



Sunbeam Institute of Information Technology
Pune and Karad

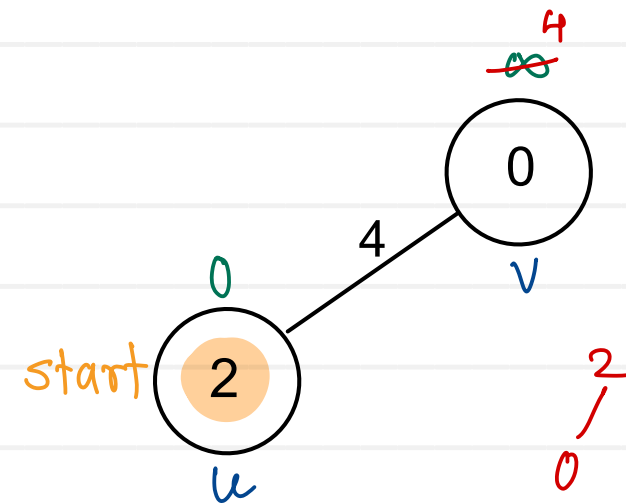
Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

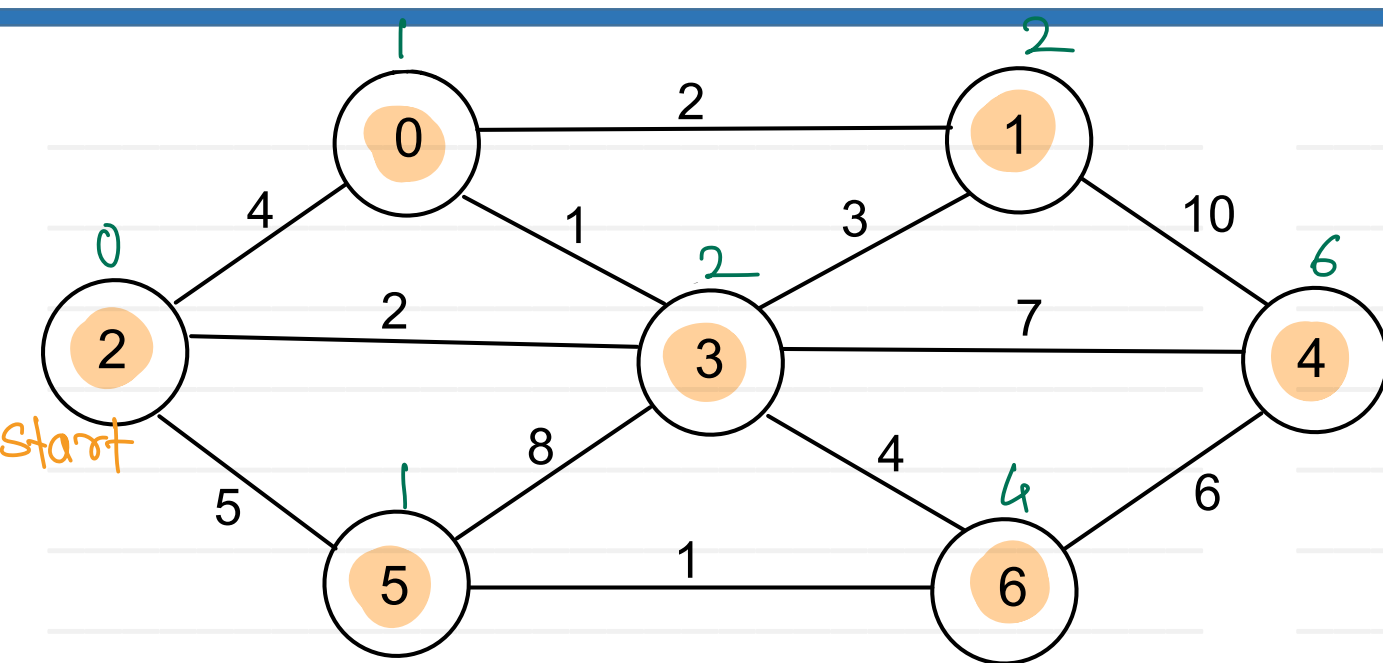
Prim's Algorithm

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.



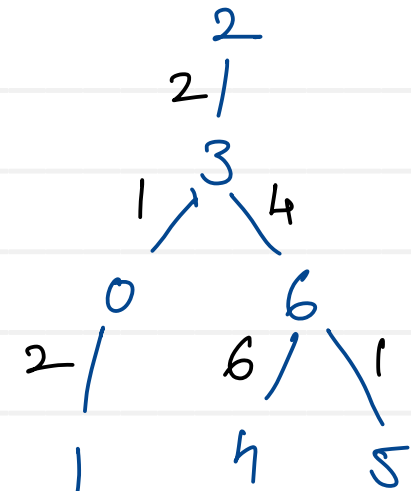
$$\text{if (adjMat}[u][v] < \text{key}[v])$$
$$\text{key}[v] = \text{adjMat}[u][v];$$

