Graph Data Structure & Algorithms

Sunbeam Infotech

Kruskal's MST – Analysis

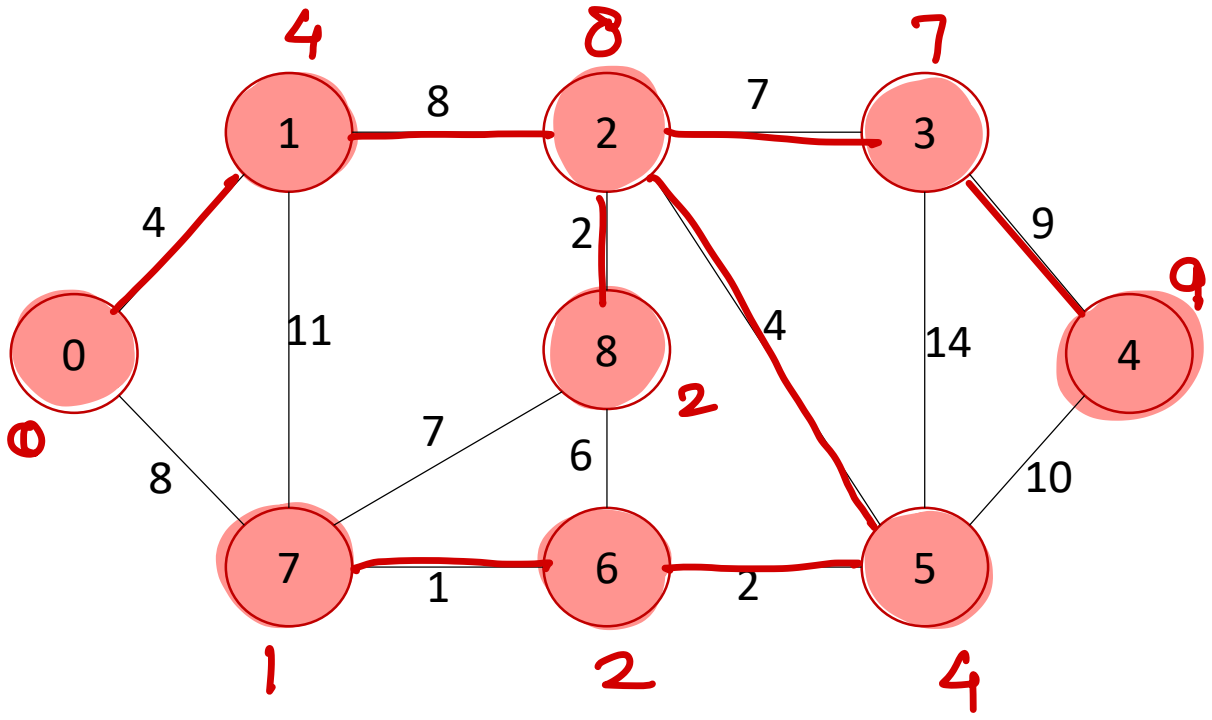
1. Sort all the edges in ascending order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step 2 until there are (V-1) edges in the spanning tree.

- Time complexity
  - Sort edges: O(E log E)
  - Pick edges (E edges): O(E)
  - Union Find: O(log V) – optimized.

- Time complexity
  - O(E log E + E log V)
  - E can max V^2.
  - So max time complexity: O(E log V).

Prim's MST

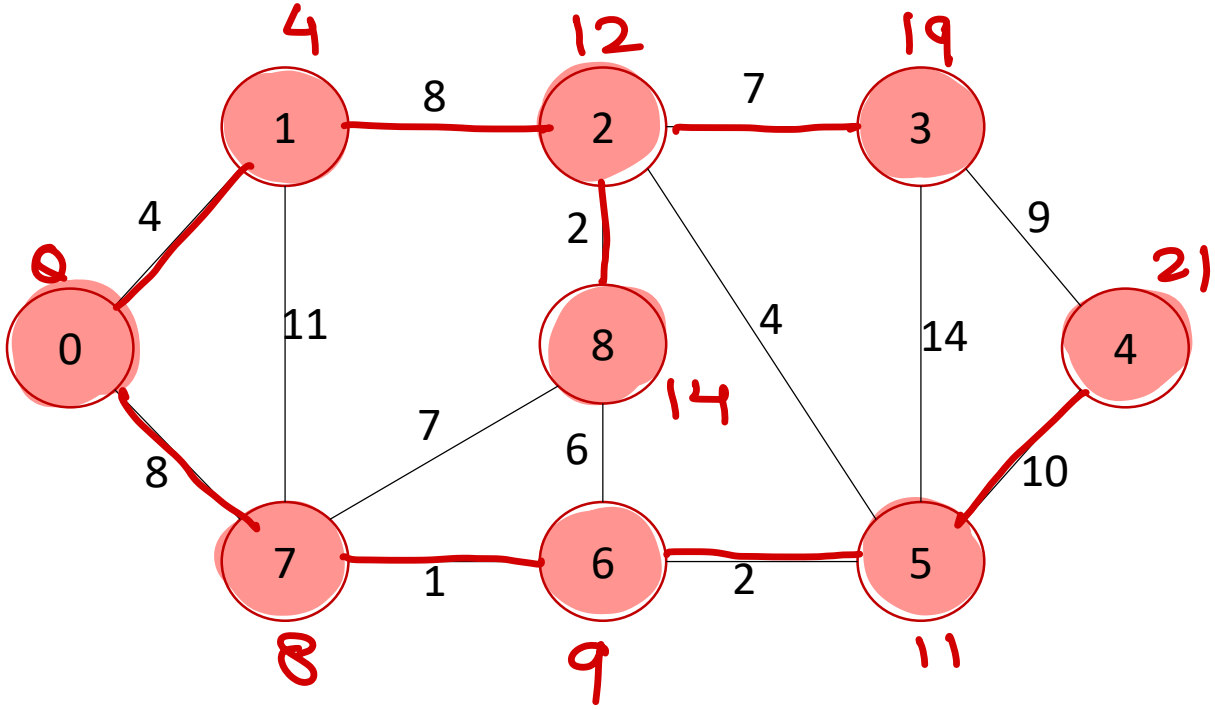
1. Create a set *mst* to keep track of vertices included in MST.
2. Also keep track of parent of each vertex. Initialize parent of each vertex -1.
3. Assign a key to all vertices in the input graph. Key for all vertices should be initialized to INF. The start vertex key should be 0.
4. While *mst* doesn't include all 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.



Dijkstra's Algorithm

→ Single Source Shortest Path also for weighted graphs.

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 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.



Dijkstra's SPT – Analysis

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 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.

- Time complexity (adjacency matrix)
  - V vertices: O(V)
  - get min key vertex: O(V)
  - update adjacent: O(V)
- Time complexity (adjacency matrix)
  - $O(V^2)$
- Time complexity (adjacency list)
  - V vertices: O(V)
  - get min key vertex: O(log V)
  - update adjacent: O(E) – E edges

E + V log V

- Time complexity (adjacency list)
  - O(E log V)

Dijkstra Algo.

dist[v] = dist[u] + weight(u,v)

① Cannot work with -ve edge weights.
e.g. chemical reaction (energy)
electronic ckt (current)

* solution → Bellman Ford. (SPT)

② Using dijkstra for finding shortest path to all vertices from each vertex :- all pair shortest path.

