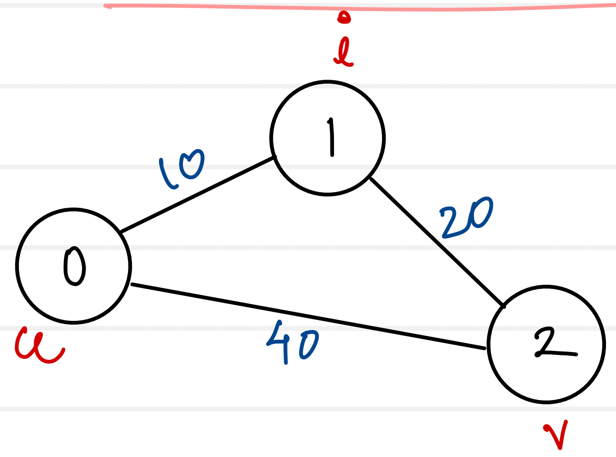
Floyd Warshall Algorithm

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).

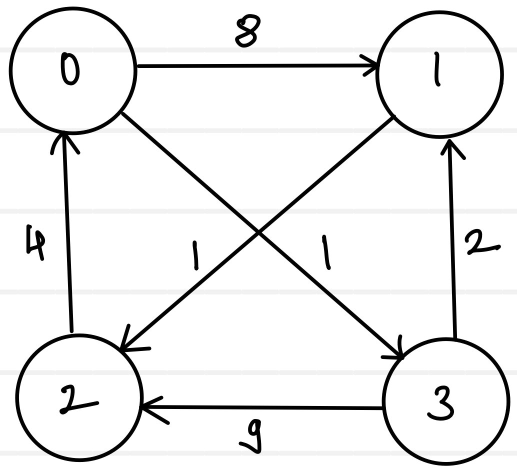
2. Consider each vertex (i) in between pair of any two vertices (u, v) and
   find the optimal distance between u & v considering intermediate vertex
   i.e. dist(u,v) = dist(u,i) + dist(i,v),
   if dist(u,i) + dist(i,v) < dist(u,v).



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

```
if( 10 + 20 < 40)
   dist[0][2] = 30
```

Floyd Warshall Algorithm



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

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

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

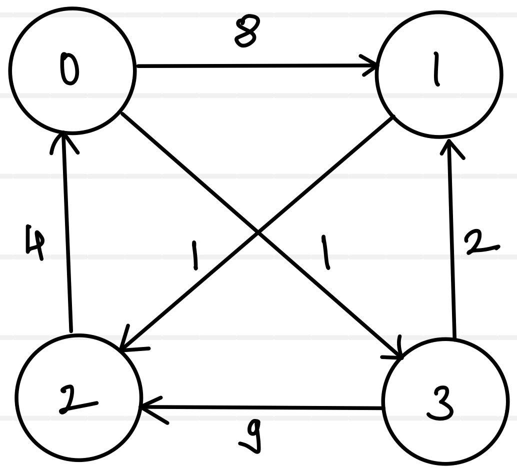
$$d_0 = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 8 & \infty & 1 \\ 1 & \infty & 0 & 1 & \infty \\ 2 & 4 & 12 & 0 & 5 \\ 3 & \infty & 2 & 9 & 0 \end{array}$$

