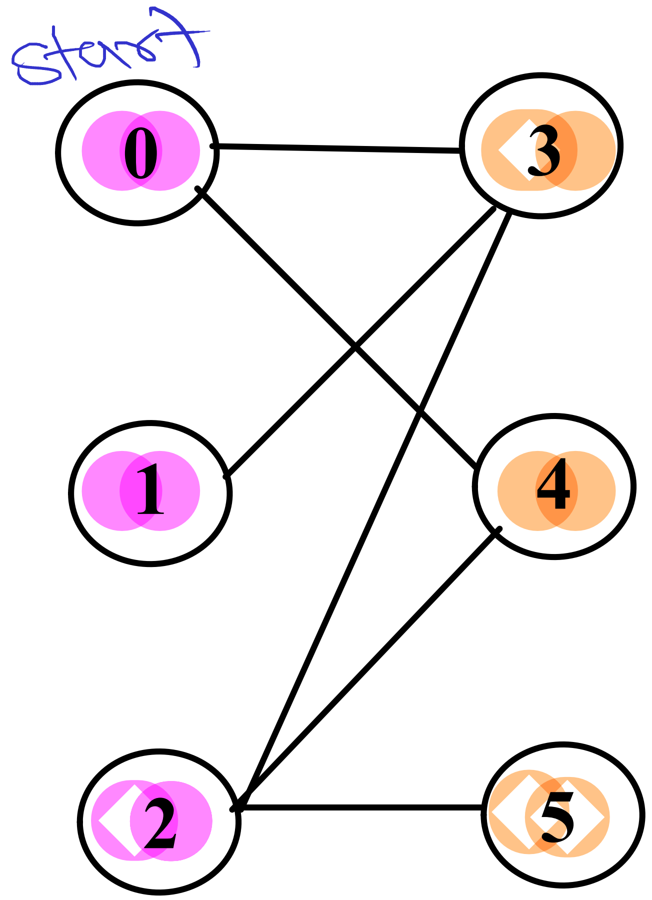
Check Bipartite-ness

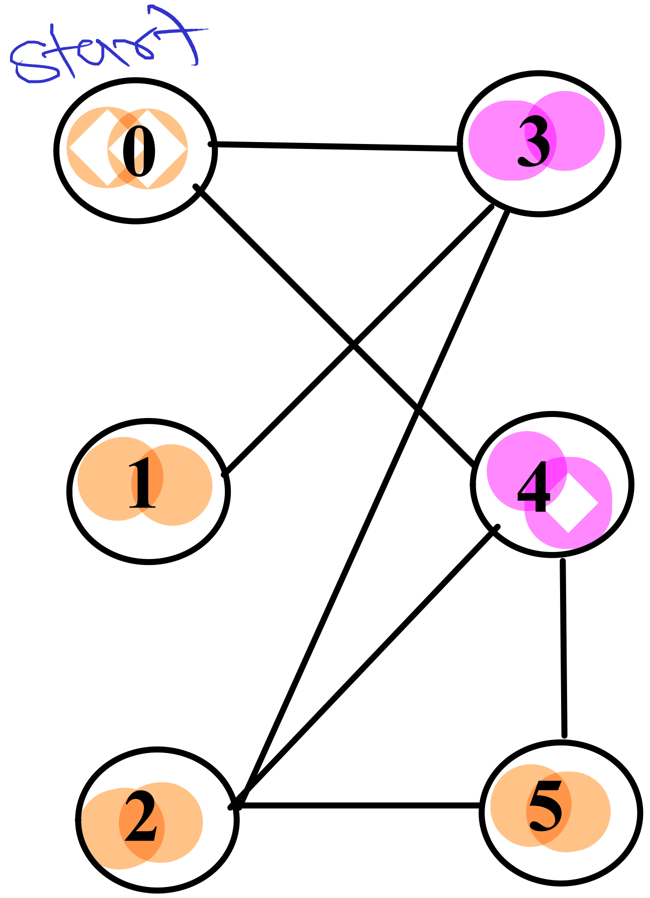
//1. keep colors of all vertices in an array. Initially vertices have no color.
//2. push start on queue & mark it. Assign it color1.
//3. pop the vertex.
//4. push all its non-marked neighbors on the queue, mark them.
//5. For each such vertex if no color is assigned yet, assign opposite color of current vertex (c1-c2, c2-c1).
//6. If vertex is already colored with same of current vertex, graph is not bipartite (return).
//7. repeat steps 3-6 until queue is empty.