* Time complexity : O(V*(E+V log V))

* solution → Warshall Floyd.
(-ve weight)

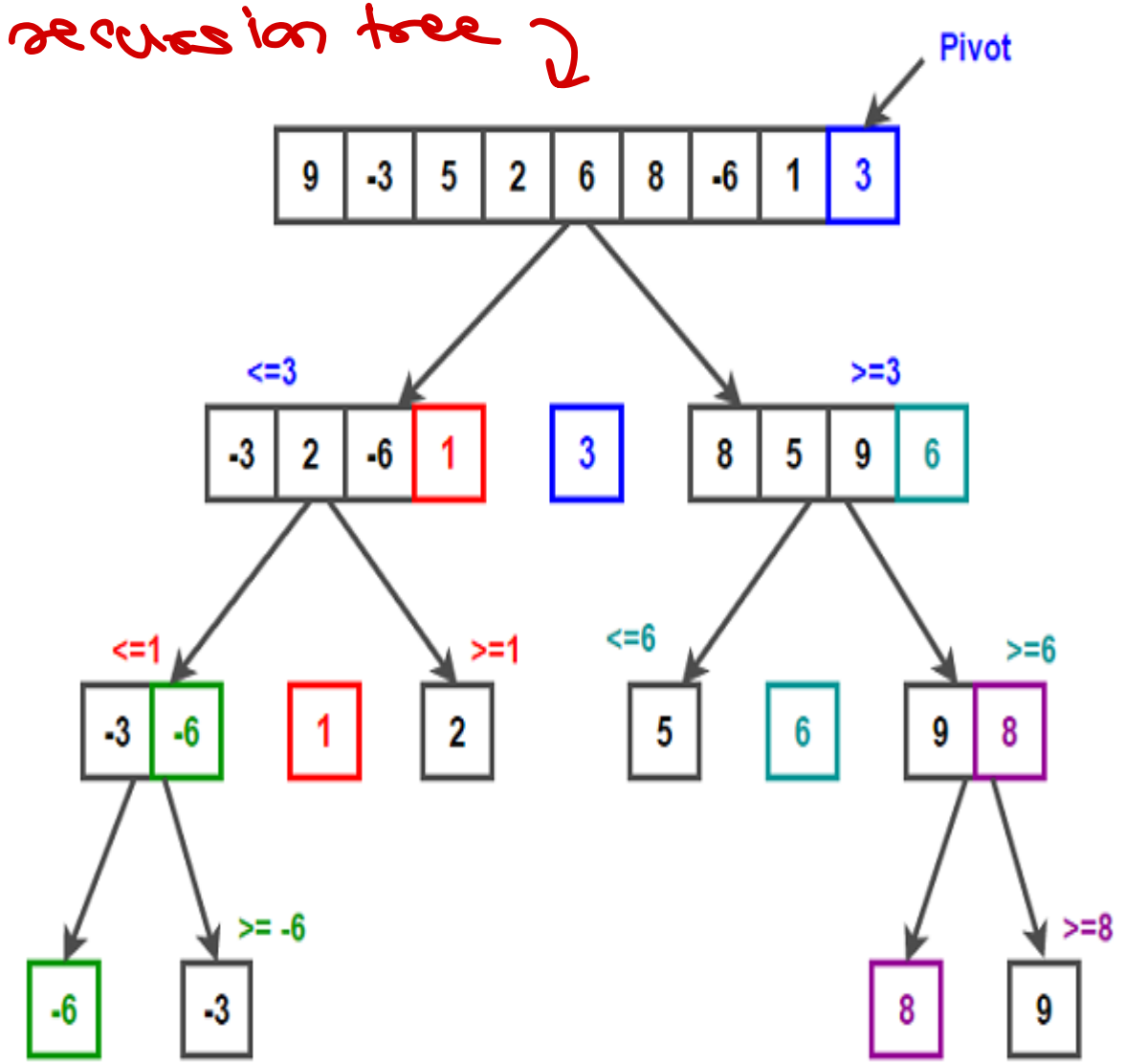
Dynamic Programming

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)

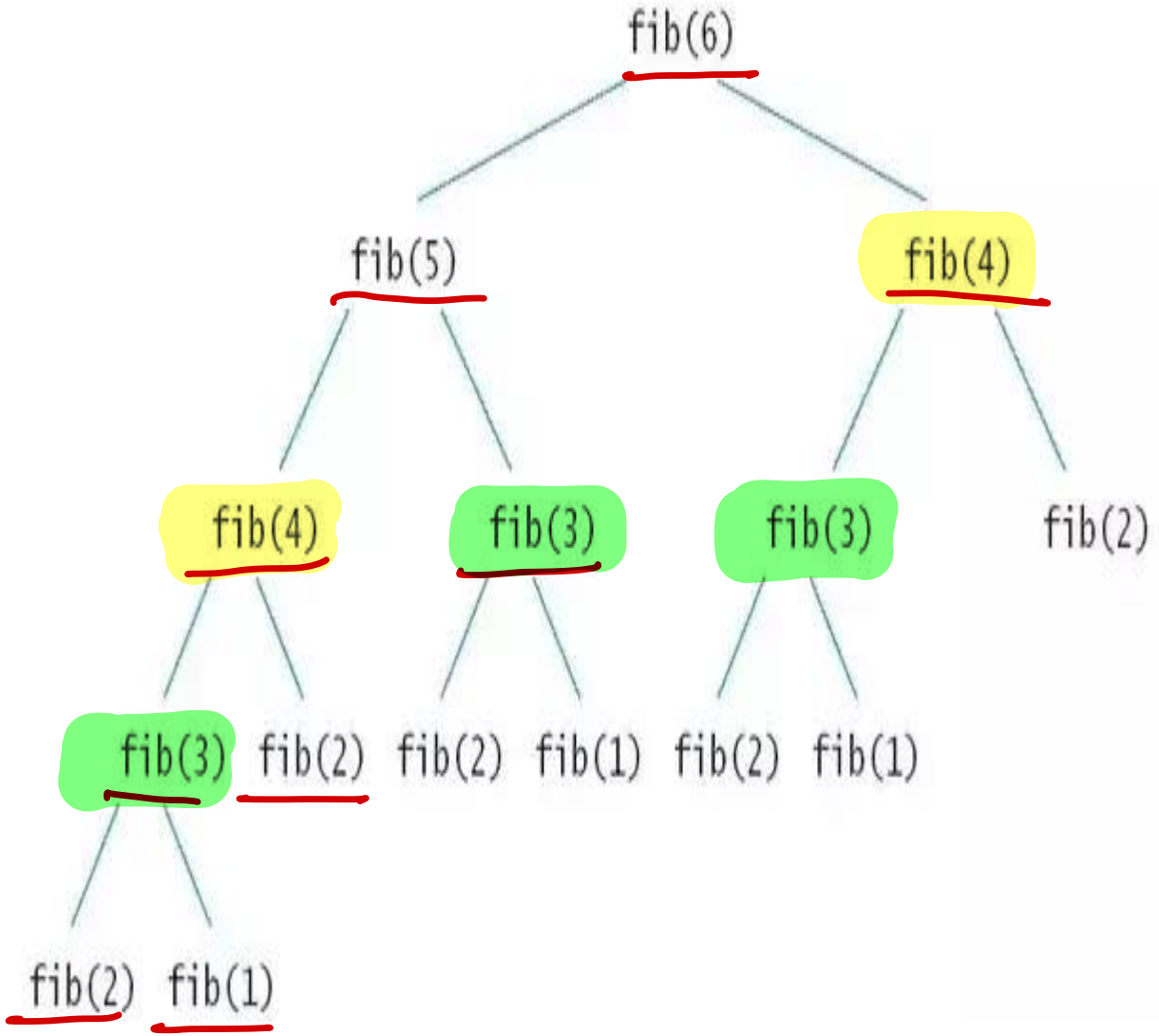
Recursion – QuickSort

- Algorithm
  1. If single element in partition, return.
  2. Last element as pivot.
  3. From left find element greater than pivot (x^{th} ele).
  4. From right find element less than pivot (y^{th} ele).
  5. Swap x^{th} ele with y^{th} ele.
  6. Repeat 2 to 4 until x < y.
  7. Swap y^{th} ele with pivot.
  8. Apply QuickSort to left partition (left to y-1).
  9. Apply QuickSort to right partition (y+1 to right).
- QS(arr, 0, 8)
  - QS(arr, 0, 3)
    - QS(arr, 0, 1)
      - QS(arr, 0, 0)
    - QS(arr, 3, 3)
  - QS(arr, 5, 8)
    - QS(arr, 5, 5)
    - QS(arr, 7, 8)
      - QS(arr, 9, 9)



