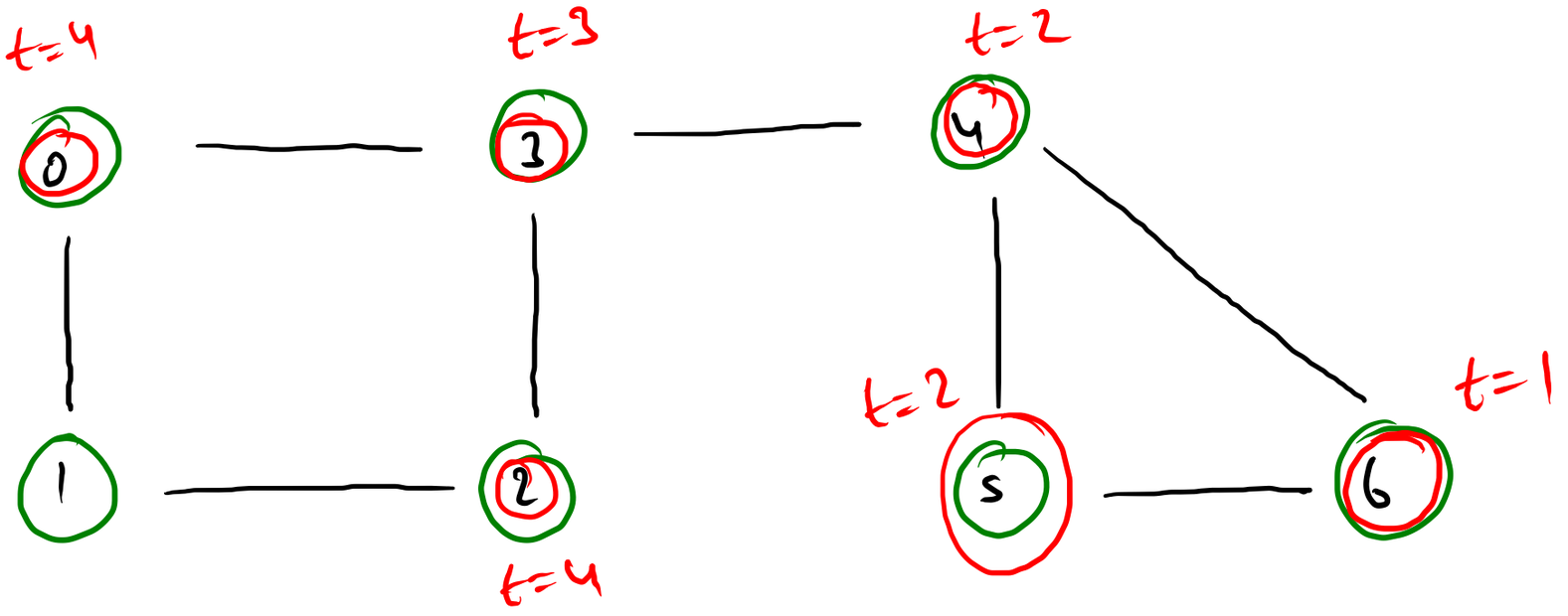
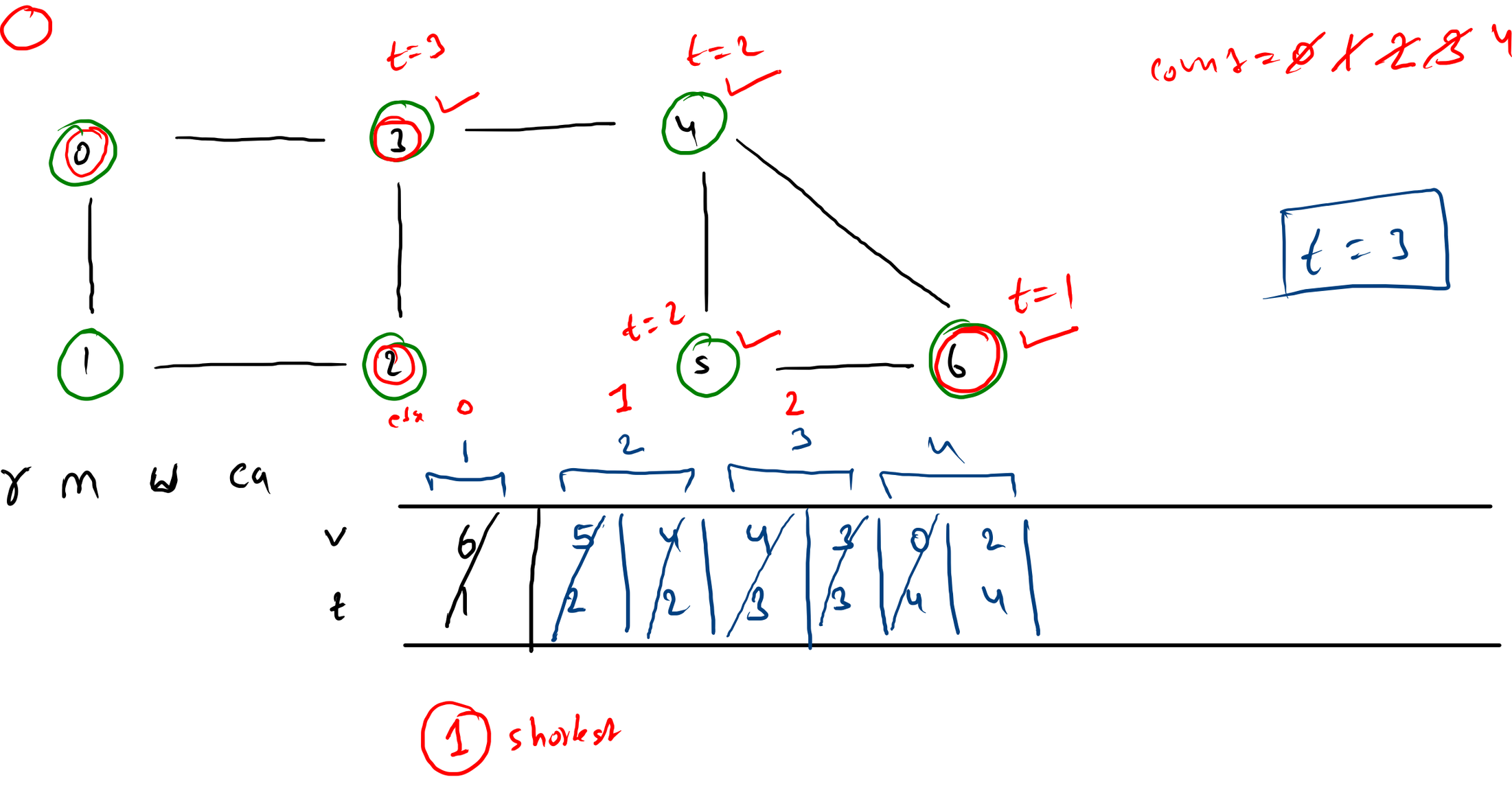


