Graph Data Structure & Algorithms

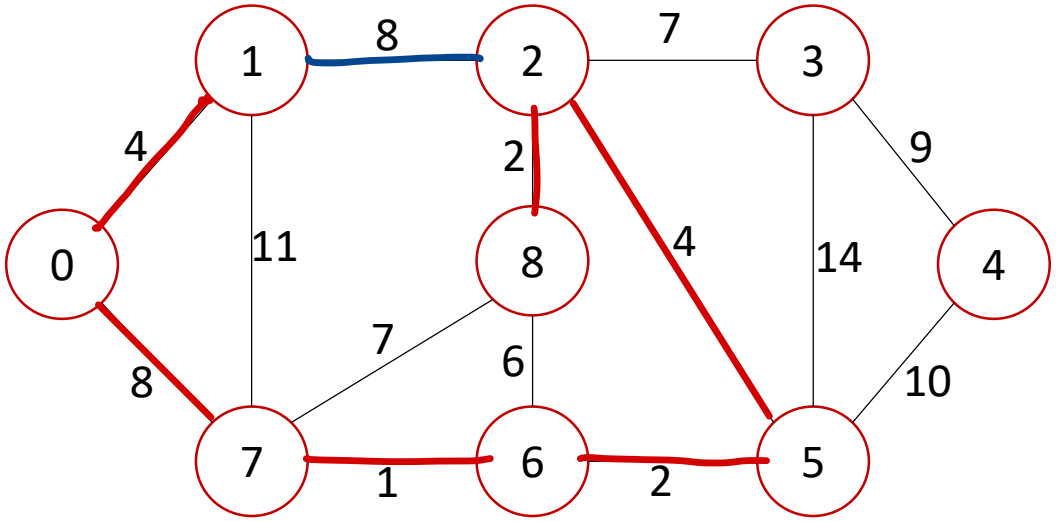
Sunbeam Infotech

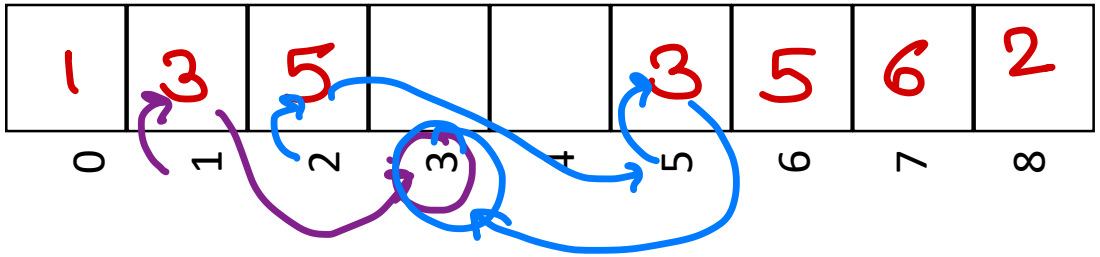
Union Find Algorithm

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

parent[src] = dest;

src root dest root

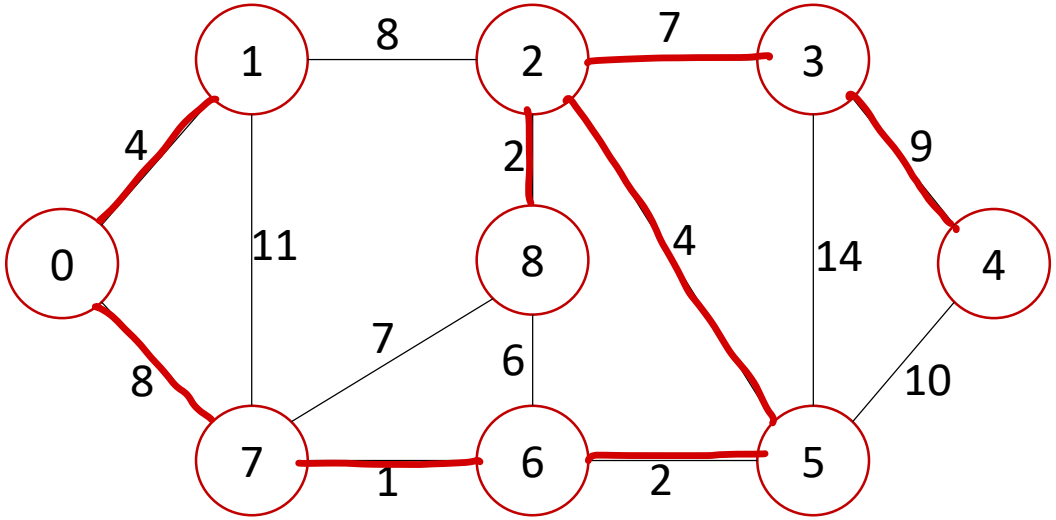




src	des	wt
7	6	1
8	2	2
6	5	2
0	1	4
2	5	4
~~8~~	~~6~~	~~6~~
2	3	7
~~7~~	~~8~~	~~7~~
0	7	8
1	2	8
3	4	9
5	4	10
1	7	11
3	5	14

Kruskal's MST

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.



src	des	wt
7	6	1
8	2	2
6	5	2
0	1	4
2	5	4
8	6	6
2	3	7
7	8	7
0	7	8
1	2	8
3	4	9
5	4	10
1	7	11
3	5	14

Union Find Algorithm – Analysis

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

- Time complexity
    - Skewed tree implementation
    - O(V)

- Improved time complexity
    - Rank based tree implementation
    - O(log V)

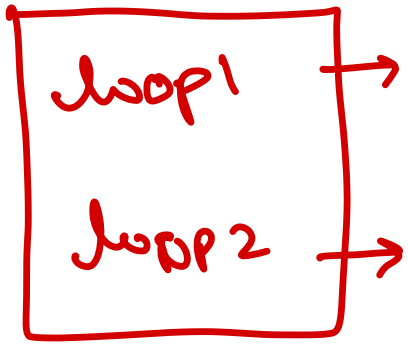
Kruskal's MST – Analysis

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.

$T \propto n + n$

$T \propto 2n$

$T \propto n$



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

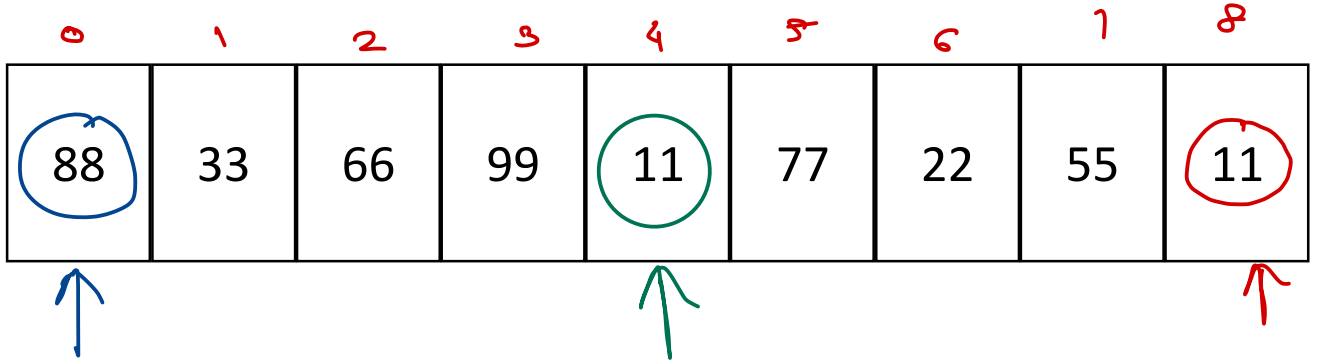
  } E log V

- Time complexity
  - O(E log E + E log V)
  - E can max V^2. $V(V-1)$
  - So max time complexity: O(E log V).

Linear Search

T α iterations

- Find a number in a list of given numbers (random order).



