Sunbeam Institute of Information Technology
Pune and Karad

Data structures and Algorithms

Trainer - Devendra Dhande

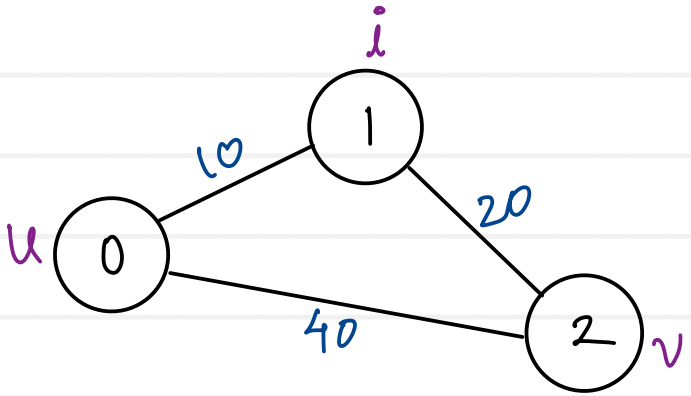
Email – devendra.dhande@sunbeaminfo.com

Floyd Warshall Algorithm

(All pair shortest path Algo)

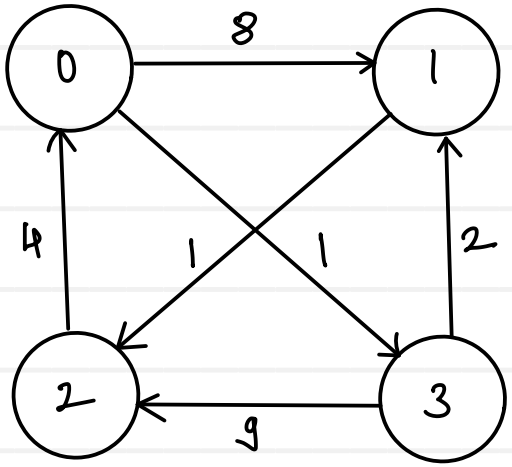
1. Create distance matrix to keep distance of every vertex from each vertex.
   Initially assign it with weights of all edges among vertices
   (i.e. adjacency matrix).

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



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

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

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

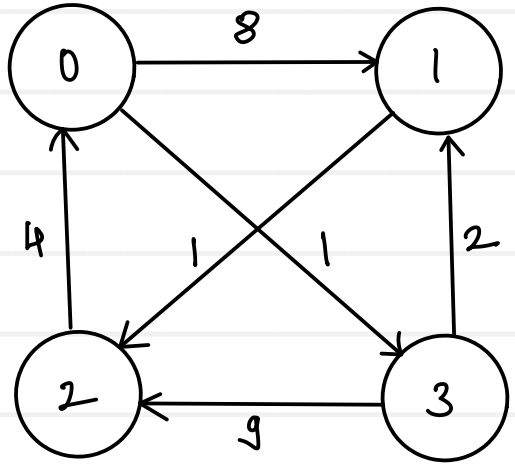
$$d_0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

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

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

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

Floyd Warshall Algorithm




```
int dist[][] = new int[vCount][vCount];   <- AS
for(int u=0 ; u < vCount ; u++){
    for(int v= 0; v < vCount ; v++){
        dist[u][v] = adjMat[u][v];
    }
    dist[u][u] = 0;
}
vtimes -> 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];
    }
  }
}
```

