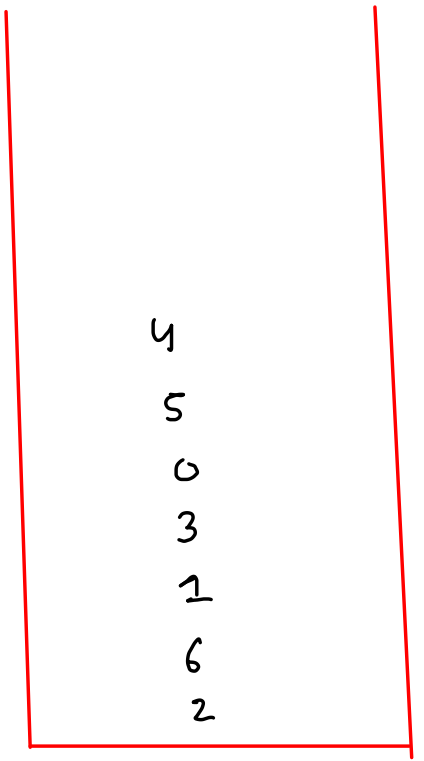
ts: 4 5 0 3 1 6 2



4.



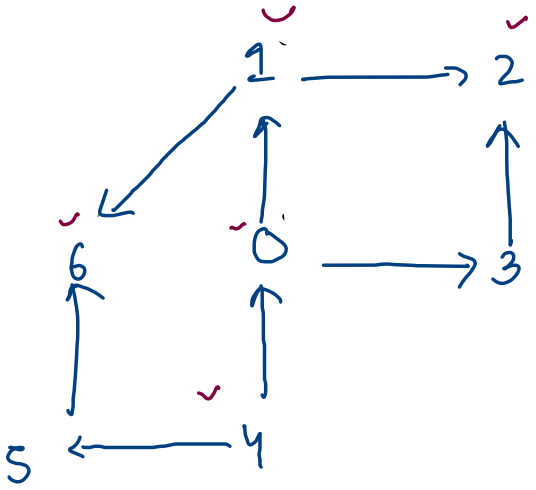


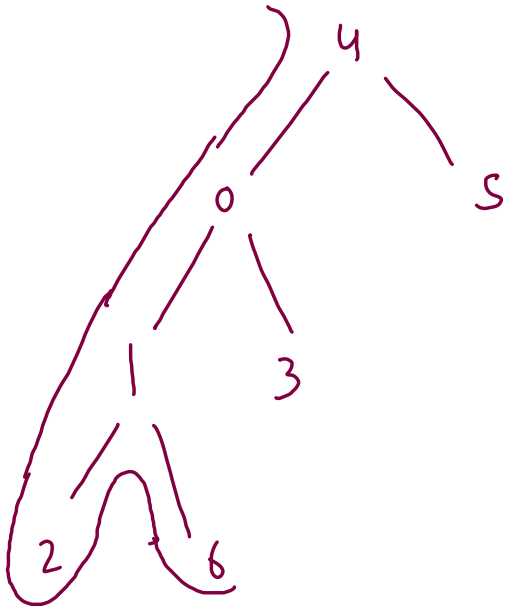


ts: 4 5 0 3 1 2 6

oc: 2 6 1 3 0 5 4

$(oc)^{-1} = ts$

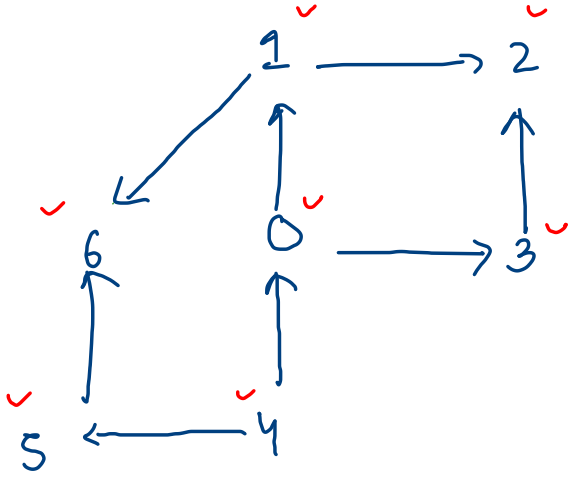




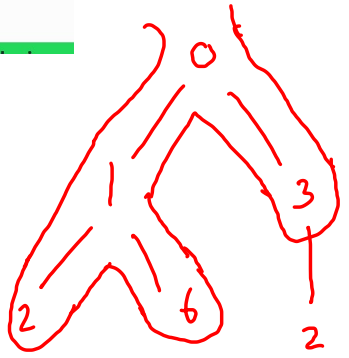
4 0 1 2 6

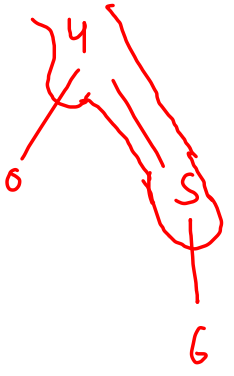
pre X

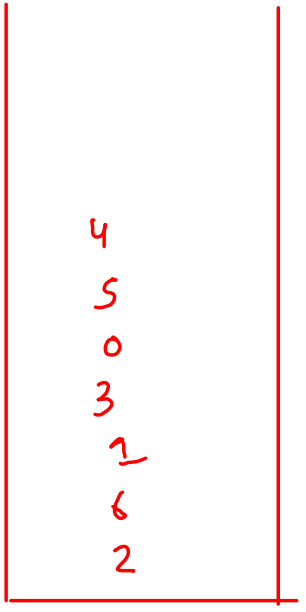
```java
public static void dfs(ArrayList<Edge>[]graph,int src,boolean[]vis,Stack<Integer>st) {

    vis[src] = true;

    for(Edge ne : graph[src]) {
        if(vis[ne.nbr] == false) {
            dfs(graph,ne.nbr,vis,st);
        }
    }

    st.push(src);
}

public static void topological_sort(ArrayList<Edge>[]graph) {
    boolean[]vis = new boolean[graph.length];
    Stack<Integer>st = new Stack<>();

    for(int i=0; i < graph.length;i++) {
        if(vis[i] == false) {
            dfs(graph,i,vis,st);
        }
    }

    //stack has ts

    while(st.size() > 0) {
        System.out.println(st.pop());
    }
}
```

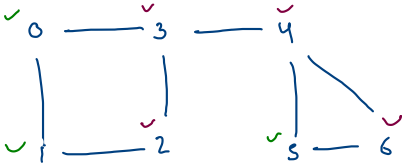


4 5 0 3 1 6 2









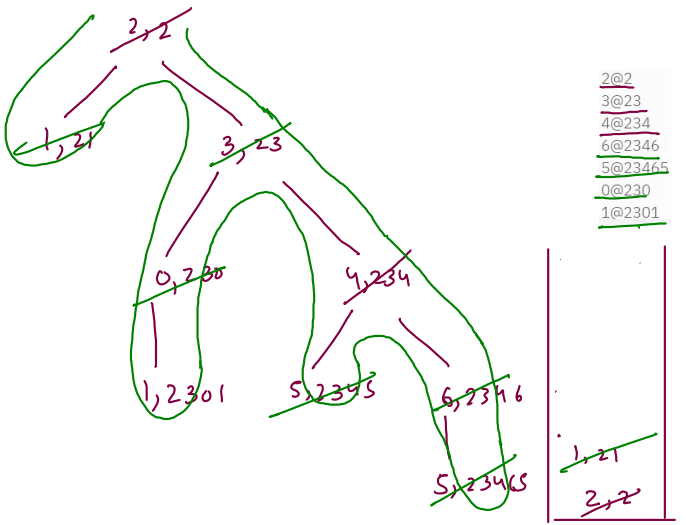
BFS (queue)

DFS (stack)

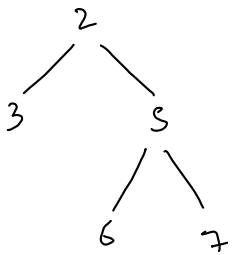
```
public static void iterative_dfs(ArrayList<Edge>[]graph,int src) {
    boolean[]vis = new boolean[graph.length];