7
8
0 1 10
1 2 10
2 3 10
0 3 10
3 4 10
4 5 10
5 6 10
4 6 10
6
3

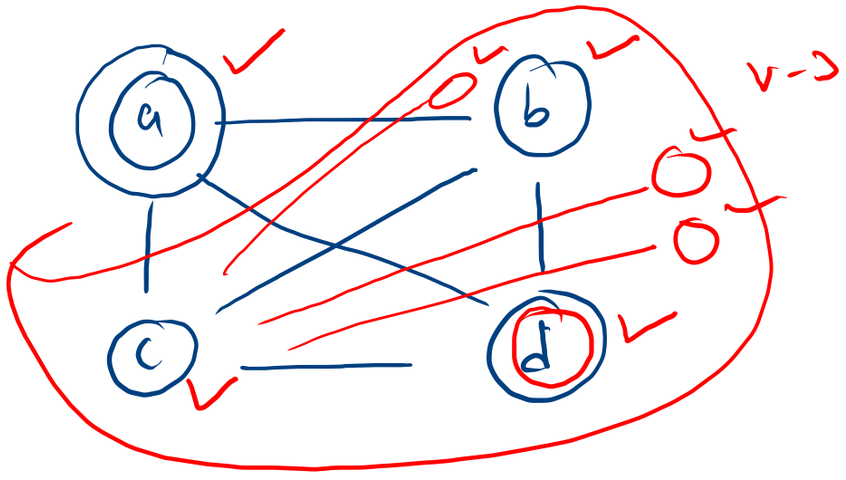
src = 6

t=3 → 4



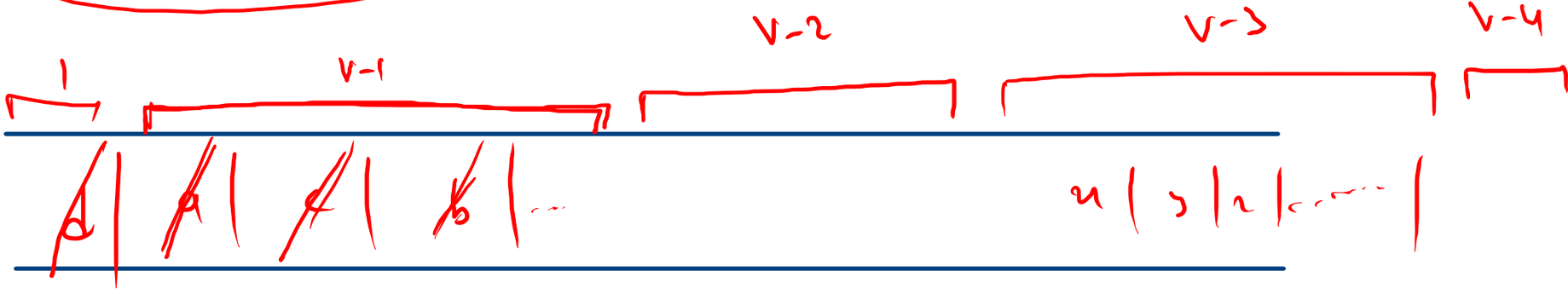
t = 3

1 shortest



vertex → V

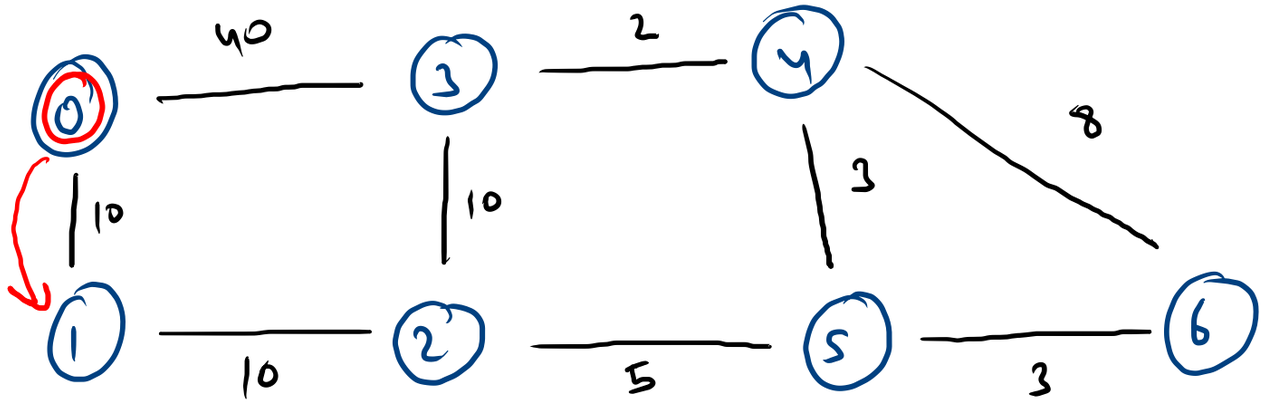
E → V(V-1)



$O(V^2)$

time ⇐

Dijkstra




```
7
9
0 1 10
1 2 10
2 3 10
 0 3 40
3 4 2
 4 5 3
 5 6 3
 4 6 8
2 5 5
 0
```

0 via "0" @ 0

1 via "01" @ 10

2 via "012" @ 20

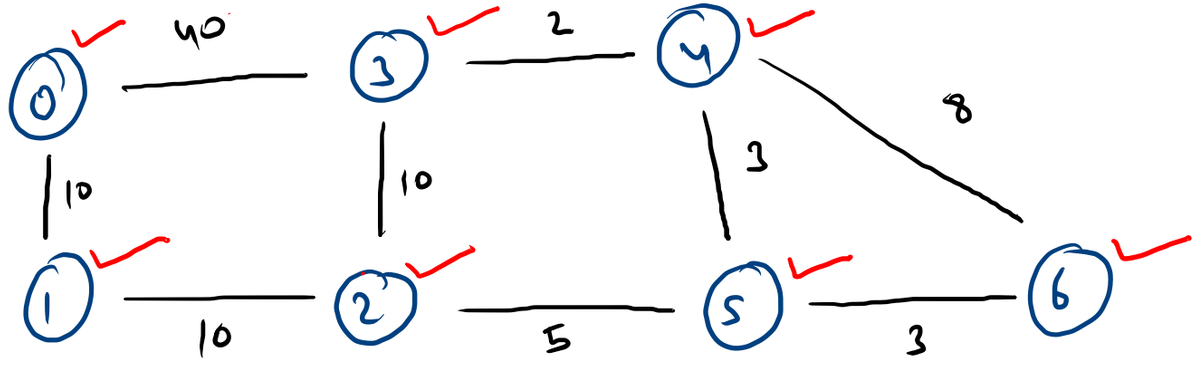
3 via "0123" @ 30

~~4 via "01234" @ 32~~

5 via "0125" @ 25

6 via 01256 @ 38

4 via 01254" @ 28



{src, nbr, wt}

2, 3, 10

2 5, 5

```
public int compareTo(Pair b){
    Pair a = this;
    if(a.wt < b.wt)return -1;
    if(a.wt > b.wt)return +1;
    return 0;
}
```

p → { 6, "0125436", 36 }

```
PriorityQueue<Pair> pq = new PriorityQueue<
pq.add(new Pair(src, src+"", 0));

boolean visited[] = new boolean[vtces];

while(pq.size() > 0){
    //r m w an
    Pair p = pq.remove();
    if(visited[p.v])continue;

    visited[p.v] = true;
    System.out.println(p.v+" via "+p.path+" @ "+p.wt);

    for(Edge e: graph[p.v]){
        if(visited[e.nbr] == false){
            pq.add(new Pair(e.nbr, p.path+e.nbr, p.wt+e.wt));
        }
    }
}
```

```
static class Pair implements Comparable<Pair> {
    int v;
    String path;
    int wt;ed
    Pair(int v, String path, int wt){
        this.v = v;
        this.path = path;
        this.wt = wt;
    }
```