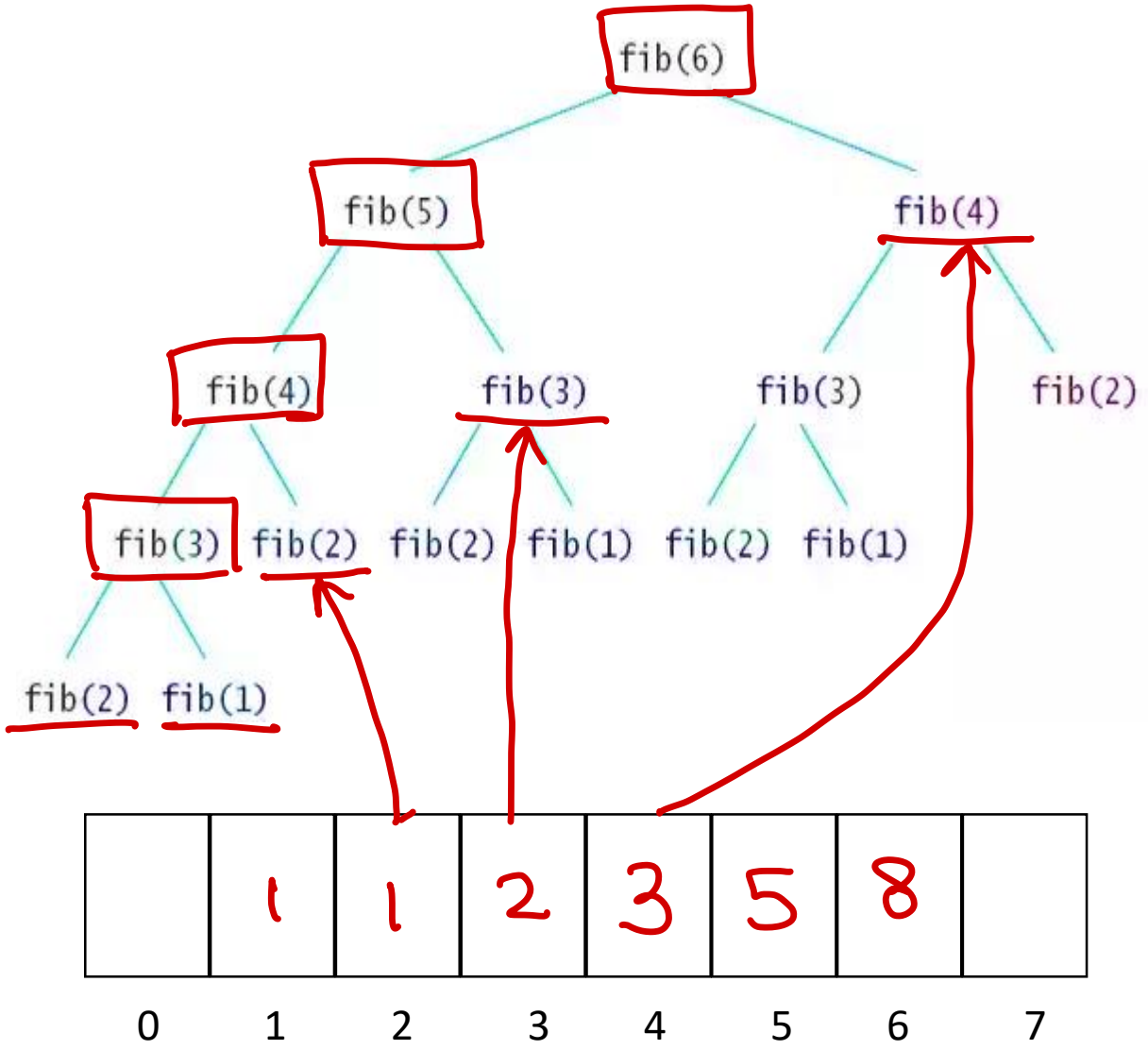
Recursion – Fibonacci Series

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



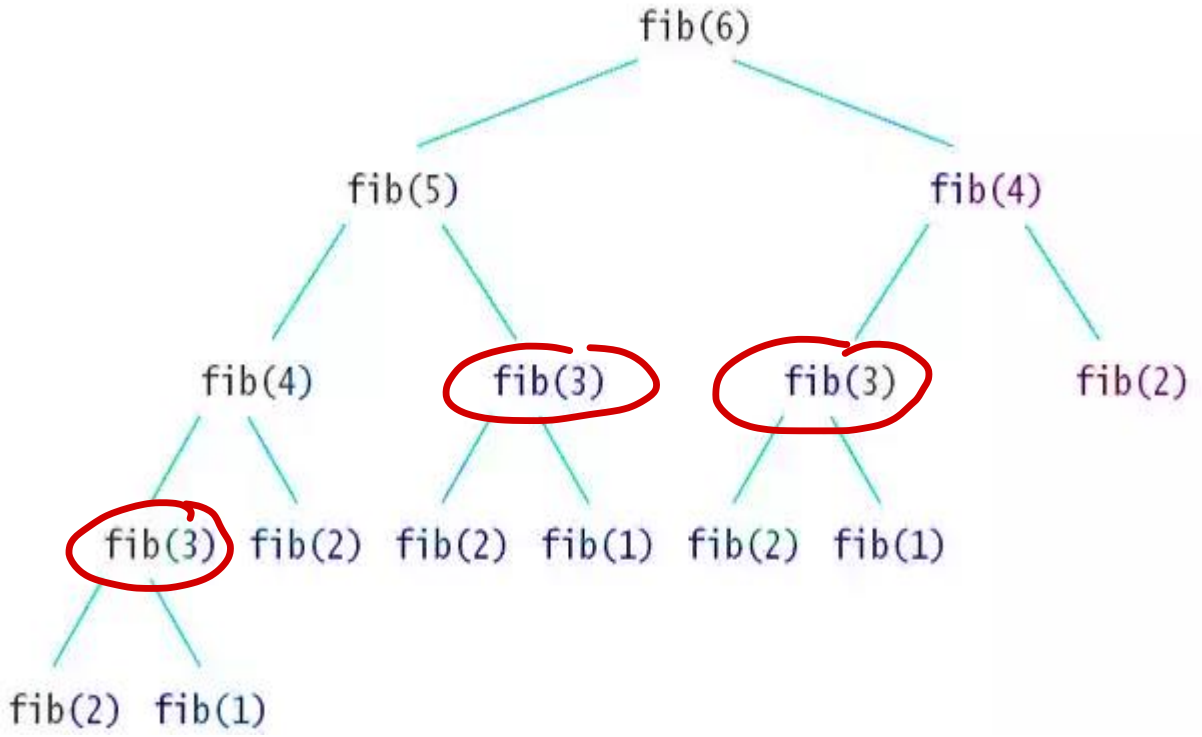
Memoization – Fibonacci Series

- 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.