color1 = -1, color = 1, no color = 0

~~0~~	~~3~~	~~4~~	~~1~~	~~2~~	~~5~~

0, 3, 4, 1, 2, 5



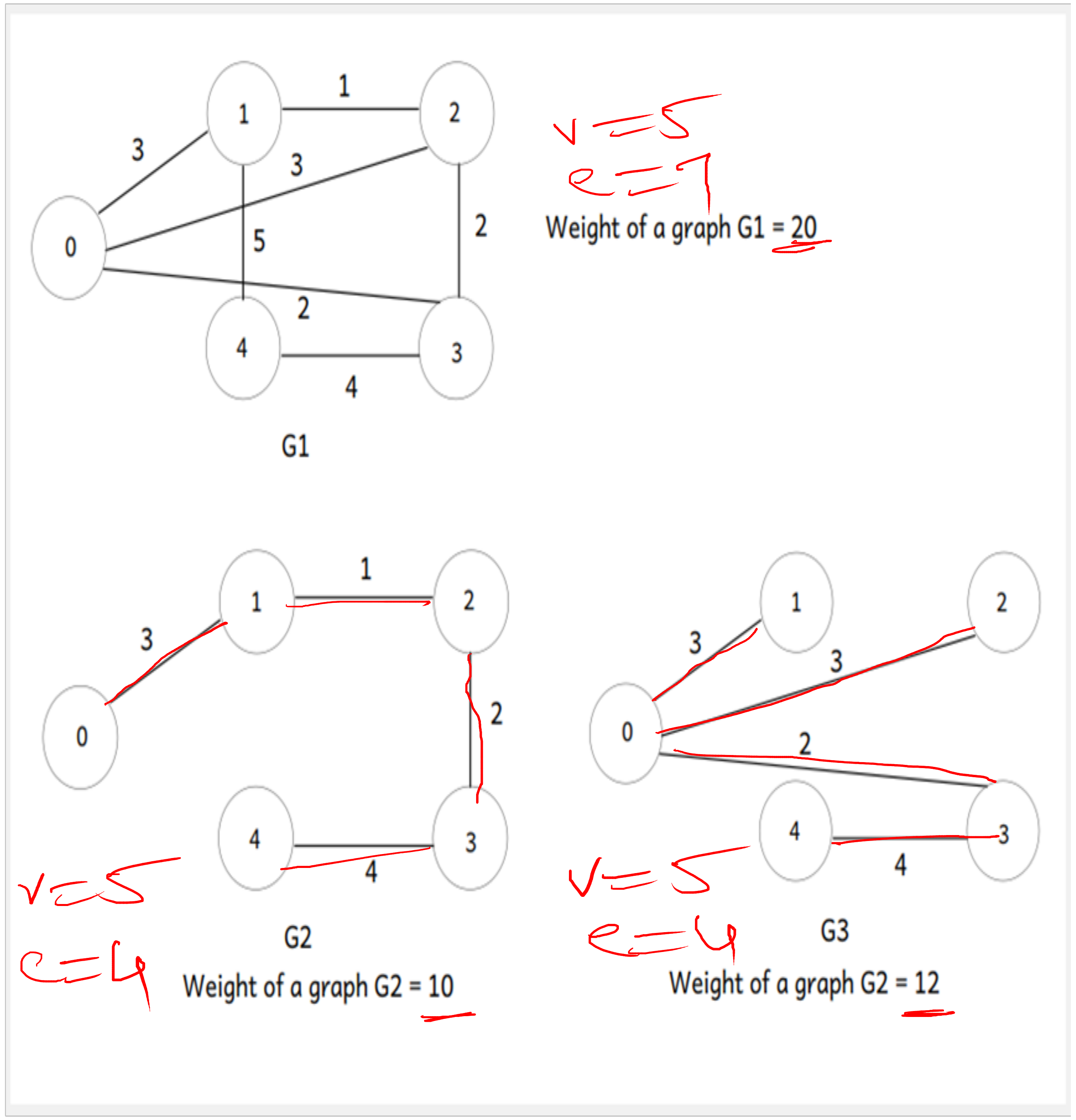
~~0~~	~~3~~	~~4~~	~~1~~	~~2~~	5

0, 3, 4, 1

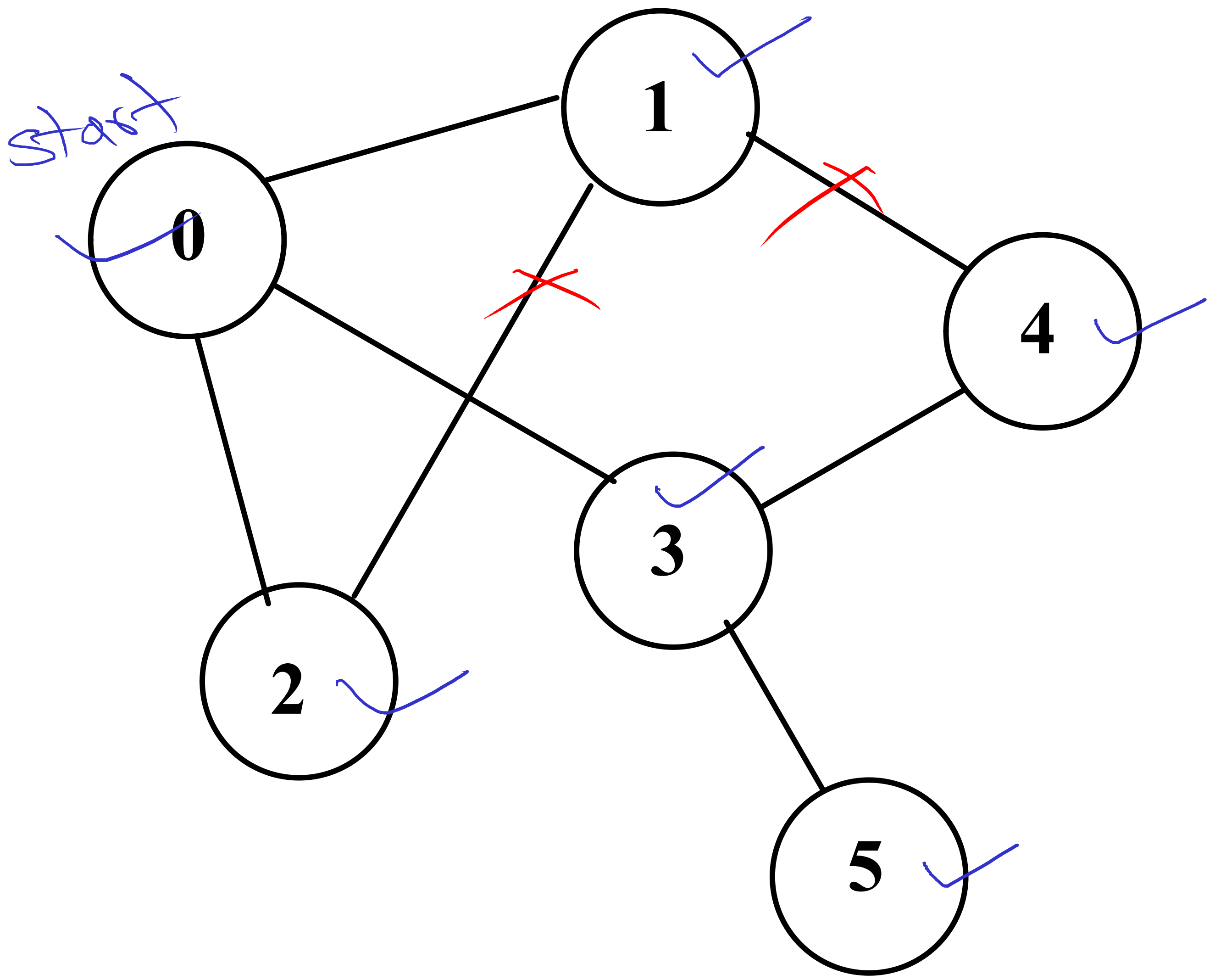
Spanning Tree

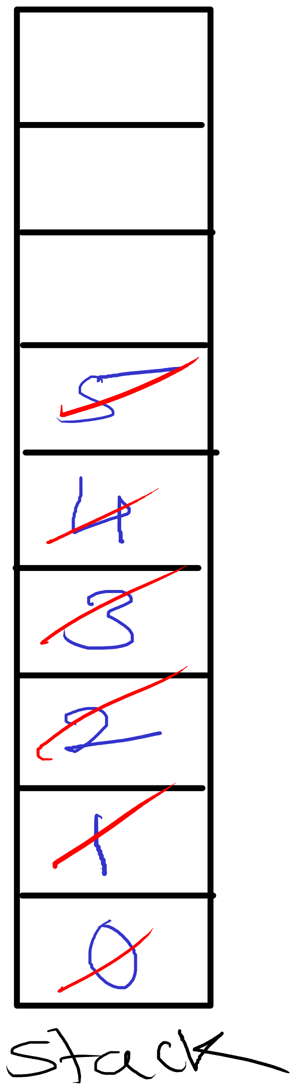
- Tree is a graph without cycles. Includes all V vertices and V-1 edges.
- Spanning tree is connected sub-graph of the given graph that contains all the vertices and sub-set of edges.
- Spanning tree can be created by removing few edges from the graph which are causing cycles to form.
- One graph can have multiple different spanning trees.
- In weighted graph, spanning tree can be made who has minimum weight (sum of weights of edges). Such spanning tree is called as Minimum Spanning Tree.
- Spanning tree can be made by various algorithms.
  - BFS Spanning tree
  - DFS Spanning tree
  - Prim's MST
  - Kruskal's MST

