Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Merge sort

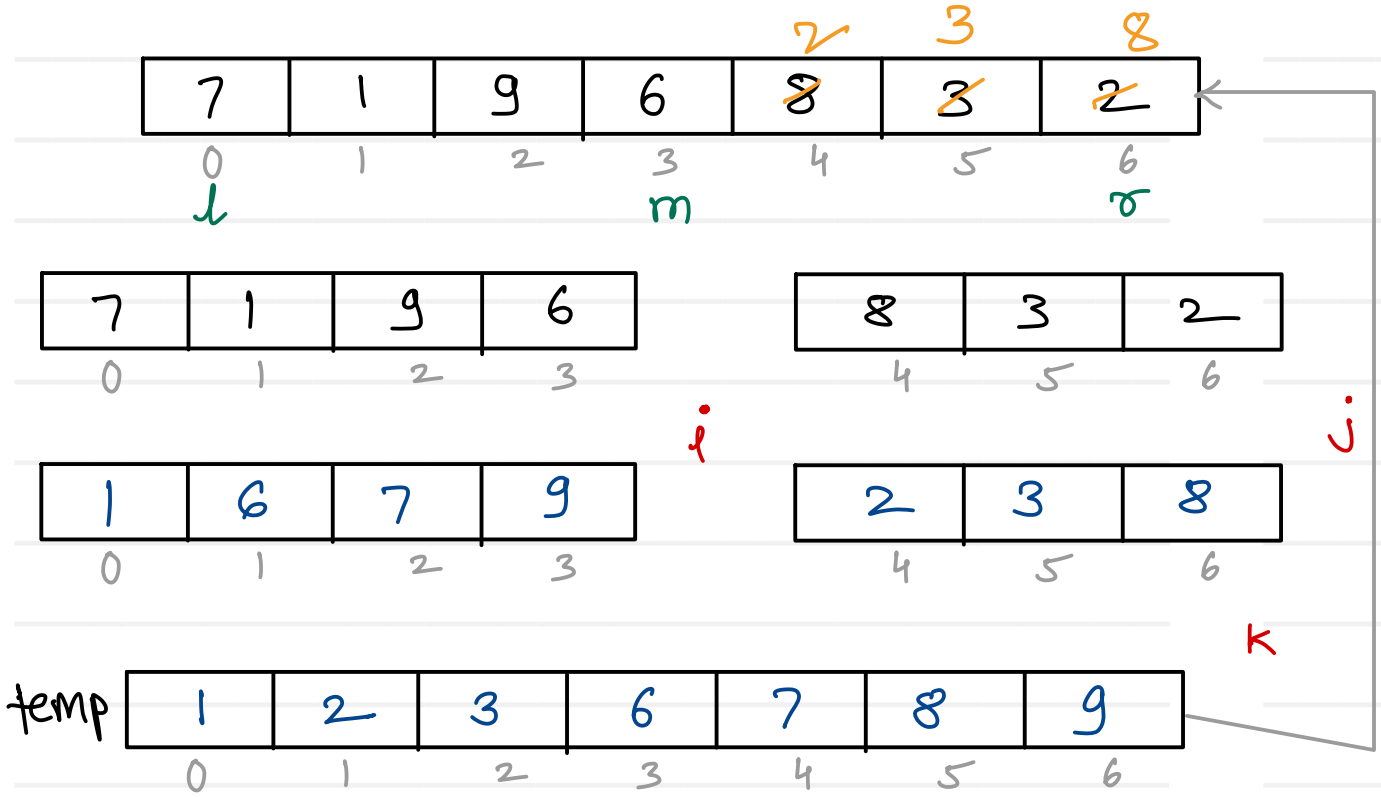
1. Divide array in two parts
2. Sort both partitions individually (by merge sort only)
3. Merge sorted partitions into temporary array
4. Overwrite temporary array into original array

$m = \frac{l+r}{2}$

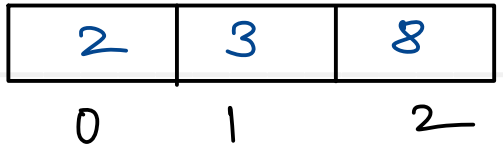
LP = l → m (i)

RP = m+1 → r (j)

size = r - l + 1



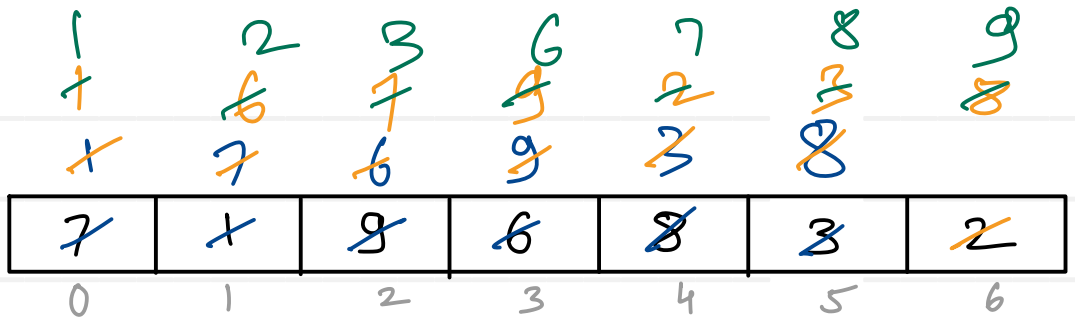
size=3




```
for(i=0; i<size; i++)
    arr[left+i] = temp[i]

arr[l+0] = temp[0]
arr[l+1] = temp[1]
arr[l+2] = temp[2]
```

Merge sort

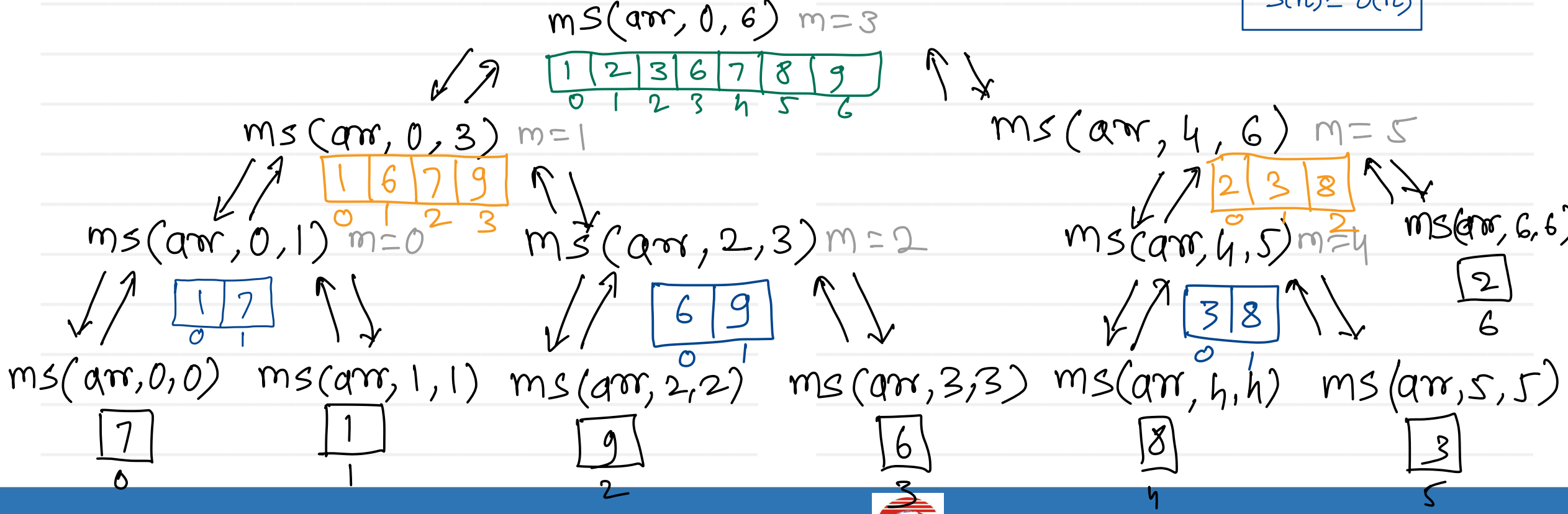


no. of levels = log n
comps per level = n
Total comps = n log n

Best / Avg / Worst: $T(n) = O(n \log n)$

temp array is auxiliary space

$S(n) = O(n)$



Quick sort

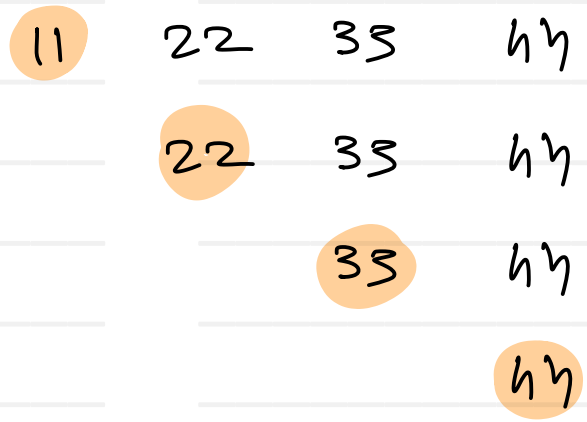
1. Select pivot/axis/reference element from array
2. Arrange lesser elements on left side of pivot
3. Arrange greater elements on right side of pivot
4. Sort left and right side of pivot again (by quick sort)