jtr = v * v * v

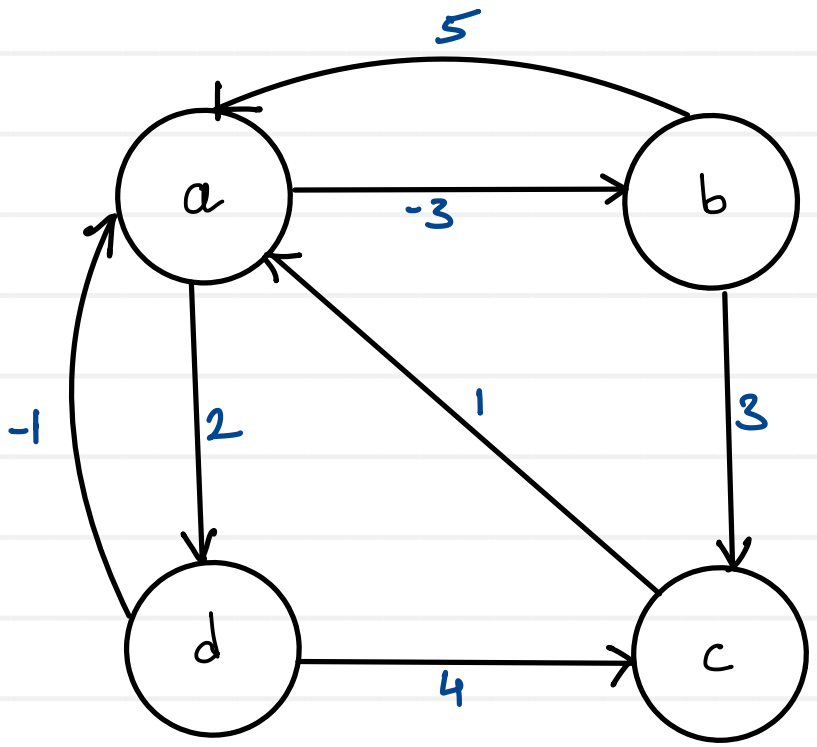
Time $\propto v^3$

$T(v) = O(v^3)$

$S(v) = O(v^2)$

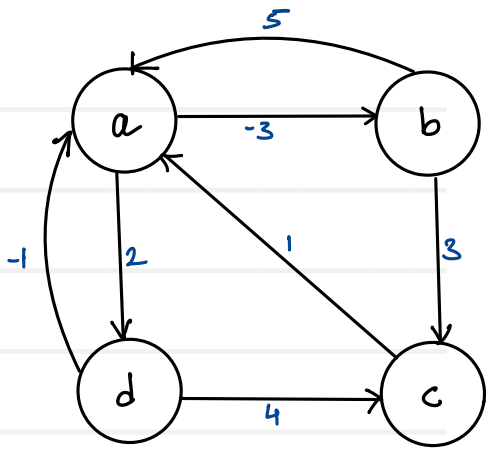
Johnson's Algorithm

- 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 uses 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)