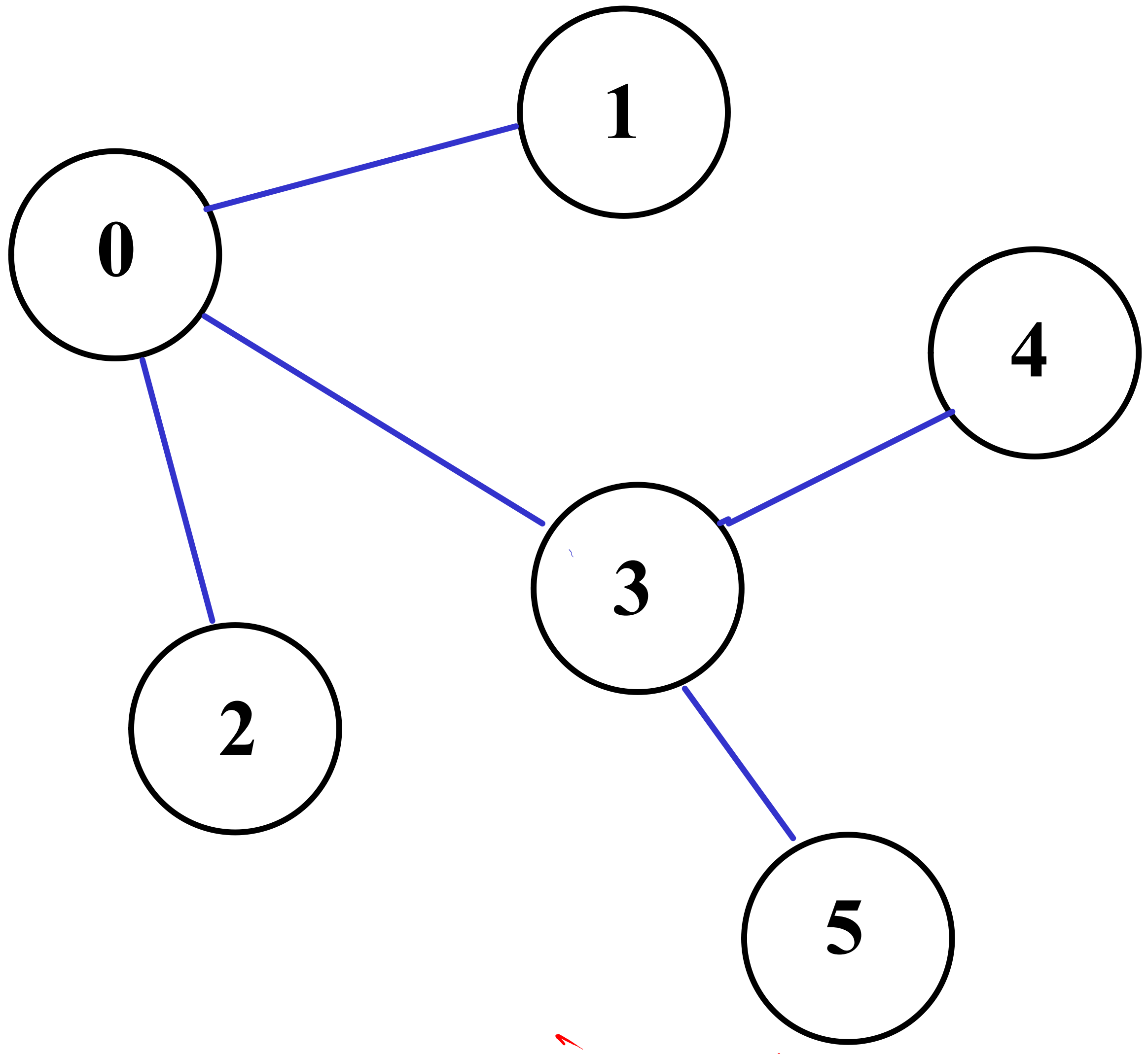
minimal spanning tree



DFS Spanning Tree





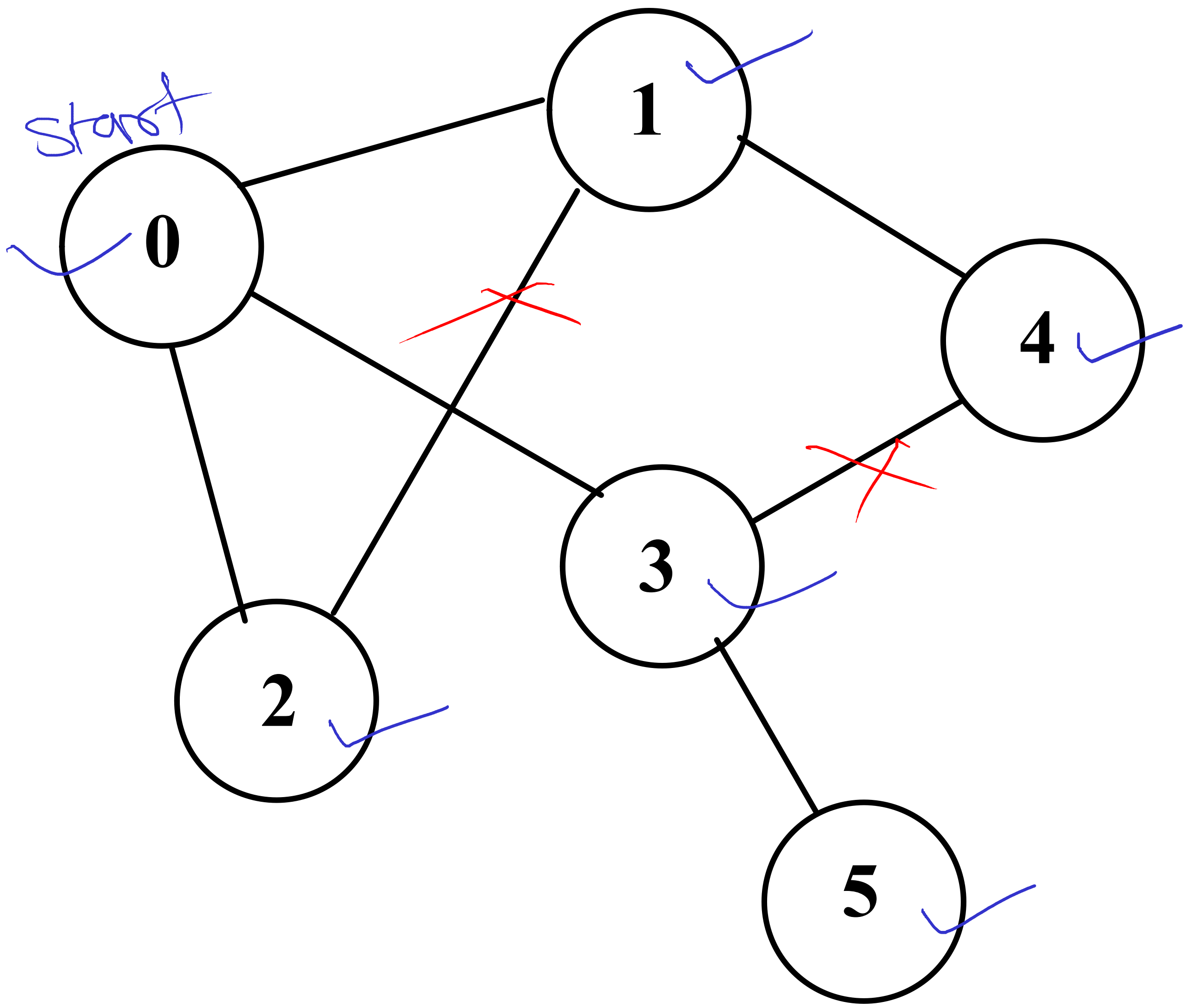