- Time complexity
  - Worst case
  - Best case
  - Average case

O(1)
T = k

O(n)

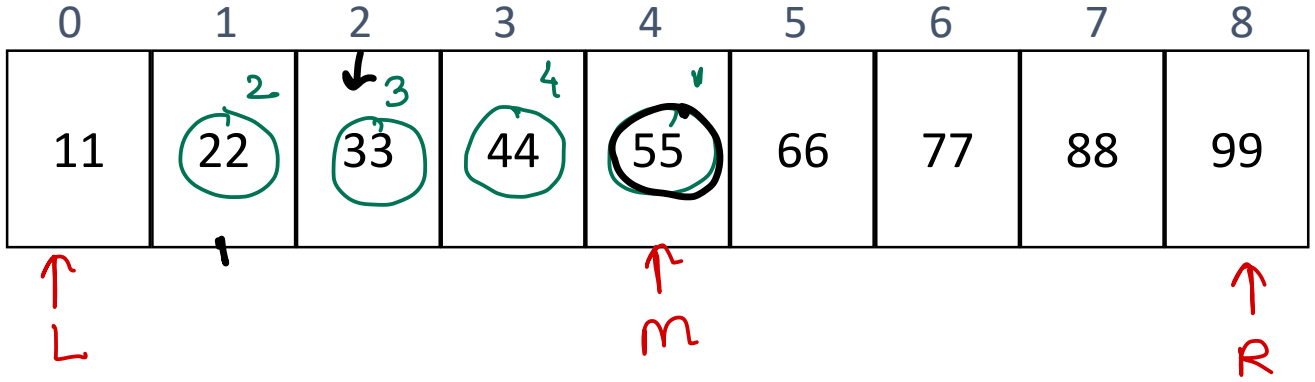
T α n/2
T α n
O(n)

Binary Search

key = 40

* array must be sorted.

```
while( L <= R )
{
    m = (L+R)/2
    if(key == a[m])
        return m;
    if( key < a[m])
        R = m-1;
    else
        L = m+1;
}
return -1;
```



16 ← 8 ← 4 ← 2 ← 1 ← x

$T \propto \log(n)$

$O(\log n)$

$2^{(i-1)} = n$

$(i-1)\log 2 = \log n$

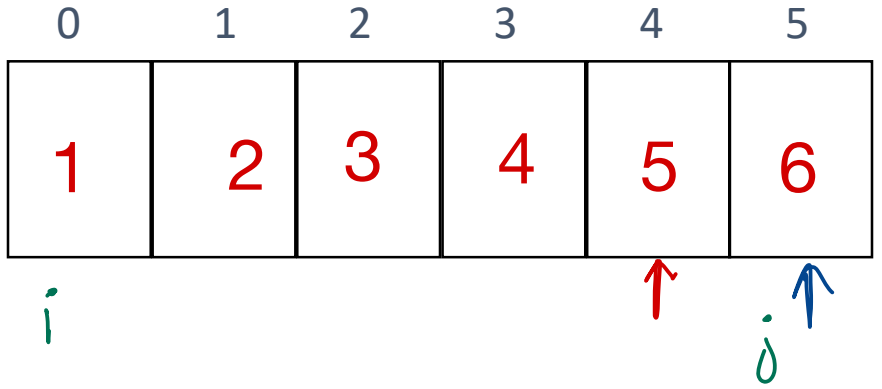
$i = \frac{\log n}{\log 2}$

$T \propto \frac{\log n}{\log 2}$

Selection Sort

5 4 2 6 3 1

```
for(i=0; i<n; i++) {
    for(j=i+1; j<n; j++) {
        if(a[i] > a[j])
            swap(a[i], a[j]);
    }
}
```



pass 1 : 5
pass 2 : 4
pass 3 : 3
pass 4 : 2
pass 5 : 1

itrs = 15

3

3

$(n-1) + (n-2) + (n-3) + \ldots + 1$

$itrs = \frac{n(n-1)}{2}$

$T \propto \frac{n(n-1)}{2}$

$T \propto n^2 - n$

$T \propto n^2$

$n >> 1$

$n^2 >>>> n$

Theory of Approximation

$O(n^2)$

Bubble Sort

5 4 2 6 3 1

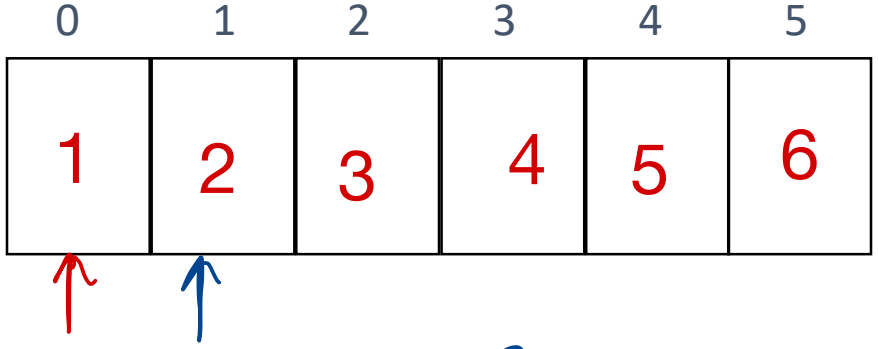
```
for(i=0; i<n-1; i++){
    flag = 0;
    for(j=0; j<n-1; j++){
        if(a[j] > a[j+1]){
            flag = 1;
            swap(a[j], a[j+1]);
        }
    }
    if(flag==0)
        return;
}
```

itrs = n-1

$T \propto n-1$

$O(n)$

best case



1

itrs = (n-1) * (n-1)

$= n^2 - 2n + 1$

$T \propto n^2 - 2n + 1$

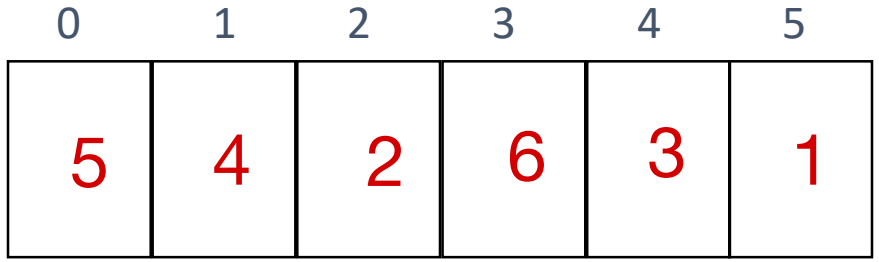
$T \propto n^2$

$O(n^2)$

Insertion Sort

insertion sort

```
for(i=1; i<n; i++) {
    temp=arr[i]
    for(j=i-1; j>=0 && a[j]>temp; j--)
        a[j+1] = a[j];
    a[j+1] = temp;
}
```



$$it = 1+2+3+\dots+(n-1)$$

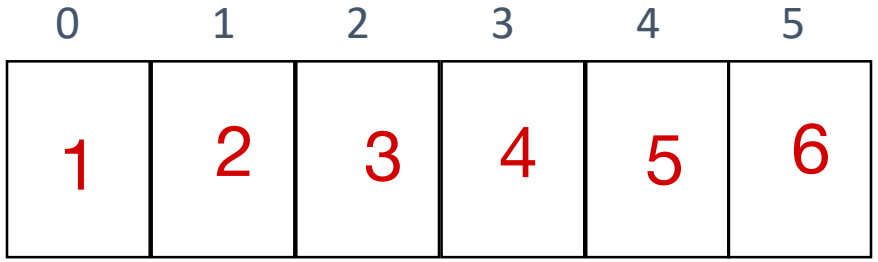
$$it = \frac{n(n-1)}{2}$$

$$O(n^2)$$

pass 1 : 1
pass 2 : 2
pass 3 : 3
⋮
n-1

Insertion Sort

```
insertion sort
for(i=1; i<n; i++) {
    temp=arr[i]
    for(j=i-1; j>=0 && a[j]>temp; j--)
        a[j+1] = a[j];
    a[j+1] = temp;
}
```



Best → O(n)

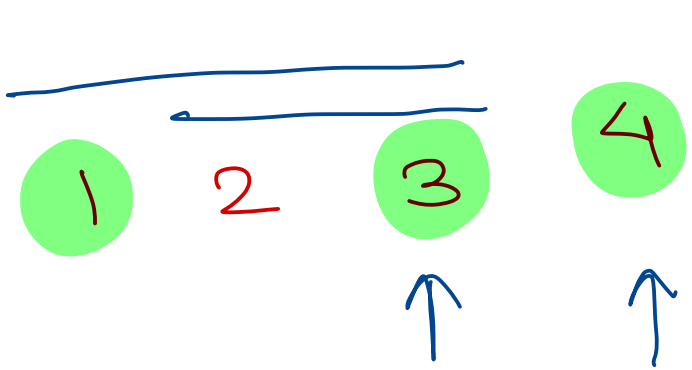
O(1)

Quick Sort

⑥ if L == R (single ele) or L > R (invalid part). then return.