no. of levels = log n

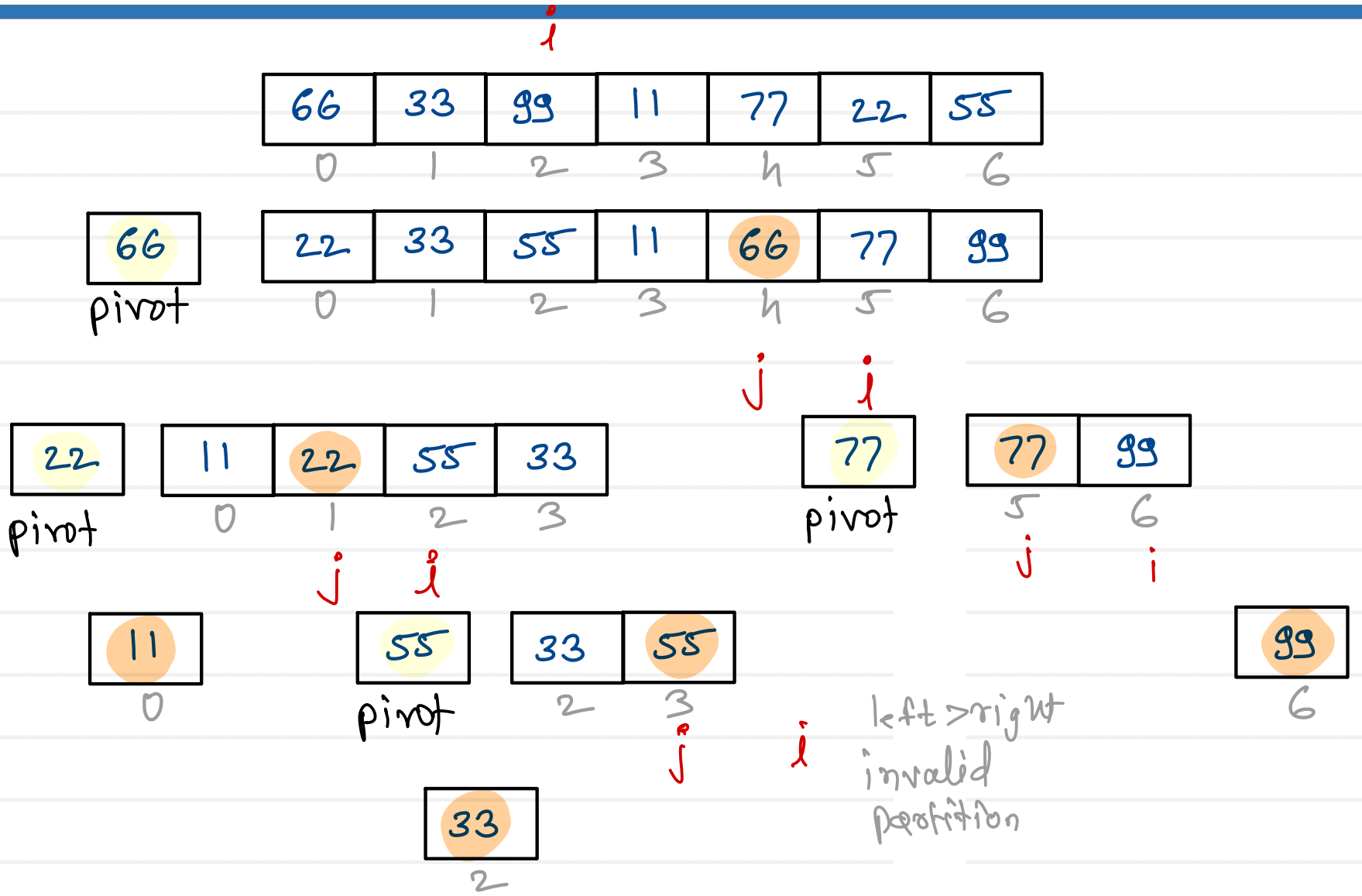
comps per level = n

Total comps = n log n

Best Avg $T(n) = O(n \log n)$



Quick sort


i
66 33 99 11 77 22 55
0 1 2 3 4 5 6
66
pivot
22 33 55 11 66 77 99
0 1 2 3 4 5 6
j
i
22
pivot
11 22 55 33
0 1 2 3
j
i
77
pivot
77 99
5 6
j
i
11
0
55
pivot
33 55
2 3
j
i
left > right
invalid
partition
99
6
33
2

	space	Time Best	Time Avg	Time Worst
selection sort	O(1)	$O(n^2)$	$O(n^2)$	$O(n^2)$
bubble sort	O(1)	$O(n)$	$O(n^2)$	$O(n^2)$
insertion sort	O(1)	$O(n)$	$O(n^2)$	$O(n^2)$
Heap sort	O(1)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	O(1)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge sort	O(n)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

in place sorting algorithm

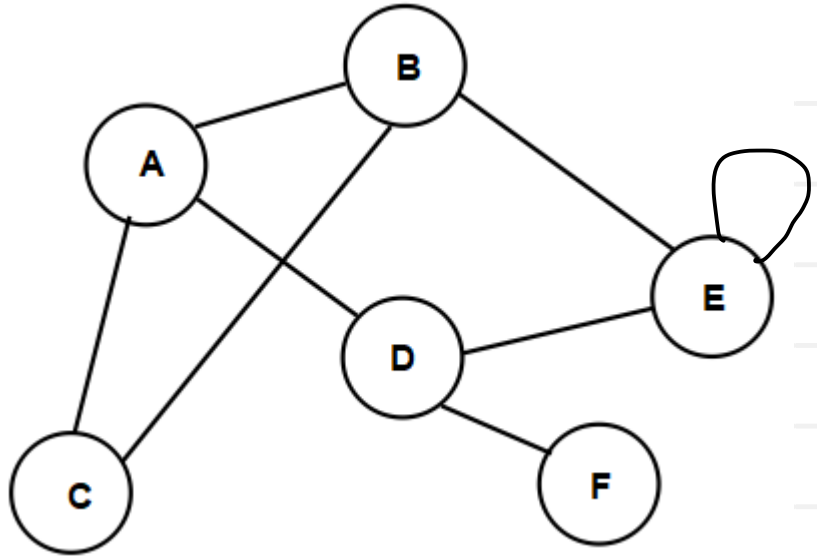
Graph : Terminologies

- **Graph** is a non linear data structure having set of vertices (nodes) and set of edges (arcs).
  - G = {V, E}

    Where V is a set of vertices and E is a set of edges
  - **Vertex (node)** is an element in the graph

    V = {A, B, C, D, E, F}
  - **Edge (arc)** is a line connecting two vertices

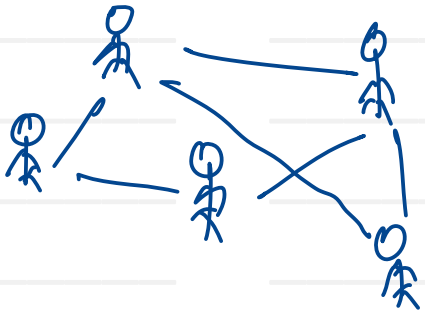
    E = {(A,B), (A,C), (B,C), (B,E), (D, E), (D,F),(A,D)}

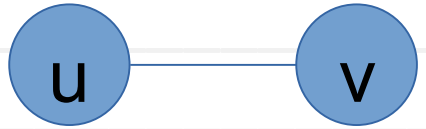


- Vertex A is set be **adjacent** to B, if and only if there is an edge from A to B.
- **Degree of vertex :-** Number of vertices adjacent to given vertex
- **Path :-** Set of edges connecting any two vertices is called as path between those two vertices.
  - Path between A to D = {(A, B), (B, E), (E, D)}
- **Cycle :-** Set of edges connecting to a node itself is called as cycle.
  - {(A, B), (B, E), (E, D), (D, A)}
- **Loop :-** An edge connecting a node to itself is called as loop. Loop is smallest cycle.

Graph : Types

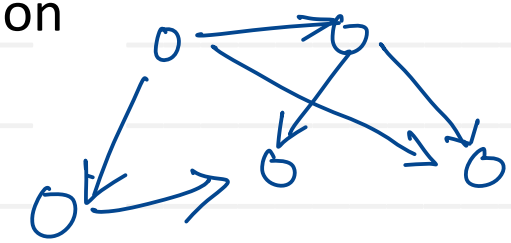
- **Undirected graph.**
  - If we can represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices i.e. undirected edge.**
  - **graph which contains undirected edges referred as undirected graph.**

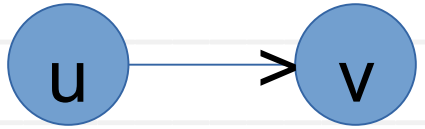




(u, v) == (v, u)

- **Directed Graph (Di-graph)**
  - If we cannot represent any edge either (u,v) OR (v,u) then it is referred as an **ordered pair of vertices** i.e. directed edge.
  - **graph which contains set of directed edges referred as directed graph (di-graph).**
  - graph in which each edge has some direction