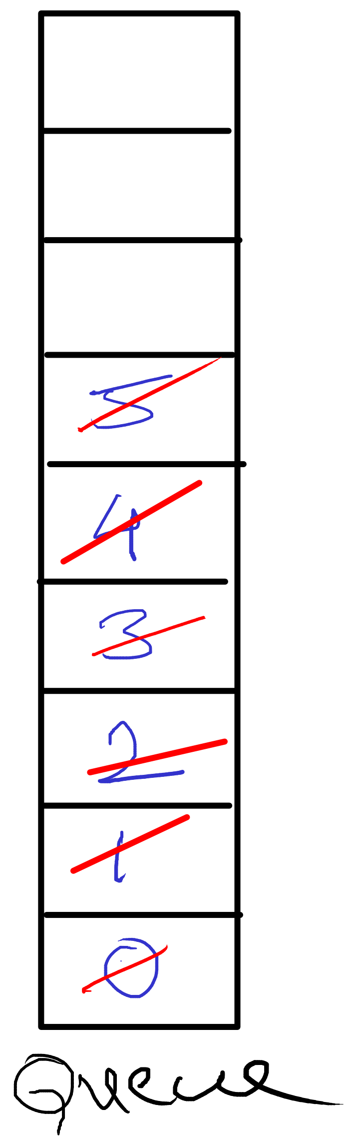
curr v: 0, 3, 5, 4, 2, 1

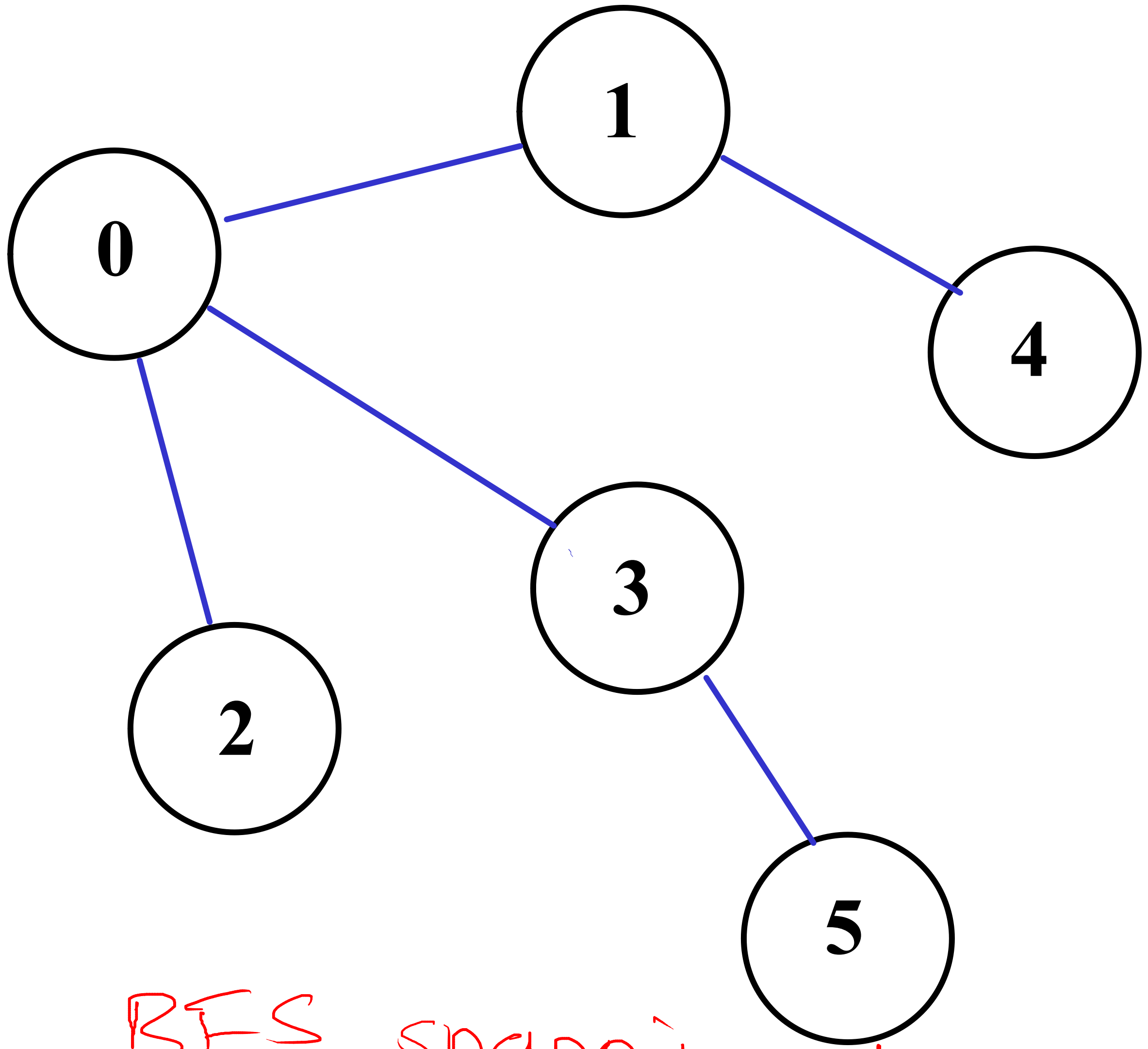
DFS Spanning tree:
0-1, 0-2, 0-3
3-4, 3-5