① Consider an ele as pivot. (a[L]).
② from left find ith ele, so that ith ele is greater than pivot.
③ from right find jth ele, so that jth ele is less or equal to the pivot.
④ if i & j are not crossed each other, swap ith ele with jth ele.
⑤ repeat steps 2 to 4, until i & j not crossing.
⑥ swap pivot ele with jth ele. This completes pass 1.
   * all eles to left of pivot are smaller than that.
   * all eles to right of pivot are greater than that
   * pivot ele is sorted.
⑦ apply quick sort to left part: L to j-1
⑧ apply quick sort to right part: j+1 to R

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



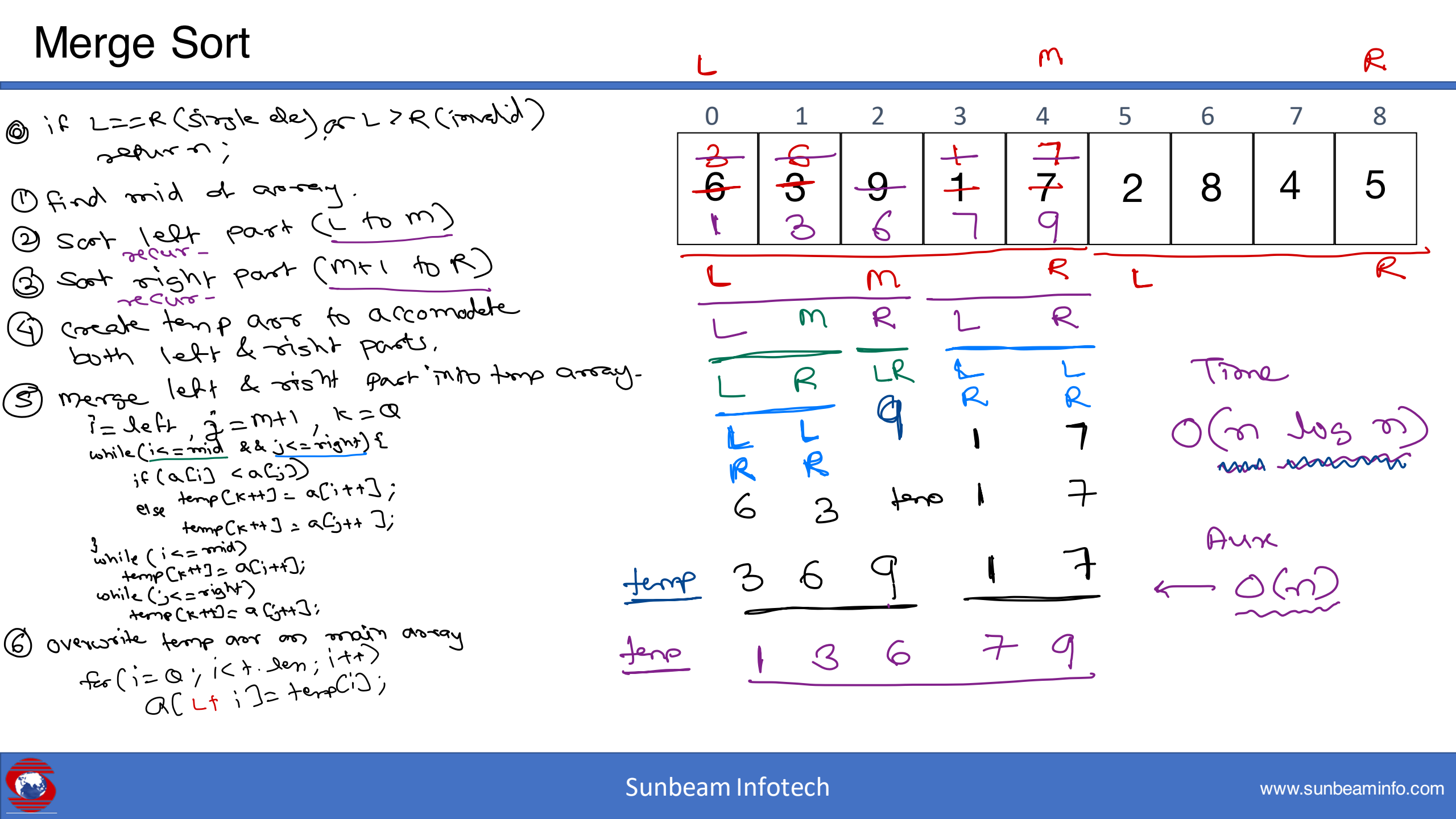
$T \propto n(n-1)$

$O(n^2)$

Quick Sort – Time complexity

- Quick sort pivot element can be
  - First element or Last element
  - Random element
  - Median of the array
- Quick sort time
  - Time to partition as per pivot – T(n)
  - Time to sort left partition – T(k)
  - Time to sort right partition – T(n-k-1)
- Worst case
  - $T(n) = T(0) + T(n-1) + O(n) \Rightarrow O(n^2)$
- Best case
  - $T(n) = T(n/2) + T(n/2) + O(n) \Rightarrow O(n \log n)$
- Average case
  - $T(n) = T(n/9) + T(9n/10) + O(n) \Rightarrow O(n \log n)$

Merge Sort


if L==R (single ele) or L > R (invalid)
return;
find mid of array.
Sort left part (L to m)
recur-
Sort right part (m+1 to R)
recur-
create temp arr to accomodate both left & right parts.
Merge left & right part into temp array.
i = left, j = m+1, k = 0
while(i<=mid && j<=right){
if(a[i] < a[j])
temp[k++] = a[i++];
else
temp[k++] = a[j++];
}
while(i<=mid)
temp[k++] = a[i++];
while(j<=right)
temp[k++] = a[j++];
overwrite temp arr on main array
for(i=0; i<t.len; i++)
a[Lf i] = temp[i];
L
m
R
0 1 2 3 4 5 6 7 8
6 3 9 1 7 2 8 4 5
1 3 6 7 9
L m R L R
L m R L R
L R LR L R
L L 9 1 7
R R
6 3 temp 1 7
temp 3 6 9 1 7
temp 1 3 6 7 9
Time
O(n log n)
Aux
O(n)

~~Insertion Sort~~

lin search → O(n) Ω(1)

Θ(n)