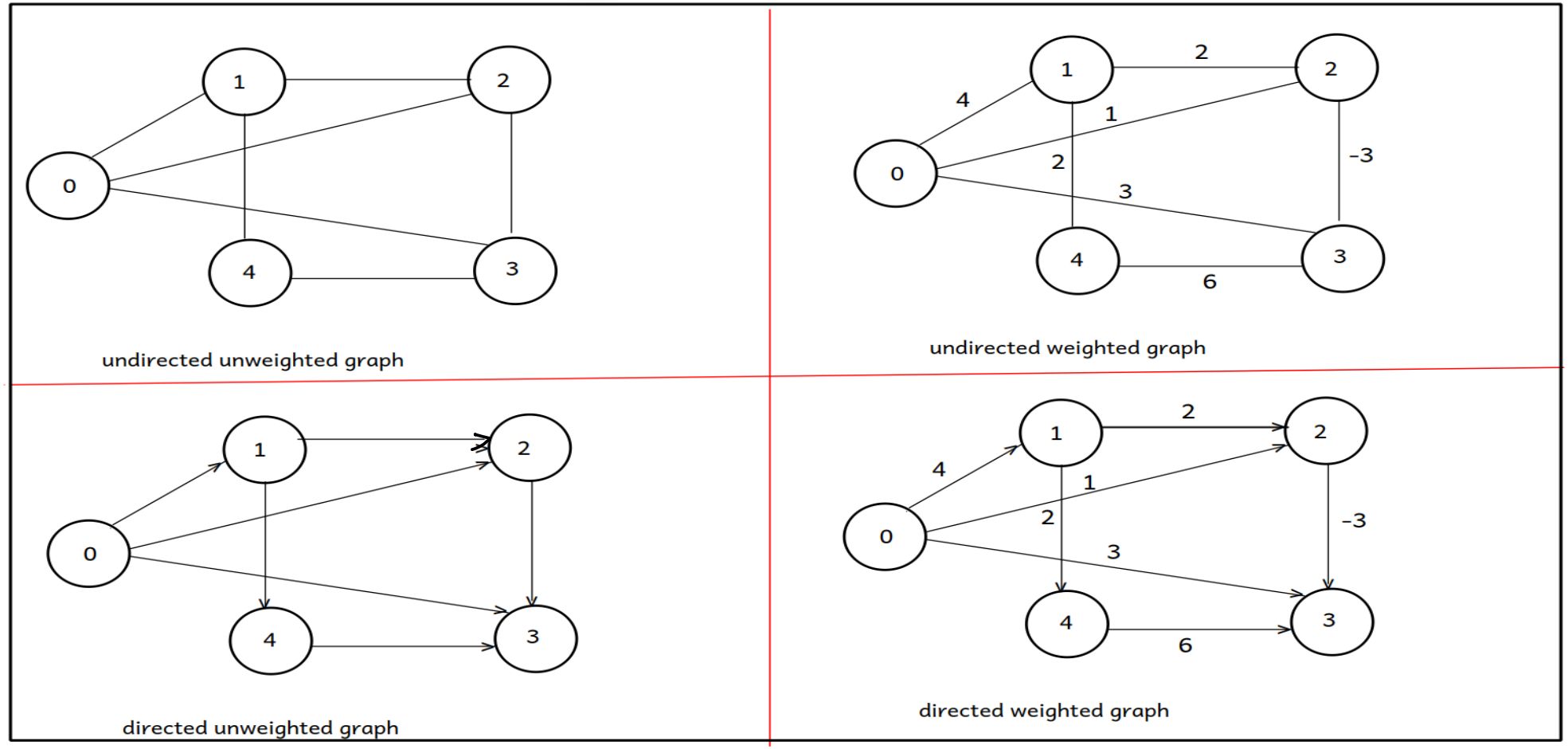




(u, v) != (v, u)

Graph : Types

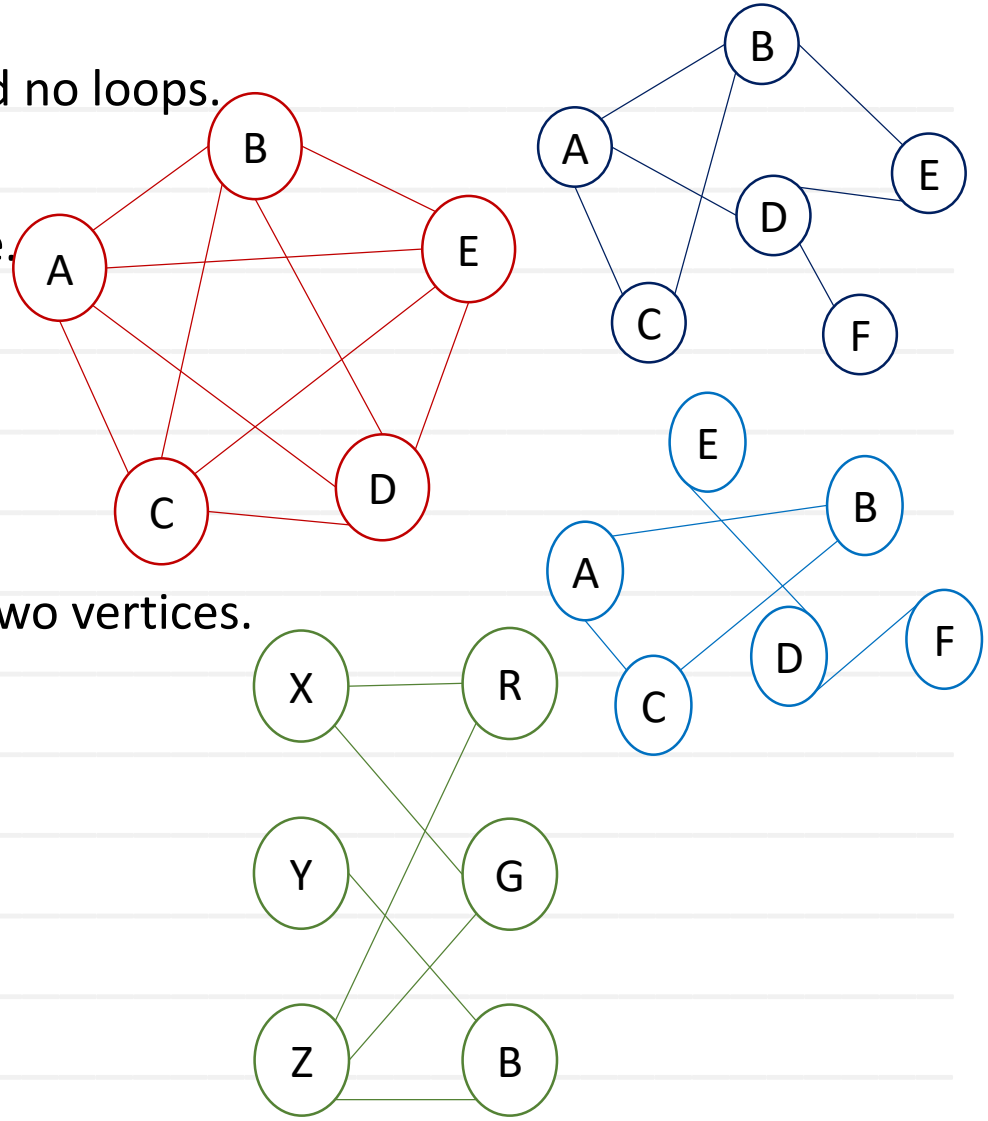
- **Weighted Graph**
  - A graph in which edge is associated with a number (ie weight)



Graph : Types

- **Simple Graph**
  - Graph not having multiple edges between adjacent nodes and no loops.
- **Complete Graph**
  - Simple graph in which node is adjacent with every other node.
  - Un-Directed graph: Number of Edges = n (n -1) / 2

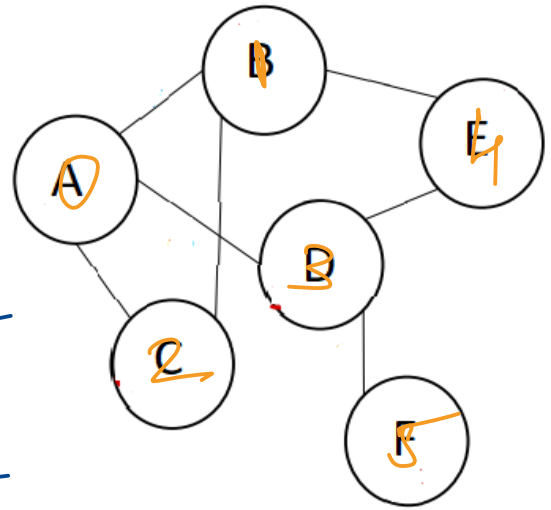
    where, n – number of vertices
  - Directed graph: Number of edges = n (n-1)
- **Connected Graph**
  - Simple graph in which there is some path exist between any two vertices.
  - Can traverse the entire graph starting from any vertex.
- **Bi-partite graph**
  - Vertices can be divided in two disjoint sets.
  - Vertices in first set are connected to vertices in second set.
  - Vertices in a set are not directly connected to each other.

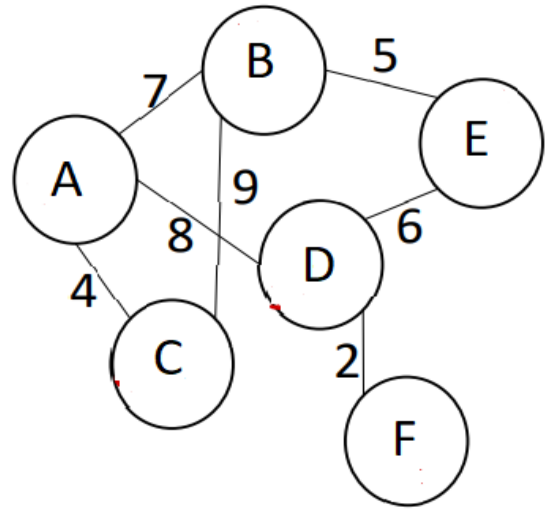


Graph Implementation – Adjacency Matrix

- If graph have V vertices, a V x V matrix can be formed to store edges of the graph.
- Each matrix element represent presence or absence of the edge between vertices.
- For non-weighted graph, 1 indicate edge and 0 indicate no edge.
- For weighted graph, weight value indicate the edge and infinity sign ∞ represent no edge.
- For un-directed graph, adjacency matrix is always symmetric across the diagonal.
- Space complexity of this implementation is O(V^2).