//1. push starting vertex on stack & mark it.
//2. pop the vertex.
//3. push all its non-marked neighbors on the stack, mark them.
//Also print the vertex to neighboring vertex edges.
4. repeat steps 2-3 until stack is empty.

BFS Spanning Tree







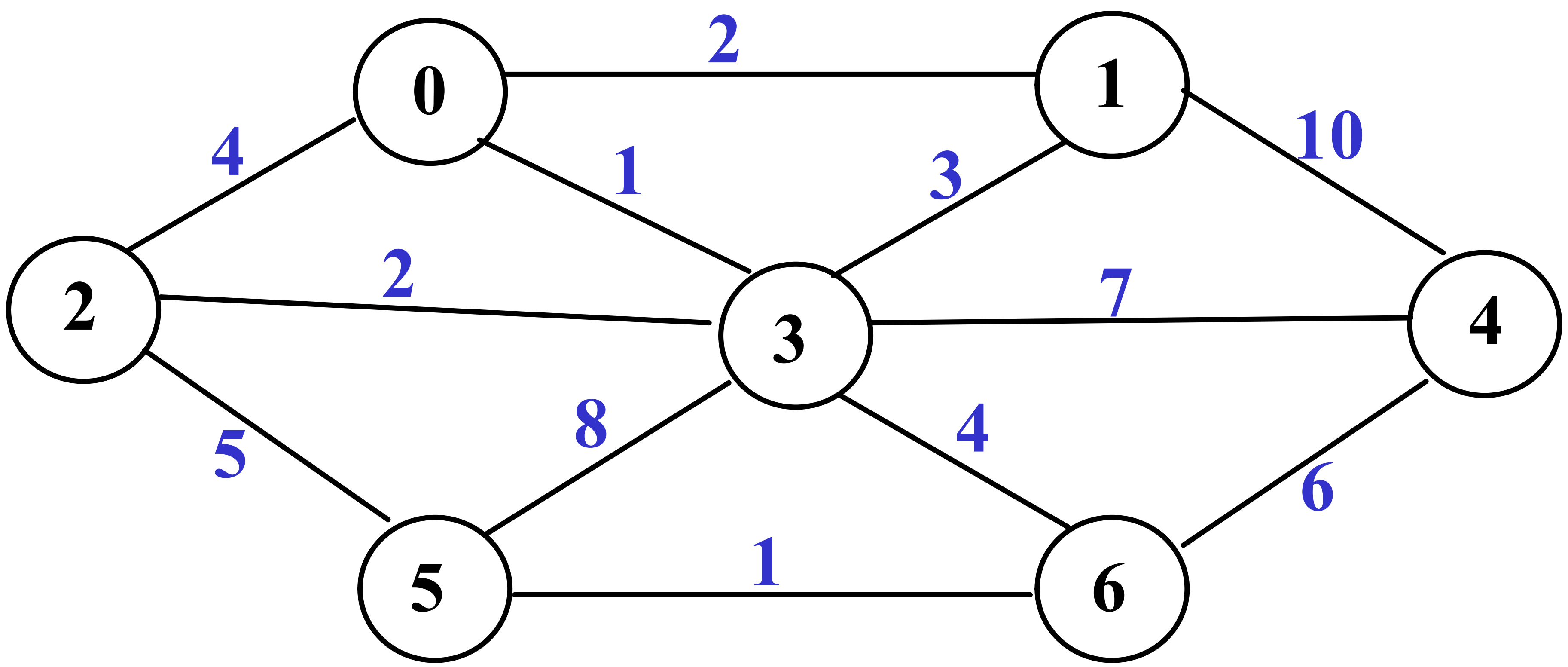
BFS spanning tree:
0-1, 0-2, 0-3
1-4, 3-5