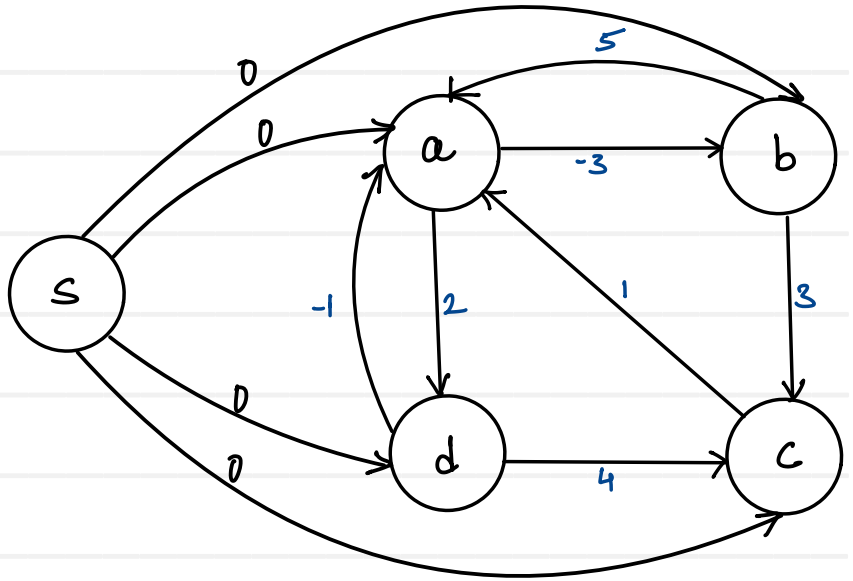


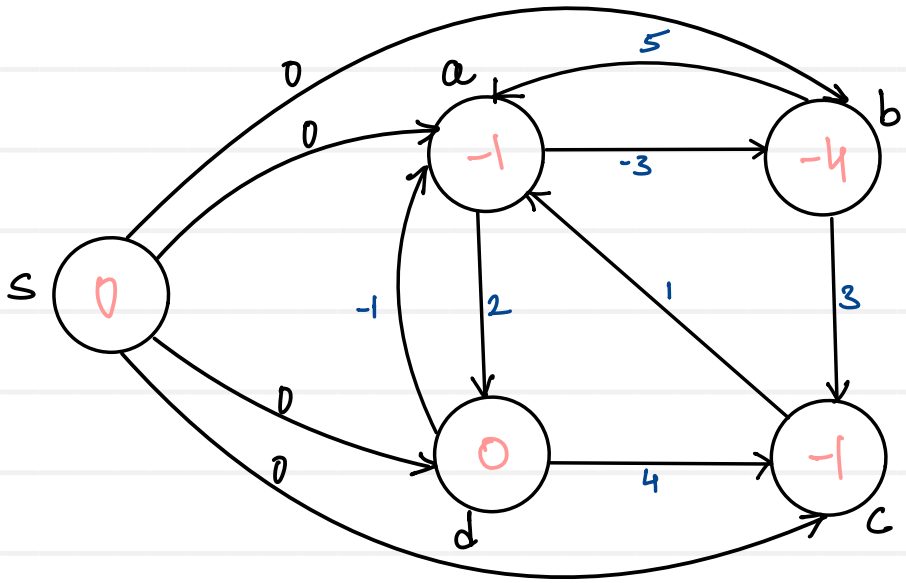
Johnson's Algorithm

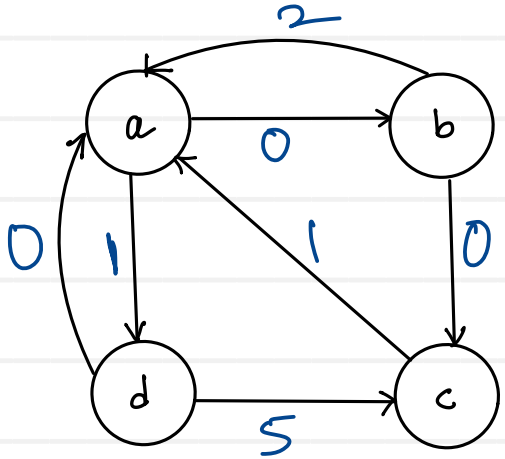
- Add a vertex (s) into a graph and add edges from it all other vertices, with weight 0.
- Find shortest distance of all vertices from (s) using Bellman Ford algorithm.
  a = -1, b = -4, c = -1, d = 0 and s = 0
- Re weight all edges (u, v) in the graph, so that they become non negative.
  weight(u,v)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







weight(u,v) = weight(u,v) + d(u) - d(v)