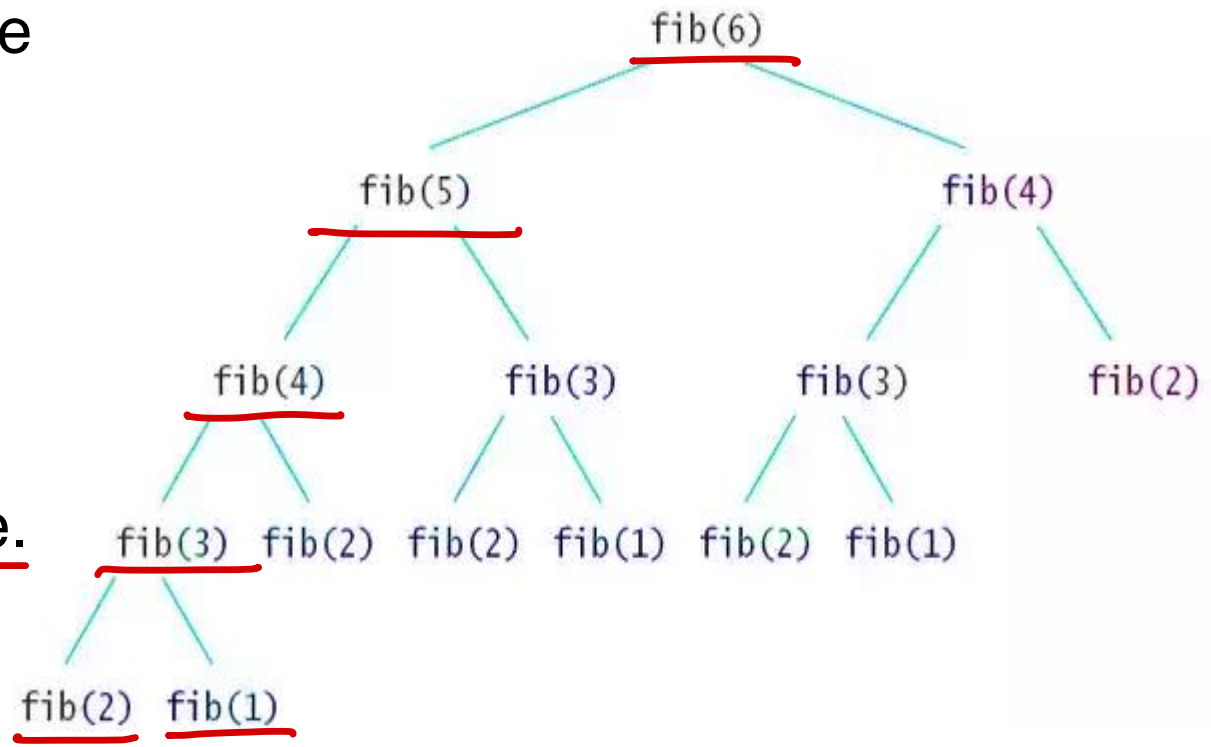
Dynamic Programming

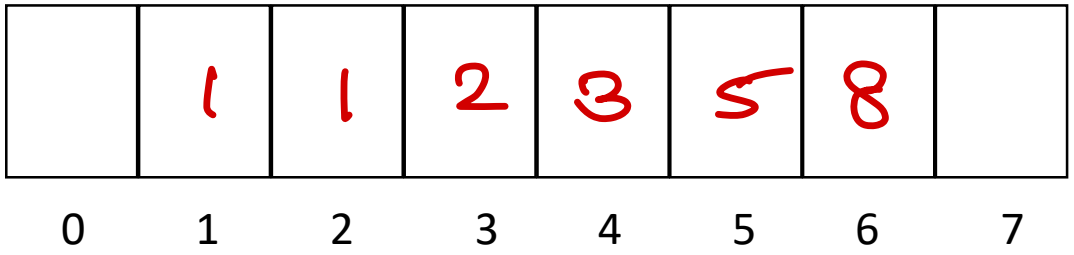
- Dynamic programming is another optimization over recursion.
- Typical DP problem give choices (to select from) and ask for optimal result (maximum or minimum).
- Technically it can be used for the problems having two properties
  - Overlapping sub-problems
  - Optimal sub-structure
- To solve problem,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



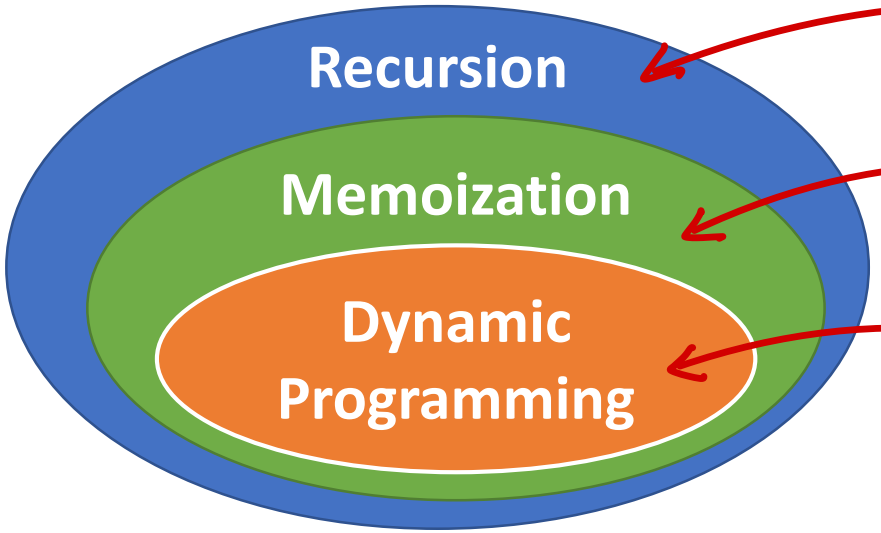
Dynamic Programming – Fibonacci Series

- Alternative solution to DP is memoizing the recursive calls. This solution needs more stack space, but similar in time complexity.
- Memoization is also referred as top-down approach.
- DP solution is bottom-up approach.
- DP use 1-d array or 2-d array to save state.
- 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

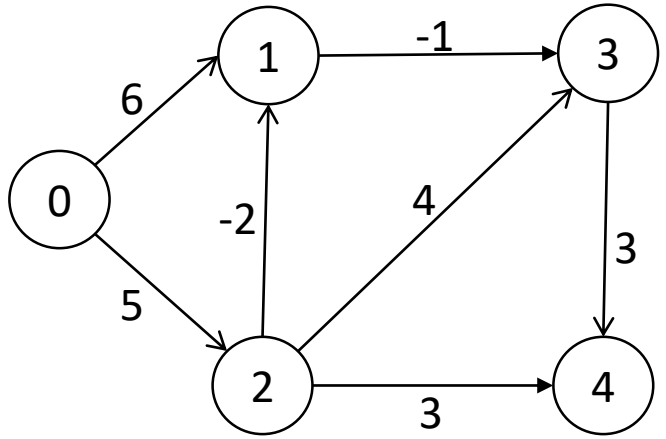


- Top down
- Optimized Top down – overlapping Sub-problems
- Bottom up – optimal substructure

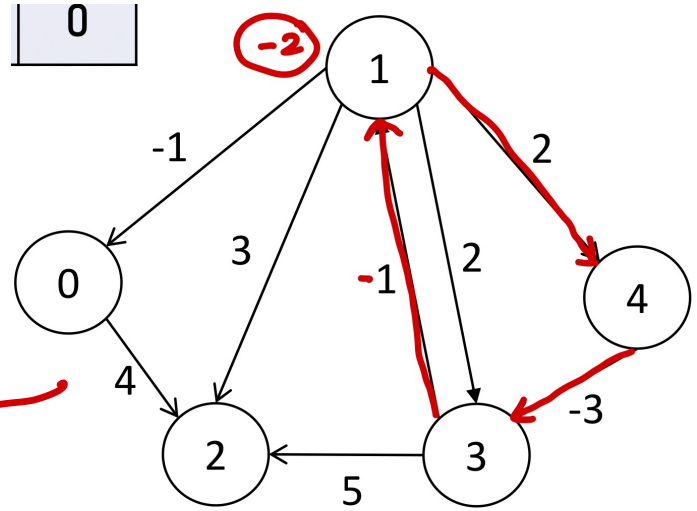
Bellman Ford Algorithm → Single source shortest path algo -ve weight edges.

- Initializes distances from the source to all vertices as infinite and distance to the source itself as 0.
- Calculates shortest distance V-1 times: For each edge u-v, if dist[v] > dist[u] + weight of edge u-v, then update dist[v], so that dist[v] = dist[u] + weight of edge u-v.
- Check if negative ~~edge~~ cycle in the graph: For each edge u-v, if dist[v] > dist[u] + weight of edge uv, then graph has –ve weight cycle.