O(1) → T = 1

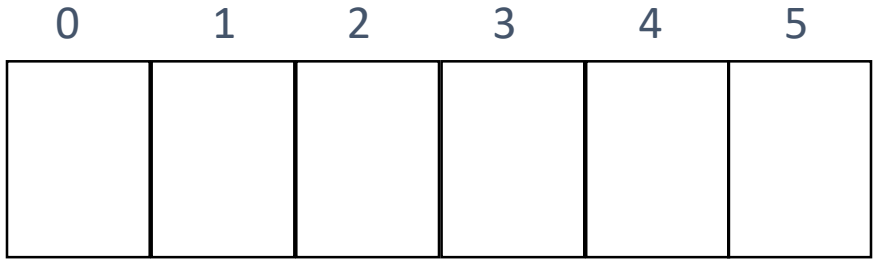
O(n) → T ∝ n → single loop

O(n²) → nested loop

O(n³) → nested triple loop

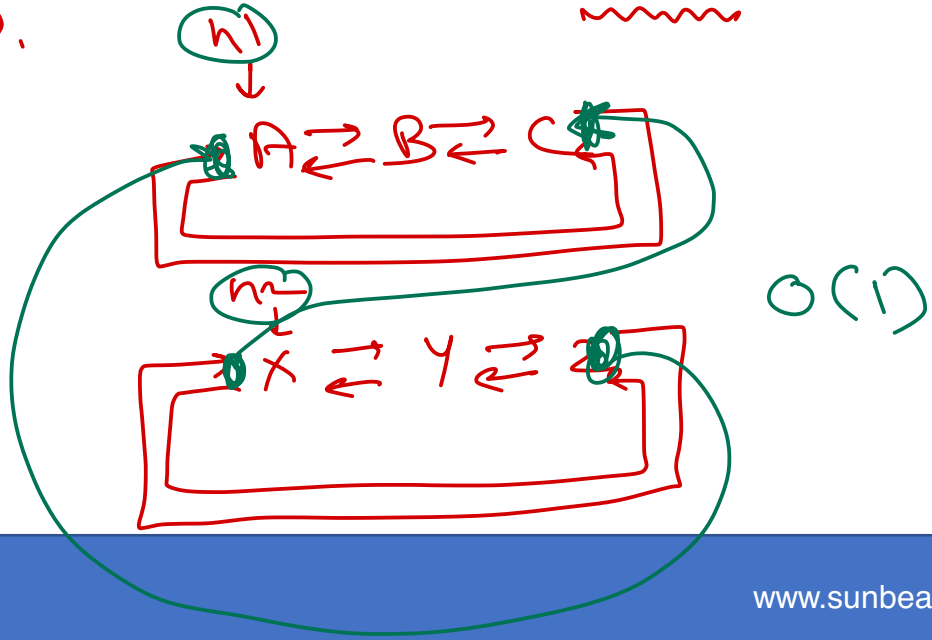
O(log n) → partitioning

O(n log n) → partitioning within loop.



print table of 2

O(1)



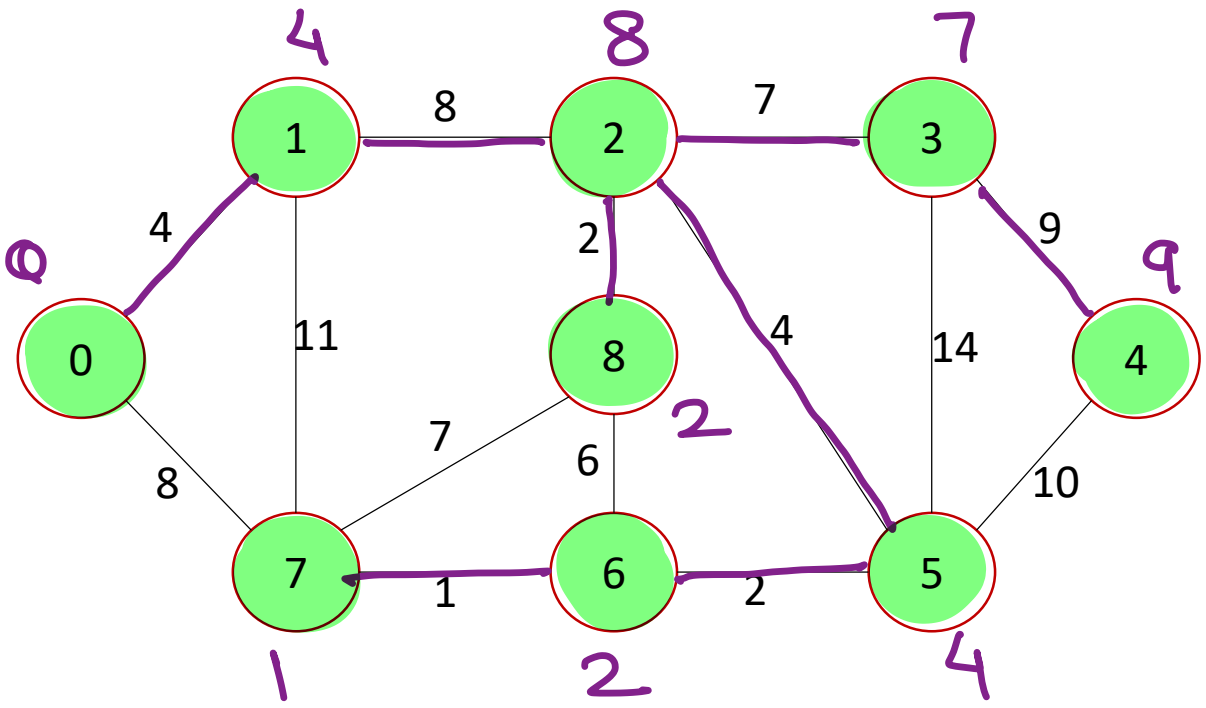
worst case → O(n) → upper bound

best case → Ω(n) → lower bound

avg case → Θ(n) → upper & lower bound

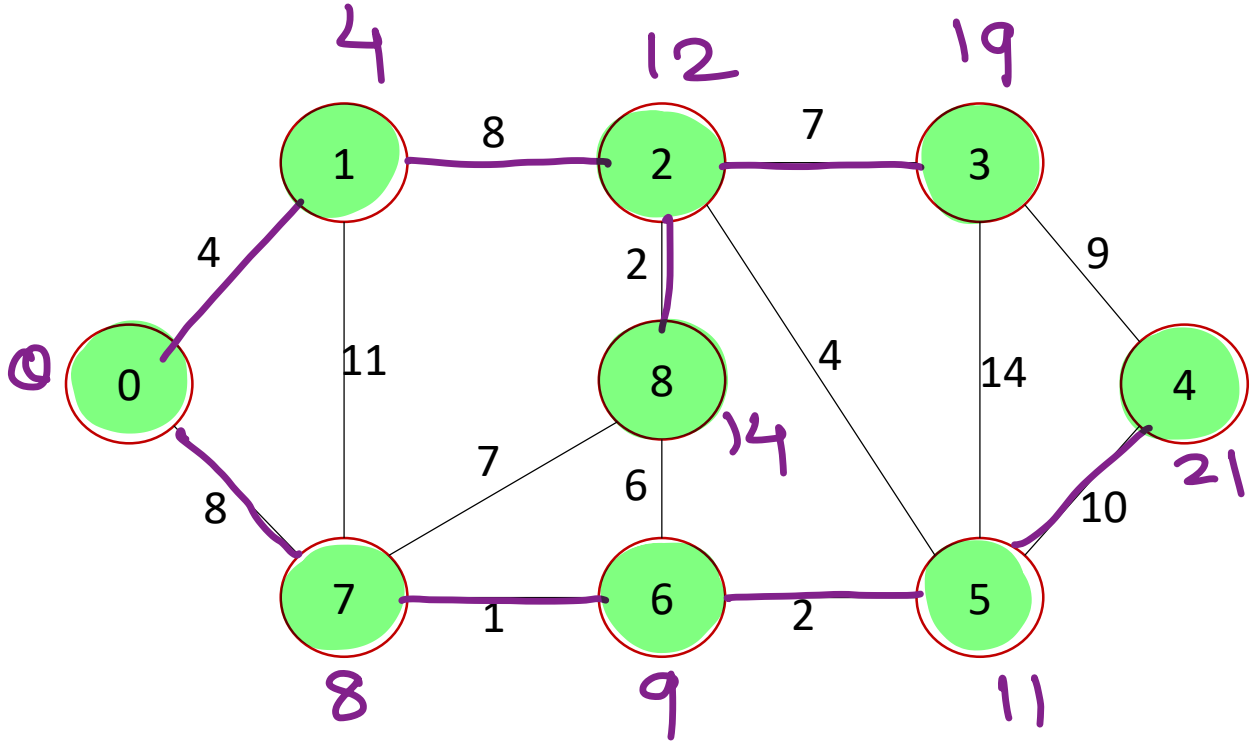
Prim's MST

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 src shortest path algo.