w(a,b) = -3 + -1 - -4 = -4 + 4 = 0

w(b,a) = 5 + -4 - -1 = 6 - 4 = 2

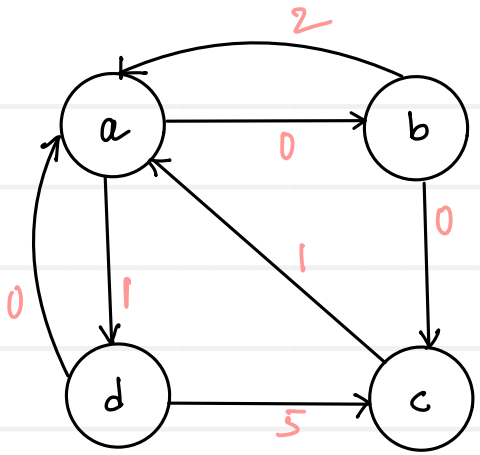
w(b,c) = 3 + -4 - -1 = 0

w(c,a) = 1 + -1 - -1 = 2 - 1 = 1

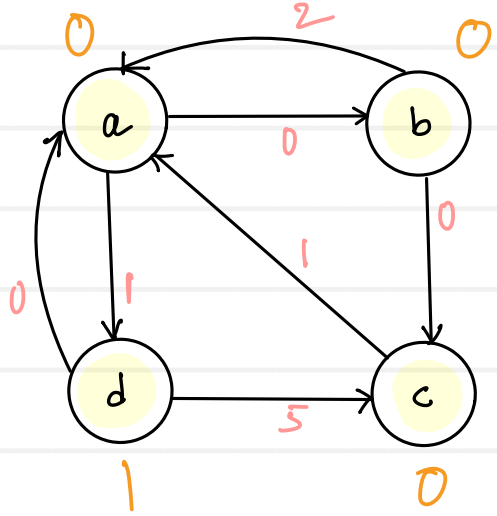
w(d,c) = 4 + 0 - -1 = 5

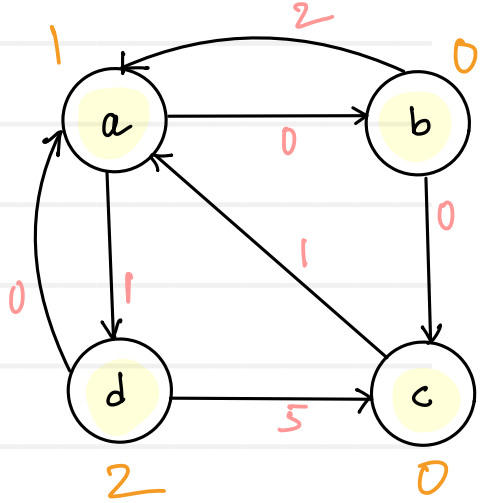
w(d,a) = -1 + 0 - -1 = 0

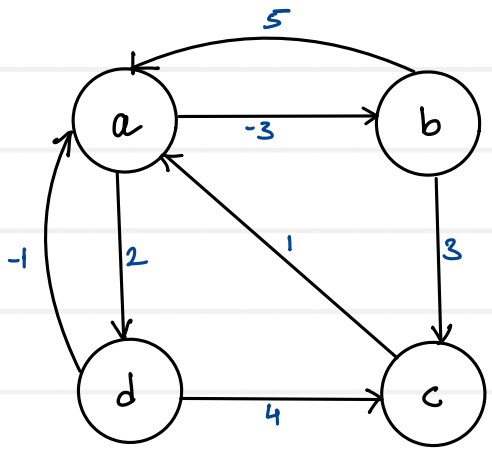
w(a,d) = 2 + -1 - 0 = 1

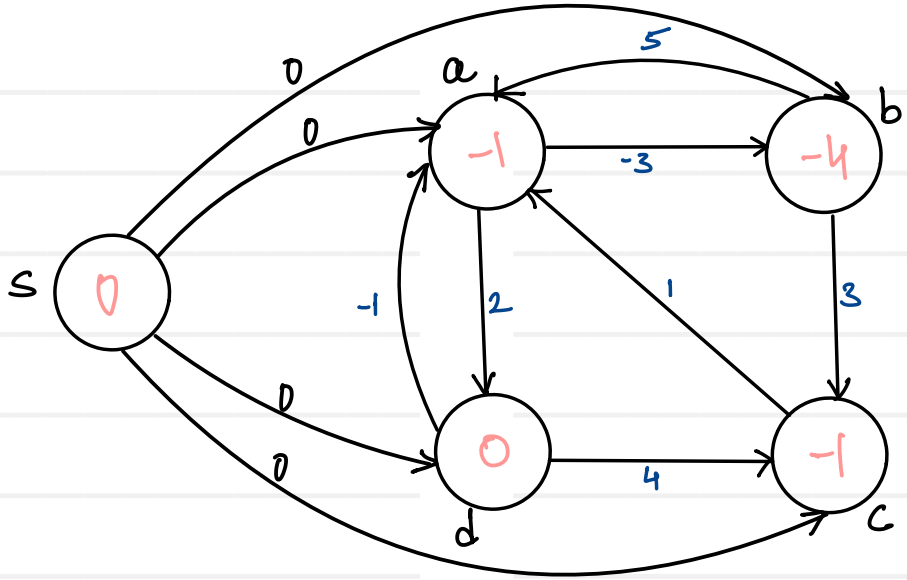


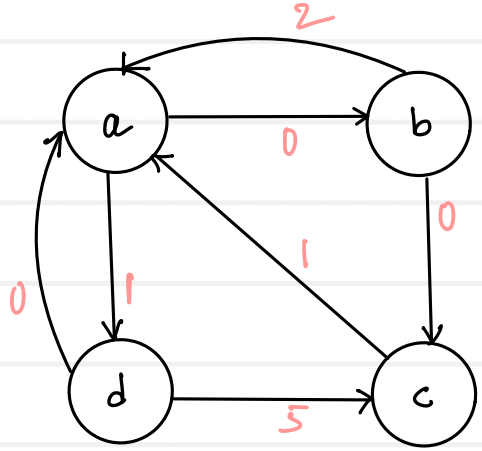
	a	b	c	d
a	0	0	0	1
b	1	0	0	2
c				
d				







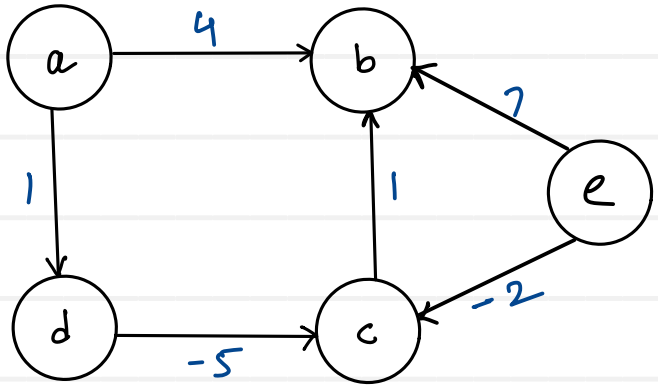




	a	b	c	d
a	0	0	0	1
b	1	0	0	2
c				
d				

$$dist(u,v) = dist(u,v) + d(v) - d(u)$$

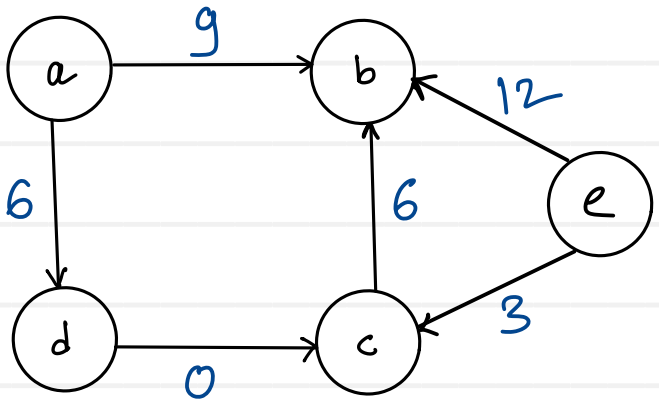
	a	b	c	d
a	0	-3	0	2
b				
c				
d				



$a - b = 4$

$a - d - c - b = 1 - 5 + 1 = -3$ ✓

After adding 5 to all wt



$a - b = 9$ $\Rightarrow 9 - 5 = 4$ ✗

$a - d - c - b = 6 + 0 + 6 = 12$ $\Rightarrow 12 - 5 = 7$

A* Algorithm

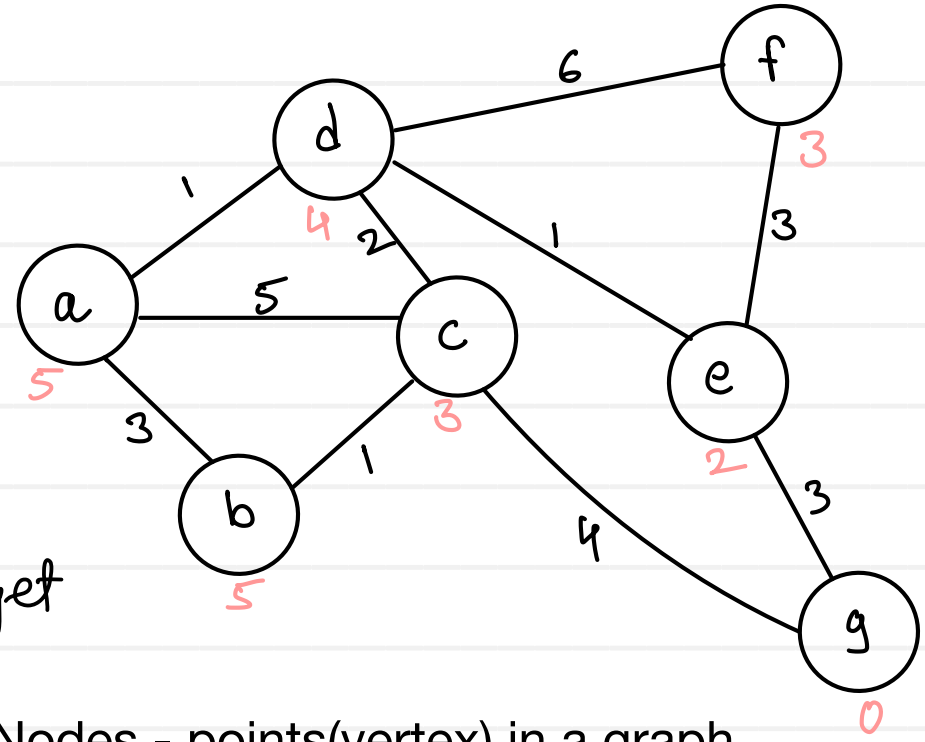
- Popular technique used in path finding and graph traversals.

- A* algorithm combines
    1. Dijkstra's algorithm