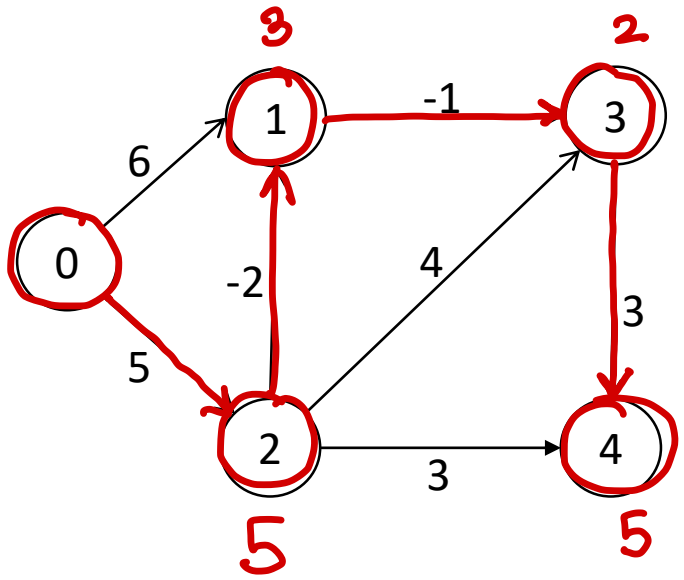
→ throw exception.

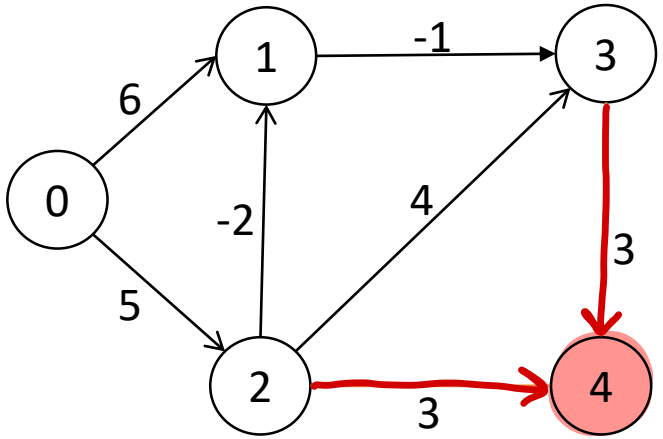


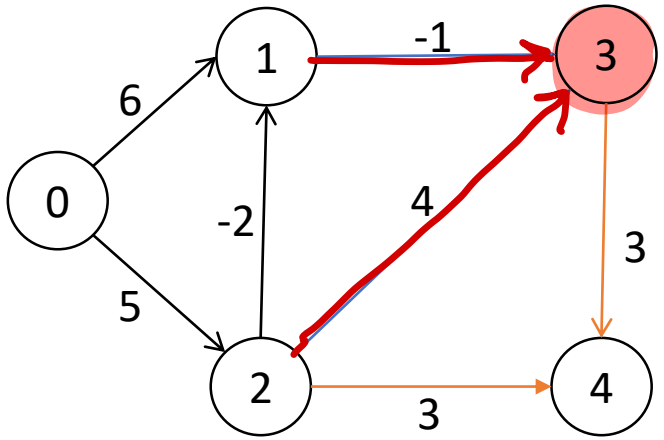
Src	Des	Wt
3	4	3
2	4	3
2	3	4
2	1	-2
1	3	-1
0	2	5
0	1	6

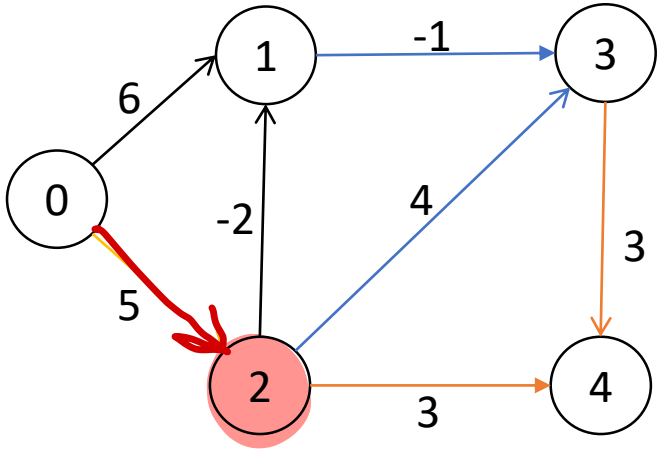


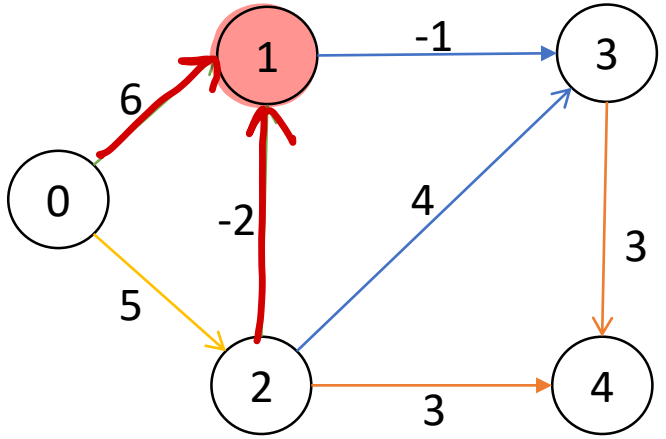
Bellman Ford Algorithm







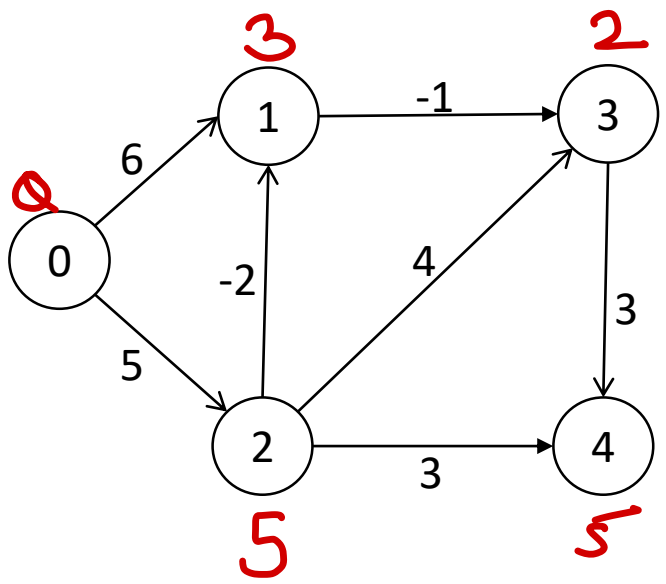




Bellman Ford Algorithm

O(VE)

	0	1	2	3	4
Pass 0	0	∞	∞	∞	∞
Pass 1	0	6	5	∞	∞
Pass 2	0	3	5	2	8
Pass 3	0	3	5	2	5
Pass 4	0	3	5	2	5



Src	Dest	Wt
3	4	3
2	4	3
2	3	4
2	1	-2
1	3	-1
0	2	5
0	1	6

Bellman Ford doesn't work with -ve weight cycle. In that case dist of vertices keep changing even after V-1 passes.

Warshall Floyd Algorithm - all pair shortest path

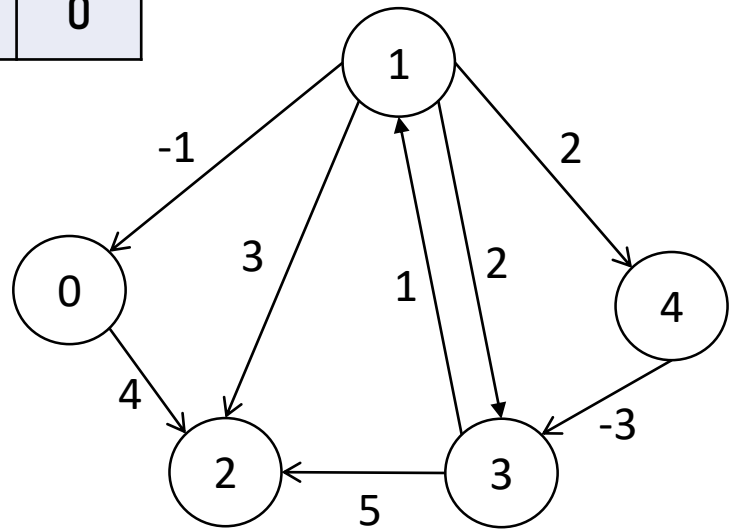
- Algorithm

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).