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

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

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

Floyd Warshall Algorithm



itr = v * v * v

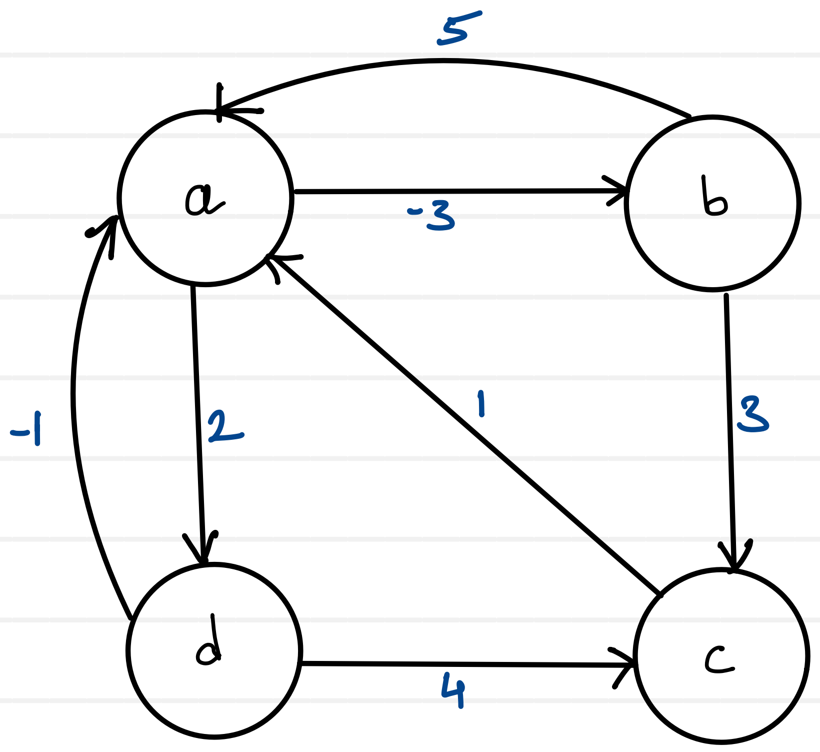
Time $\propto v^3$

$\boxed{T(v) = O(v^3)}$

```
int dist[][] = new int[vCount][vCount];
for(int u=0; u < vCount ; u++) {
        for(int v=0; v< vCount ; v++) {
            dist[u][v] = adjMat[u][v];
        }
        dist[u][u] = 0;
}
v times → for( int i=0; i< vCount ; i++) {
v times   → for(int u=0; u< vCount ; u++) {
v times       → for( int v=0; v< vCount ; v++){
                    if( dist[u][i] + dist[i][v] < dist[u][v])
                        dist[u][v] = dist[u][i] + dist[i][v];
                }
          }
}
```

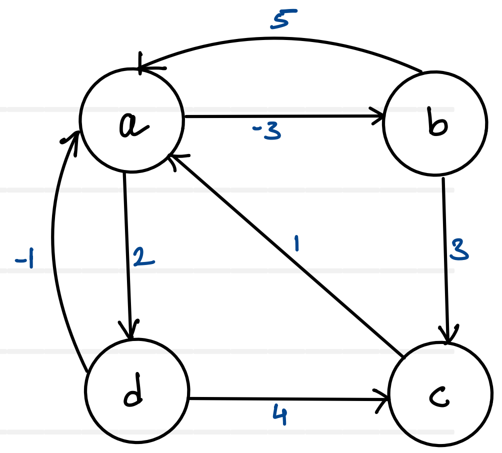
Johnson's Algorithm

- Time complexity of Floyd Warshall is O(V^3).
- Applying Dijkstra's algorithm on V vertices will cause time complexity O(V * V log V). This is faster than Floyd Warshall.
- However Dijkstra's algorithm can't work with negative edges.
- Johnson use Bellman ford to re weight all edges in graph to remove negative edges. Then apply Dijkstra's algorithm to all vertices to calculate shortest distance.
- Finally re weight distance to consider original edge weights.
- Time complexity of the algorithm:
  O(VE + V^2 log V)