0 "0" 0

1 "01" 10

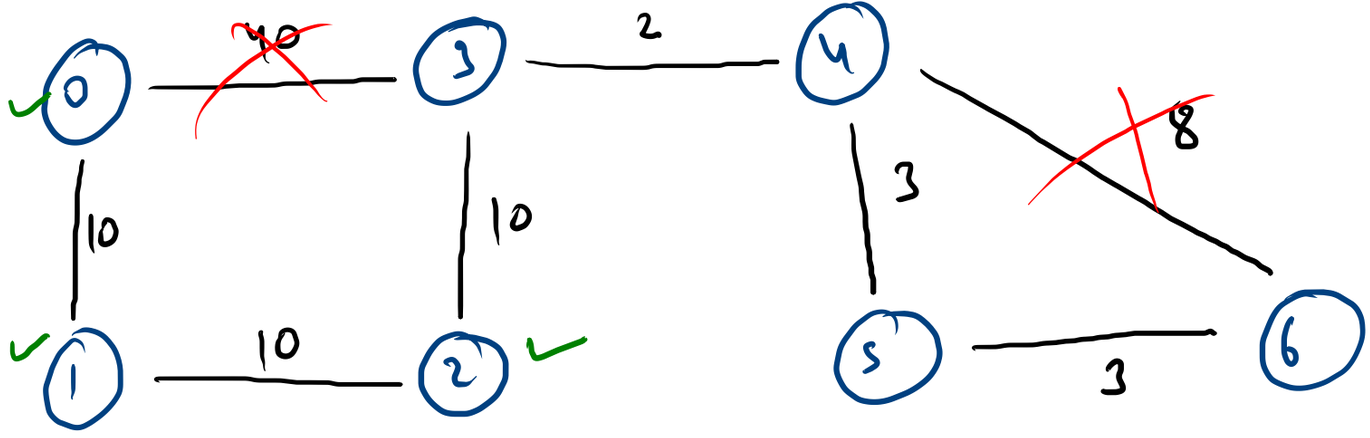
2 "012" 20

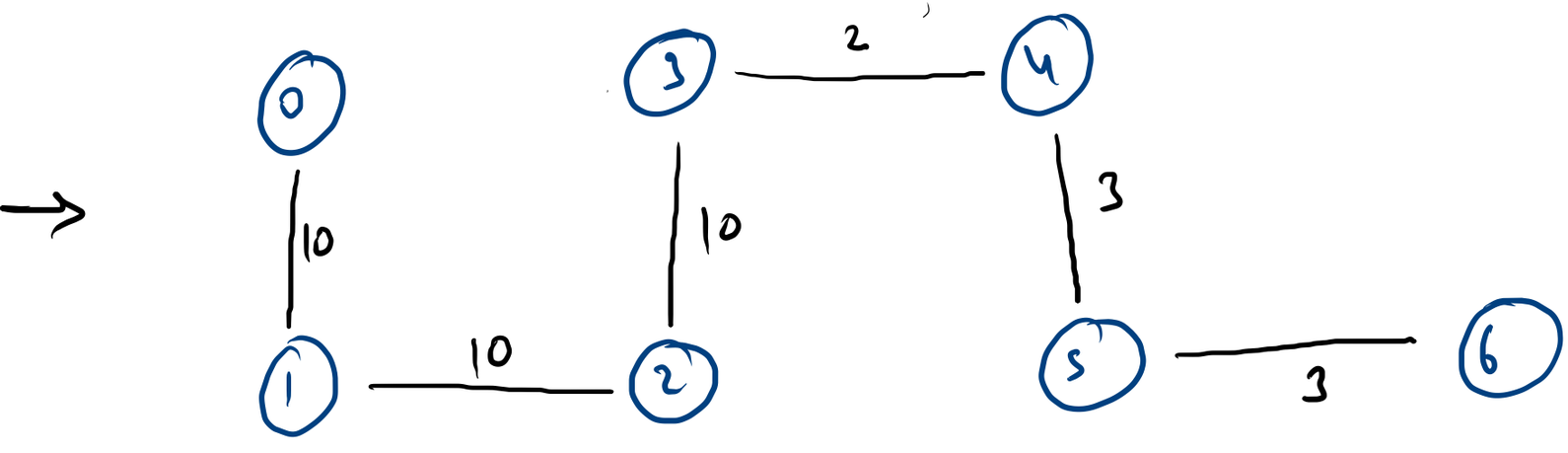
5 "0125" 25

4 "01254" 28

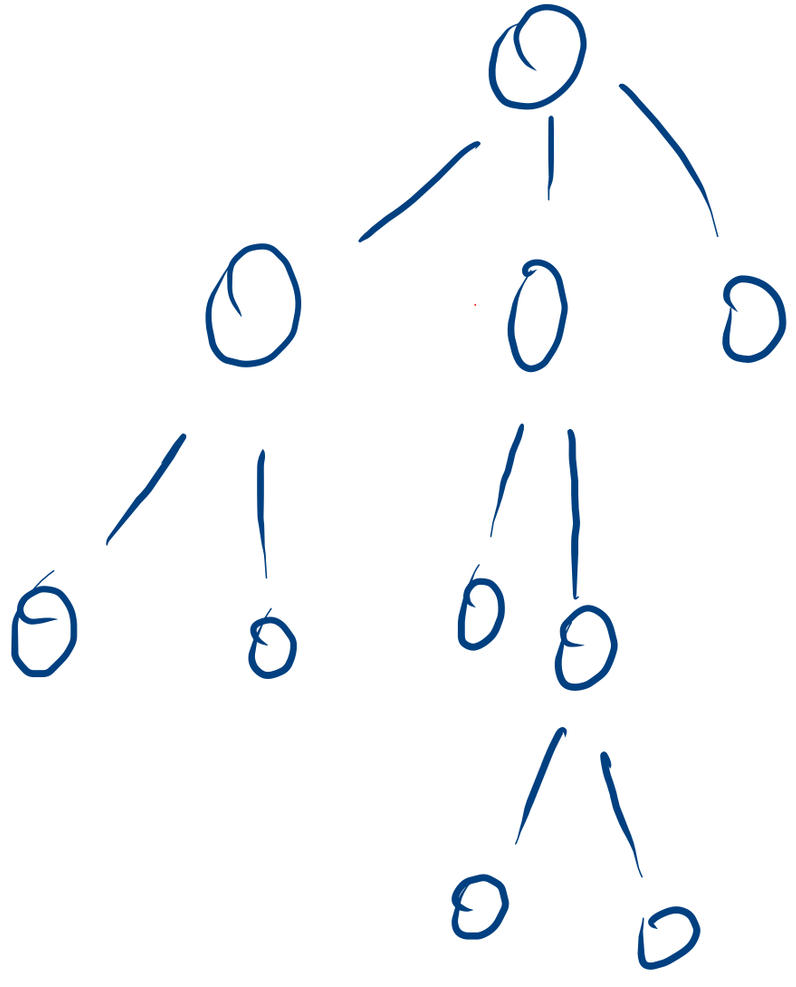
6 "01256", 28

3 0123 , 30





[1-0@10]
[2-1@10]
[3-2@10]
[4-3@2]
[5-4@3]
[6-5@3]