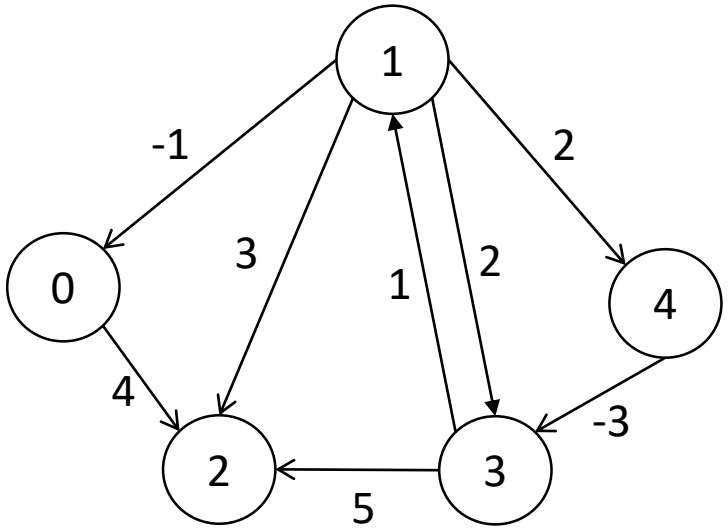
2. Consider each vertex (i) in between pair of any two vertices (s, d) and find the optimal distance between s & d considering intermediate vertex i.e. dist(s,d) = dist(s,i) + dist(i,d), if dist(s,i) + dist(i,d) < dist(s,d).

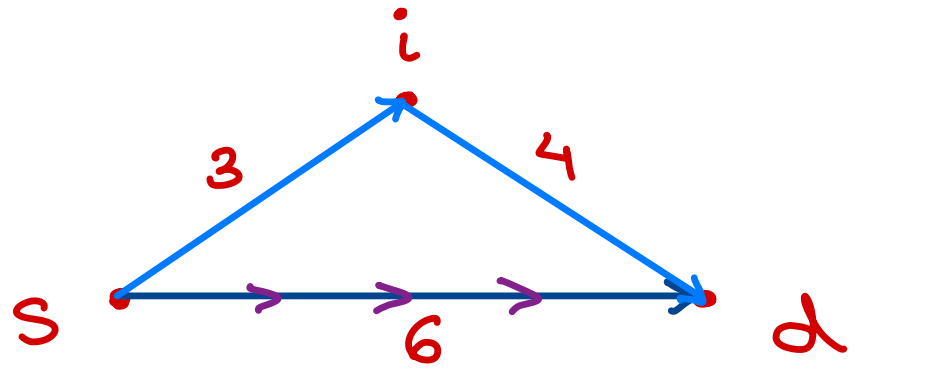
	0	1	2	3	4
0	0	∞	4	∞	∞
1	-1	0	3	2	2
2	∞	∞	0	∞	∞
3	∞	1	5	0	∞
4	∞	∞	∞	-3	0

$O(V^3)$

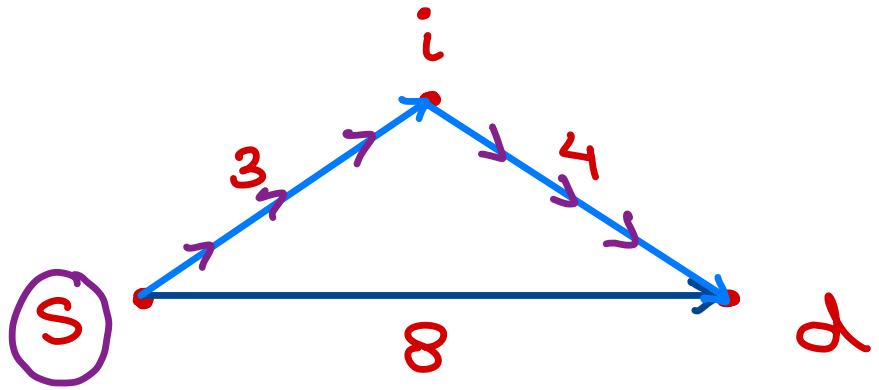


Warshall Floyd Algorithm





dist[s,d] = dist[s,d]



dist[s,d] = dist[s,i] + dist[i,d]

Johnson's Algorithm

→ all pair shortest path.

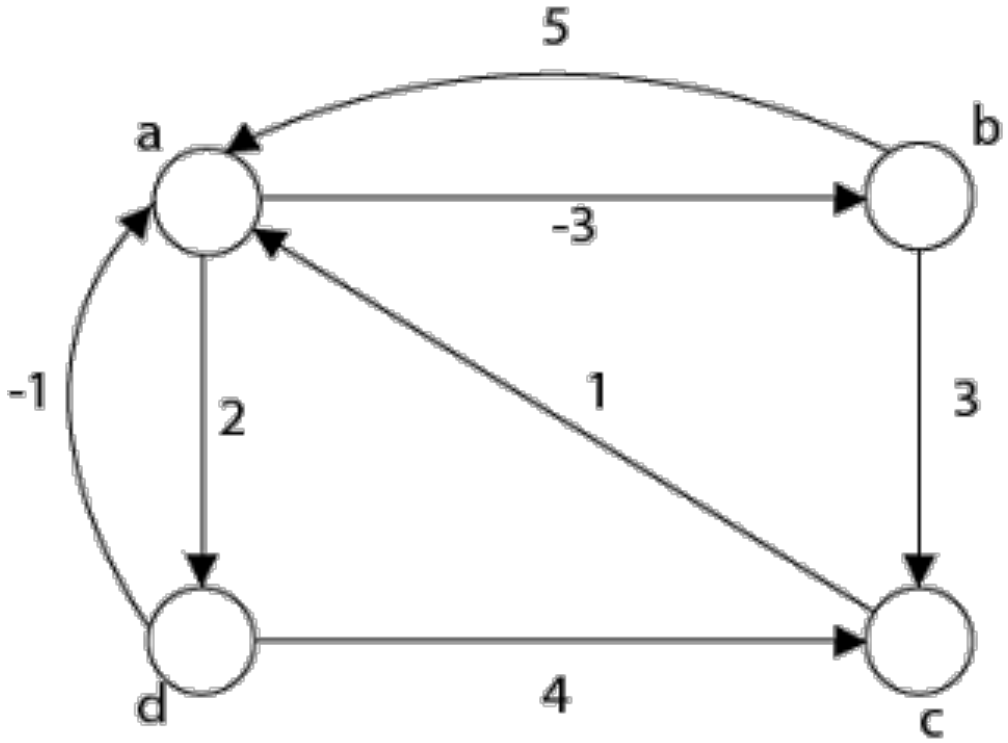
Using Bellaman for all vertices: O(V * V * E)

↳ slower than warshall floyd.

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

$O(VE + V^2 \log V)$. → faster than Warshall Floyd.