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



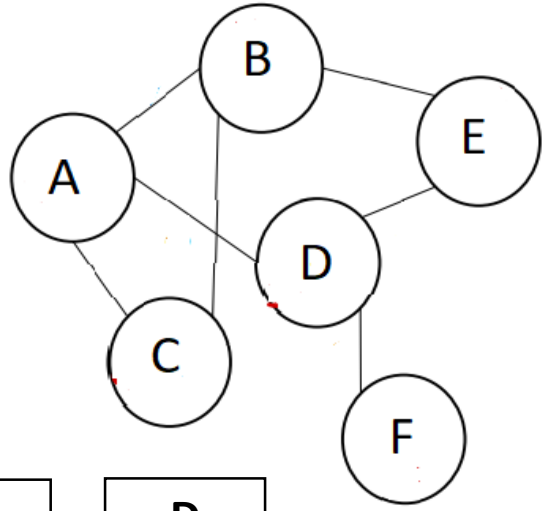


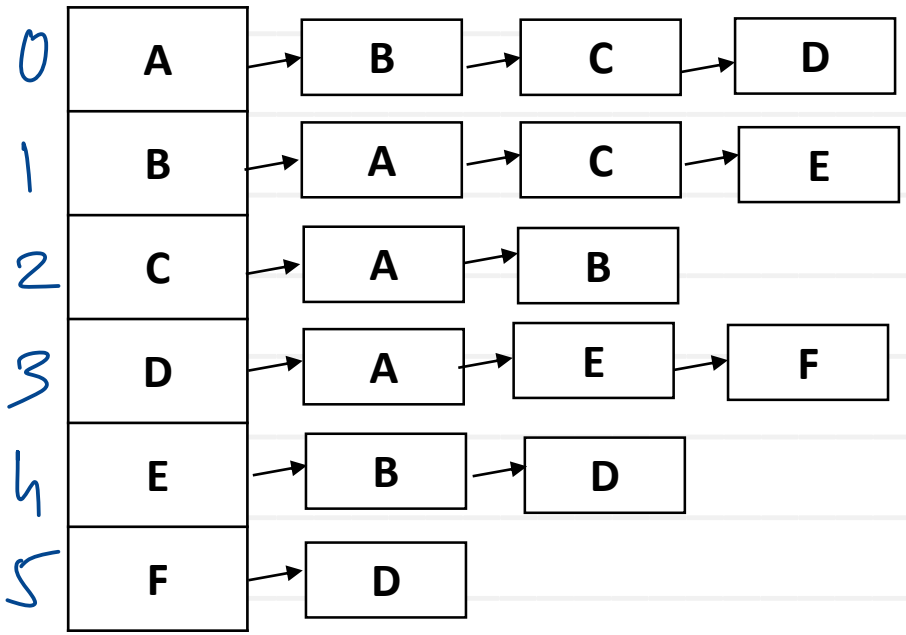
	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	0	1	0
C	1	1	0	0	0	0
D	1	0	0	0	1	1
E	0	1	0	1	0	0
F	0	0	0	1	0	0

	A	B	C	D	E	F
A	∞	7	4	8	∞	∞
B	7	∞	9	∞	5	∞
C	4	9	∞	∞	∞	∞
D	8	∞	∞	∞	6	2
E	∞	5	∞	6	∞	∞
F	∞	∞	∞	2	∞	∞

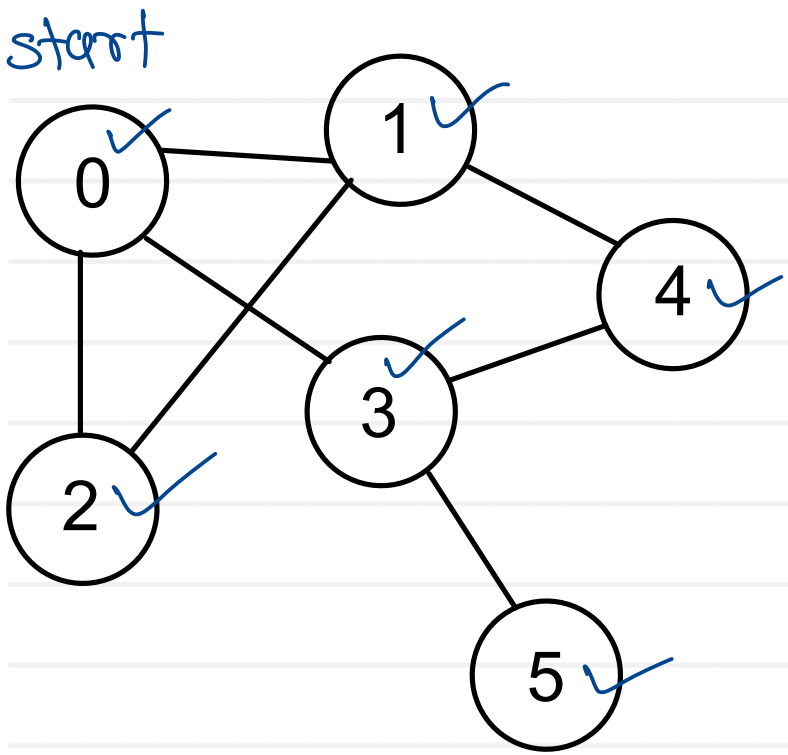
Graph Implementation – Adjacency List

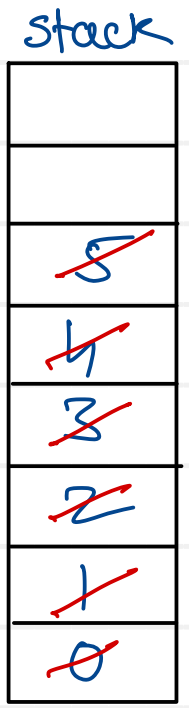
- Each vertex holds list of its adjacent vertices.
- For non-weighted graphs only, neighbor vertices are stored.
- For weighted graph, neighbor vertices and weights of connecting edges are stored.
- Space complexity of this implementation is O(V+E).
- If graph is sparse graph (with fewer number of edges), this implementation is more efficient (as compared to adjacency matrix method).





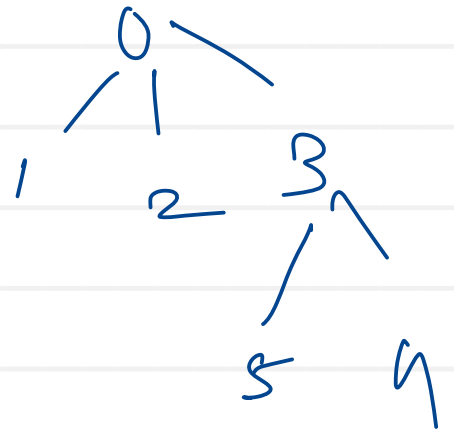
DFS Traversal



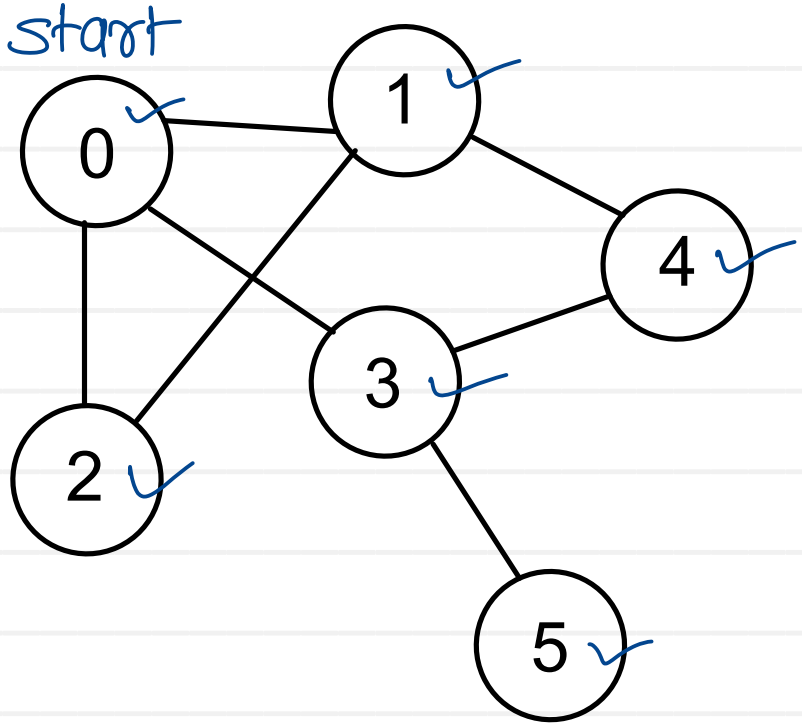


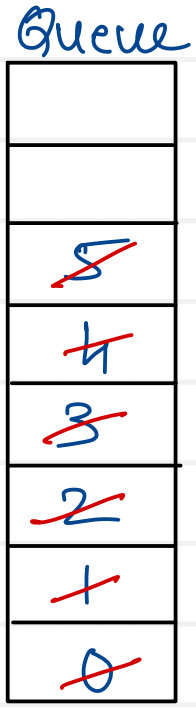
Traversal : 0, 3, 5, 4, 2, 1

1. Choose a vertex as start vertex.
2. Push start vertex on stack & mark it.
3. Pop vertex from stack.
4. Print the vertex.
5. Put all non-visited neighbours of the vertex on the stack and mark them.
6. Repeat 3-5 until stack is empty.



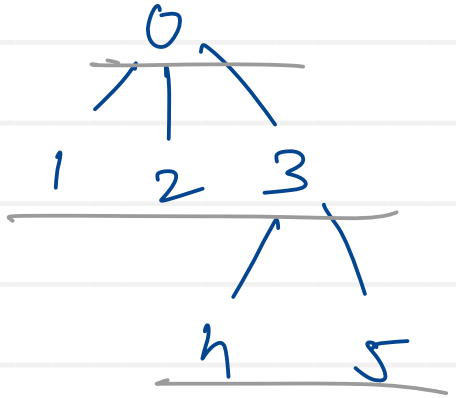
BFS Traversal





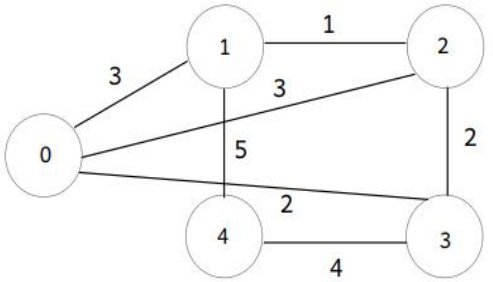
Traversal : 0, 1, 2, 3, 4, 5

1. Choose a vertex as start vertex.
2. Push start vertex on queue & mark it
3. Pop vertex from queue.
4. Print the vertex.
5. Put all non-visited neighbours of the vertex on the queue and mark them.
6. Repeat 3-5 until queue is empty.