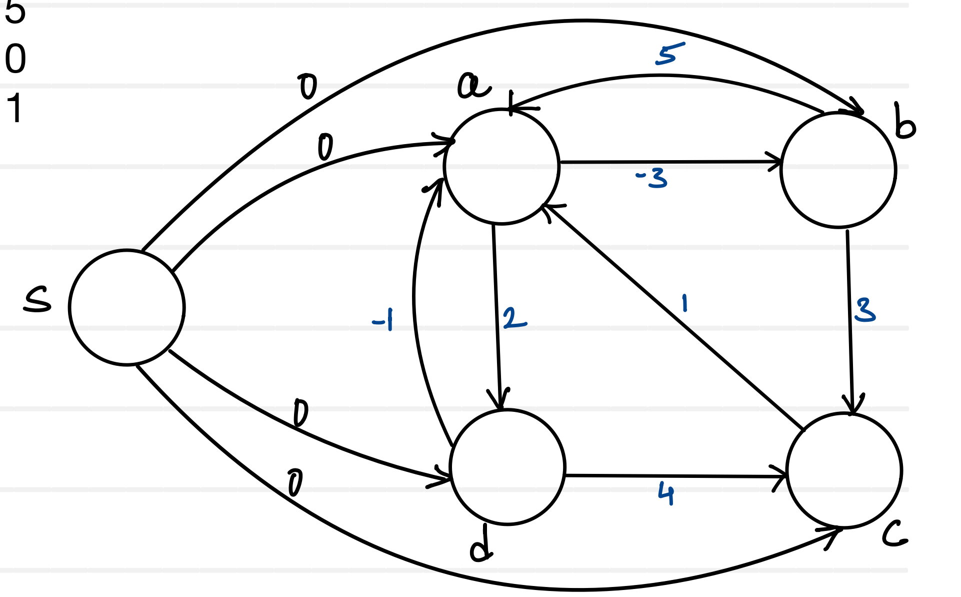


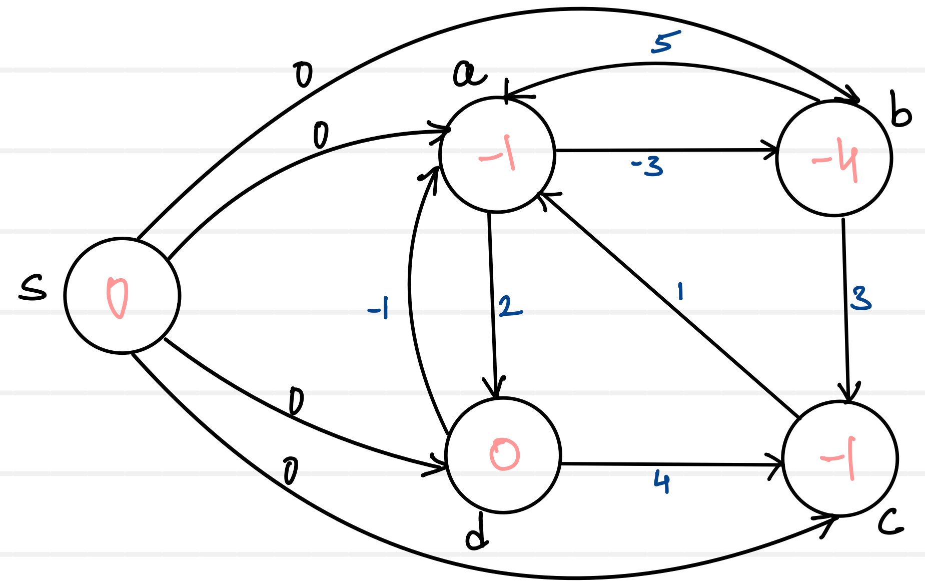
Johnson's Algorithm

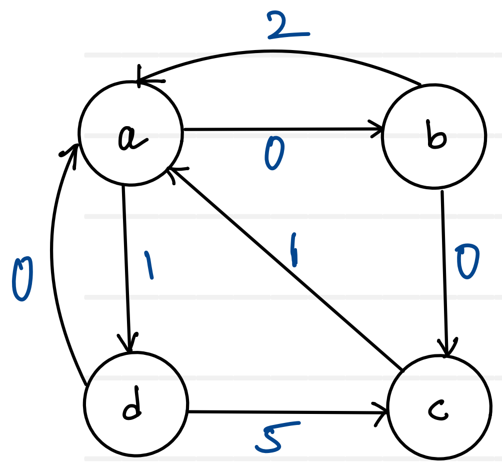
- Add a vertex (s) into a graph and add edges from it all other vertices, with weight 0.
- Find shortest distance of all vertices from (s) using Bellman Ford algorithm.
  a = -1, b = -4, c = -1, d = 0 and s = 0
- Re weight all edges (u, v) in the graph, so that they become non negative.
  weight(u,v) = weight(u,v) + d(u) - d(v)
- Apply Dijkstra's algorithm on every vertex of graph and find distances.
- Re weight all distances against original graph
  dist(u,v) = dist(u,v) + d(v) - d(u)



w(a, b) = 0
w(b, a) = 2
w(b, c) = 0
w(c, a) = 1
w(d, c) = 5
w(d, a) = 0
w(a, d) = 1