    2. Greedy Breadth First Approach

- It considers both:
    - distance already traveled from the start
    - estimate of the remaining distance to the goal /target

- This combination helps A* to make informed decisions about which path to explore next.

- This makes A* both efficient and accurate

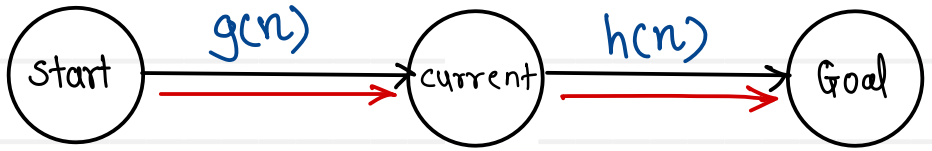


- Nodes - points(vertex) in a graph
- Edges - links/connections between nodes
- Path cost - actual cost of moving from one node to another
- Heuristic - estimated cost from any node to the goal

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

$$g(n_k) = \sum_{i=0}^{k} w(n_i)$$

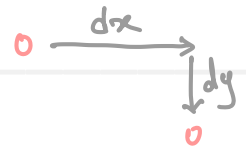
h(n) : heuristic function - estimated cost from current to goal vertex.

- common heuristic functions:

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

1. Manhattan distance

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

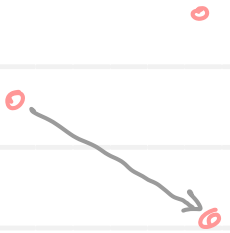


2. Diagonal distance

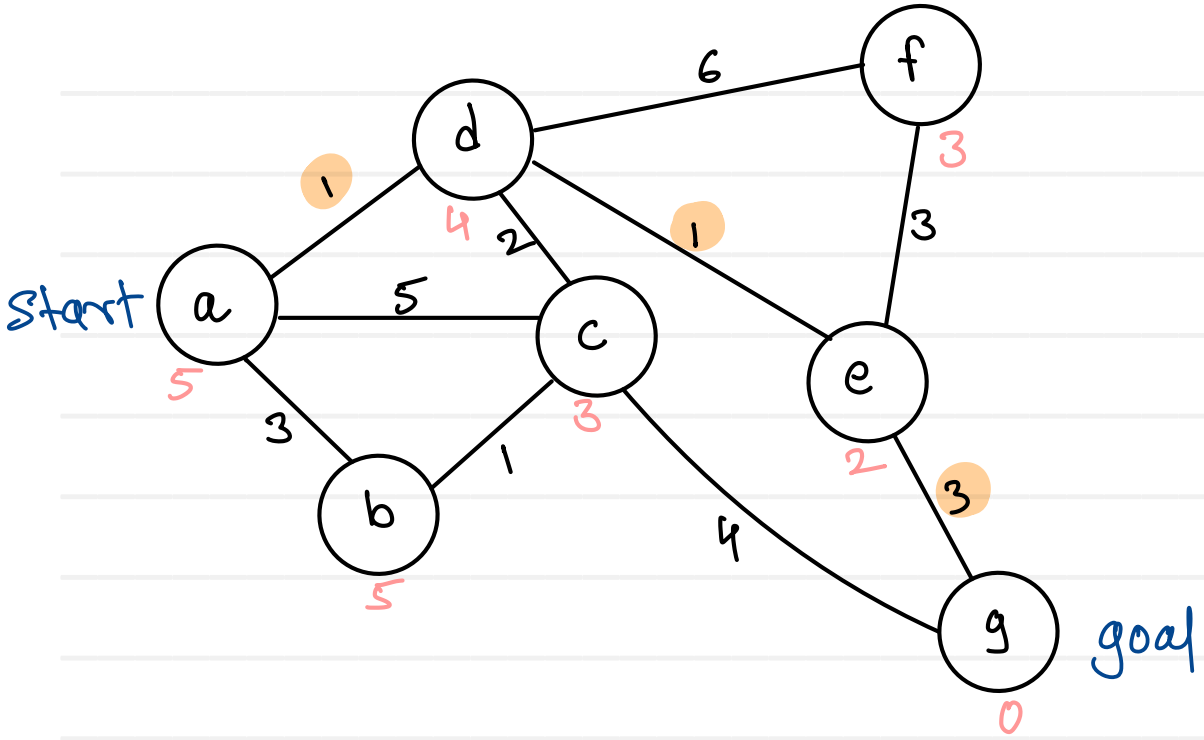
$$h(n) = MAX(|x_1 - x_2|, |y_1 - y_2|)$$

