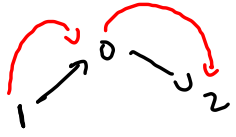
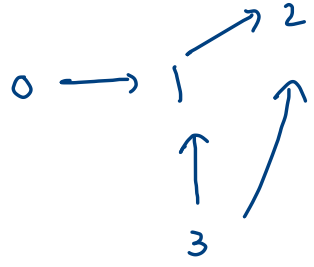
Order Of Compilation

DAG

topological sort : In a directed acyclic graph, a permutation of vertices such that ∀ u→v, u should come before v.





topological sort

order of compilation

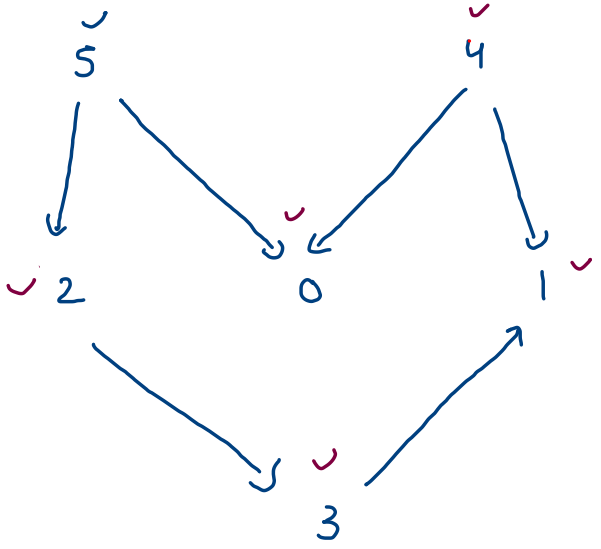
topological sort.

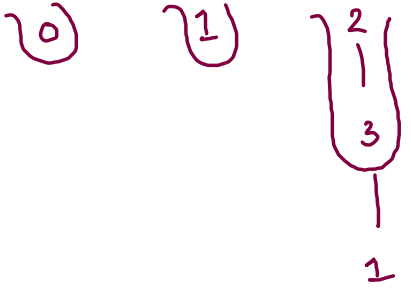
u→v, u is dependent on v.

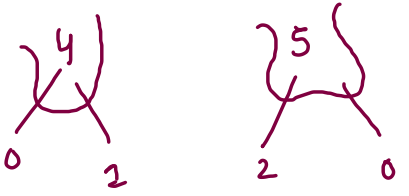
order of compilation = reverse (topological sort)

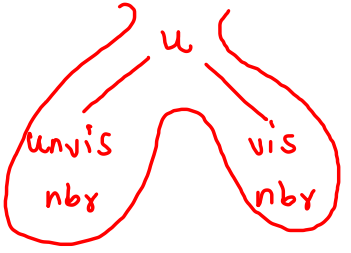
(the file which is compiled first, should come first)

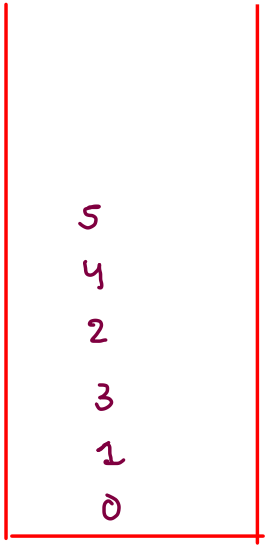
algo : (i) push vertices in stack in postorder.











ts: 5 4 2 3 1 0

ooc: 0 1 3 2 4 5

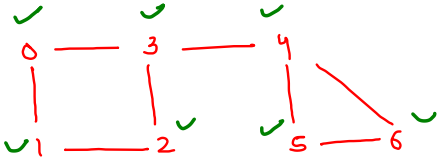
Iterative Depth First Traversal

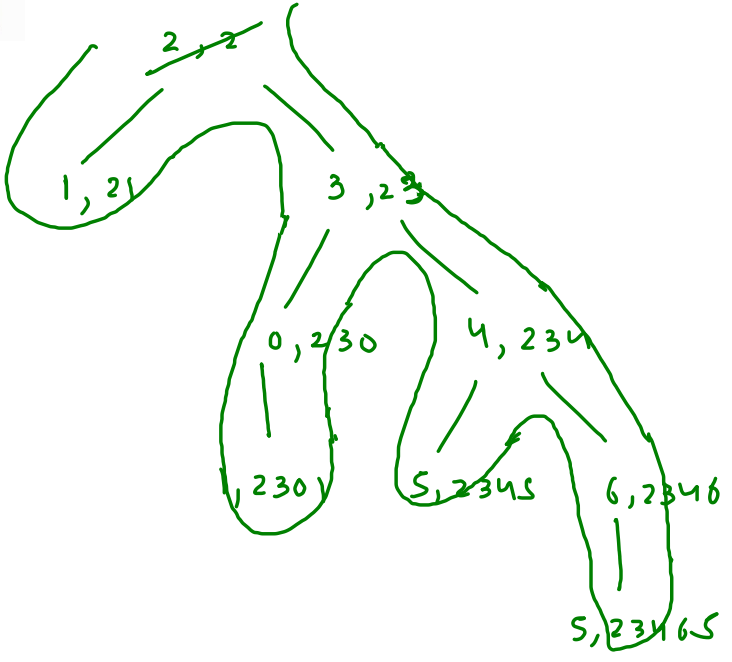
```
while(st.size() > 0) {
    //remove
    Pair rem = st.pop();

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

    //work
    System.out.println(rem.v + "@" + rem.psf);

    //add nbr*
    for(Edge edge : graph[rem.v]) {
        int nbr = edge.nbr;

        if(vis[nbr] == false) {
            st.push(new Pair(nbr,rem.psf + nbr));
        }
    }
}
```