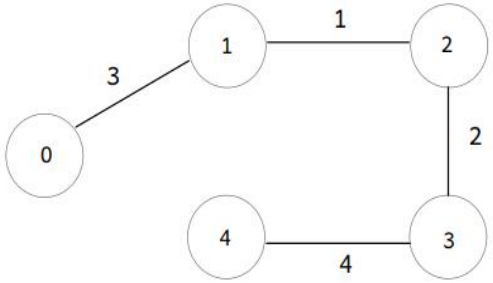
Spanning Tree

- 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



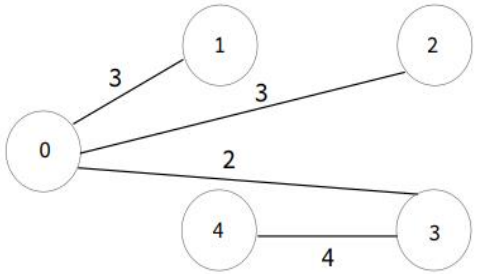
G1

Weight of a graph G1 = 20



G2

Weight of a graph G2 = 10

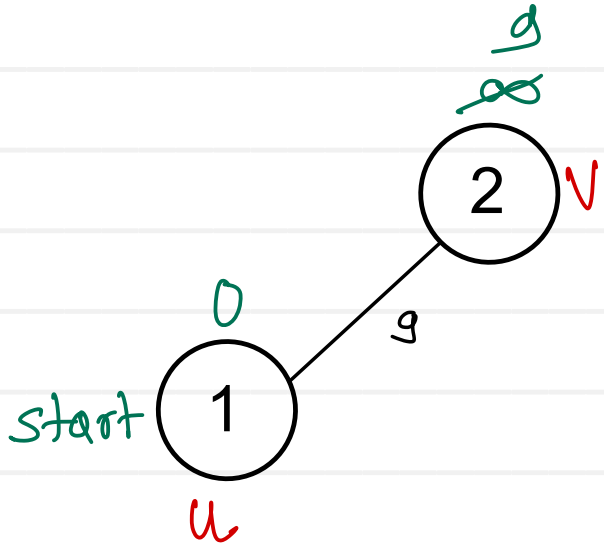


G3

Weight of a graph G2 = 12

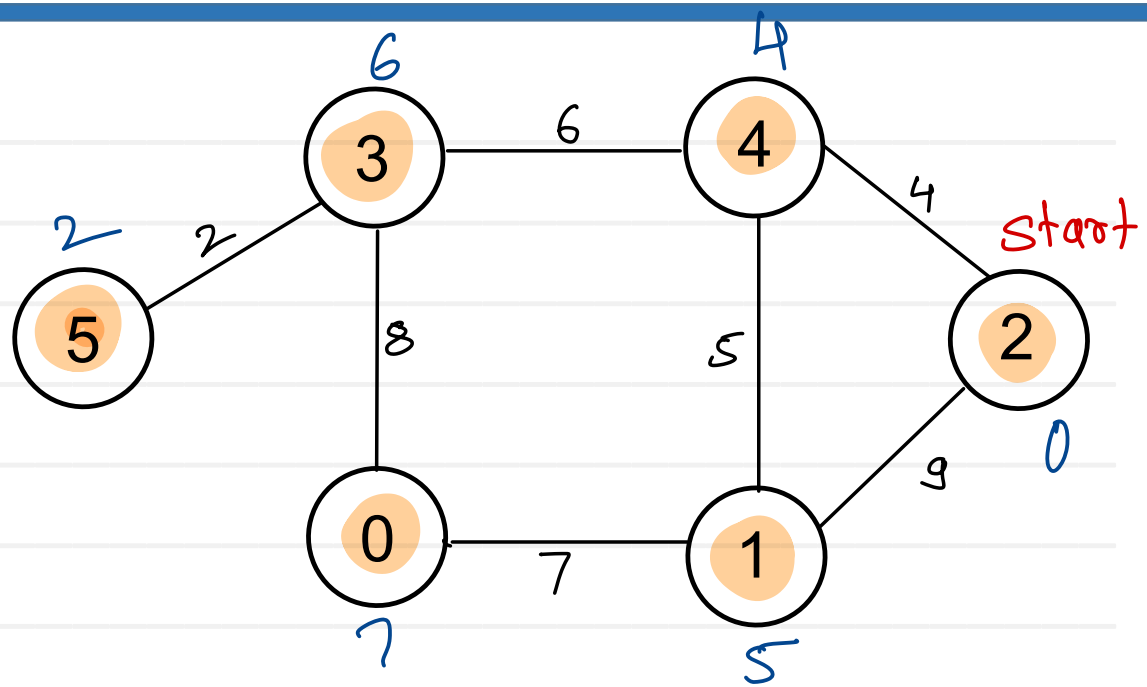
Prim's Algorithm

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.

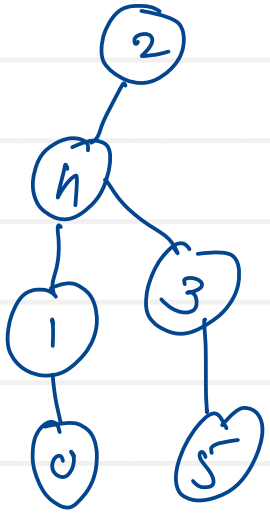



```
if(adjMat[u][v] < key[v])
    key[v] = adjMat[u][v];
```

Prim's Algorithm



V	key	parent
0	7	1
1	5	4
2	0	-1
3	6	4
4	4	2
5	2	3



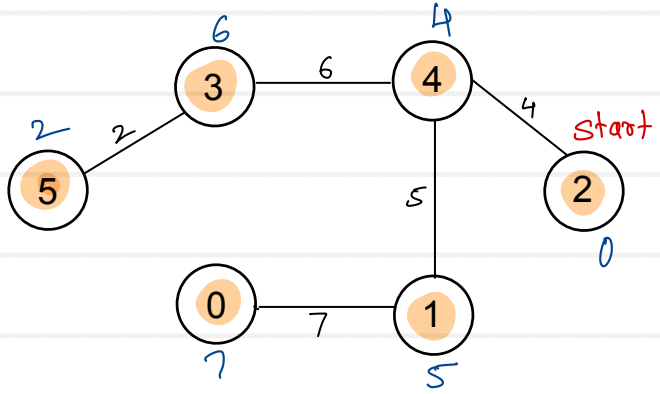
V	key	parent
0	∞	-1
1	9	2
2	0	-1
3	∞	-1
4	4	2
5	∞	-1

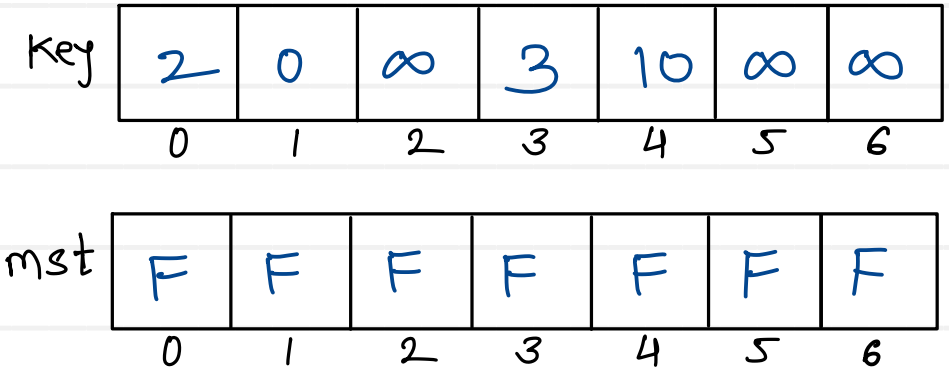
V	key	parent
0	∞	-1
1	5	4
2	0	-1
3	6	4
4	4	2
5	∞	-1

V	key	parent
0	7	1
1	5	4
2	0	-1
3	6	4
4	4	2
5	∞	-1

V	key	parent
0	7	1
1	5	4
2	0	-1
3	6	4
4	4	2
5	2	3

V	key	parent
0	7	1
1	5	4
2	0	-1
3	6	4
4	4	2
5	2	3



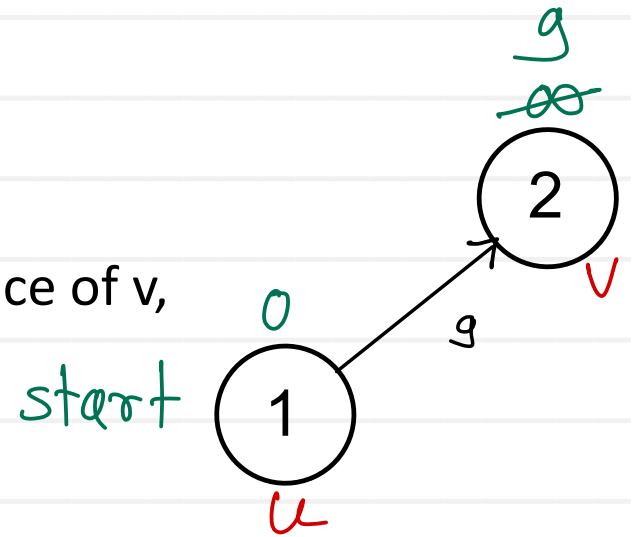


minkey minkeyVertex i condition

```
int findMinKeyVertex(int key[], boolean mst[]){
    int minkey = ∞, minkeyVertex = -1;
    for(int i=0; i<vertexCount; i++){
        if( !mst[i] && key[i] < minkey ){
            minkey = key[i]
            minkeyVertex = i;
        }
    }
    return minkeyVertex;
}
```