    Stack<Pair>st = new Stack<>();

    st.push(new Pair(src,src+""));

    while(st.size() > 0) {
        //remove
        Pair top = st.pop();

        //mark*
        if(vis[top.vtx] == true) {
            continue;
        }

        vis[top.vtx] = true;

        //work
        System.out.println(top.vtx + "@" + top.psf);

        //add unvisited nbr
        for(Edge ne : graph[top.vtx]) {
            int nbr = ne.nbr;
            if(vis[nbr] == false) {
                st.push(new Pair(nbr,top.psf + nbr));
            }
        }
    }
}
```

iterative DFS

src = 2



2@2
3@23
4@234
6@2346
5@23465
0@230
1@2301



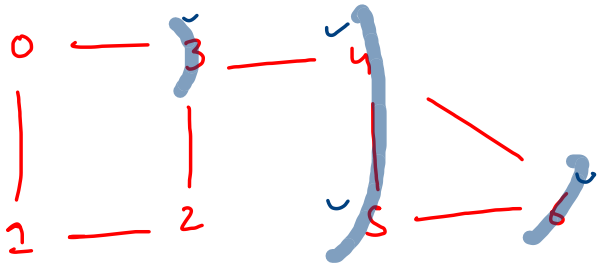
regular post:

LRN

reverse pre

NRL

Spread of Injection




```
while(q.size() > 0) {
    int count = q.size();

    if(lev > time) {
        break;
    }

    while(count-- > 0) {
        //remove
        int rem = q.remove();

        //mark*
        if(vis[rem] == true) {
            continue;
        }
        vis[rem] = true;

        //work
        ans += 1;

        //add nbr
        for(Edge ne : graph[rem]) {
            if(vis[ne.nbr] == false) {
                q.add(ne.nbr);
            }
        }
    }

    lev++;
}
```

inj = 6

t = 3

ans = 1 + 1 + 1 + 1

count : 2

lev = ~~1 1 1~~ 4

~~6~~	~~5~~	~~4~~	~~4~~	~~3~~	0	2

1 2 3

4