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.

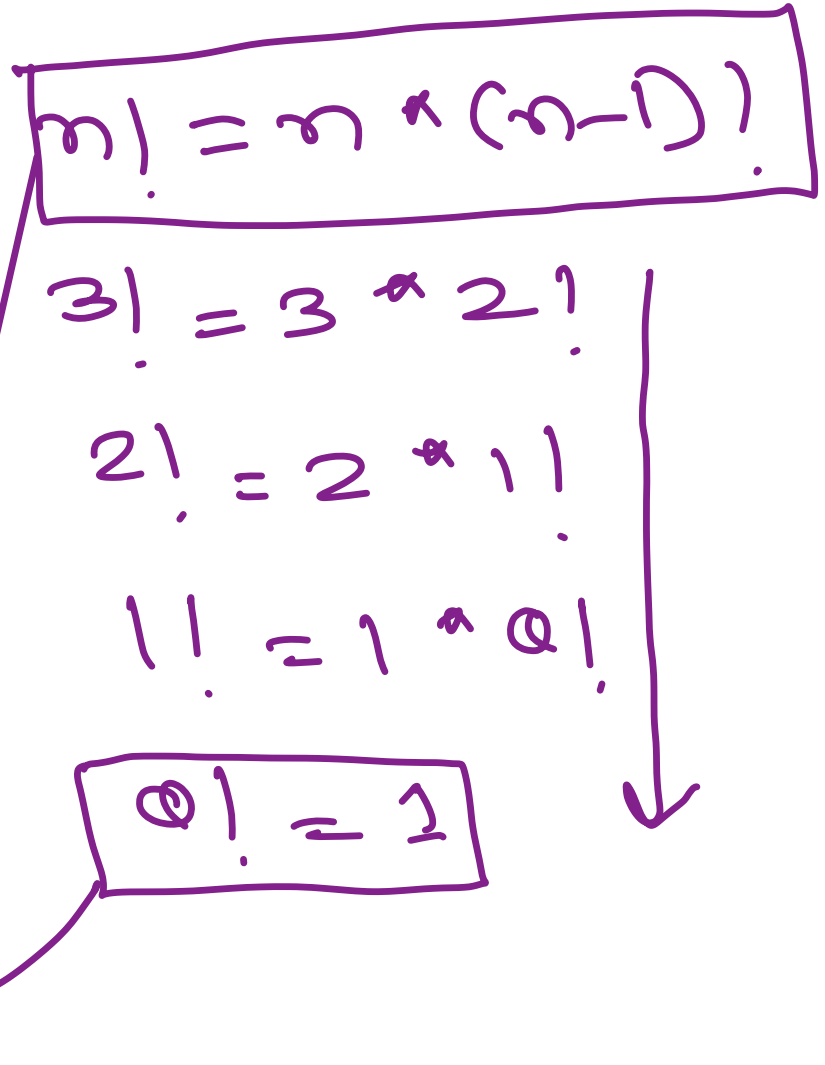
- Time complexity (adjacency matrix)
  - V vertices: O(V)
  - get min key vertex: O(V)
  - update adjacent: O(V)

  } O(2V)
- Time complexity (adjacency matrix)
  - $O(V^2)$

- Time complexity (adjacency list)
  - V vertices: O(V) →
  - get min key vertex: O(log V) ← using heap.
  - update adjacent: O(E) – E edges ← for all items of outer.
- Time complexity (adjacency list)
  - O(E log V) ✓

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)