m

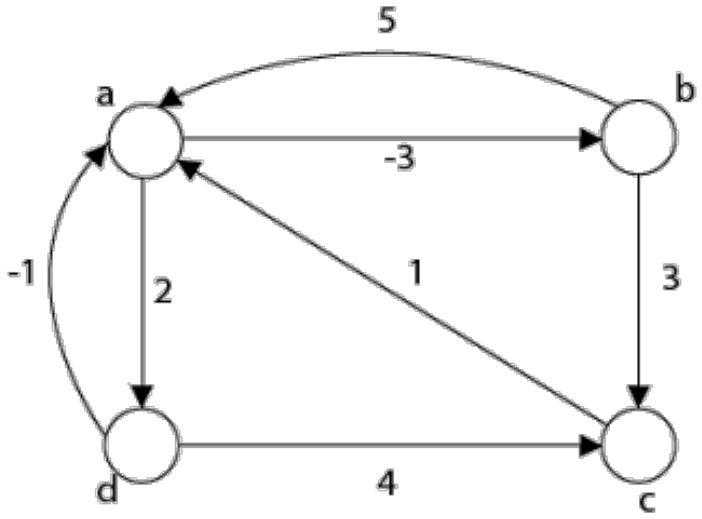


Johnson's Algorithm

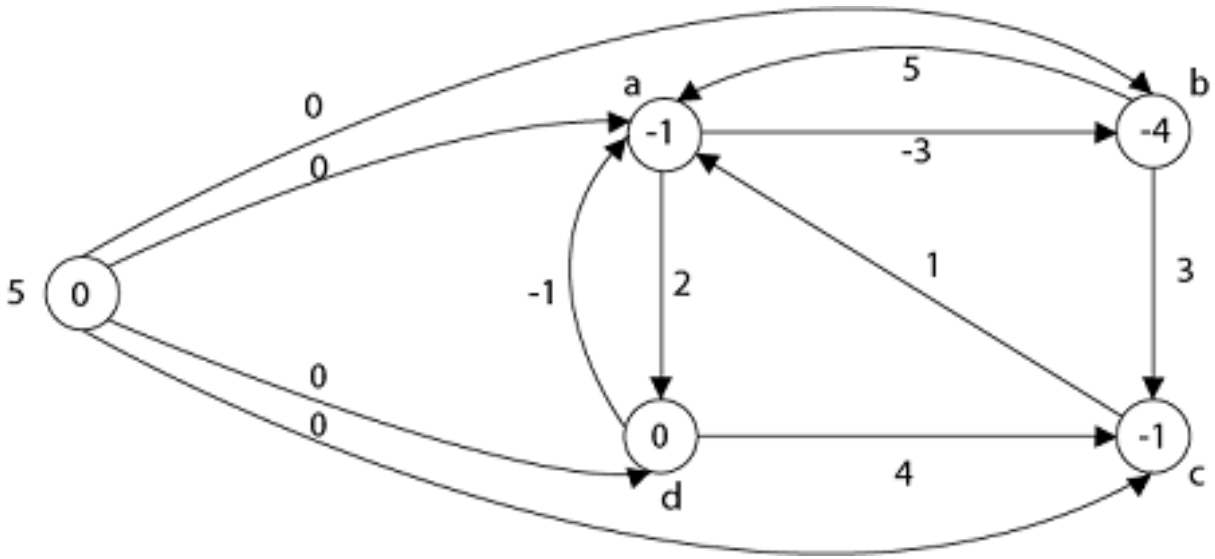
1. Add a vertex (s) into a graph and add edges from it all other vertices, with weight 0.
2. Find shortest distance of all vertices from (s) using Bellman Ford algorithm.

   a = -1, b = -4, c = -1, d = 0 and s = 0
3. Reweight all edges (u, v) in the graph, so that, they become non negative.

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



- 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

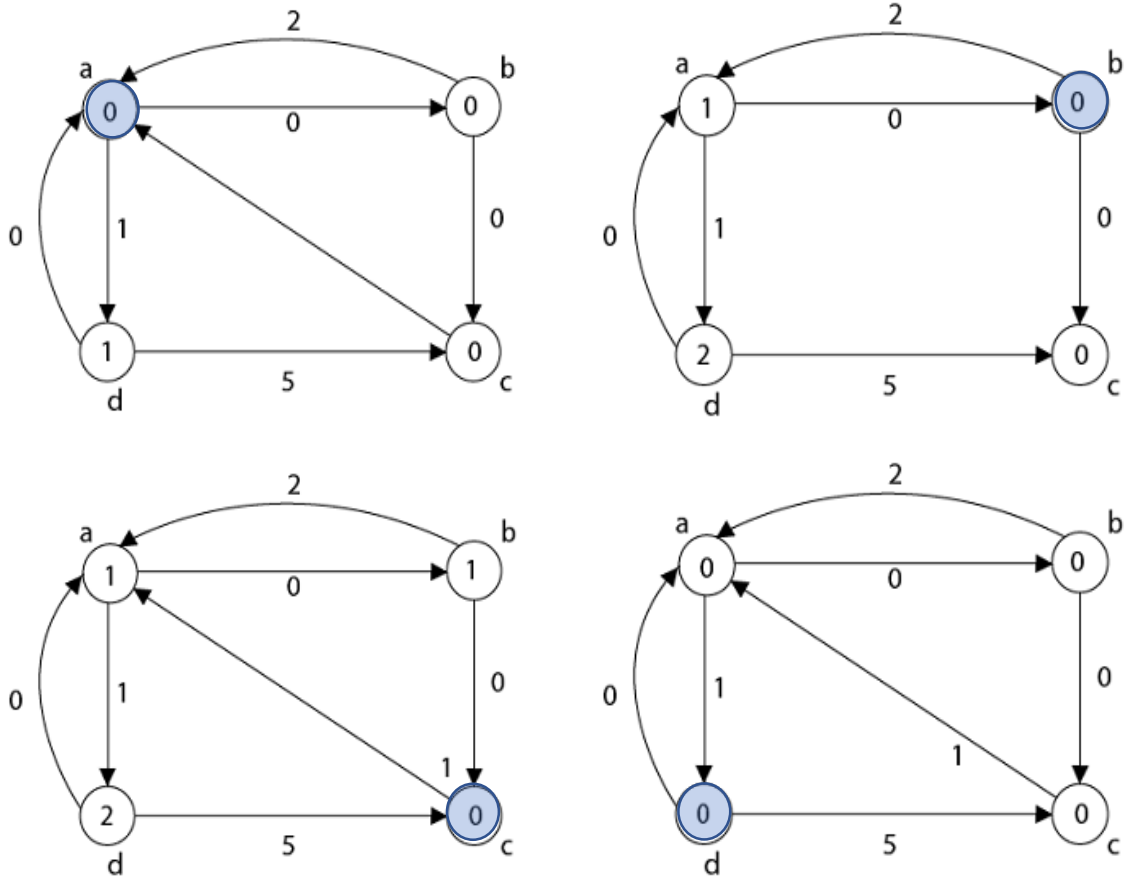


Johnson's Algorithm

4. Apply Dijkstra on each vertex to calculate shortest distance to all other vertices.
5. Reweight all distances to consider original weights.

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

	a	b	c	d
a	0 -> 0	0 -> -3	0 -> 0	1 -> 2
b	1 -> 4	0 -> 0	0 -> 3	2 -> 6
c	1 -> 1	1 -> -2	0 -> 0	2 -> 3
d	0 -> -1	0 -> -4	0 -> -1	0 -> 0