Prim's Algorithm



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

16



	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

Key

0	1	2	3	4	5	6

mst

0	1	2	3	4	5	6

minkey minkeyVertex i condition

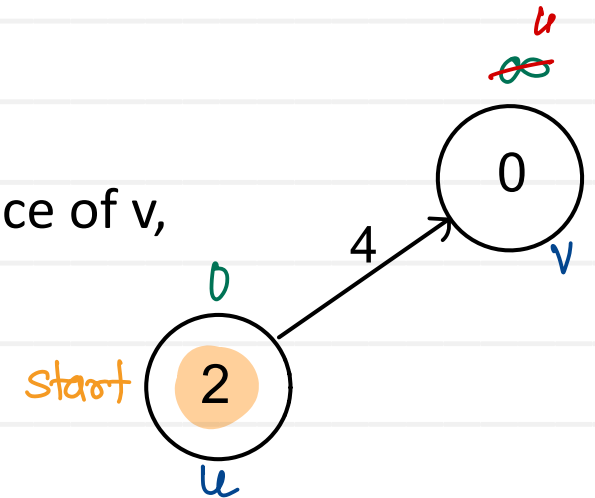
```
int findMinKeyVertex(int key[], boolean mst[]) {
    int minkey = ∞, minkeyVertex = -1;
    for (int i = 0; i < vertexCount; i++) {
        if (!mst[i] && key[i] < minkey) {
            minkey = key[i];
            minkeyVertex = i;
        }
    }
    return minkeyVertex;
}
```

Dijkstra's Algorithm ← single source shortest path 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 .

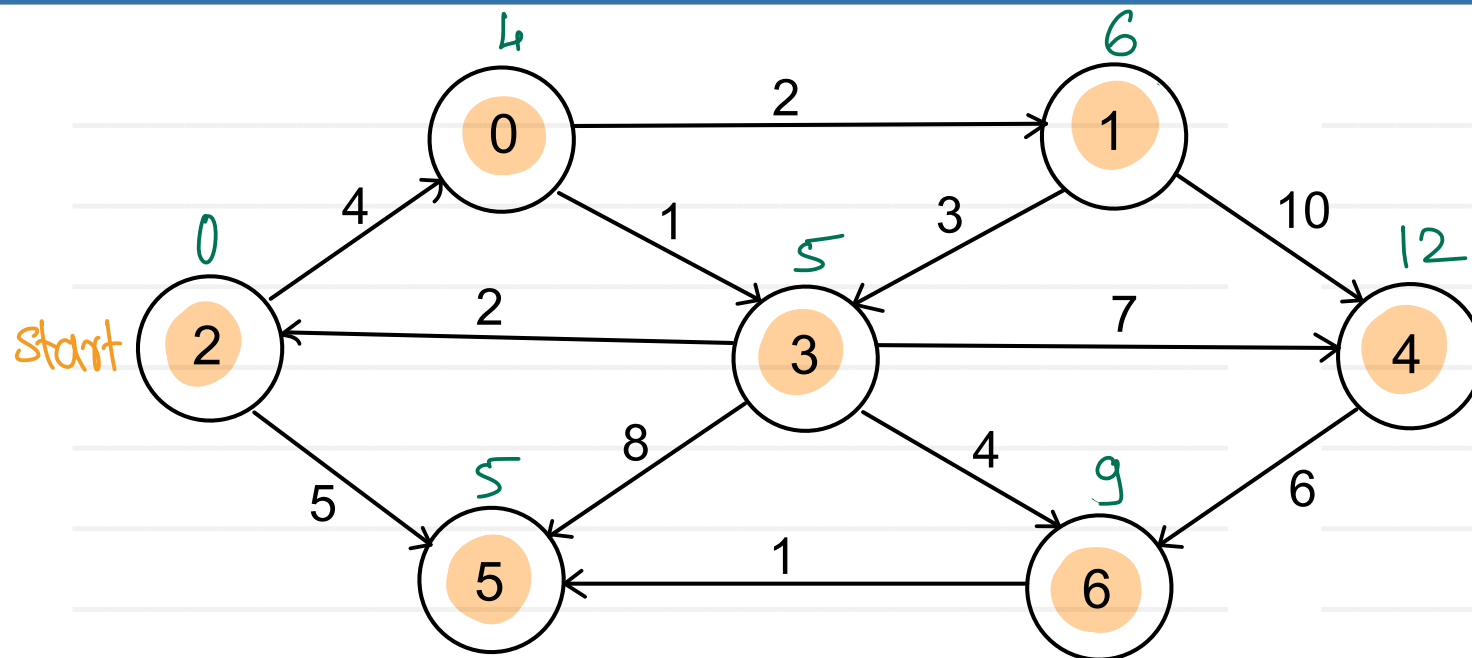
For each adjacent vertex v ,

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 $u-v$.



$$\text{if}(\text{dist}[u] + \text{adjMat}[u][v] < \text{dist}[v])$$
$$\text{dist}[v] = \text{dist}[u] + \text{adjMat}[u][v];$$