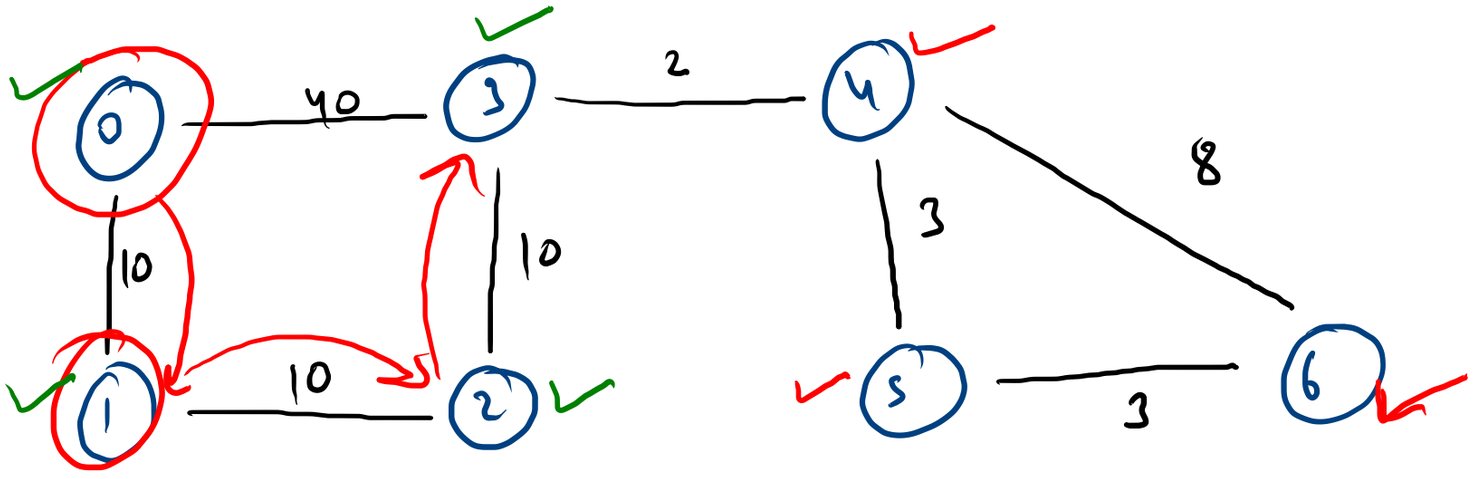
spanning tree

min

max

u v wt

{-1, 0, 0}



v u wt

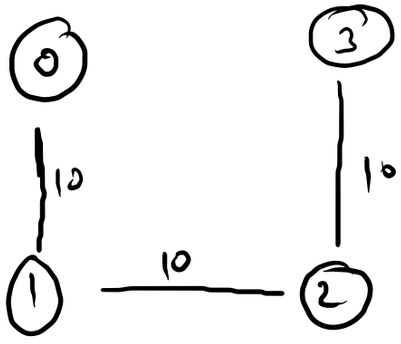
{ 1 2 10 }

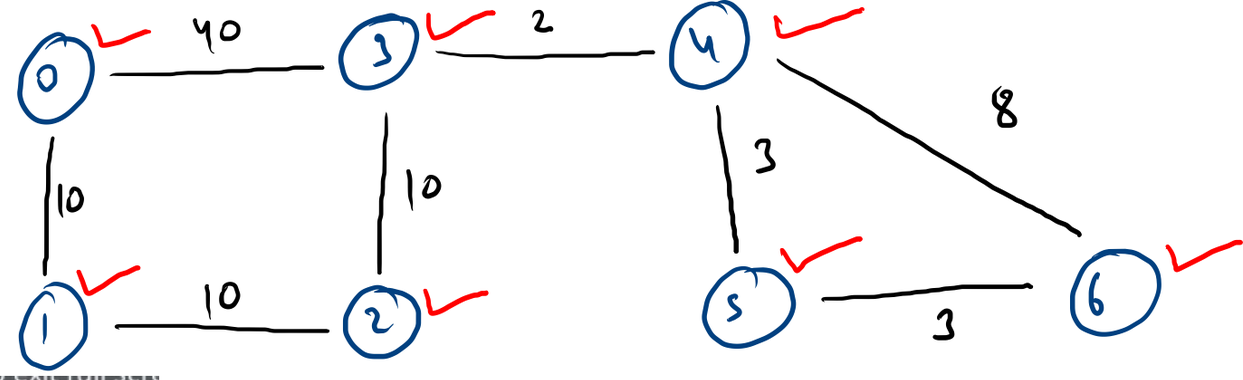
1 → 2

v u wt

{ 2 3 10 }

Prim's



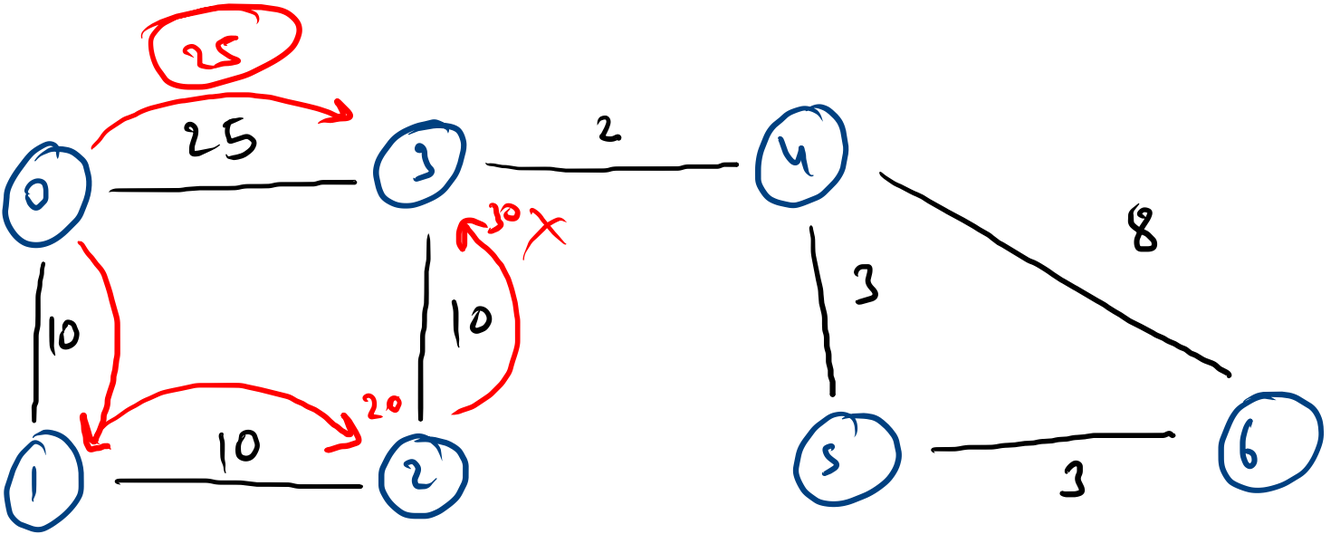



```java
PriorityQueue<Pair> pq = new PriorityQueue<>();
pq.add(new Pair(-1, 0, 0));

boolean visited[] = new boolean[vtces];

while(pq.size() > 0){
    // r m w an
    Pair p = pq.remove();

    if(visited[p.u])continue;
    visited[p.u] = true;

    if(p.v != -1){
        System.out.println("["+p.u+"-"+p.v+"@"+p.wt+"]");
    }

    for(Edge e: graph[p.u]){
        pq.add(new Pair(p.u, e.nbr, e.wt));
    }
}
```