$$x^y = x * x^{y-1}$$

$$\vdots$$

$$x^0 = 1 \quad [y == 0]$$

$$factors(n) = prime\_factor\_of\_n * factors\left(\frac{n}{prime}\right)$$

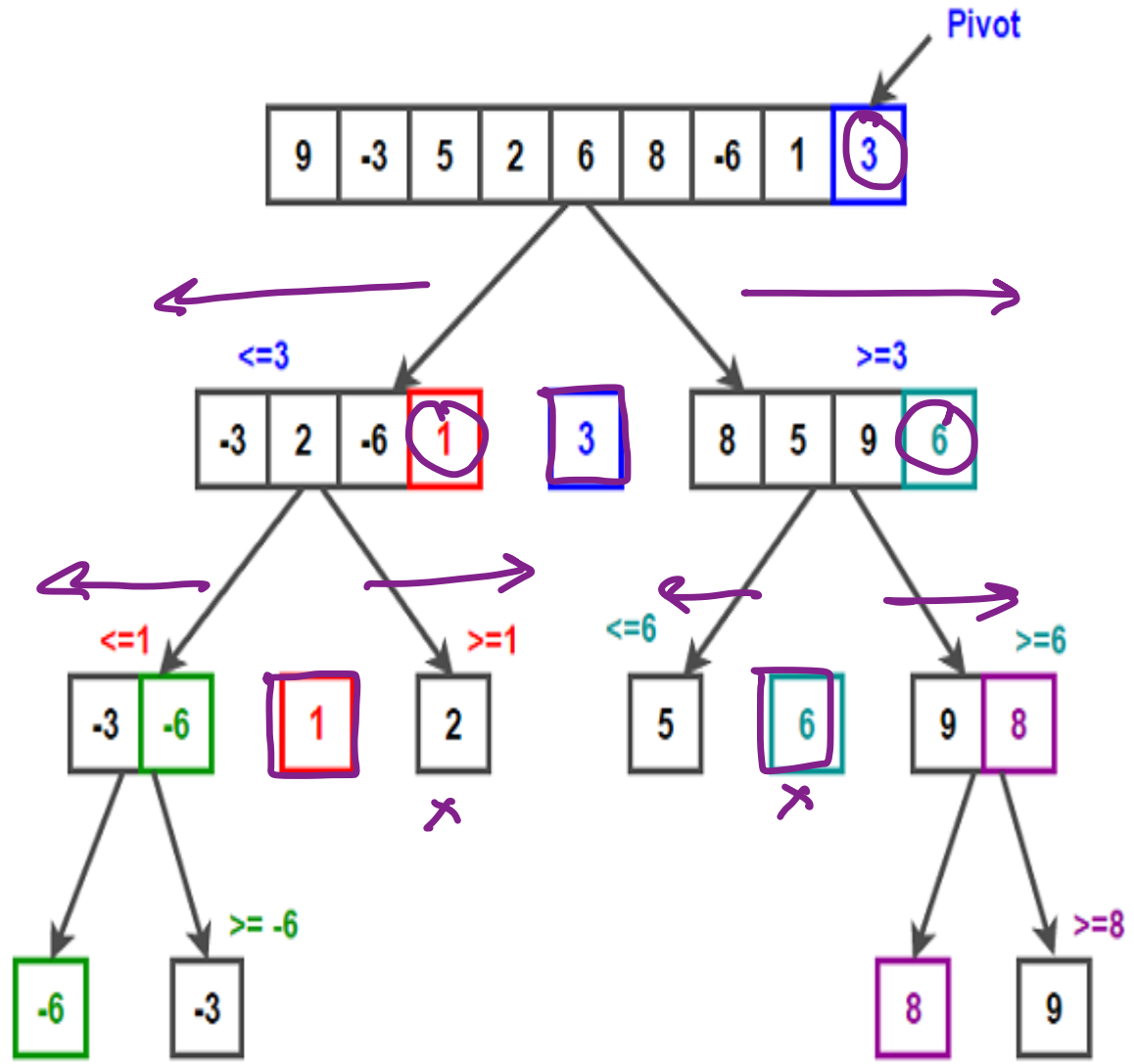
$$factors(1) = 1$$

$$T_n = T_{n-1} + T_{n-2}$$

$$T_1 = T_2 = 1$$

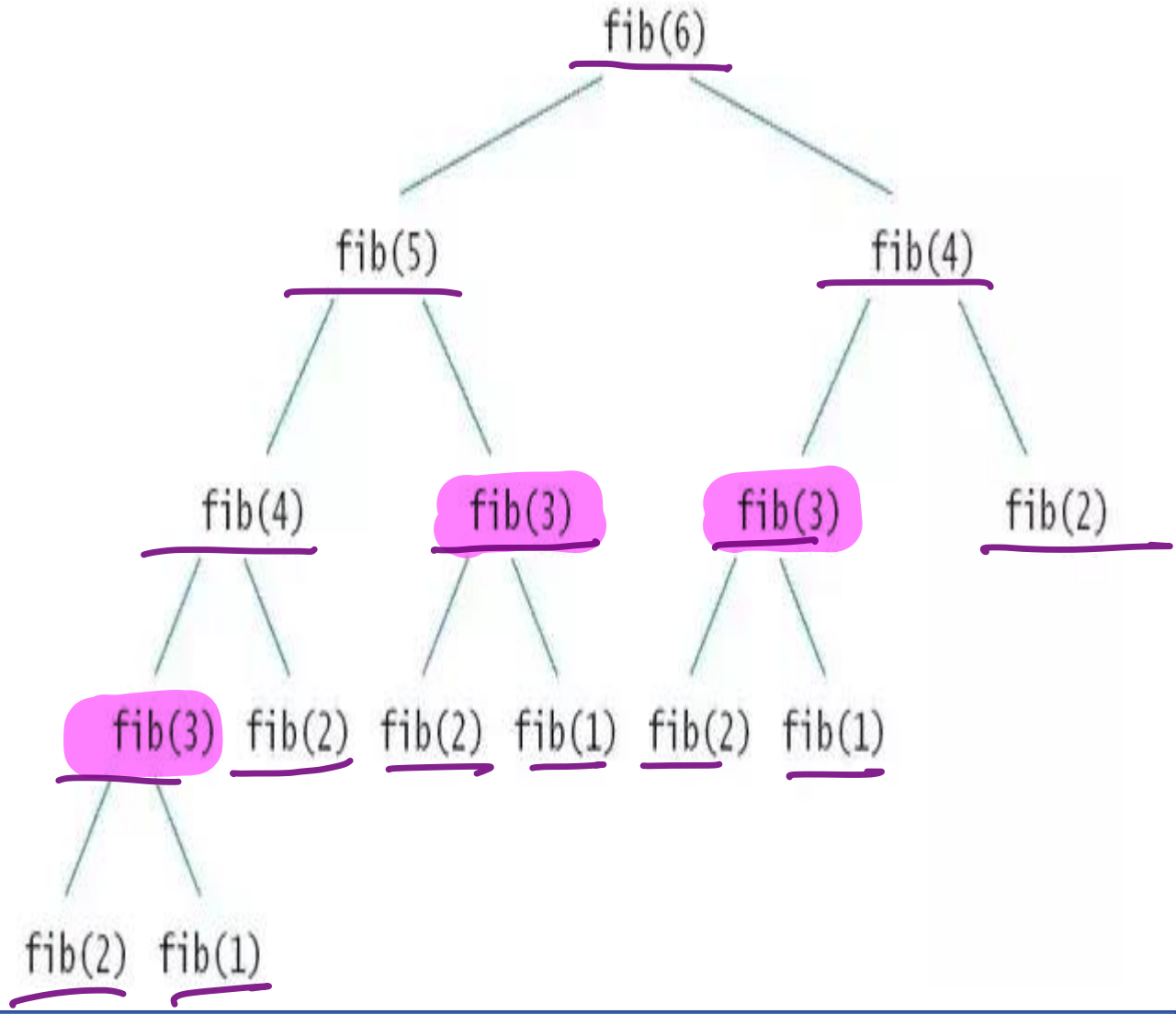
Recursion – QuickSort

- 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

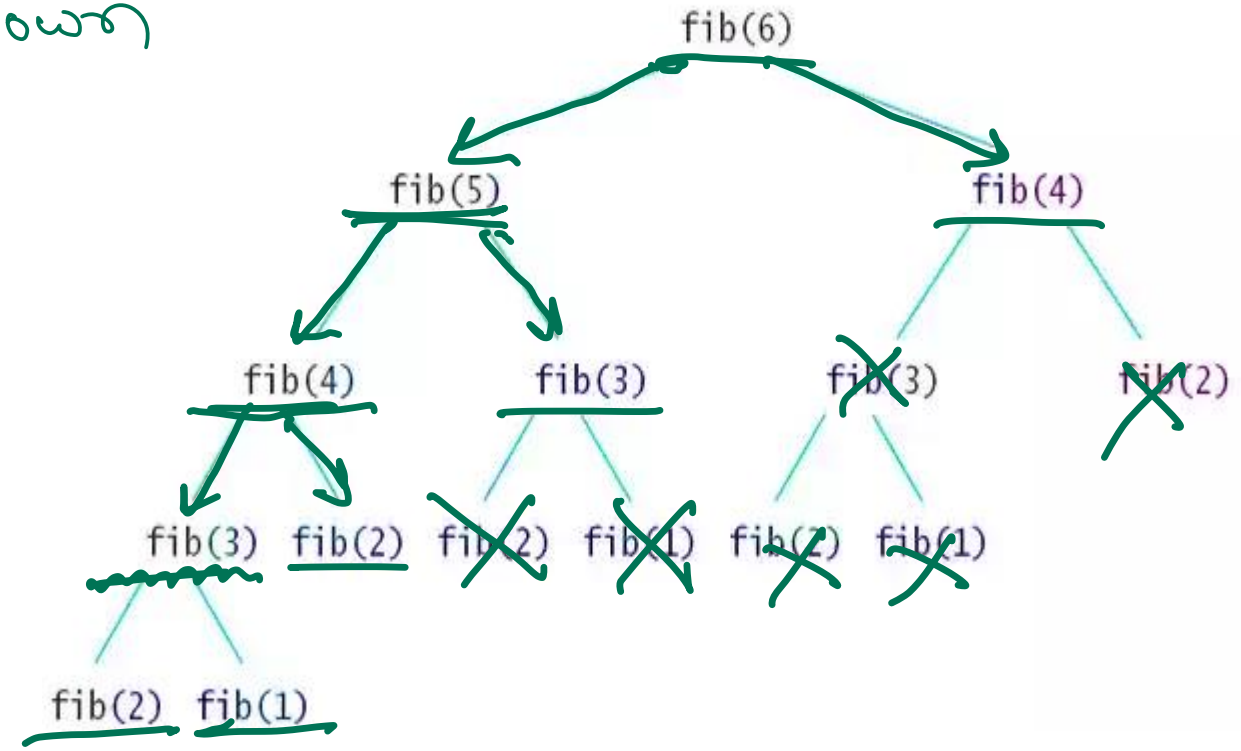
- Recursive formula
  - $T_n = T_{n-1} + T_{n-2}$
- Terminating condition
  - $T_1 = T_2 = 1$
- Overlapping sub-problem

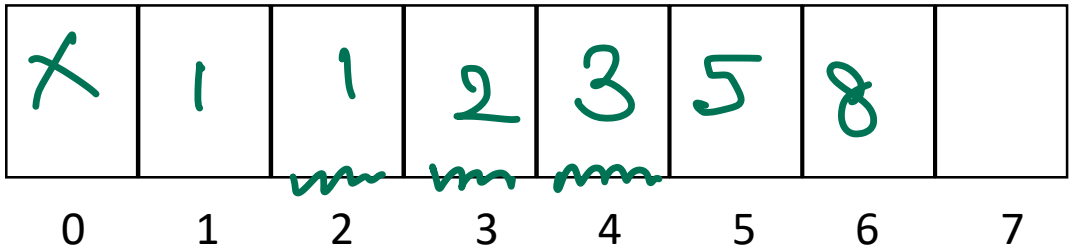


Memoization – Fibonacci Series

- 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.