Dijkstra's Algorithm



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

	D
0	4
1	∞
2	0
3	∞
4	∞
5	5
6	∞

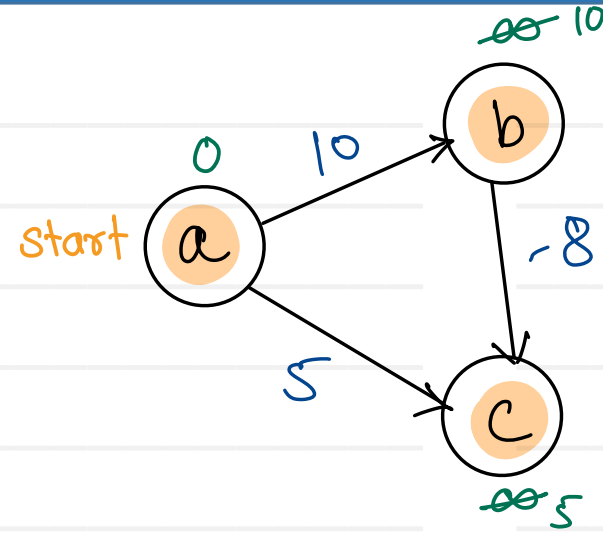
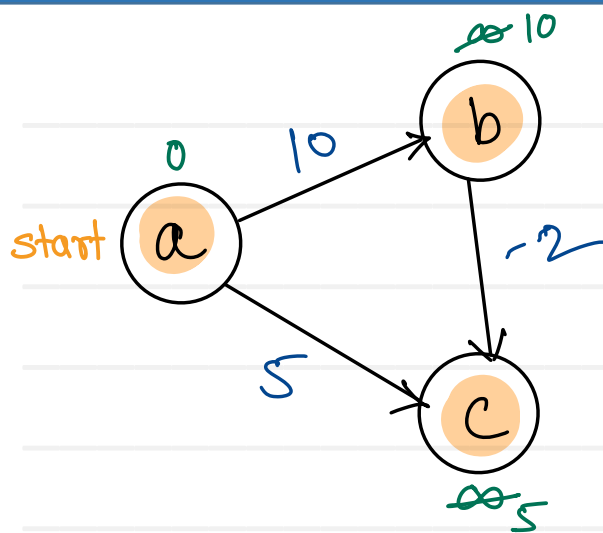
	D
0	4
1	6
2	0
3	5
4	∞
5	5
6	∞

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

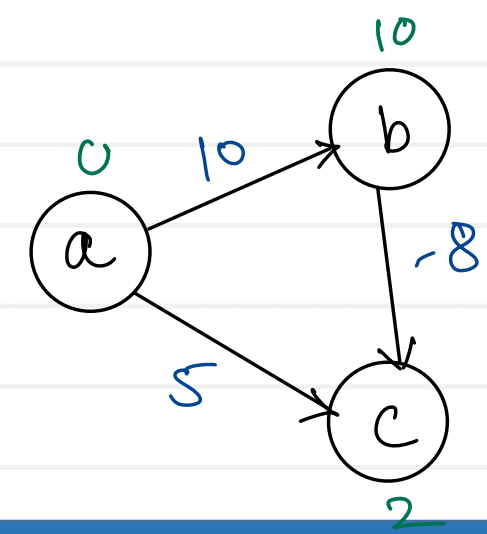
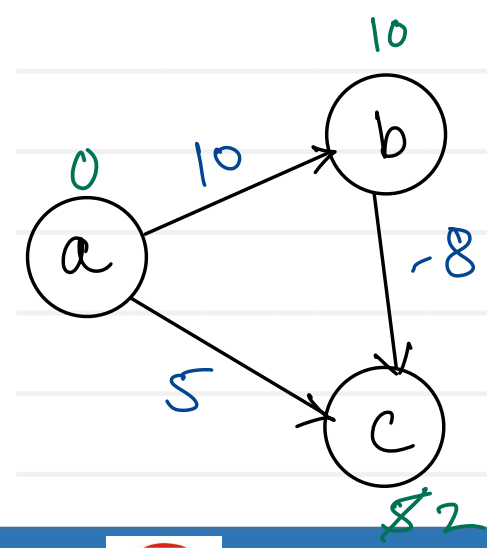
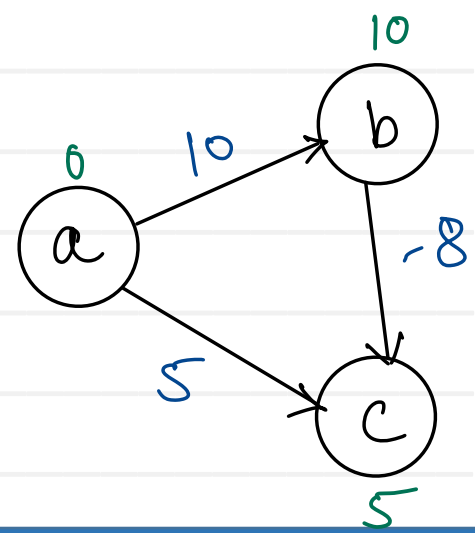
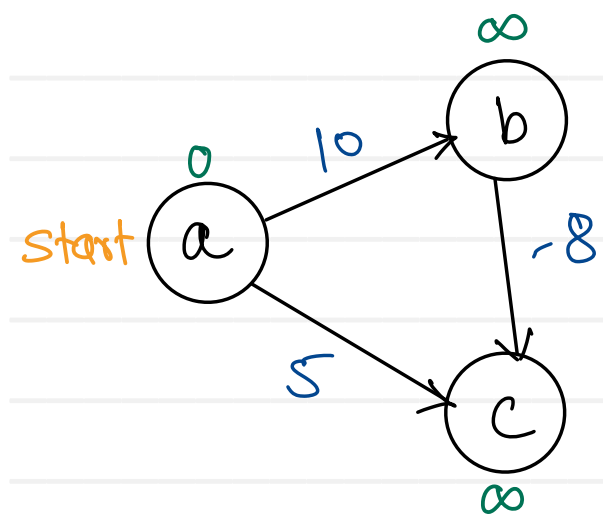
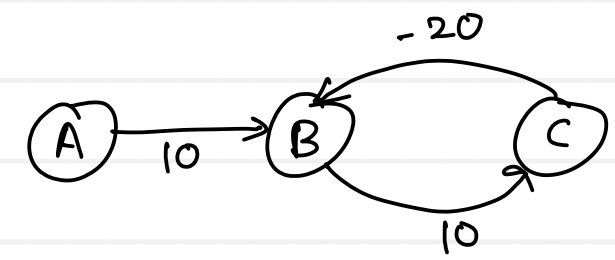
	D
0	4
1	6
2	0
3	5
4	12
5	5
6	9