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	5	3	8
a d	1	4	5
c	3	3	6
a d e	2	2	4
f	7	3	10
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.

- 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.

Merge Sorted Array

Given two sorted integer arrays nums1 and nums2 in ascending order, and two integers m and n representing the number of elements in nums1 and nums2 respectively.

Merge nums1 and nums2 into a single array sorted in ascending order.

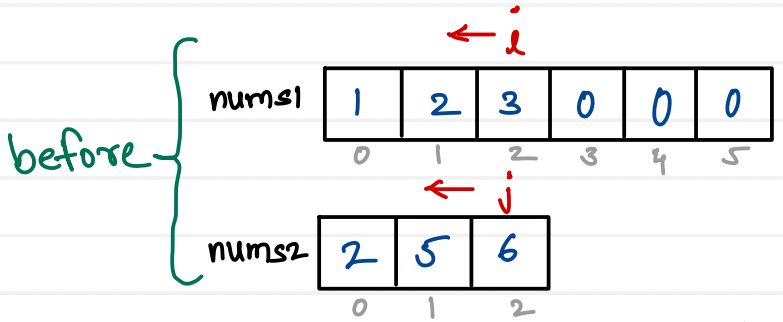
The final sorted array should be stored inside the array nums1. To accommodate this, nums1 has a length of m + n.

Example 1:
Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]

Example 2:
Input: nums1 = [1], m = 1, nums2 = [], n = 0
Output: [1]

Example 3:
Input: nums1 = [0], m = 0, nums2 = [1], n = 1
Output: [1]

As sorted array to be stored in nums1 again, need to start from last elements of the arrays.

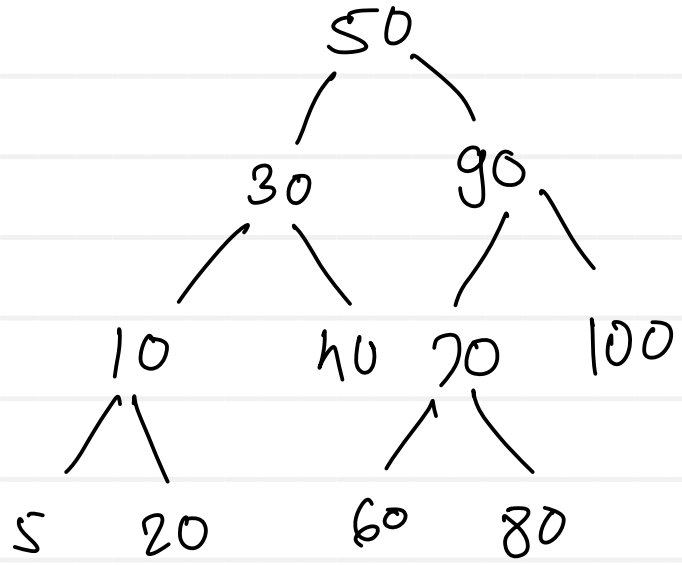


compare ith & jth element, whichever is maximum put it into nums1 from last index

Time complexity:

$T(n) = O(n)$

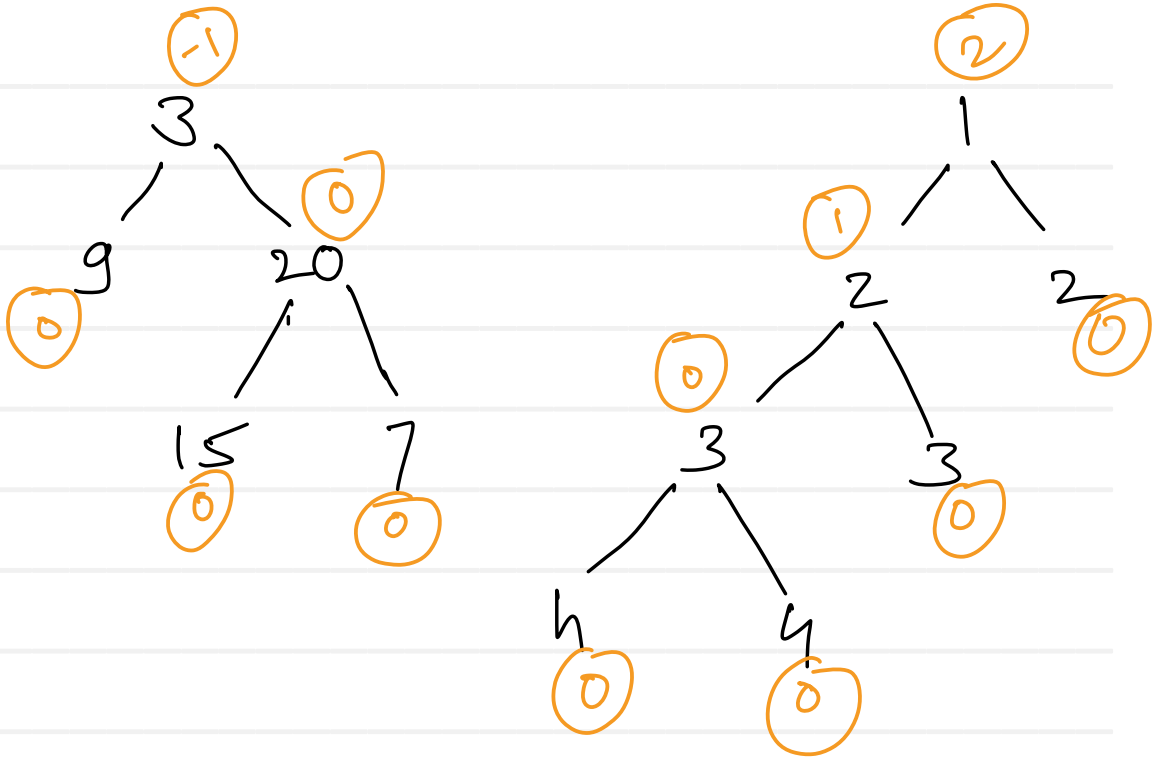
Strictly binary tree or not



traverse each node in tree using BFS
if node has single child,
return false

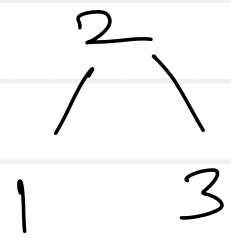
return true - when all nodes are traversed

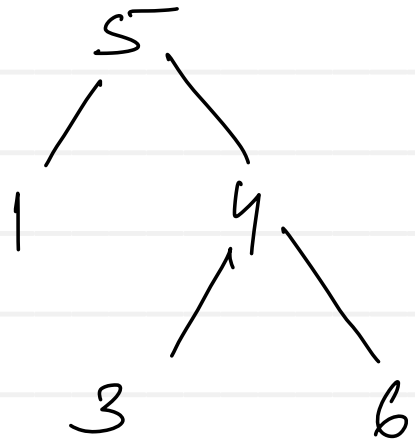
Balanced binary tree or not




```
int height(Node trav) {
    if(trav == null) return 0;
    int hl = height(trav.left);
    int hr = height(trav.right);

    if(hl == -1 || hr == -1)
        return -1;

    if(Math.abs(hl-hr) > 1)
        return -1;

    int max = hl > hr ? hl : hr;
    return max + 1;
}

boolean isBalanced(root) {
    return height(root) != -1;
}
```