cur v: 0, 1, 2, 3, 4, 4

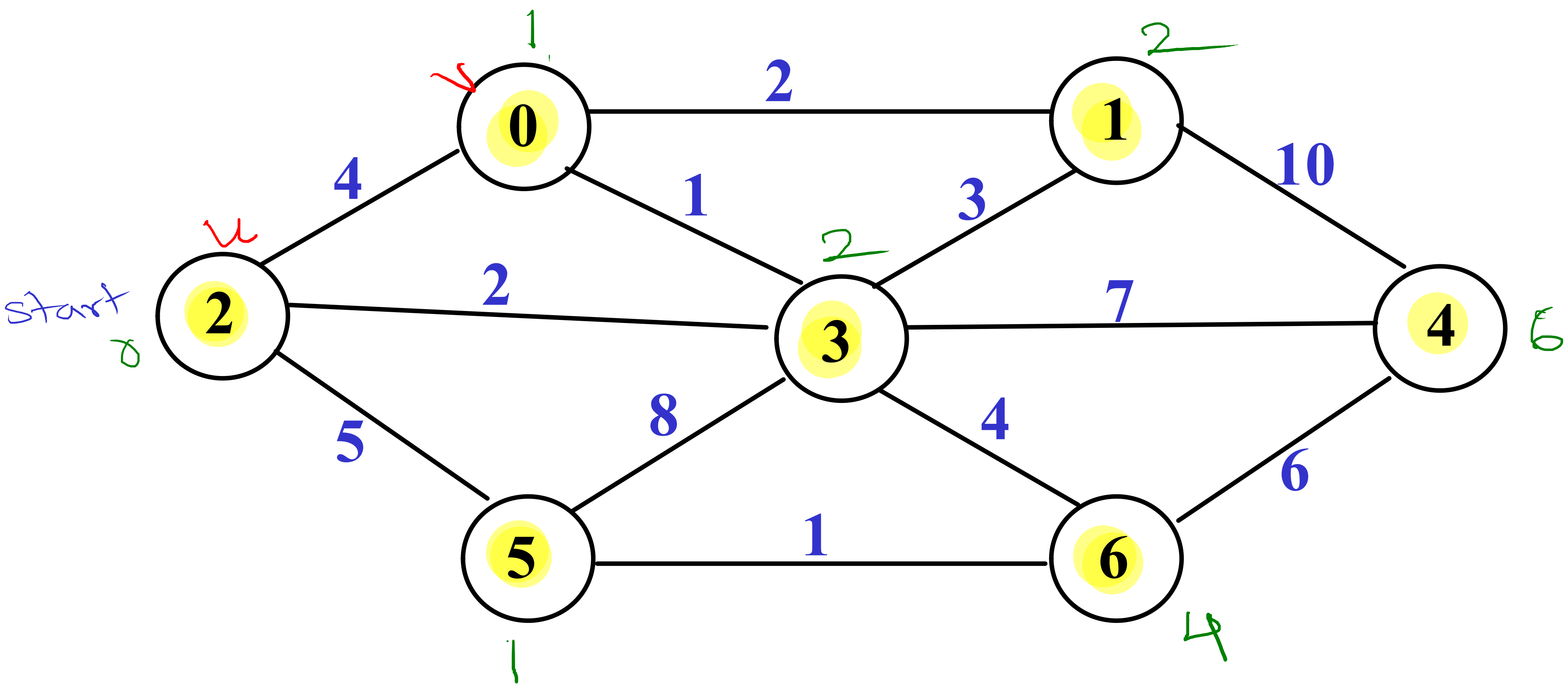
//1. push starting vertex on queue & mark it.
//2. pop the vertex.
//3. push all its non-marked neighbors on the queue, mark them.
//Also print the vertex to neighboring vertex edges.
//4. repeat steps 2-3 until queue is empty.

Prim's MST

1. Create a set mst to keep track of vertices included in MST.
2. Also keep track of parent of each vertex. Initialize parent of each vertex -1.
3. Assign a key to all vertices in the input graph. Key for all vertices should be initialized to INF. The start vertex key should be 0.
4. While mst doesn't include all the vertices
    i. Pick a vertex u which is not there in mst and has minimum key.
    ii. Include vertex u to mst.
    iii. Update key and parent of all adjacent vertices of u.
        a. For each adjacent vertex v,
            if weight of edge u-v is less than the current key of v,
            then update the key as weight of u-v.
        b. Record u as parent of v.



Prim's MST



	K	P
0	1	3
1	2	0
2	0	-1
3	2	2
4	6	6
5	1	6
6	4	3

	K	P
0	4	2
1	∞	-1
2	0	-1
3	2	2
4	∞	-1
5	5	2
6	∞	-1

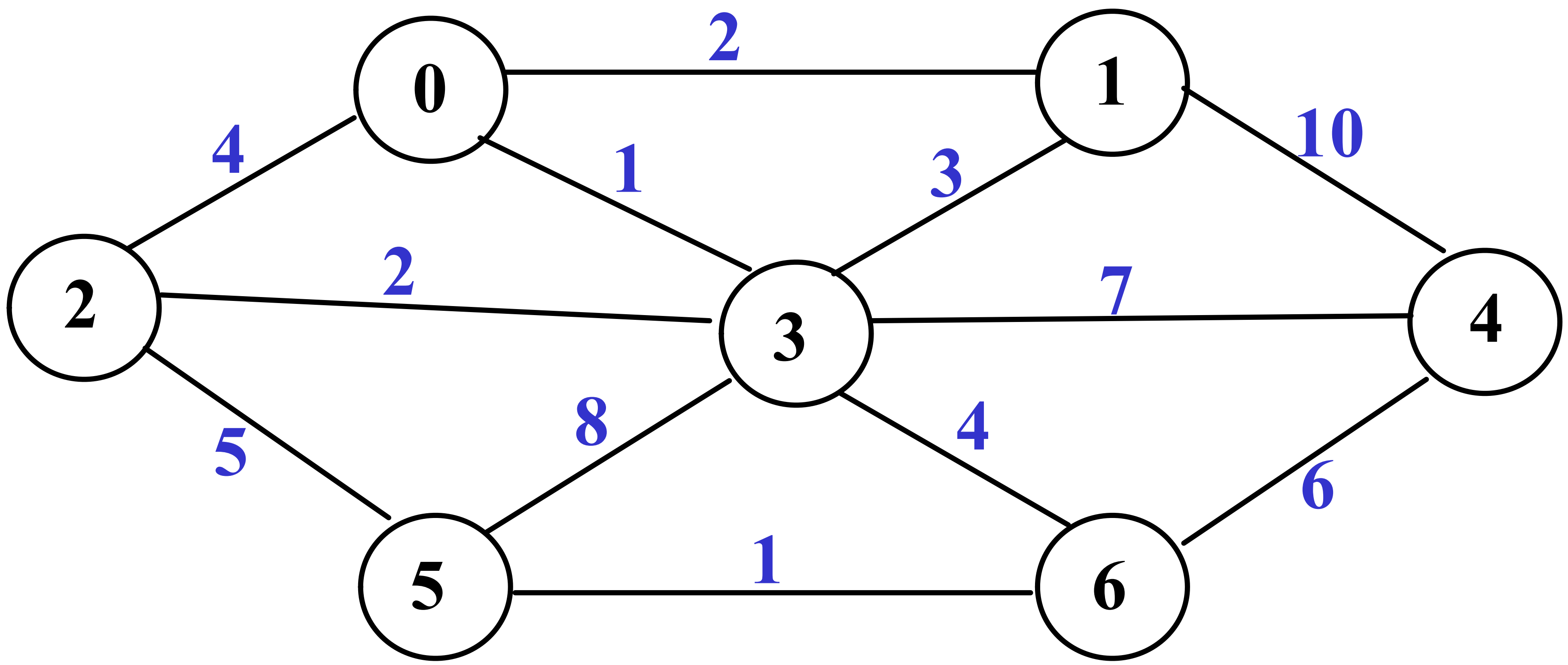
	K	P
0	1	3
1	3	3
2	0	-1
3	2	2
4	7	3
5	5	2
6	4	3