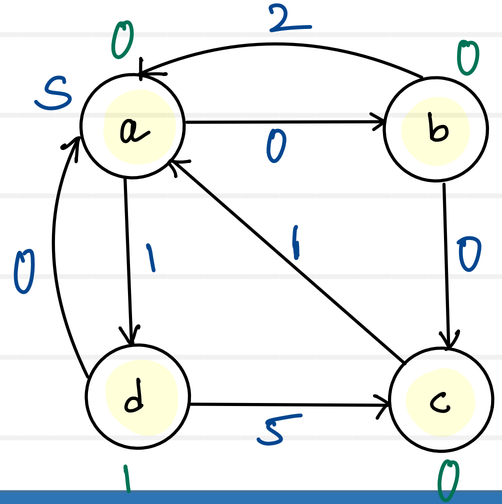






$$\text{dist}[a,b] = 0 + -4 - -1$$
$$= -3$$

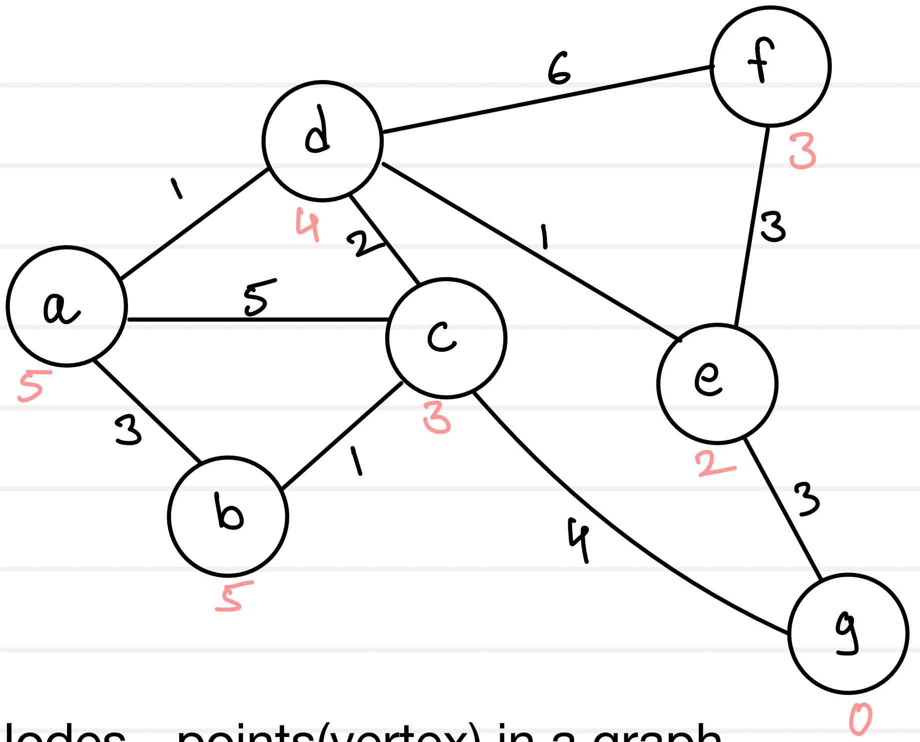
$$\text{dist}[a,c] = 0 + -1 - -1$$
$$= 0$$



	a	b	c	d
a	0	0	0	1
b				
c				
d				

A* Algorithm

- Popular technique used in path finding and graph traversals.
- A* algorithm combines
  1. Dijkstra's algorithm
  2. Greedy ~~Best~~ Breadth First Approach
- It considers both:
  - distance already traveled from the start
  - estimate of the remaining distance to the goal / targe
- This combination helps A* to make informed decisions about which path to explore next.
- This makes A* both efficient and accurate

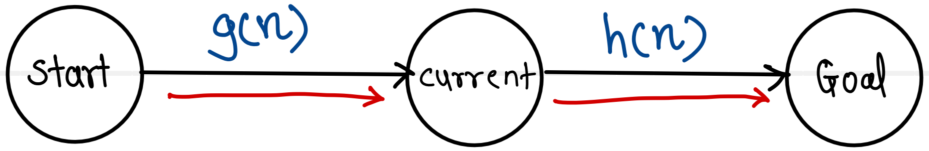


- Nodes - points(vertex) in a graph
- Edges - links/connections between nodes
- Path cost - actual cost of moving from one node to another (wt)
- Heuristic - estimated cost from any node to the goal

A* Algorithm

- Below three functions work together to guide the search process towards the most promising paths
  1. f(n) - total estimated cost
  2. g(n) - path cost / traveled distance from start to current node
  3. h(n) - estimated cost from current to goal / target

$$f(n) = g(n) + h(n)$$



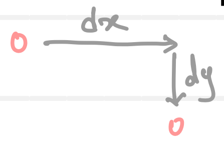
g(n) : known distance from start to current node

$$g(n_K) = \sum_{i=0}^{K} w(n_i)$$

h(n) : heuristic function
– estimated cost from current to goal vertex.