v u wt

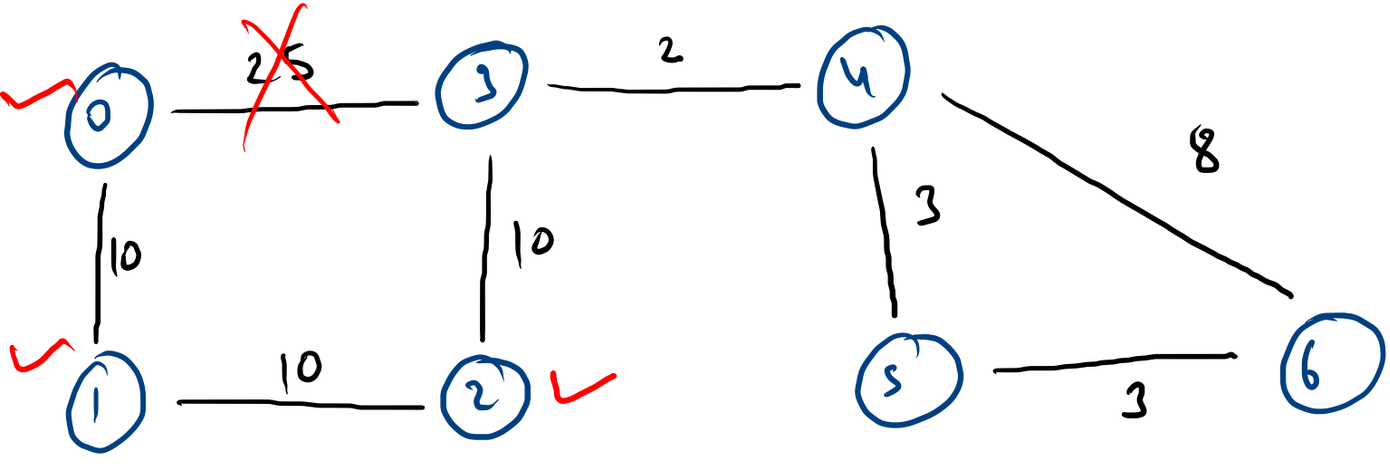
{0, 3, 40}

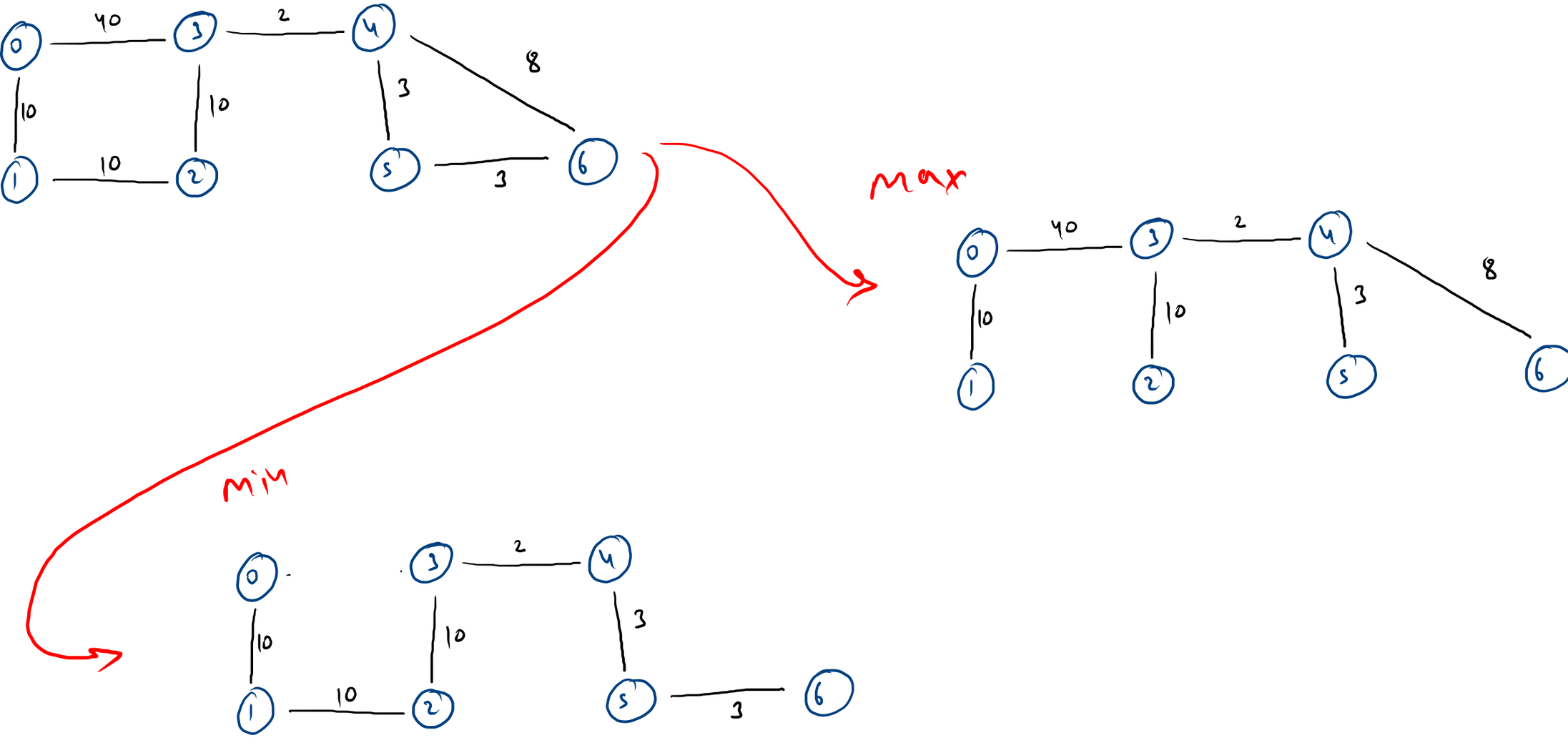
1-0 @ 10
2-1 @ 10
2-3 @ 10
4-3 @ 2
5-4 @ 3
6-5 @ 3

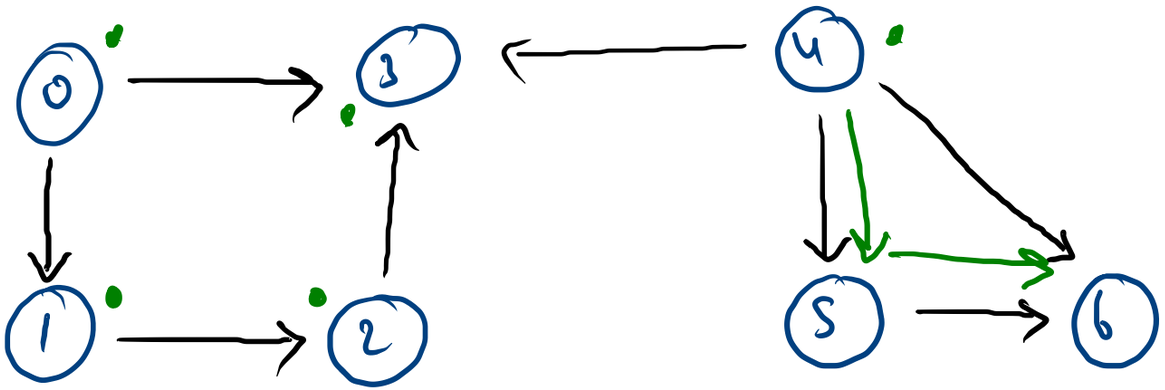


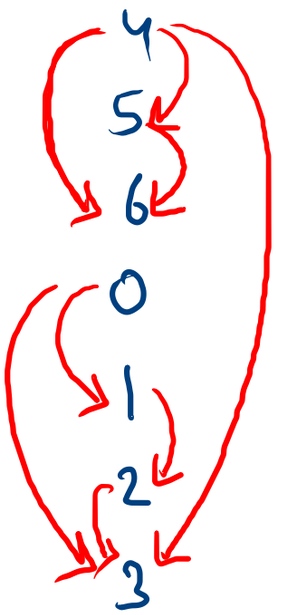
dykstra

0 → 3 25



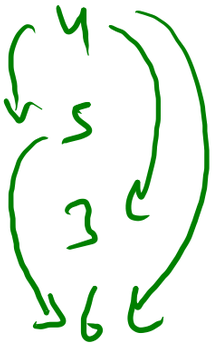

0
40
3
2
4
8
10
10
3
1
10
2
5
3
6
max
0
40
3
2
4
8
10
10
3
1
2
5
6
min
0
3
2
4
10
10
3
1
10
2
5
3
6