	K	P
0	1	3
1	2	0
2	0	-1
3	2	2
4	7	3
5	5	2
6	4	3

	K	P
0	1	3
1	2	0
2	0	-1
3	2	2
4	7	3
5	5	2
6	4	3

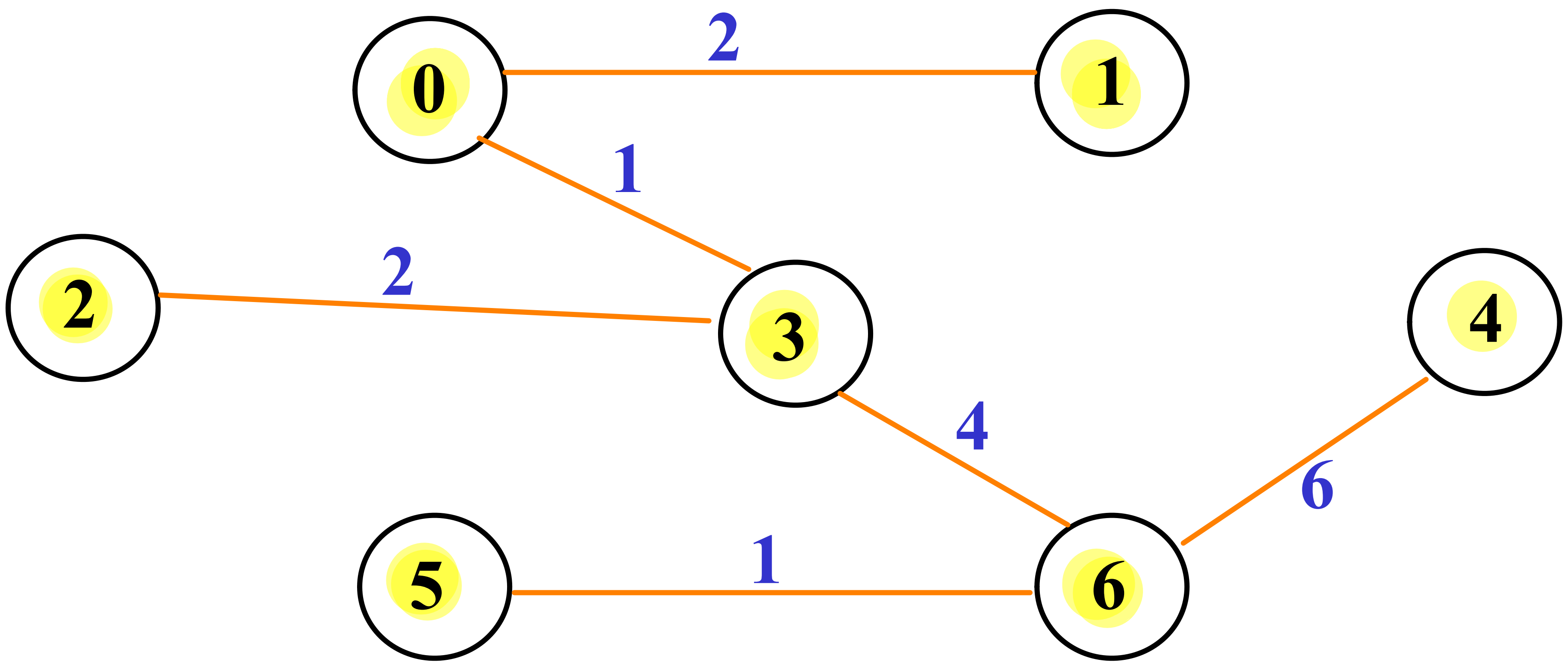
	K	P
0	1	3
1	2	0
2	0	-1
3	2	2
4	6	6
5	1	6
6	4	3