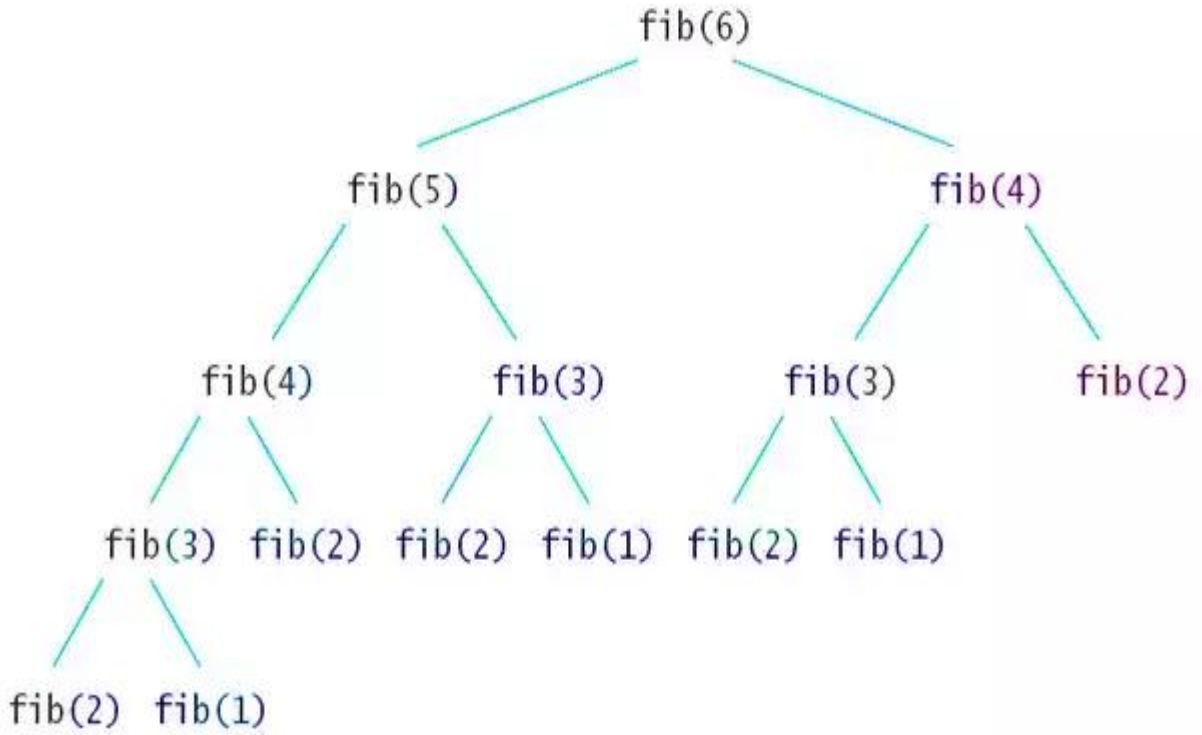
top-down





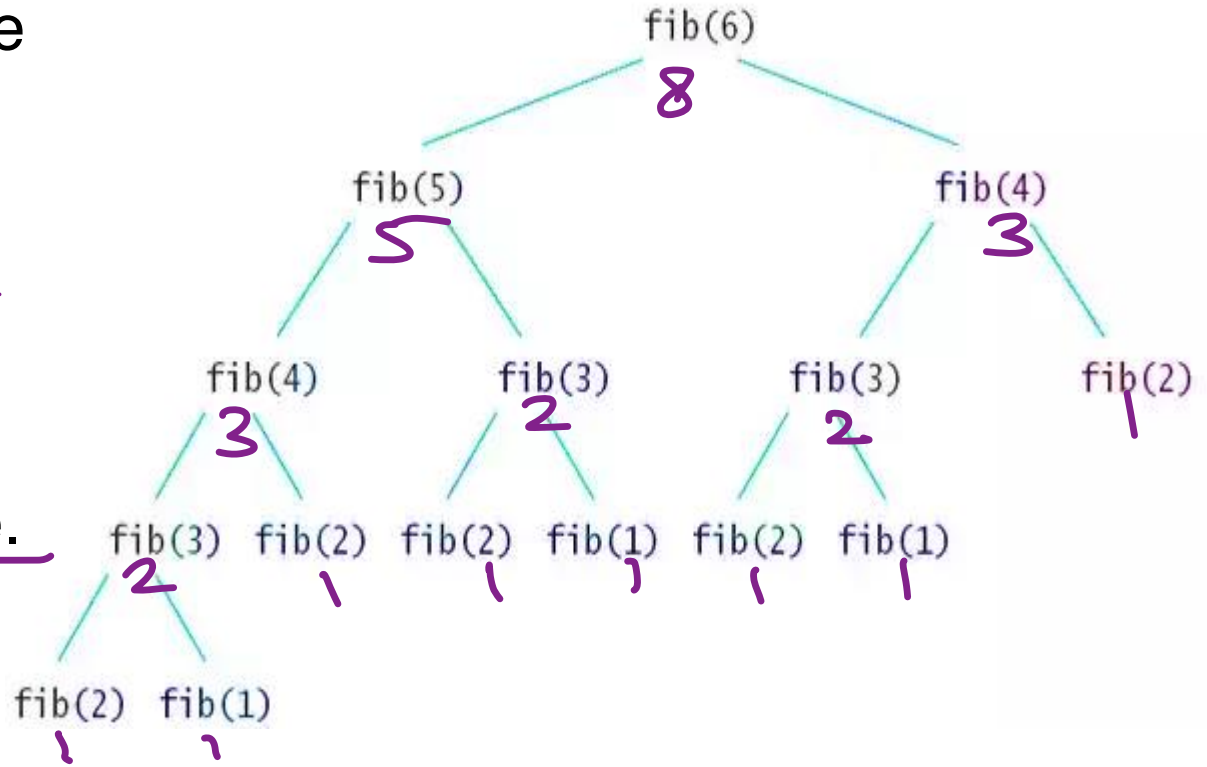
Dynamic Programming

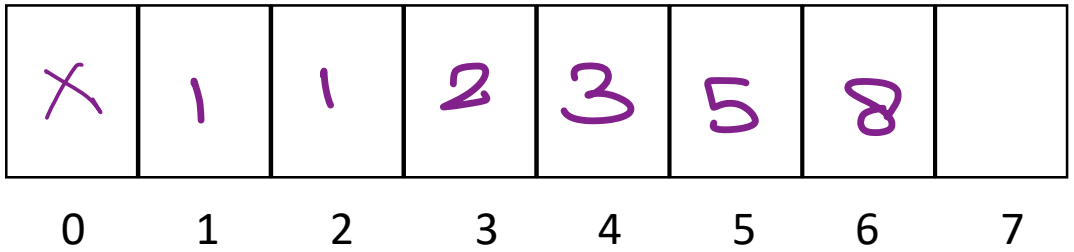
- Dynamic programming is another optimization over recursion.
- Typical DP problem give choices (to select from) and ask for optimal result (maximum or minimum).
- Technically it can be used for the problems having two properties
  - Overlapping sub-problems
  - Optimal sub-structure
- To solve problem,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



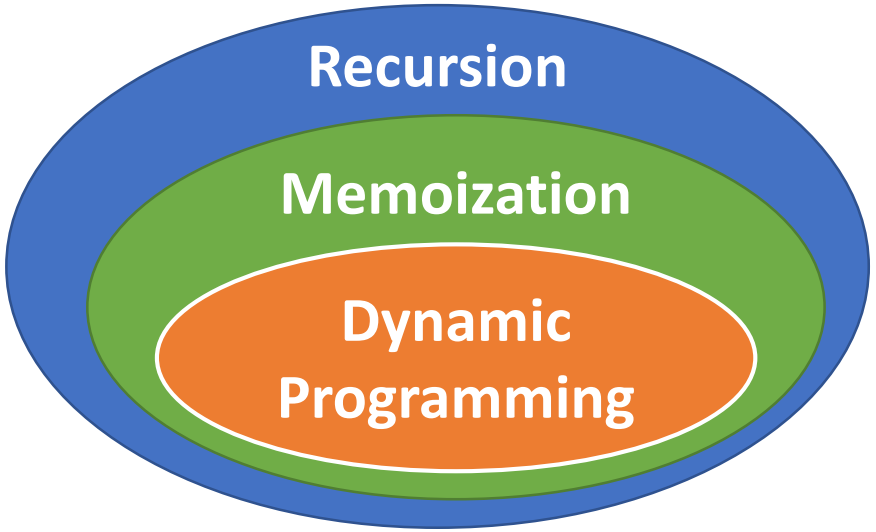
Dynamic Programming – Fibonacci Series

- 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.