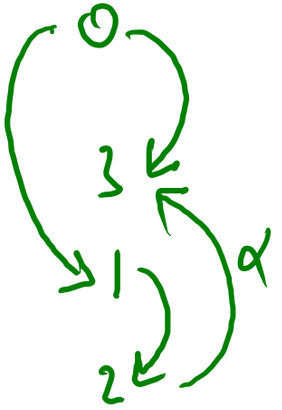




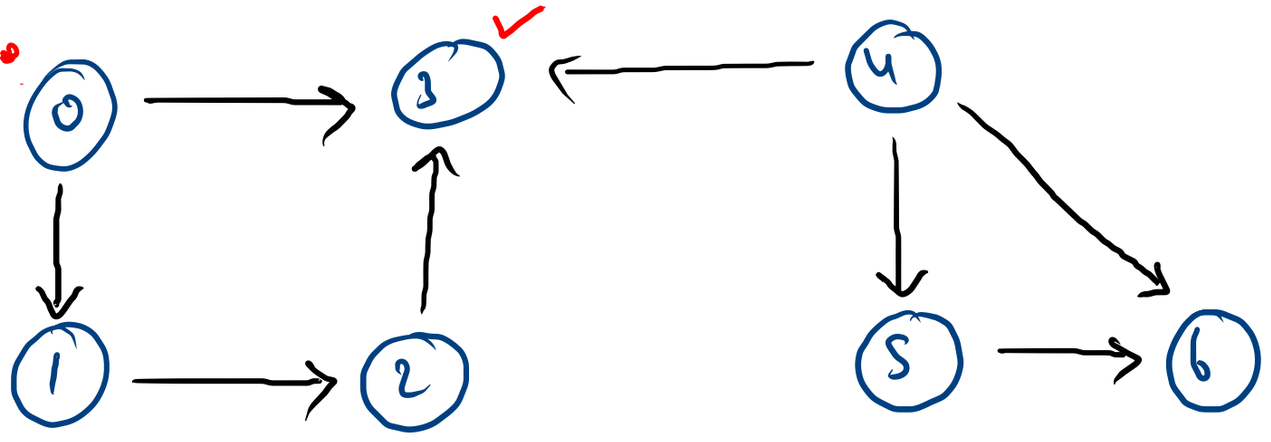
4 5 6 0 1 2 3

u → v



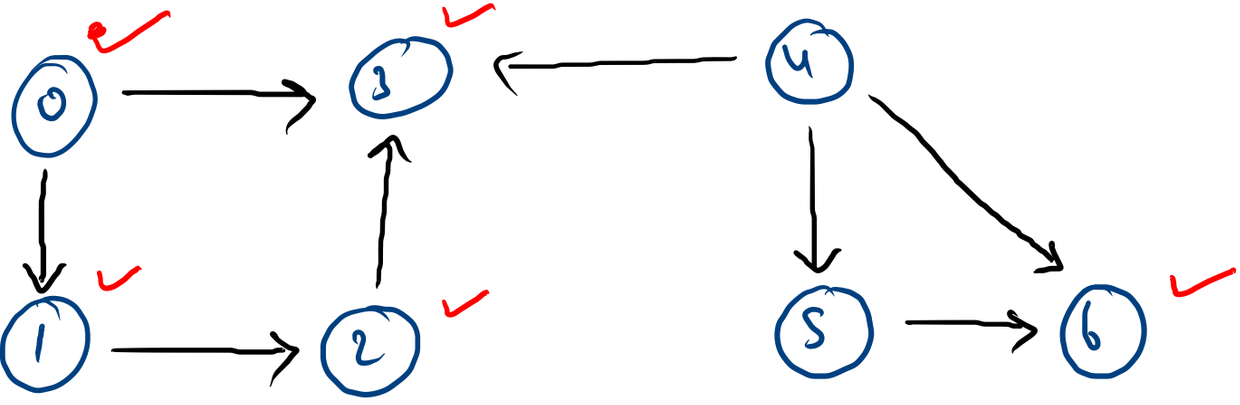


4 5 6 0 1 2 3



{ 4, ~~3~~, 5, 6 }

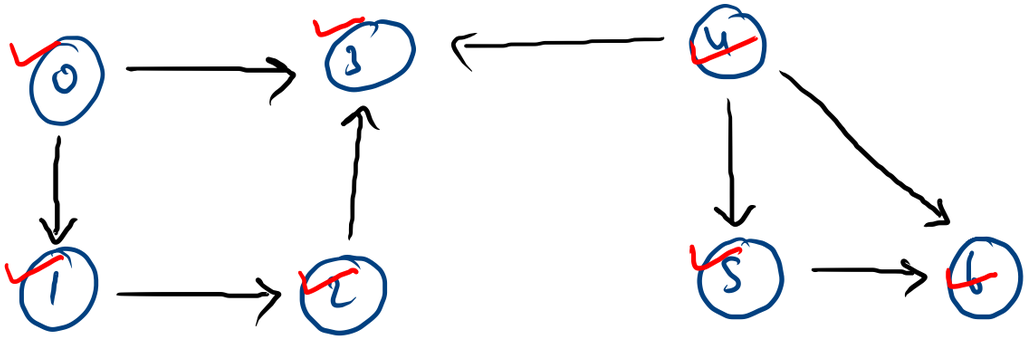
4
5
6
0
1
2
3

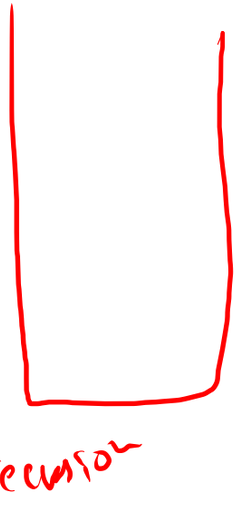


4
5
6
0
1
2
3

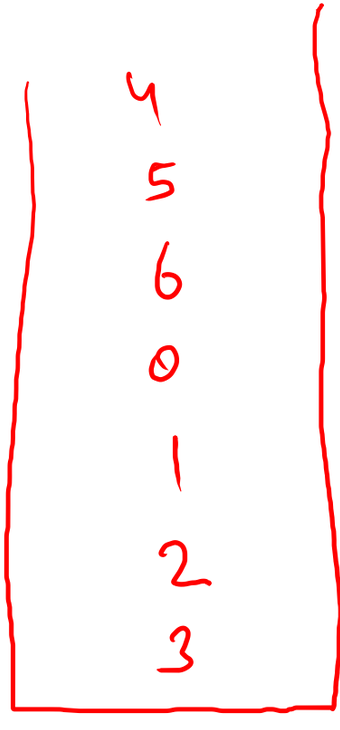
i = 0 1 2 3 4 5 6

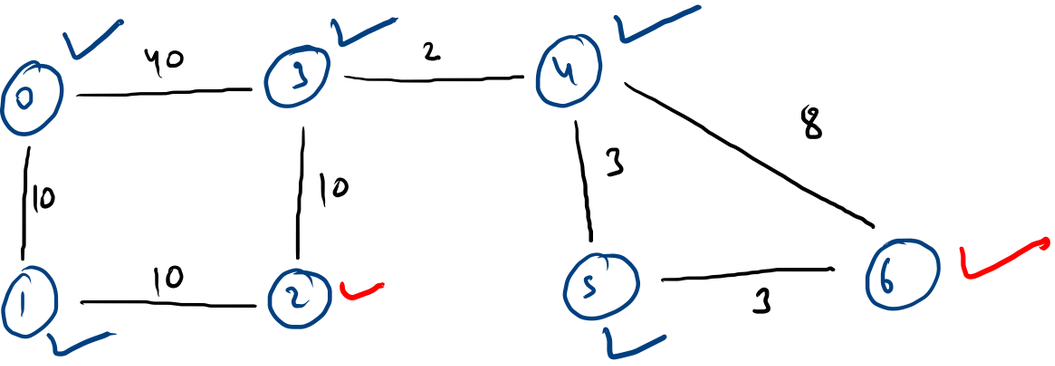
```
boolean visited[] = new boolean[vtces];
Stack<Integer> st = new Stack<>();
for(int i=0;i<vtces;i++){
    if(visited[i] == false){
        dfs(graph, i, st, visited);
    }
}

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



atic void dfs(ArrayList<Edge> graph[], int sr

    visited[src] = true;

    for(Edge e: graph[src]){
        if(visited[e.nbr] == false){
            dfs(graph, e.nbr, st, visited);
        }
    }
    st.push(src);
```