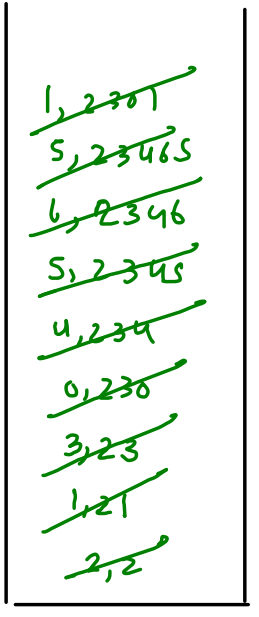
order

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

BFS with stack







994. Rotting Oranges

2 -> rotten orange

1 -> fresh orange

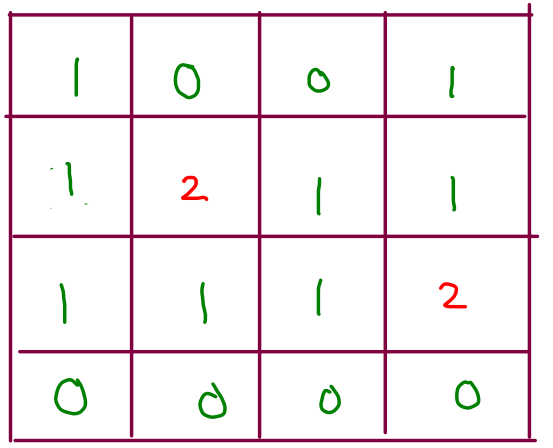
0 -> nothing

You are given an `m x n` `grid` where each cell can have one of three values:

- `0` representing an empty cell,
- `1` representing a fresh orange, or
- `2` representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return *the minimum number of minutes that must elapse until no cell has a fresh orange*. If *this is impossible, return* `-1`.



multiple src bfs

t = 0

	0	1	2	3
0	1	0	0	1
1	1	2	1	1
2	1	1	1	2
3	0	0	0	0

normal BFS

jo = 8 7 6 5 4 3 2 1
0

Pair {
int i j int j j
int t j
}

remove
mark*
work
add nbr*

1,1,0 | 2,3,0 | 1,0,1 | 2,1,1 | 1,2,1 | 1,3,1 | 2,2,1 | 0,0,2 | 2,0,2 | 2,0,2 | 2,2,2 | 1,3,2 | 2,2,2 | 0,3,2

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

    if(grid[rem.r][rem.c] == -1) {
        continue;
    }
    else if(grid[rem.r][rem.c] == 1) {
        fo--;
    }
    grid[rem.r][rem.c] = -1;

    if(fo == 0) {
        return rem.t;
    }

    for(int i=0; i < 4;i++) {
        int nr = rem.r + dir[i][0];
        int nc = rem.c + dir[i][1];

        if(nr >= 0 && nr < grid.length && nc >= 0 && nc < grid[0].length && grid[nr][nc] == 1) {
            q.add(new Pair(nr,nc,rem.t+1));
        }
    }

}
```

	0	1	2	3
0	~~1~~	0	~~2~~	0
1	~~2~~	~~2~~	~~1~~	~~1~~
2	0	~~1~~	~~1~~	0
3	0	0	~~1~~	~~2~~

fo = ~~7~~ ~~6~~ ~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ 0

0,2,0 | 1,1,0 | 3,2,0 | 1,2,1 | 1,0,1 | 2,1,1 | 1,2,1 | 3,2,1 | 2,2,2 | 1,3,2 | 0,0,2 | 2,2,2 | 2,2,2

Fractional Knapsack - Official

↳ greedy

	0	1	2	3	4
wt	10	7	3	5	2
value	30	56	45	50	15
value / wt	3	8	15	10	7.5
	⑤	⑨	①	②	③

cap: 7

(i) items are breakable

(ii) limited amount of each items.

rc = ~~7~~ ~~4~~ 0

mp = 45 + 40

(3·45) (5-50) → $\frac{4}{5}$

Fractional Knapsack

```
int rc = W;
double mp = 0.0;

for(int i = n-1; i >= 0;i--) {
    Pair p = items[i];

    if(p.wt <= rc) {
        rc -= p.wt;
        mp += p.val;
    }
    else {
        mp += p.r * rc;
        rc = 0;
    }
}
```

```
N = 3, W = 50
values[] = {60,100,120}
weight[] = {10,20,30}
```

rc = ~~50~~ ~~40~~ ~~20~~ 0

mp = 0 + 60 + 100 + 80

```
static class Pair implements Comparable<Pair>{
    int wt;
    int val;
    double r;

    Pair(int wt,int val) {
        this.wt = wt;
        this.val = val;
        this.r = (val * 1.0) / wt;
    }

    public int compareTo(Pair o) {
        if(this.r < o.r) {
            return -1;
        }
        else if(this.r > o.r) {
            return 1;
        }
        else {
            return 0;
        }
    }
}
```

	0	1	2
wt	10	20	30
value	60	100	120
val/wt	6	5	4