	K	P
0	1	3
1	2	0
2	0	-1
3	2	2
4	6	6
5	1	6
6	4	3



	K	P
0	1	3
1	2	0
2	0	-1
3	2	2
4	6	6
5	1	6
6	4	3

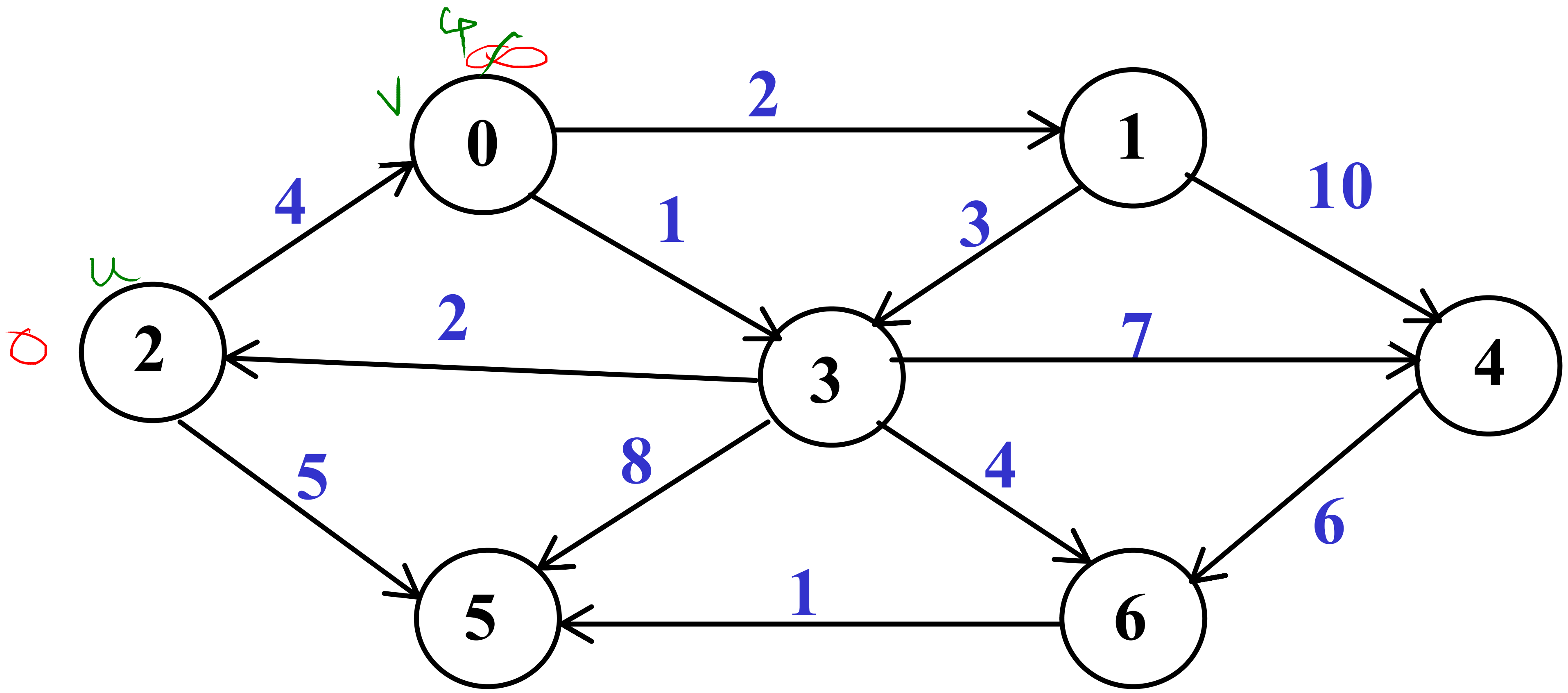
start



$wt = 2+1+1+2+4+6 = \underline{\underline{16}}$

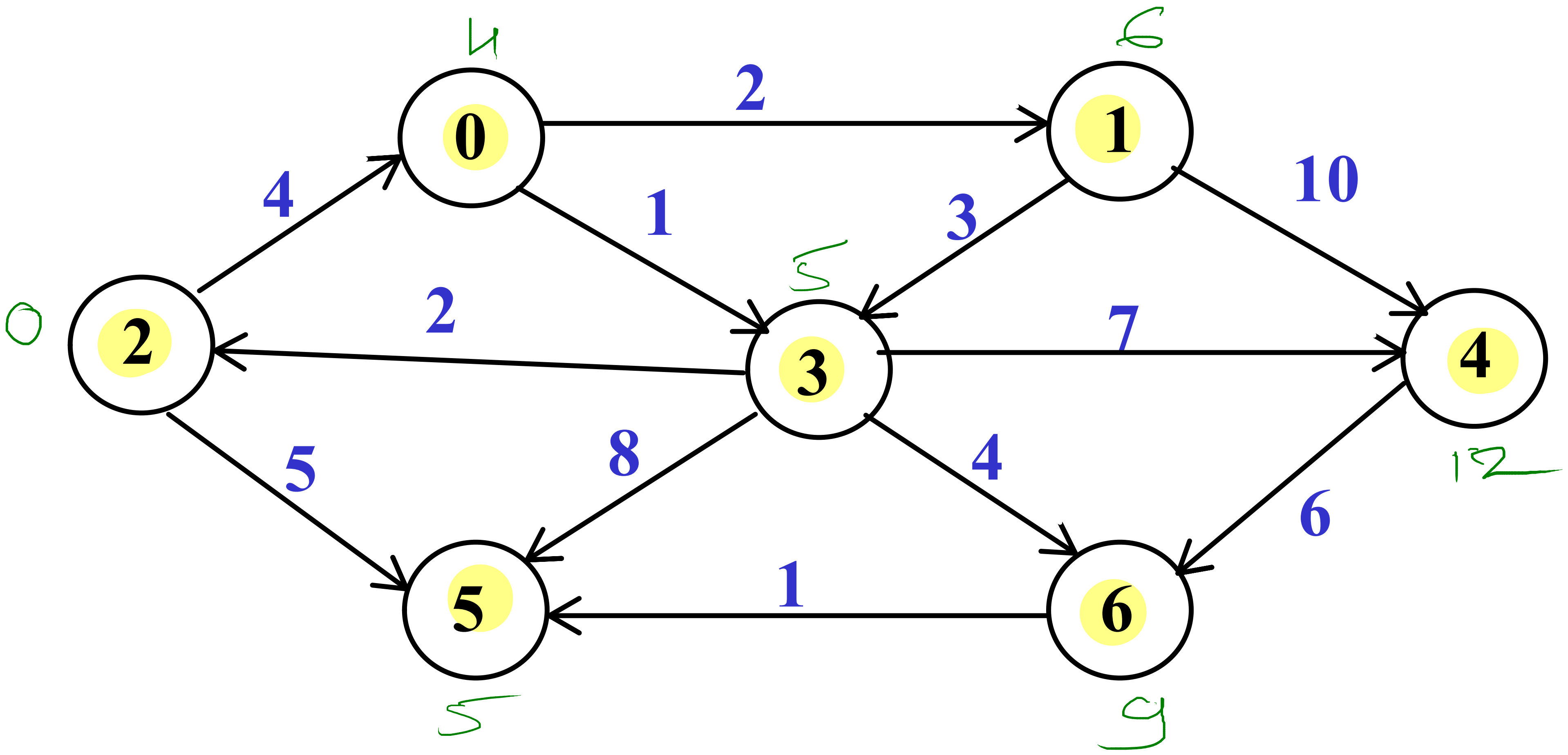
Dijkstra's Algorithm

1. Create a set spt to keep track of vertices included in shortest path tree.
2. Track distance of all vertices in the input graph. Distance for all vertices should be initialized to INF. The start vertex distance should be 0.
3. While spt doesn't include all the vertices
    i. Pick a vertex u which is not there in spt and has minimum distance.
    ii. Include vertex u to spt.
    iii. Update distances of all adjacent vertices of u.
    For each adjacent vertex v,
        if distance of u + weight of edge u-v is less than the current distance of v,
        then update its distance as distance of u + weight of edge u-v.



0 + 4 < ∞

Dijkstra's Algorithm



	D	P
0	4	2
1	6	0
2	0	-1
3	5	0
4	12	3
5	5	2
6	9	3

	D	P
0	4	2
1	∞	-1
2	0	-1
3	∞	-1
4	∞	-1
5	5	2
6	∞	-1

	D	P
0	4	2
1	6	0
2	0	-1
3	5	0
4	∞	-1
5	5	2
6	∞	-1

	D	P
0	4	2
1	6	0
2	0	-1
3	5	0
4	12	3
5	5	2
6	9	3

	D	P
0	4	2
1	6	0
2	0	-1
3	5	0
4	12	3
5	5	2
6	9	3

	D	P
0	4	2
1	6	0
2	0	-1
3	5	0
4	12	3
5	5	2
6	9	3

	D	P
0	4	2
1	6	0
2	0	-1
3	5	0
4	12	3
5	5	2
6	9	3

Problem solving technique: Greedy approach

- A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.
- We can make choice that seems best at the moment and then solve the sub-problems that arise later.
- The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the sub-problem.
- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one.
- A greedy algorithm never reconsiders its choices.
- A greedy strategy may not always produce an optimal solution.

eg. Greedy algorithm decides minimum number of coins to give while making change.

coins available : 50, 20, 10, 5, 2, 1

Rs - 36

36 - 20 = 16   20

16 - 10 = 6   20   10

6 - 5 = 1   20   10   5

1 - 1 = 0   20   10   5   1

travel — Pune → Delhi

- by road
  - personal vehicle
  - public transport
- by train
- by flight