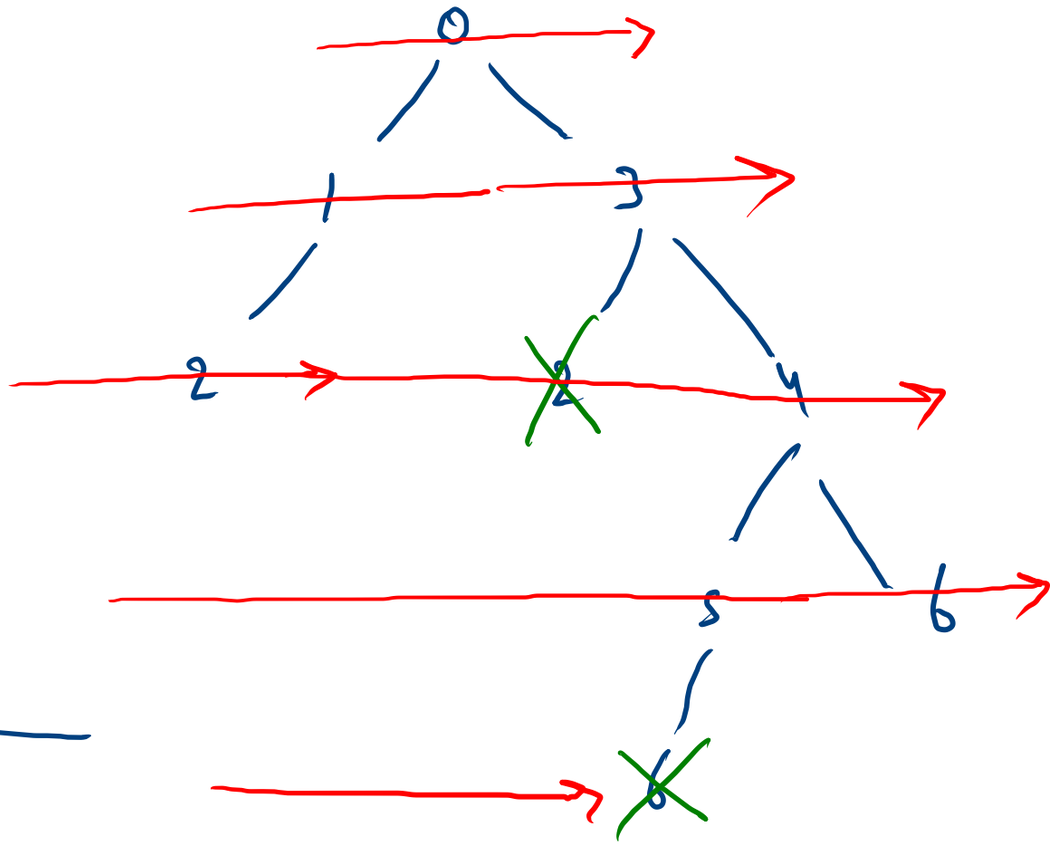


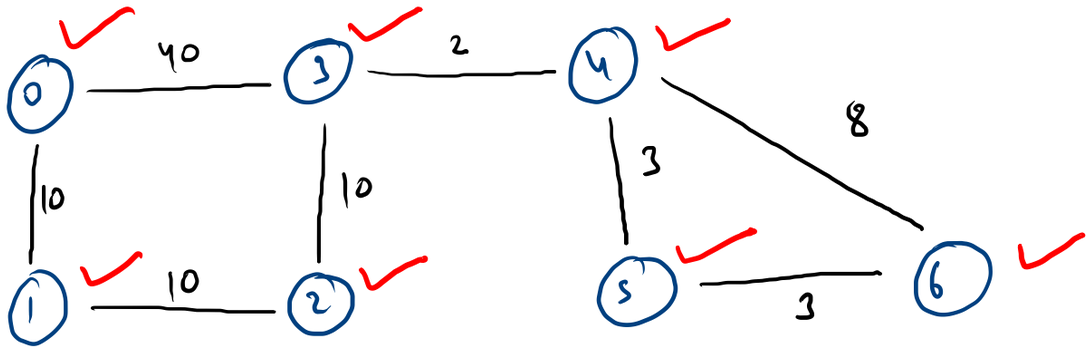
recursion





γ m ω αh

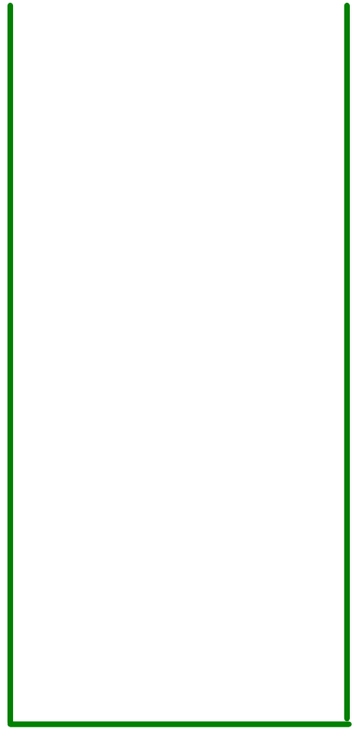




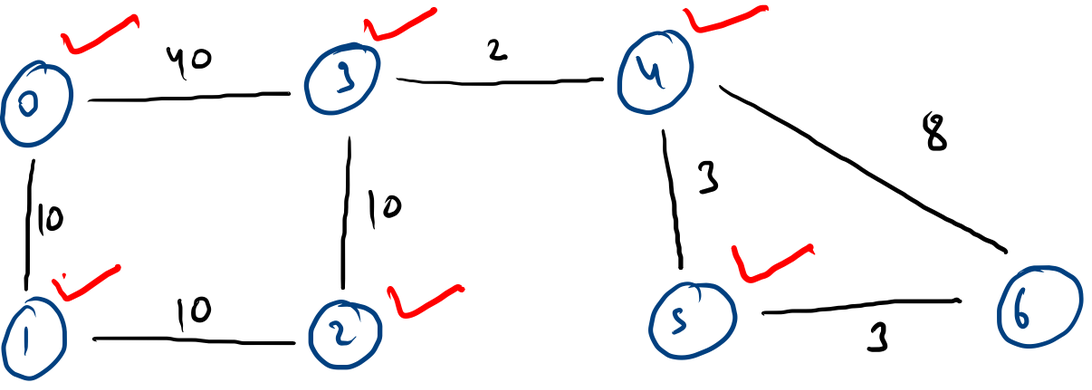
γ m ω ah

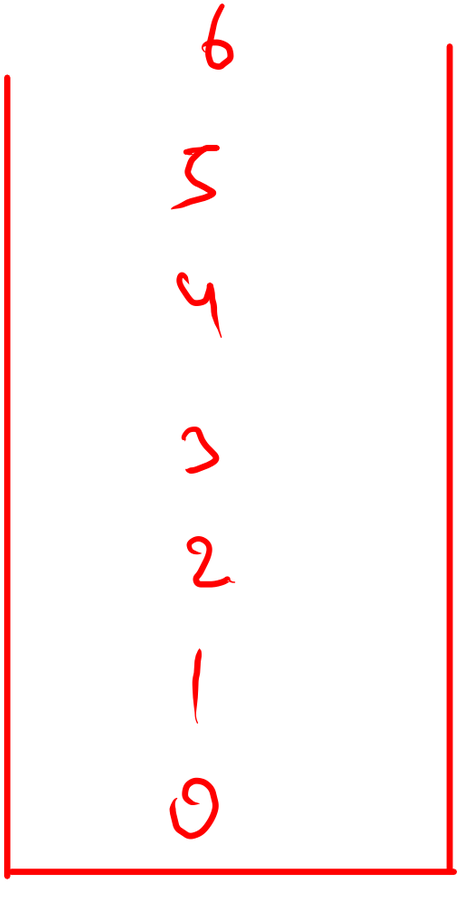
Iterative

Left right

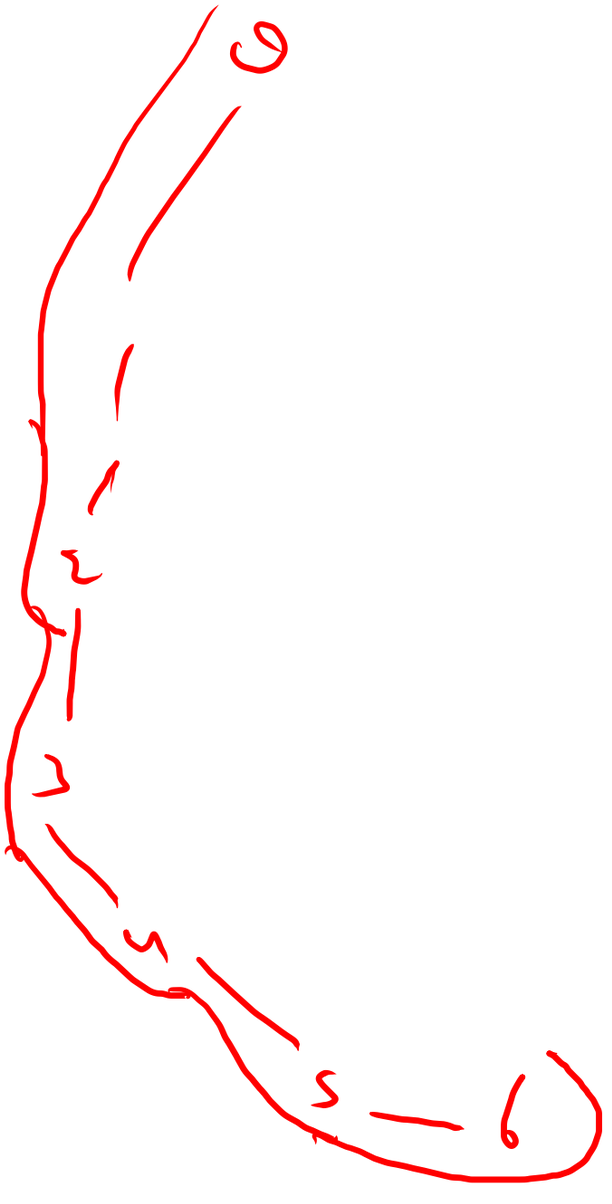