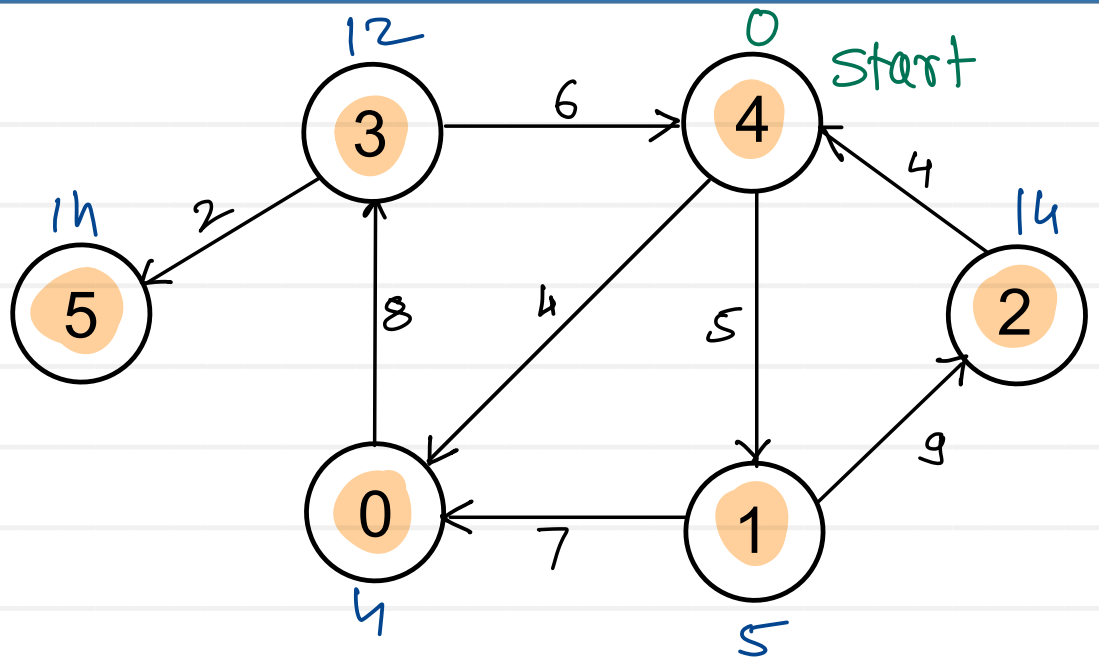
Dijkstra's Algorithm

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.

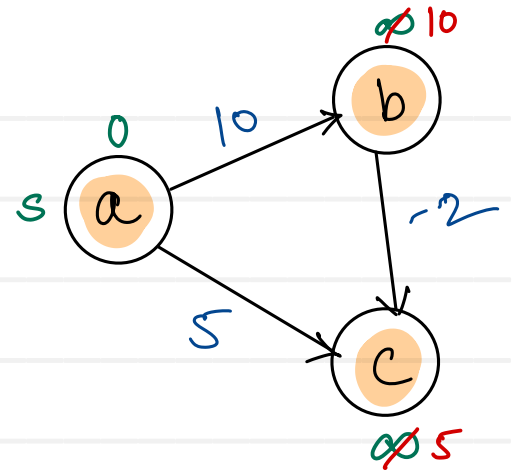


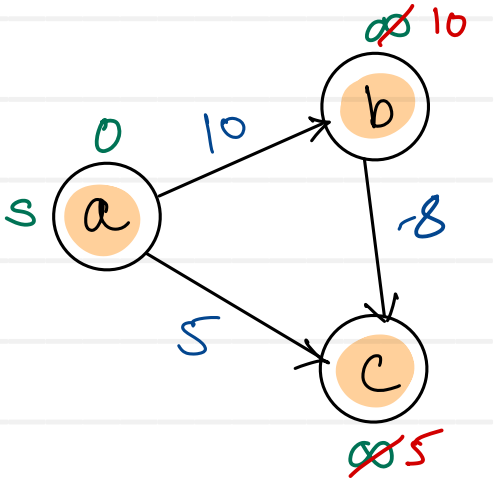

```
if(dist[u] + adjMat[u][v] < dist[v])
    dist[v] = dist[u] + adjMat[u][v]
```

Dijkstra's Algorithm



	D
0	4
1	5
2	14
3	12
4	0
5	14





	D
0	4
1	5
2	∞
3	∞
4	0
5	∞

	D
0	4
1	5
2	∞
3	12
4	0
5	∞

	D
0	4
1	5
2	14
3	12
4	0
5	∞

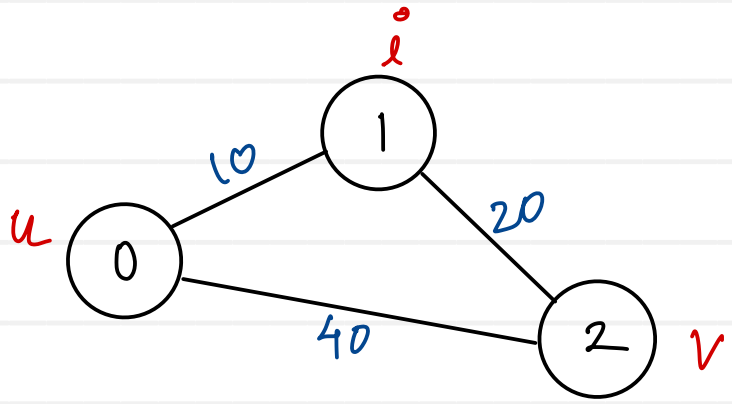
	D
0	4
1	5
2	14
3	12
4	0
5	14

	D
0	4
1	5
2	14
3	12
4	0
5	14

Floyd Warshall Algorithm

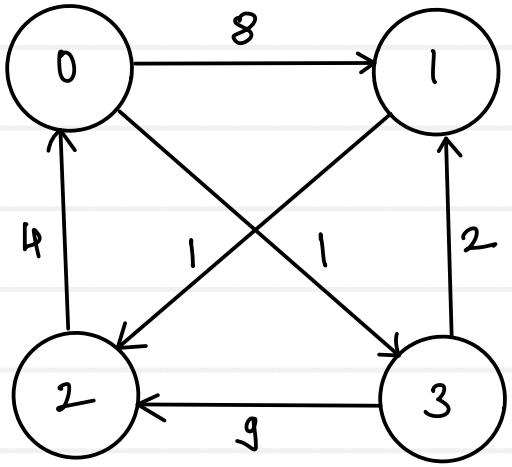
1. Create distance matrix to keep distance of every vertex from each vertex.
   Initially assign it with weights of all edges among vertices
   (i.e. adjacency matrix).

2. Consider each vertex (i) in between pair of any two vertices (u, v) and
   find the optimal distance between u & v considering intermediate vertex
   i.e. dist(u,v) = dist(u,i) + dist(i,v),
   if dist(u,i) + dist(i,v) < dist(u,v).




```
if(dist[u][i] + dist[i][v] < dist[u][v])
    dist[u][v] = dist[u][i] + dist[i][v];
```

Floyd Warshall Algorithm



$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \infty & 8 & \infty & 1 \\ \infty & \infty & 1 & \infty \\ 4 & \infty & \infty & \infty \\ \infty & 2 & 9 & \infty \end{bmatrix} \end{array}$$

$$d = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$d_0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$d_0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

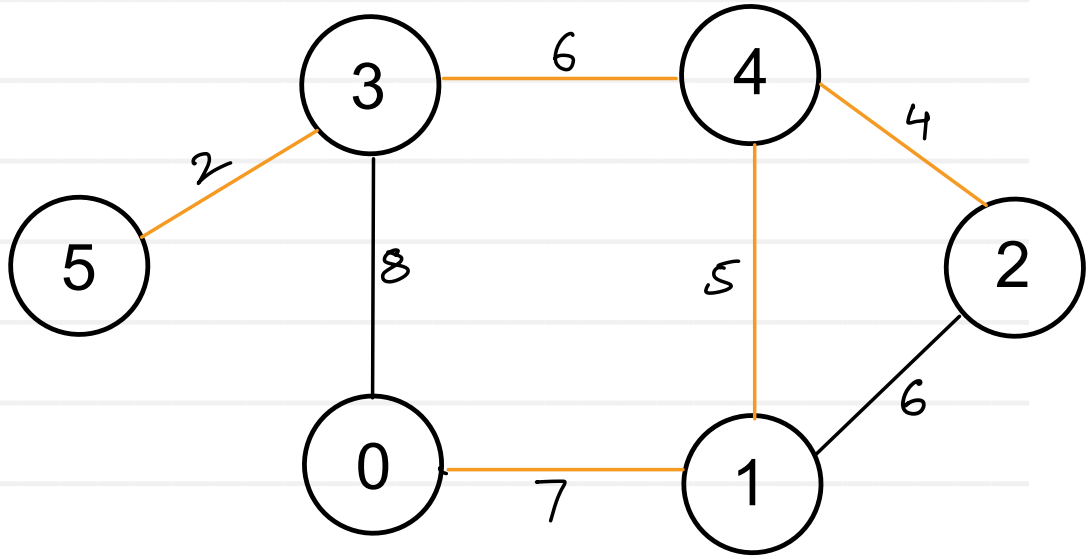
$$d_1 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$d_2 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$d_3 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

Kruskal's Algorithm

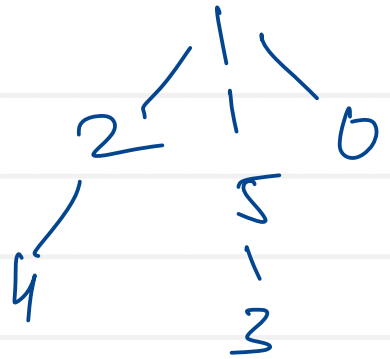
1. Sort all the edges in ascending order of their weight.
2. Pick the smallest edge.
   Check if it forms a cycle with the spanning tree formed so far.
   If cycle is not formed, include this edge.
   Else, discard it.
3. Repeat step 2 until there are (V-1) edges in the spanning tree.