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

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

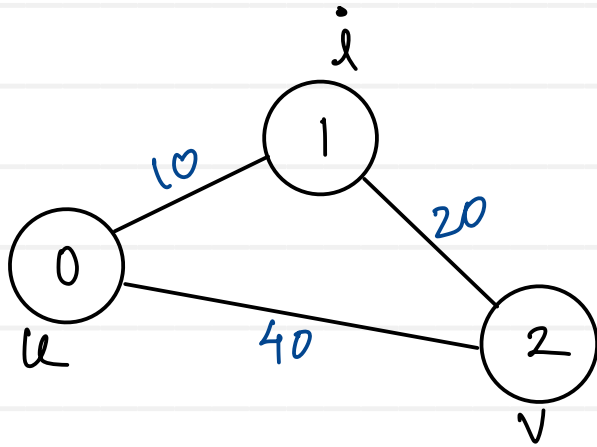


- Dijkstra's algorithm may fail if graph is having -ve edges



Floyd Warshall Algorithm

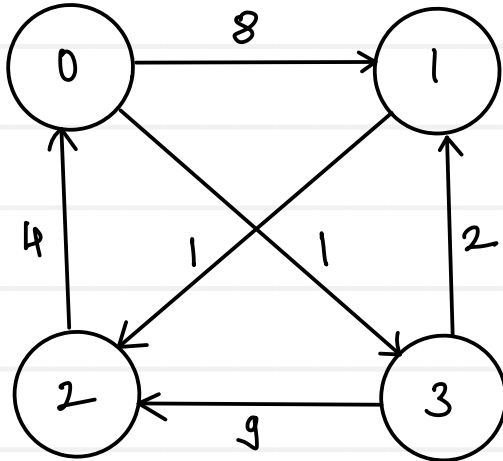
1. Create distance matrix to keep distance of every vertex from each vertex.
Initially assign it with weights of all edges among vertices
(i.e. adjacency matrix).
2. Consider each vertex (i) in between pair of any two vertices (u, v) and find the optimal distance between u & v considering intermediate vertex
i.e. $\text{dist}(u, v) = \text{dist}(u, i) + \text{dist}(i, v)$,
if $\text{dist}(u, i) + \text{dist}(i, v) < \text{dist}(u, v)$.



$\text{if} (\text{dist}[u][i] + \text{dist}[i][v] < \text{dist}[u][v])$
 $\text{dist}[u][v] = \text{dist}[u][i] + \text{dist}[i][v];$

Floyd Warshall Algorithm

$$\begin{matrix} 3-0 & + & 0-2 & & 3-2 \\ \infty & + & \infty & < & 9 \end{matrix}$$