Validate binary search tree

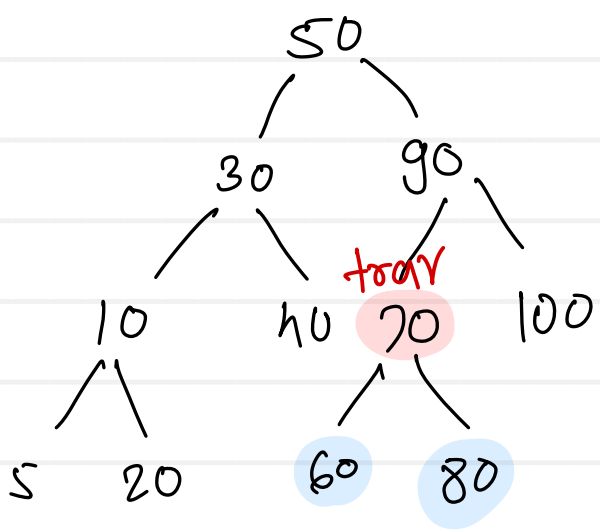




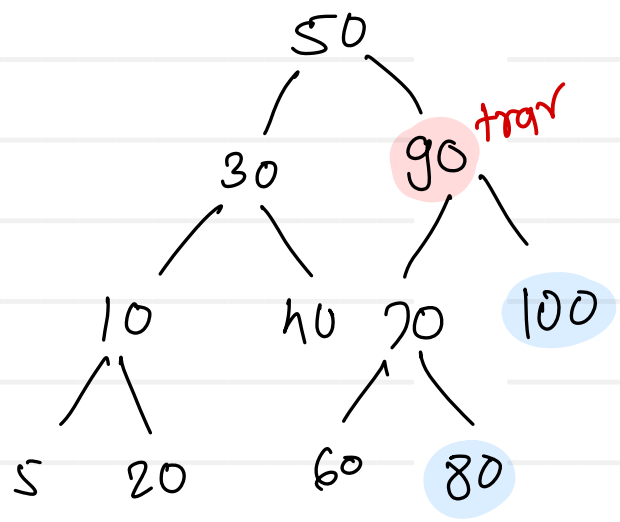
trav	left	Right
&2	1	3
&1	null	null
&3	null	null

trav	left	right
&5	1	4 X

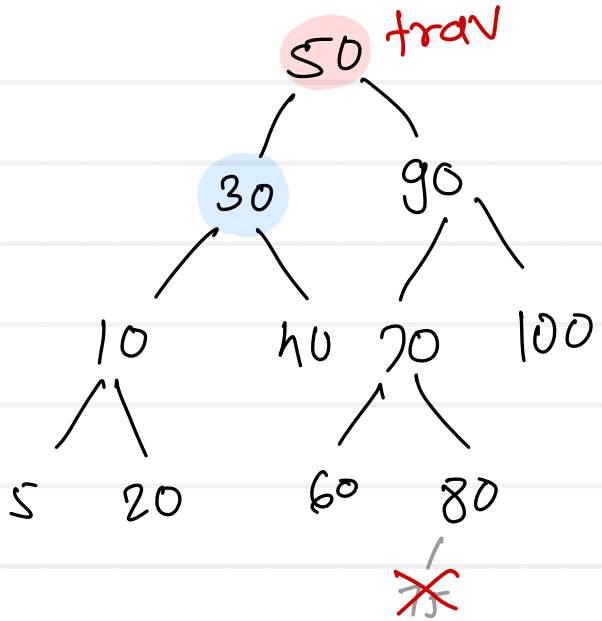
Nearest common ancestor



60, 80 → 70

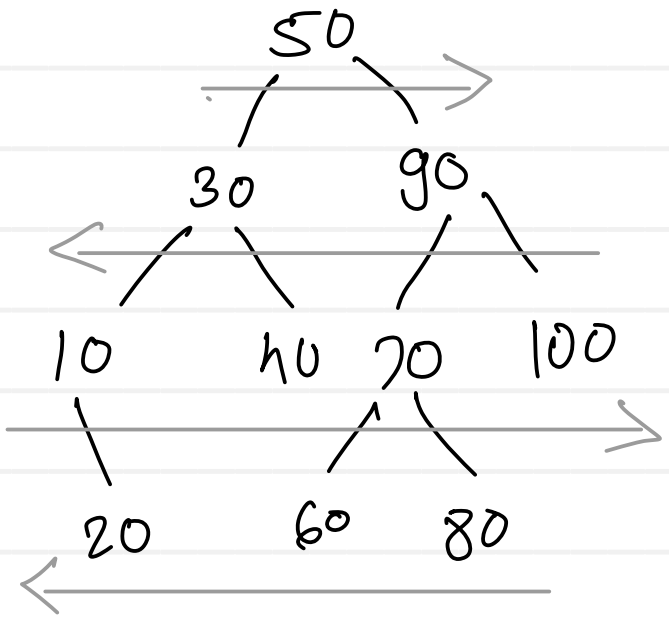


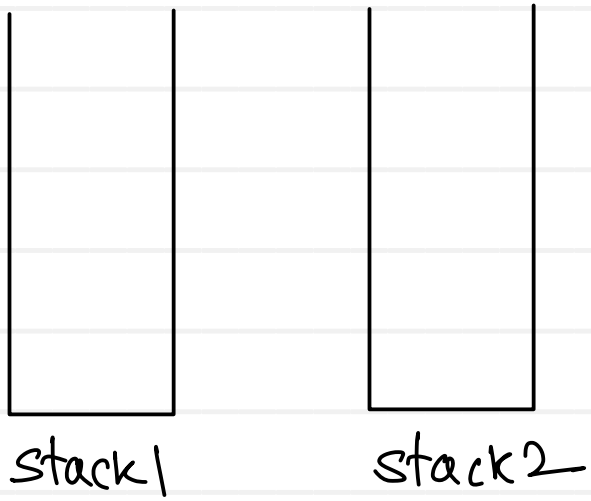
80, 100 → 90



30, 75 → X

Binary tree zigzag level order traversal (spiral order)





output: [50] [90, 30] [10, 40, 70, 100] [80, 60, 20]