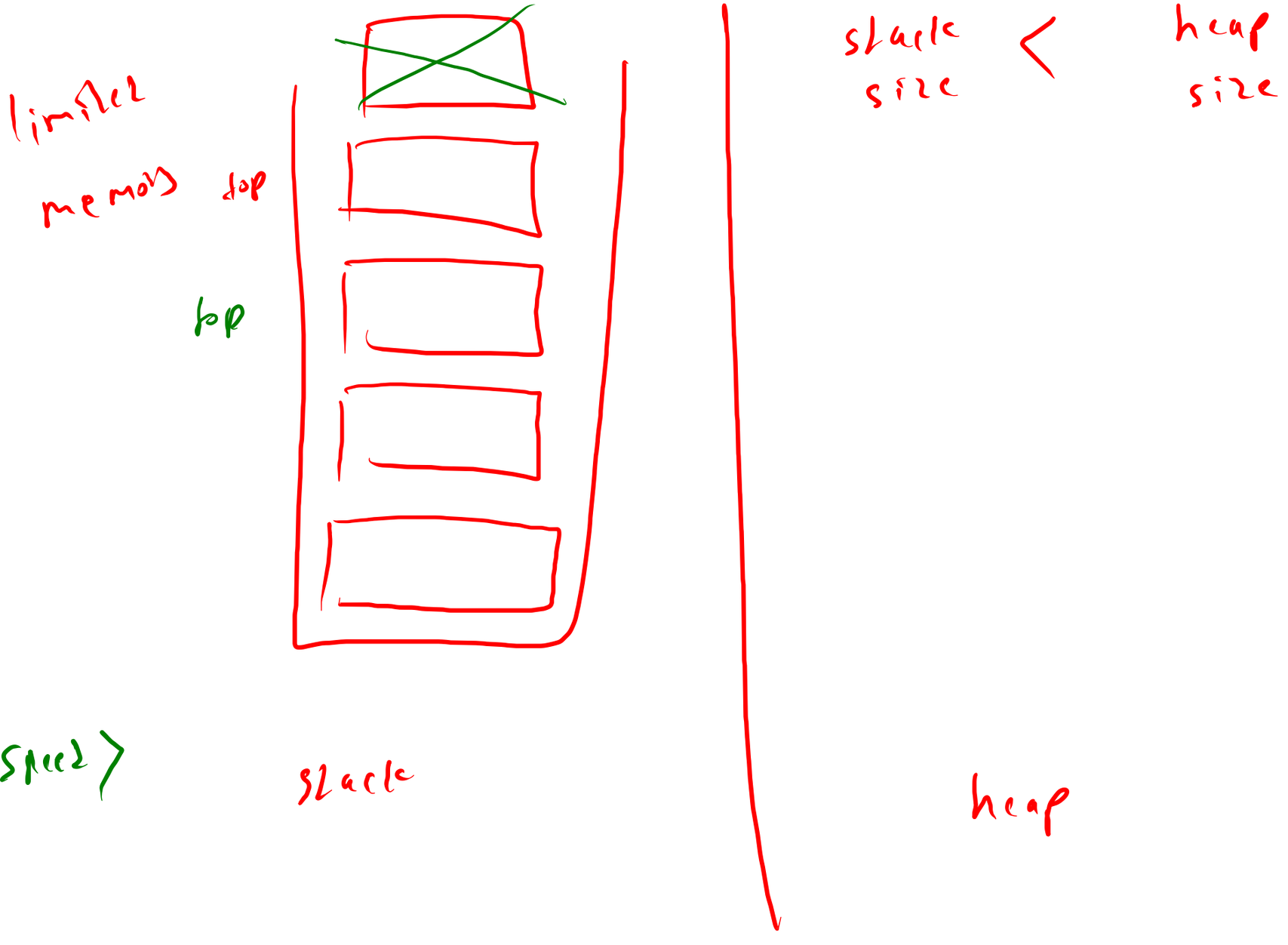
← Iterative stack

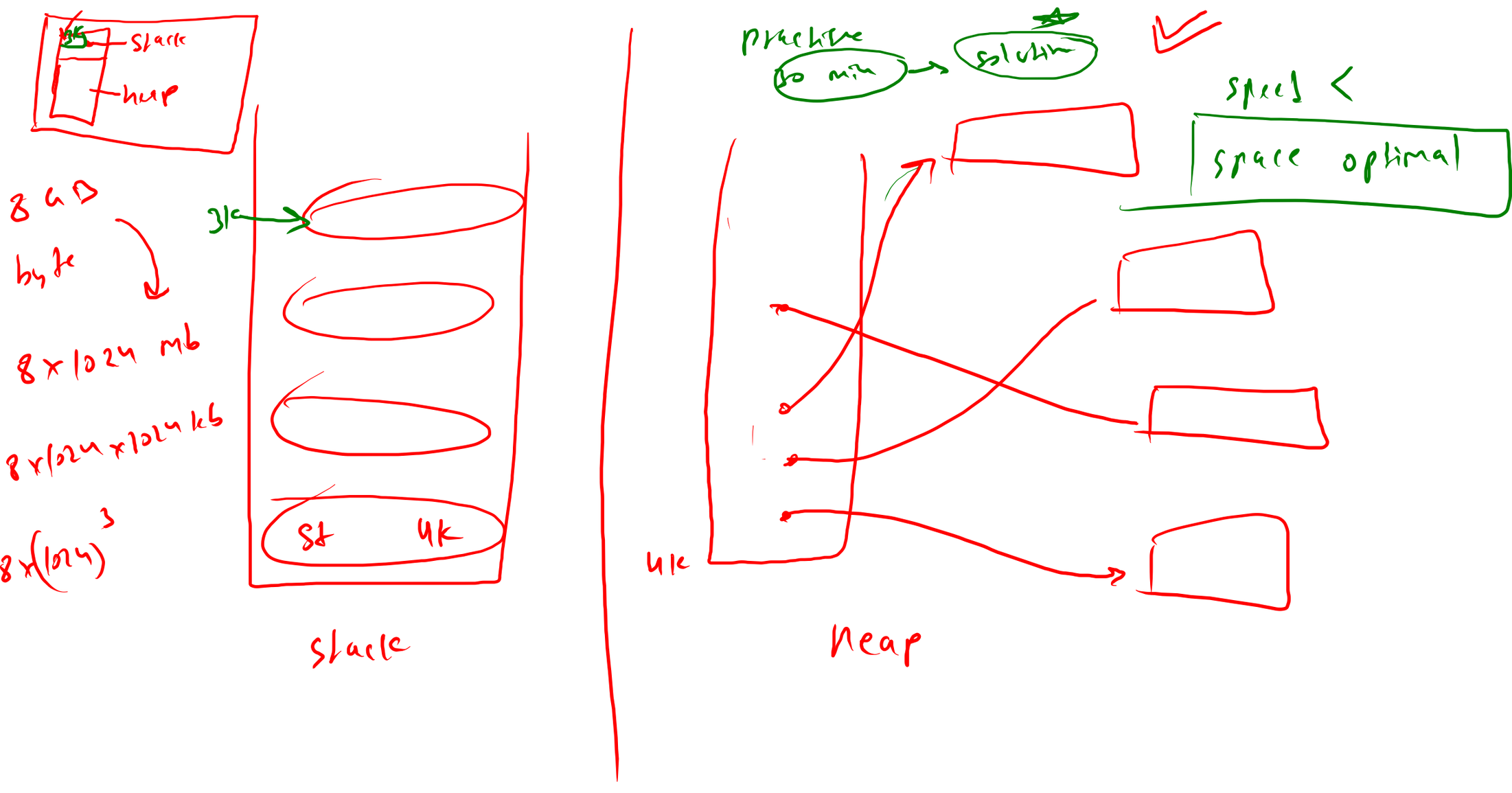




← recursive stack




limited
memory
top
top
space
stack
stack
size
<
heap
size
heap


stack
heap
8 GB
byte
8 x 1024 mb
8 x 1024 x 1024 kb
8 x (1024)^3
3k
8k
4k
stack
practice
10 min
solution
speed <
space optimal
4k
heap