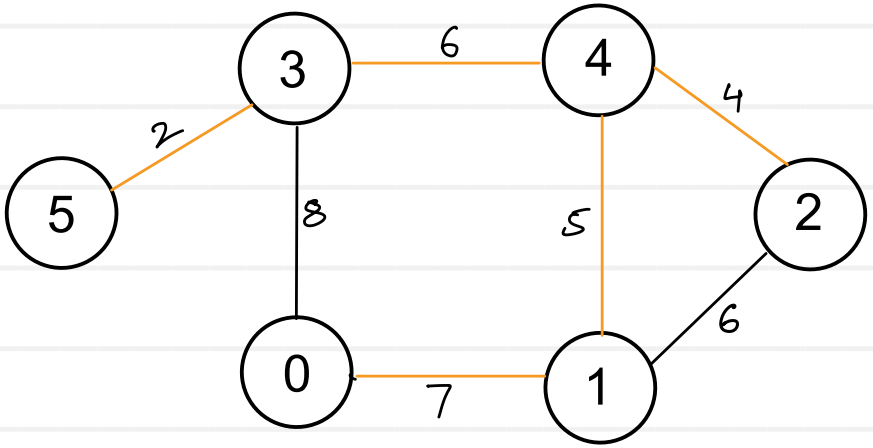
s	d	wt	
3	5	2	✓
4	2	4	✓
4	1	5	✓
1	2	6	✗
3	4	6	✓
0	1	7	✓
0	3	8	

Union Find Algorithm

1. Consider all vertices as disjoint sets (parent = -1).
2. For each edge in the graph / MST
   1. Find set(root) of first vertex.
   2. Find set(root) of second vertex.
   3. If both are in same set(same root), cycle is detected.
   4. Otherwise, merge(Union) both the sets i.e. add root of first set under second set



	0	1	2	3	4	5
Parents	1	-1	1	5	2	1



sr	dr	s	d	wt
3	5	3	5	2
4	2	4	2	4
2	1	4	1	5
1	1	1	2	6
5	1	3	4	6
0	1	0	1	7
		0	3	8

```
int find(int v, int parent[]){
    while(parent[v] != -1)
        v = parent[v];
    return v;
}

void union(int sr, int dr, int parent[]){
    parent[sr] = dr;
}
```

Graph applications

- Graph represents flow of computation/tasks. It is used for resource planning and scheduling. MST algorithms are used for resource conservation. DAG are used for scheduling in Spark or Tez.

- In OS, process and resources are treated as vertices and their usage is treated as edges. This resource allocation algorithm is used to detect deadlock.

- In social networking sites, each person is a vertex and their connection is an edge. In Facebook person search or friend suggestion algorithms use graph concepts.

- In world wide web, web pages are like vertices; while links represents edges. This concept can be used at multiple places.
  - Making sitemap
  - Downloading website or resources
  - Developing web crawlers
  - Google page-rank algorithm

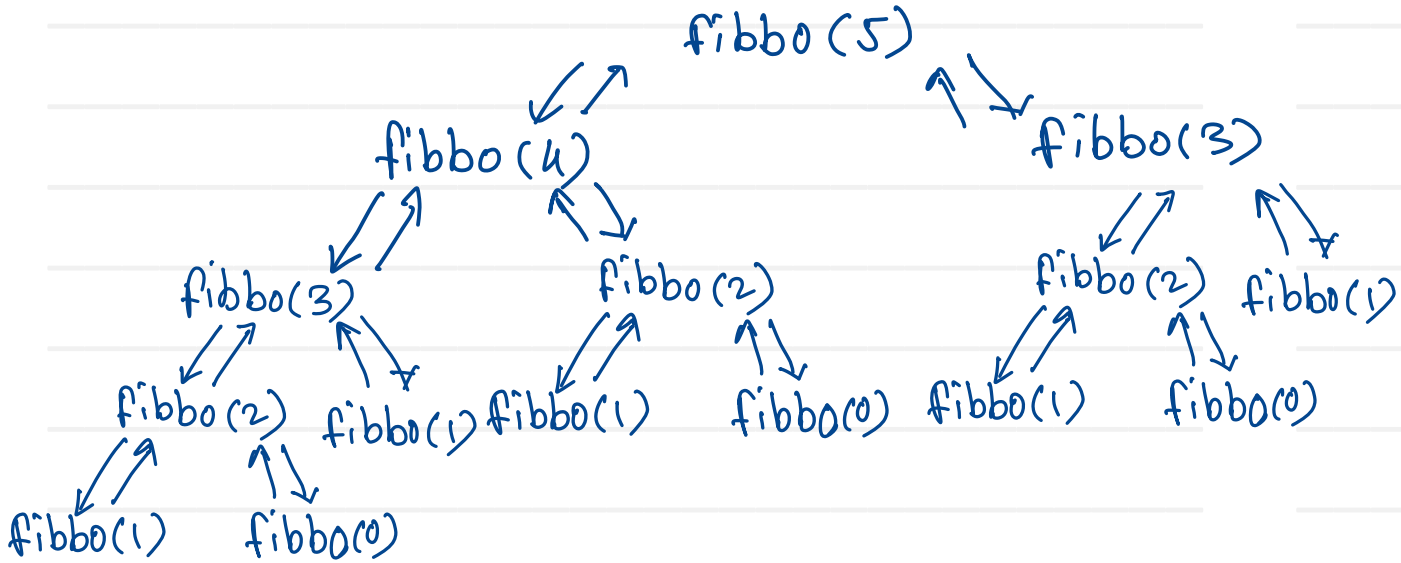
- Maps uses graphs for showing routes and finding shortest paths. Intersection of two (or more) roads is considered as vertex and the road connecting two vertices is considered to be an edge.

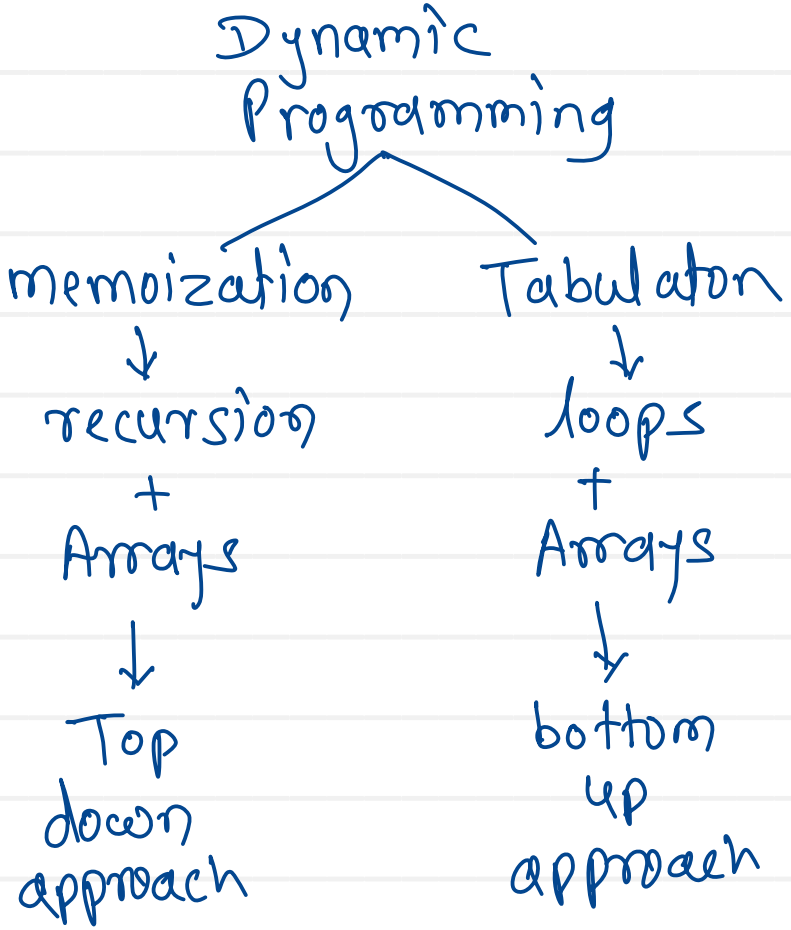
Problem solving technique : Greedy approach

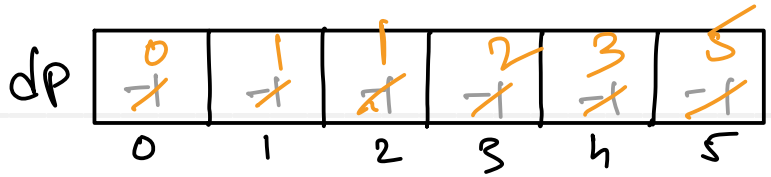
- A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.
- We can make choice that seems best at the moment and then solve the sub-problems that arise later.
- The choice made by a 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.

e.g. Greedy algorithm decides minimum number of coins to give while making change.
coins available : 50, 20, 10, 5, 2, 1

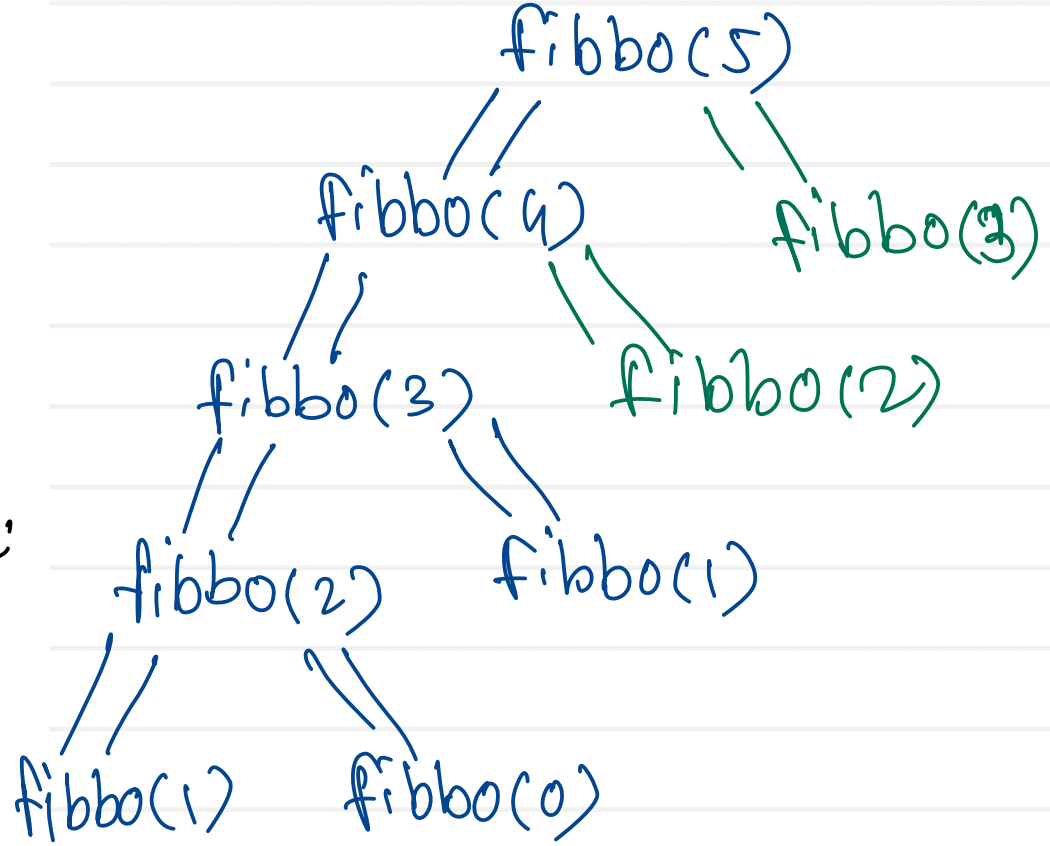
```
int fibbo(int n){
    if(n==0 || n==1)
        return 0;
    return fibbo(n-1)+fibbo(n-2);
}
```