	0	1	2	3
0	∞	8	∞	1
1	∞	∞	1	∞
2	4	∞	∞	∞
3	∞	2	9	∞

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

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

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

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

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

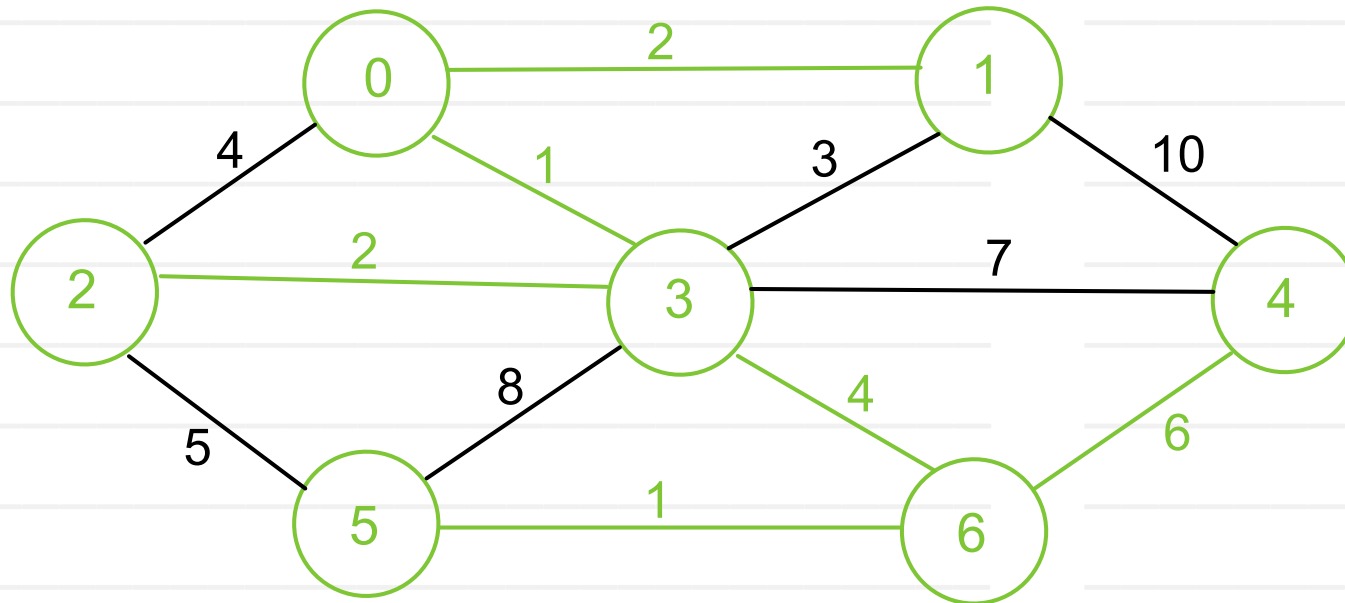
$$d_3 = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 0 & 3 & 4 & 1 \\ 1 & 5 & 0 & 1 & 6 \\ 2 & 4 & 7 & 0 & 5 \\ 3 & 7 & 2 & 3 & 0 \end{bmatrix}$$

```
int dist[vCount][vCount];
for( u=0 ; u < vCount ; u++ ) {
    for( v=0 ; v < vCount ; v++ )
        dist[u][v] = adjMat[u][v];
    dist[u][u] = 0;
}

for( i=0 ; i < vCount ; i++ ) {
    for( u=0 ; u < vCount ; u++ ) {
        for( v=0 ; v < vCount ; v++ ) {
            if( dist[u][i] + dist[i][v] < dist[u][v] )
                dist[u][v] = dist[u][i] + dist[i][v];
        }
    }
}
```

Kruskal's Algorithm

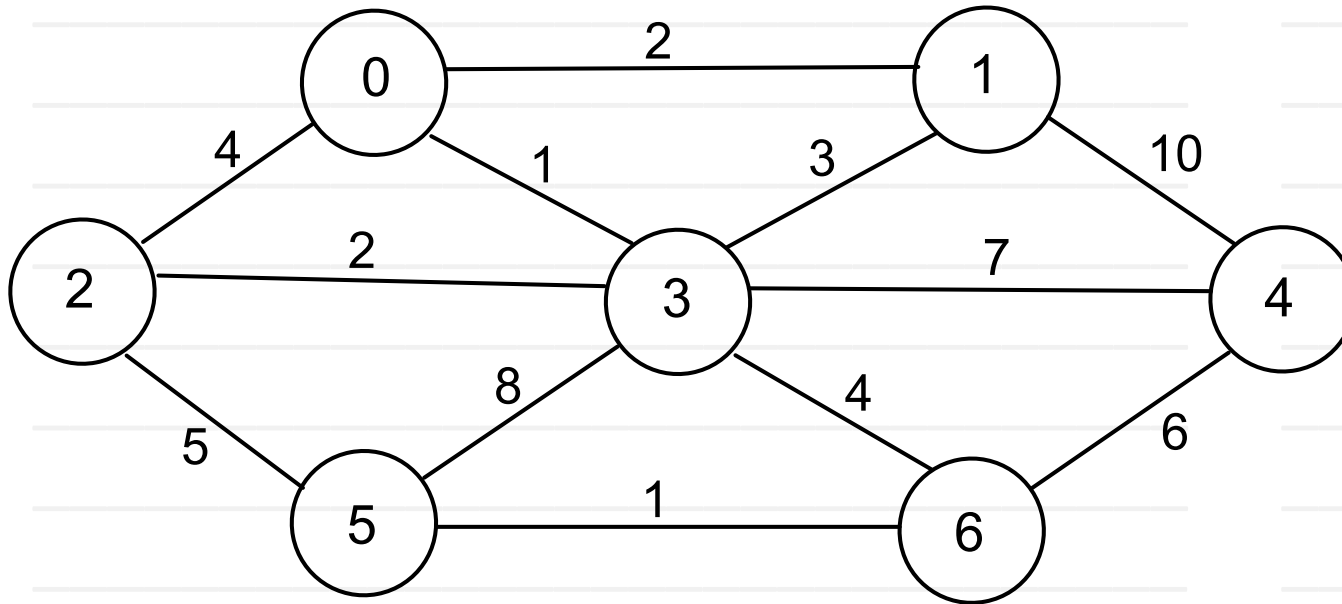
1. Sort all the edges in ascending order of their weight.
2. Pick the smallest edge.
Check if it forms a cycle with the spanning tree formed so far.
If cycle is not formed, include this edge.
Else, discard it.
3. Repeat step 2 until there are $(V-1)$ edges in the spanning tree.