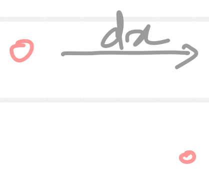
- common heuristic functions:

coordinates of current node - (x_1, y_1)
coordinates of goal node - (x_2, y_2)



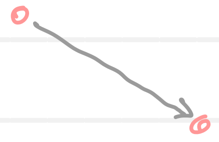
1. Manhattan distance

$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$



2. Diagonal distance

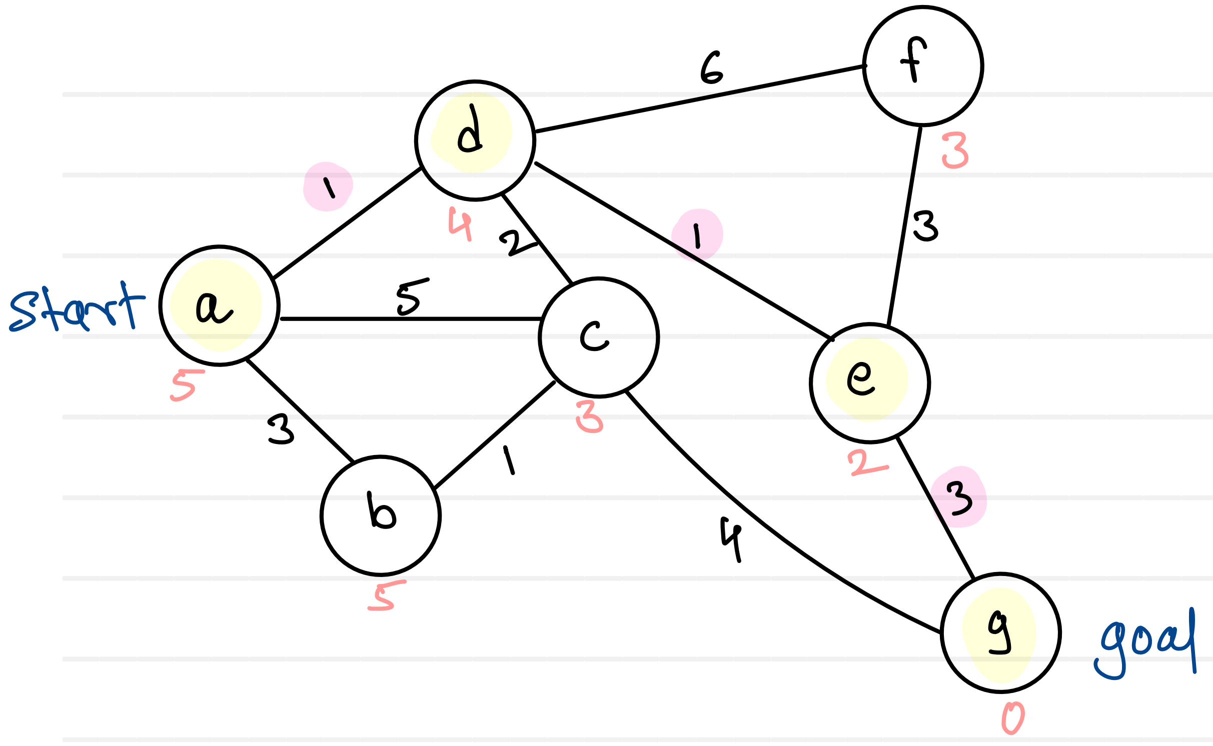
$$h(n) = MAX(|x_1 - x_2|, |y_1 - y_2|)$$



3. Euclidean distance

$$h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

A* Algorithm



	g(n)	h(n)	f(n)
a	0	5	5
b	3	5	8
c	3	3	6
d	1	4	5
e	2	2	4
f	5	3	8
g	5	0	5

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.

  (Directed Acyclic Graph)