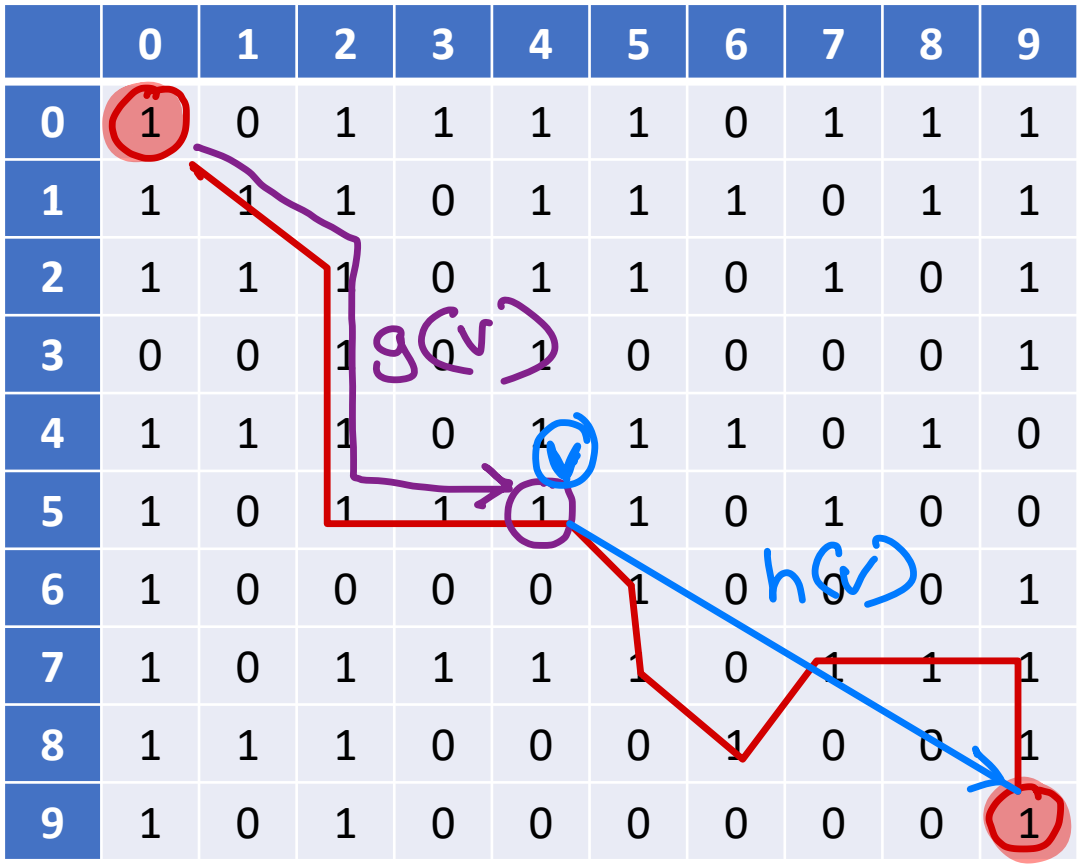


A* Search Algorithm → point to point

- Point to point approximate shortest path finder algorithm.
- This algorithm is used in artificial intelligence.
- Commonly used in games or maps to find shortest distance in faster way.
- It is modification of BFS.
- Put selected adjacent vertices on queue, based on some heuristic.
- A math function is calculated for vertices
  - f(v) = g(v) + h(v) → vertex with min f(v) is picked.
  - g(v) → cost of source to vertex v
  - *h(v)* → estimated cost of vertex v to destination.

→ Robot path Games

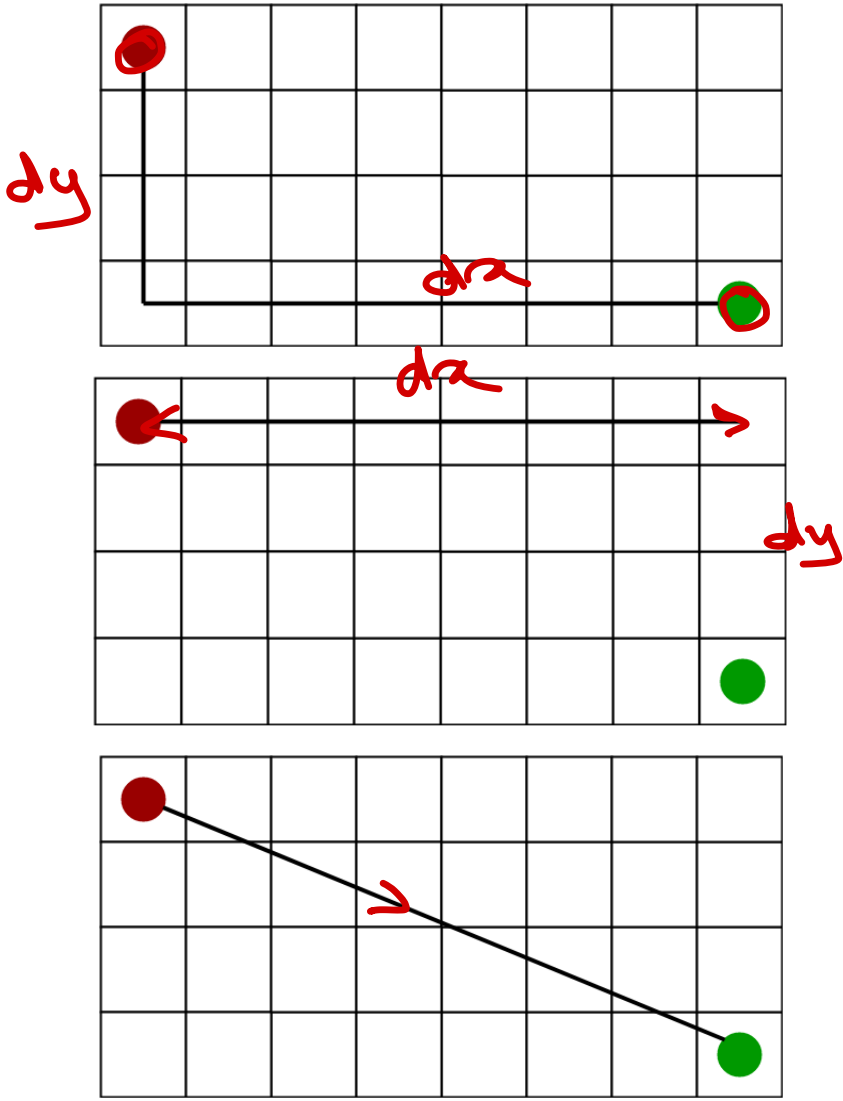
priority



	0	1	2	3	4	5	6	7	8	9
0	1	0	1	1	1	1	0	1	1	1
1	1	1	1	0	1	1	1	0	1	1
2	1	1	1	0	1	1	0	1	0	1
3	0	0	1	0	1	0	0	0	0	1
4	1	1	1	0	1	1	1	0	1	0
5	1	0	1	1	1	1	0	1	0	0
6	1	0	0	0	0	1	0	0	0	1
7	1	0	1	1	1	1	0	1	1	1
8	1	1	1	0	0	0	1	0	0	1
9	1	0	1	0	0	0	0	0	0	1

A* Search Algorithm

- h(v) represent heuristic and depends on problem domain. Three common techniques to calculate heuristic:
- Manhattan distance
  - When moves are limited in four directions only.
  - h = dx + dy
- Diagonal distance
  - When moves are allowed in all eight directions (one step).
  - h = MAX(dx, dy)
- Euclidean distance
  - When moves are allowed in any direction.
  - h = $\sqrt{dx^2 + dy^2}$
- Note that heuristic may result in longer paths in typical cases.



A* search algorithm

- Start point g(v) = 0, h(v) = 0 & f(h) = 0.
- Push start point vertex on a priority queue (by f(v)).
- Until queue is empty
  - Pop a point (v) from queue.
  - Add v into the path.
  - For each adjacent point (u)
    - If u is destination, build the path.
    - If u is invalid or already on path or blocked, skip it.
    - Calculate newg = g(v) + 1, newh = *heuristic* and newf = newg + newh.
    - If newf is less than f(u), f(u) = newf and also parent(u) = v. Rearrange elements in priority queue.

	0	1	2	3	4	5	6	7	8	9
0	1	0	1	1	1	1	0	1	1	1
1	1	1	1	0	1	1	1	0	1	1
2	1	1	1	0	1	1	0	1	0	1
3	0	0	1	0	1	0	0	0	0	1
4	1	1	1	0	1	1	1	0	1	0
5	1	0	1	1	1	1	0	1	0	0
6	1	0	0	0	0	1	0	0	0	1
7	1	0	1	1	1	1	0	1	1	1
8	1	1	1	0	0	0	1	0	0	1
9	1	0	1	0	0	0	0	0	0	1

A* Search Algorithm

	0	1	2	3	4	5	6	7	8	9
0	1	0	1	1	1	1	0	1	1	1
1	1	1	1	0	1	1	1	0	1	1
2	1	1	1	0	1	1	0	1	0	1
3	0	0	1	0	1	0	0	0	0	1
4	1	1	1	0	1	1	1	0	1	0
5	1	0	1	1	1	1	0	1	0	0
6	1	0	0	0	0	1	0	0	0	1
7	1	0	1	1	1	1	0	1	1	1
8	1	1	1	0	0	0	1	0	0	1
9	1	0	1	0	0	0	0	0	0	1

	0	1	2	3	4	5	6	7	8	9
0	1	0	1	1	1	1	0	1	1	1
1	1	1	1	0	1	1	1	0	1	1
2	1	1	1	0	1	1	0	1	0	1
3	0	0	1	0	1	0	0	0	0	1
4	1	1	1	0	1	1	1	0	1	0
5	1	0	1	1	1	1	0	1	0	0
6	1	0	0	0	0	1	0	0	0	1
7	1	0	1	1	1	1	0	1	1	1
8	1	1	1	0	0	0	1	0	0	1
9	1	0	1	0	0	0	0	0	0	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.

Thank you!

Nilesh Ghule <Nilesh@sunbeaminfo.com>