S	d	wt
0	3	- 1 ✓
6	5	- 1 ✓
0	1	- 2 ✓
3	2	- 2 ✓
1	3	- 3 ✗
2	0	- 4 ✗
3	6	- 4 ✓
2	5	- 5 ✗
4	6	- 6 ✓
3	4	- 7
3	5	- 8
1	4	- 10

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

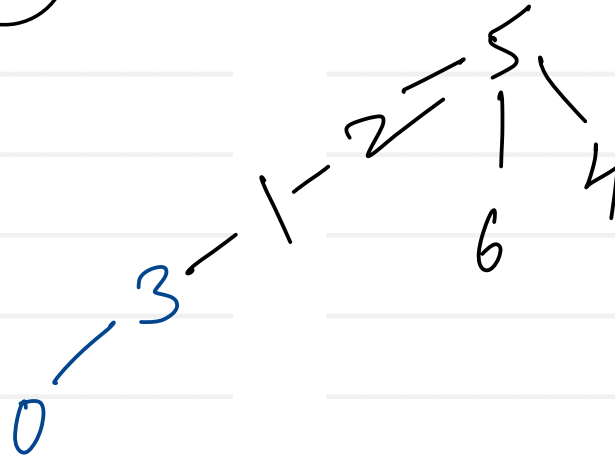
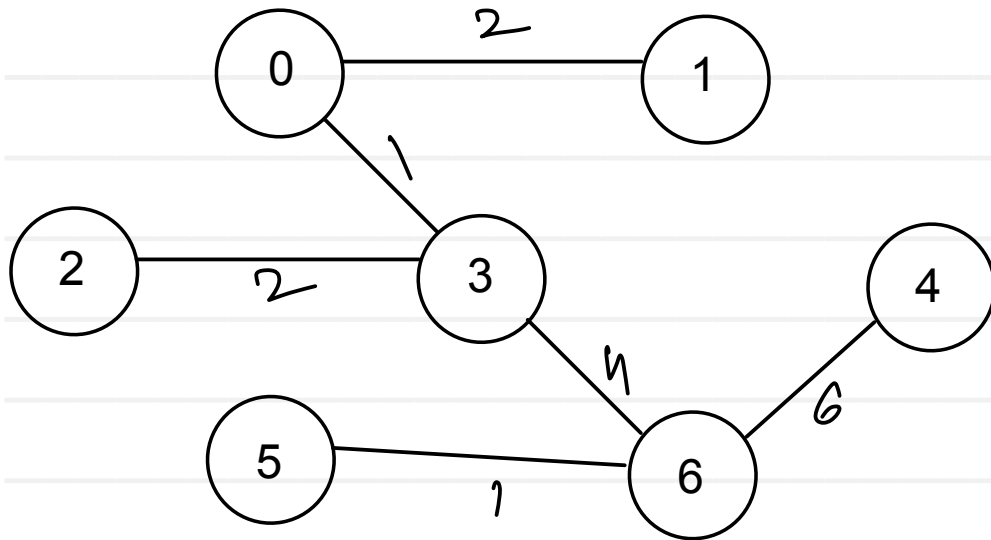


```
int find (int v, int parent[]) {
    while (parent[v] != -1)
        v = parent[v];
    return v;
}
```

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

Union Find Algorithm

	0	1	2	3	4	5	6
Parents	3	2	5	1	5	1	5



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

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.