- 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

Recursion

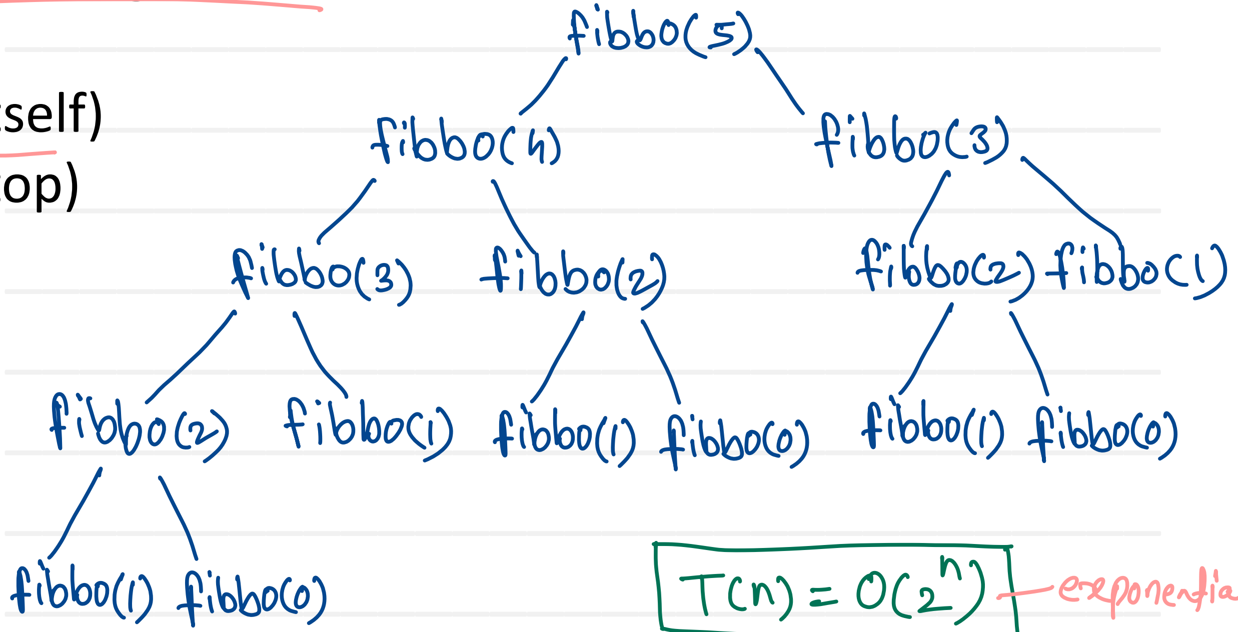
- Function calling itself is called as recursive function.
- For each function call stack frame is created on the stack.
- Thus it needs more space as well as more time for execution.
- However recursive functions are easy to program.
- Typical divide and conquer problems are solved using recursion.
- 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);
}
```



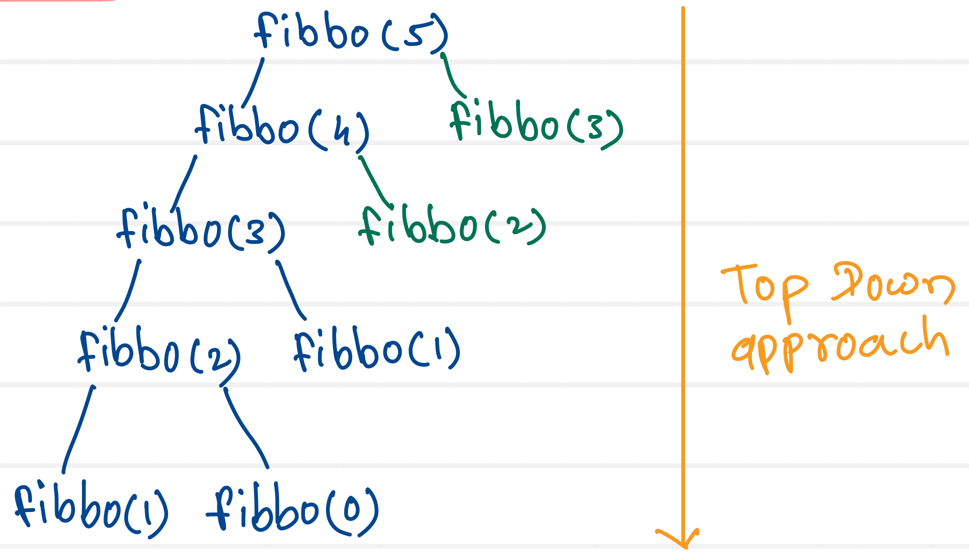
$\boxed{T(n) = O(2^n)}$ — exponential

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.

dp	0	1	1	2	3	5
	~~-1~~	~~-1~~	~~-1~~	~~-1~~	~~-1~~	~~-1~~
	0	1	2	3	4	5

```
int fibbo(int n){
    if(n==0 || n==1)
        return 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

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

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

    return dp[n];
}
```

$\uparrow$ bottom-up approach

Greedy

1) Kruskal's MST
2) Dijkstra's SPT
3) Prim's MST

4) A* Algorithm

Dynamic programming

1) Bellman Ford (1D array)
2) Floyd Warshall (2D array)

3) Johnson's Algo

SUNBEAM

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com