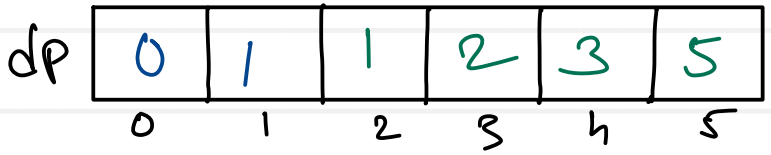







```
int fibbo(int n){
    if(n==0 || n==1){
        dp[n]=n;
        return dp[n];
    }
    if(dp[n]!=-1)
        return dp[n];
    return dp[n]=fibbo(n-1)+fibbo(n-2);
}
```






```
int fibbo(int n) {
    dp[0] = 0;
    dp[1] = 1;
    for( i=2; i<=5; i++)
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}
```

i
2
3
4
5

Recursion

- Function calling itself is called as recursive function.
- For each function call stack frame is created on the stack.
- Thus it needs more space as well as more time for execution.
- However recursive functions are easy to program.
- Typical divide and conquer problems are solved using recursion.
- For recursive functions two things are must
  - Recursive call (Explain process it terms of itself)
  - Terminating or base condition (Where to stop)

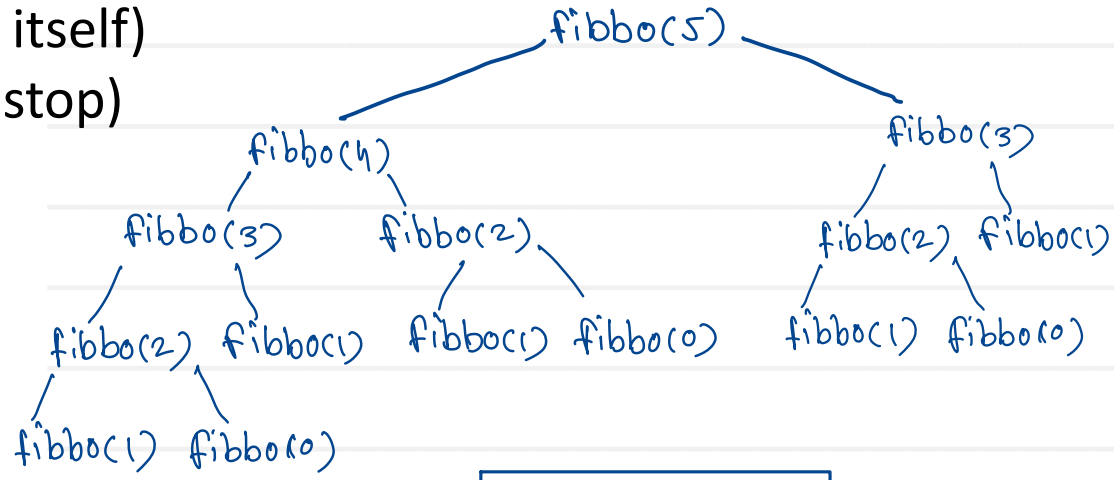
e.g. Fibonacci Series

- Recursive formula

  Tn = Tn-1 + Tn-2
- Terminating condition

  T1 = T2 = 1
- Overlapping sub-problem

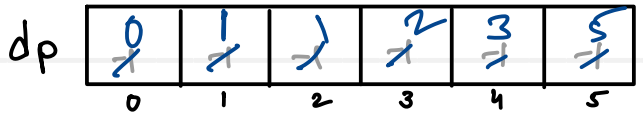
```
int fibbo(int n) {
  if(n==0 || n==1)
    return n;
  return fibbo(n-1)+fibbo(n-2);
}
```



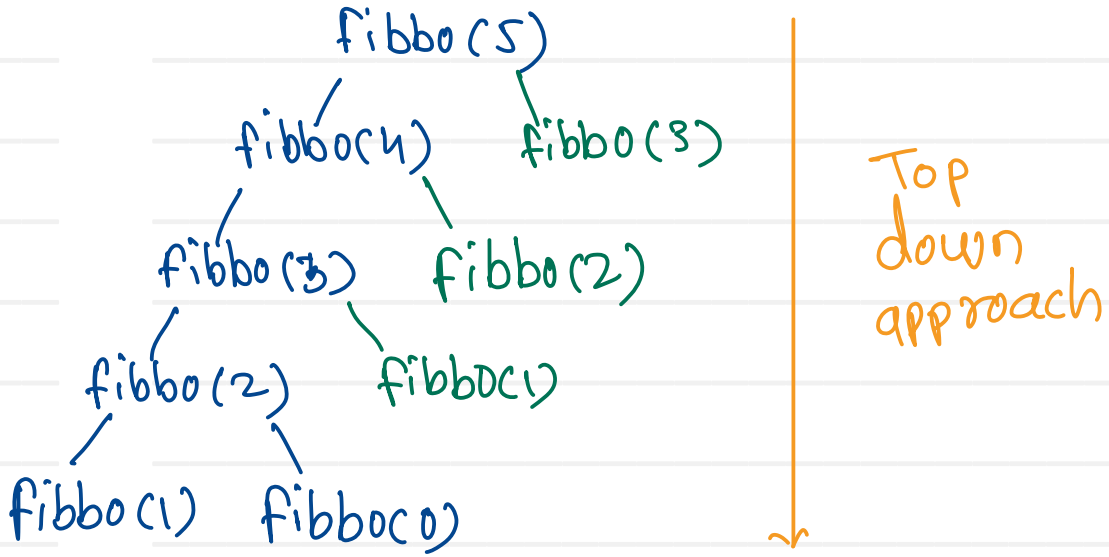
$$T(n) = 2^n$$ — exponential

Memoization

- It's based on the Latin word memorandum, meaning "to be remembered".
- Memoization is a technique used in computing to speed up programs.
- This is accomplished by memorizing the calculation results of processed input such as the results of function calls.
- If the same input or a function call with the same parameters is used, the previously stored results can be used again and unnecessary calculation are avoided.
- Need to rewrite recursive algorithm. Using simple arrays or map/dictionary.




```
int fibbo(int n) {
    if(n==0 || n==1){
        dp[n] = n;
        return dp[n];
    }
    if(dp[n] != -1)
        return dp[n];
    dp[n] = fibbo(n-1)+fibbo(n-2);
    return dp[n];
}
```

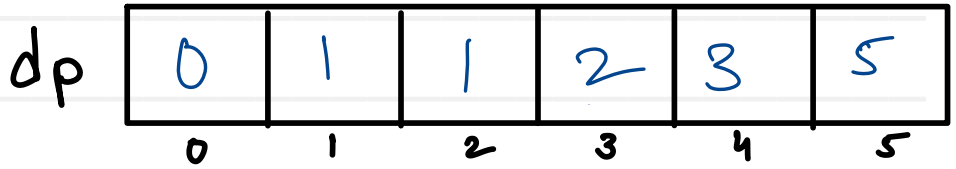


Dynamic Programming

- Dynamic programming is another optimization over recursion.
- Typical DP problem give choices (to select from) and ask for optimal result (maximum or minimum).
- Technically it can be used for the problems having two properties
  - Overlapping sub-problems
  - Optimal sub-structure
- To solve problem, we need to solve its sub-problems multiple times.
- Optimal solution of problem can be obtained using optimal solutions of its sub-problems.

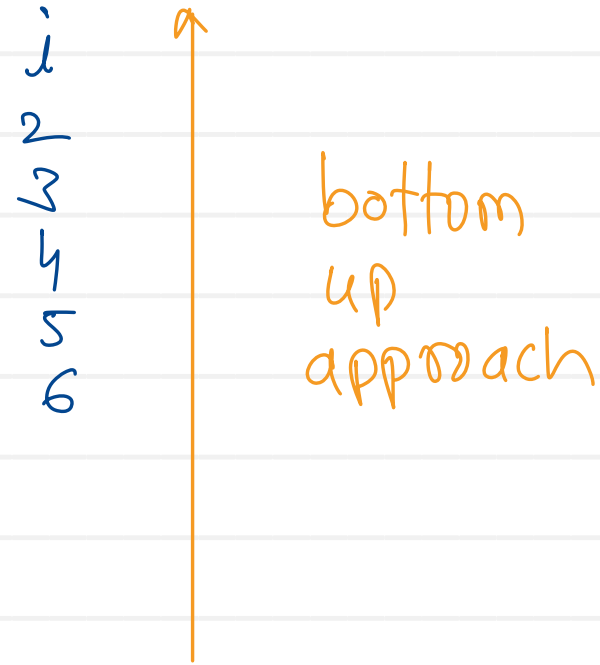
- Greedy algorithms pick optimal solution to local problem and never reconsider the choice done.
- DP algorithms solve the sub-problem in a iteration and improves upon it in subsequent iterations.

Dynamic programming




```
int fibbo( int n ) {
    dp[0] = 0;
    dp[1] = 1;
    for( i=2; i<=n; i++)
        dp[i] = dp[i-1] + dp[i-2];

    return dp[n];
}
```



Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com