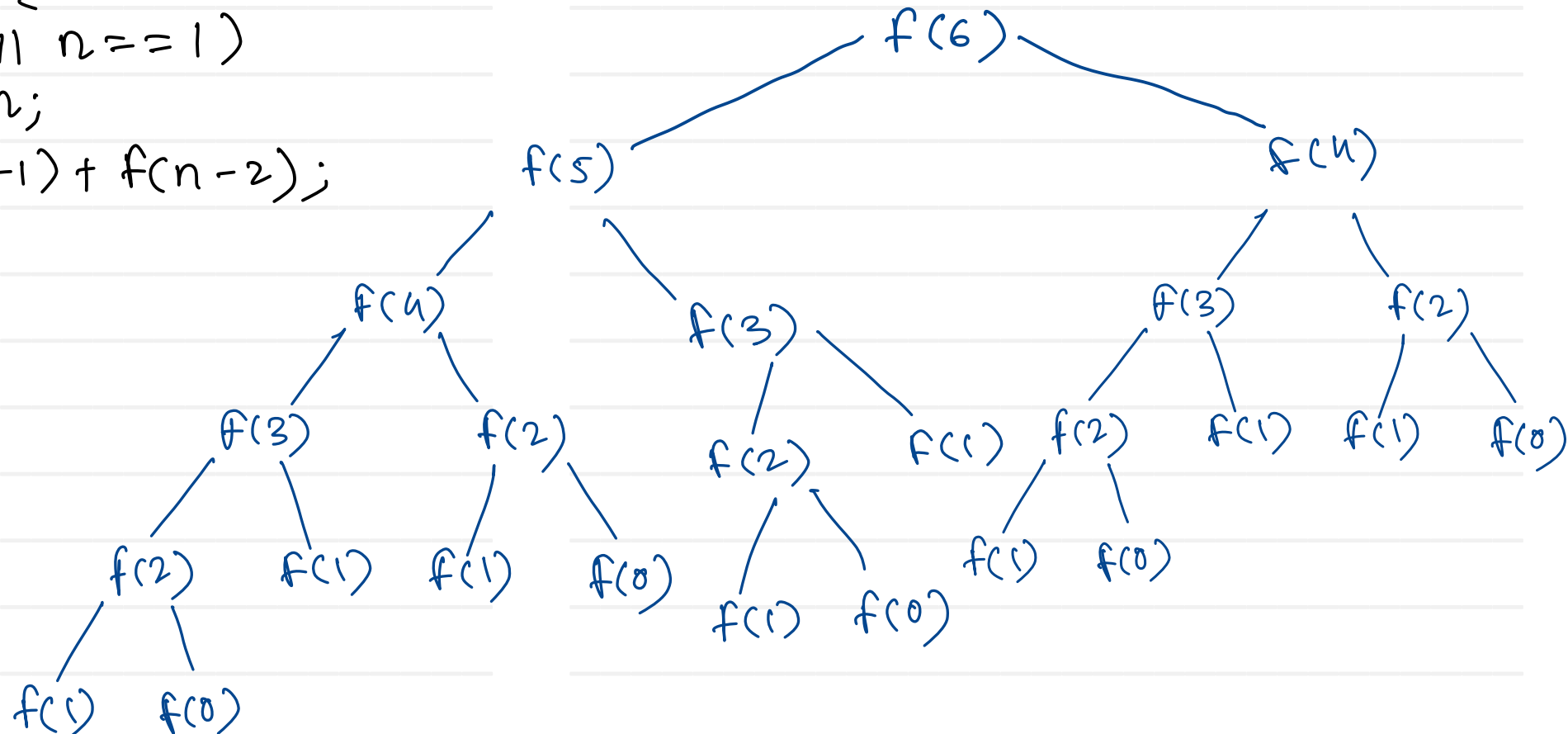
coins available : 50, 20, 10, 5, 2, 1

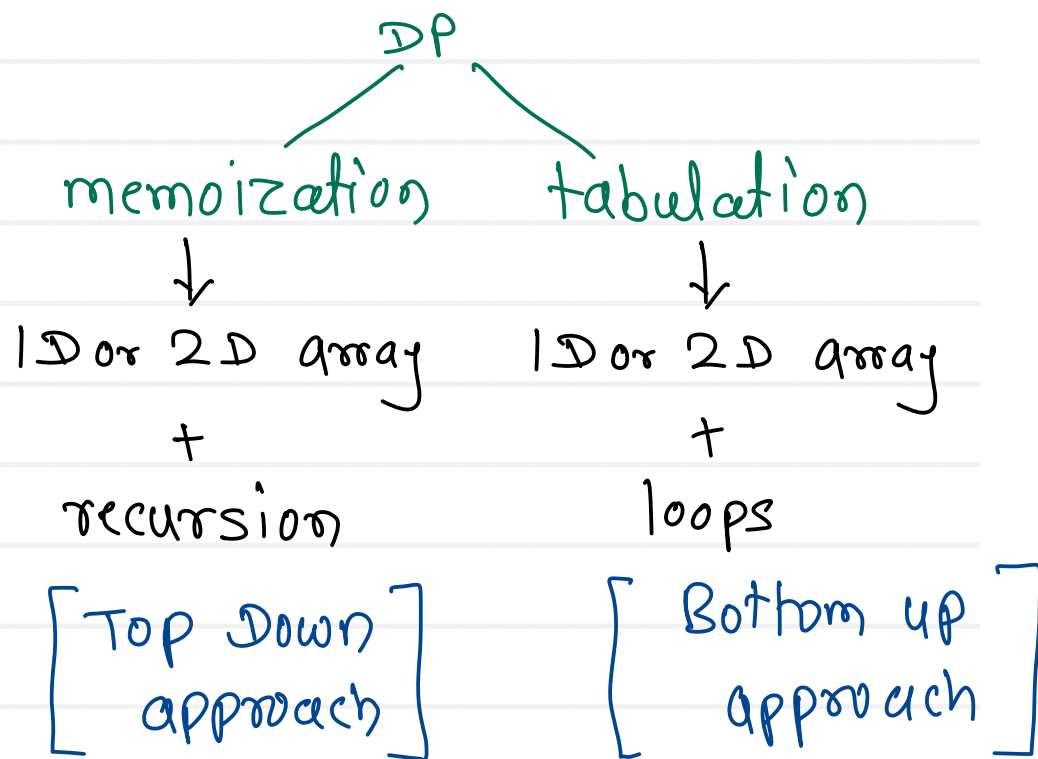
```

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

```

3





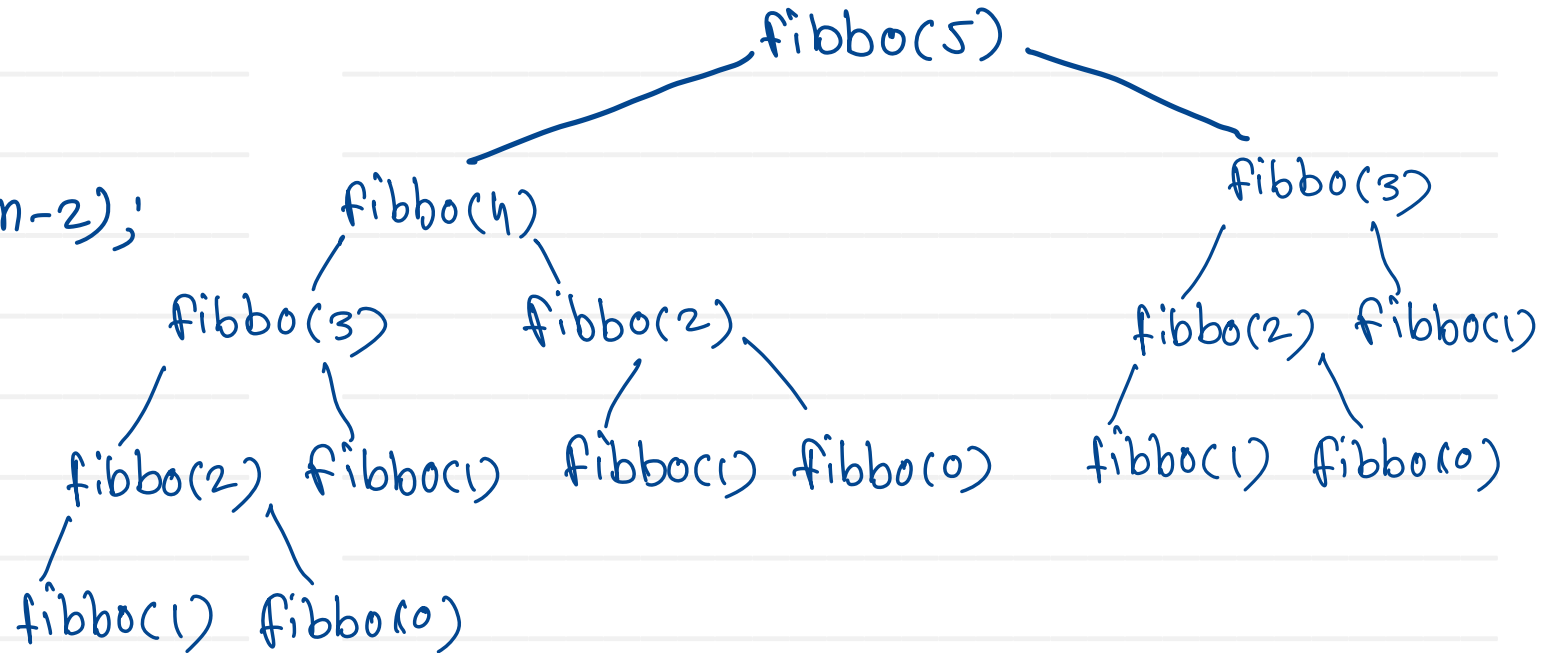
- Function calling itself is called as recursive function.
- For each function call stack frame is created on the stack.
- Thus it needs more space as well as more time for execution.
- However recursive functions are easy to program.
- Typical divide and conquer problems are solved using recursion.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

e.g. Fibonacci Series

- Recursive formula
$$T_n = T_{n-1} + T_{n-2}$$
- Terminating condition
$$T_1 = T_2 = 1$$
- Overlapping sub-problem

```

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



$$T(n) = O(2^n) \quad \text{— exponential}$$

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.

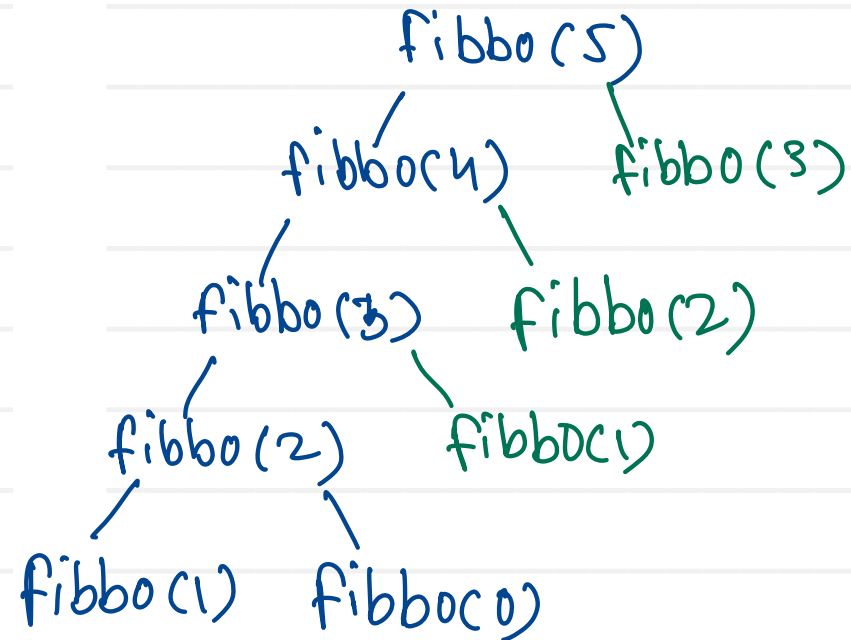
dp

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

```

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];
}

```



Top
down
approach

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

dp	0	1	1	2	3	5
	0	1	2	3	4	5

```
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];  
}
```

i
2
3
4
5
6

bottom
up
approach



Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com