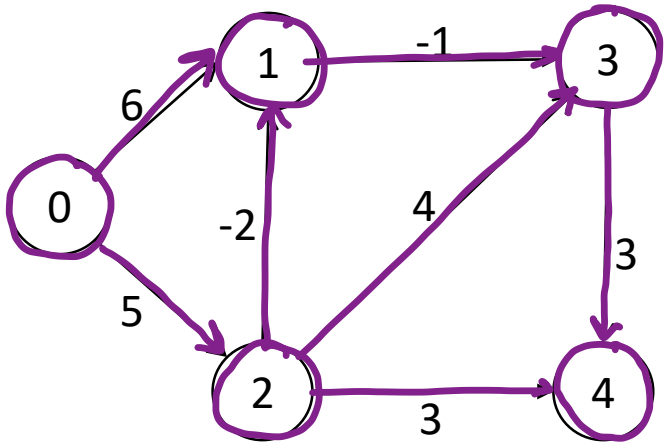
Dynamic Programming



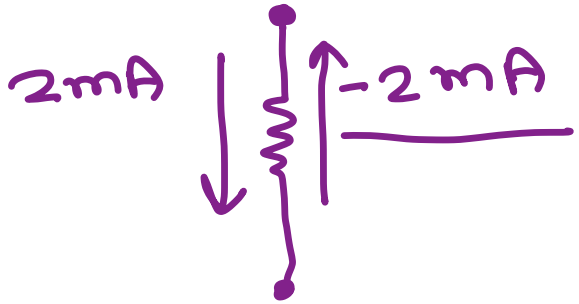
- Top down
- Optimized Top down
- Bottom up

Bellman Ford Algorithm

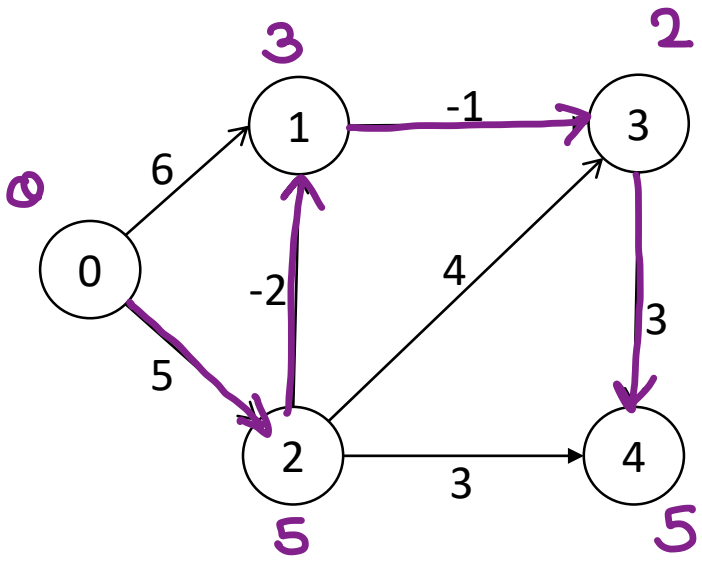
- Initializes distances from the source to all vertices as infinite and distance to the source itself as 0.
- Calculates shortest distance V-1 times: For each edge u-v, if dist[v] > dist[u] + weight of edge u-v, then update dist[v], so that dist[v] = dist[u] + weight of edge u-v.
- Check if negative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

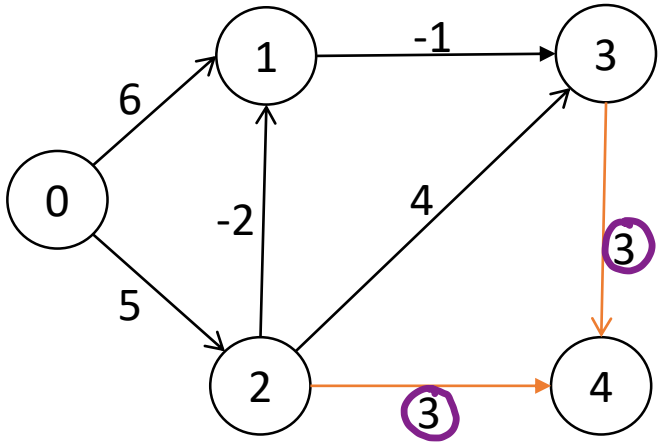


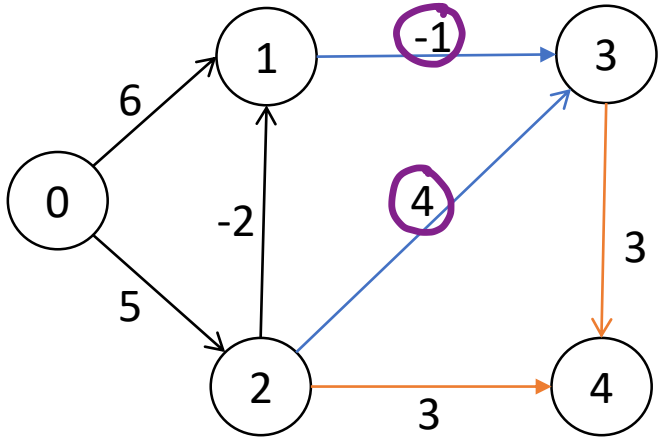
Src	Des	Wt
3	4	3
2	4	3
2	3	4
2	1	-2
1	3	-1
0	2	5
0	1	6

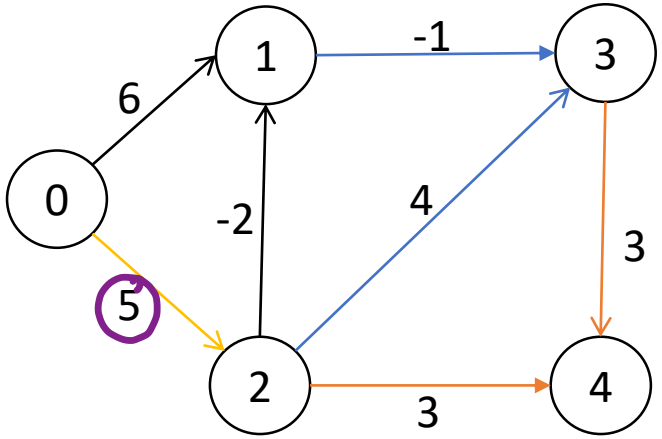


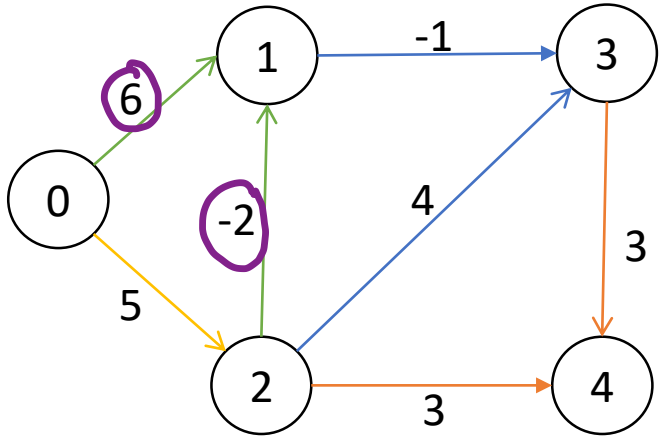
Bellman Ford Algorithm





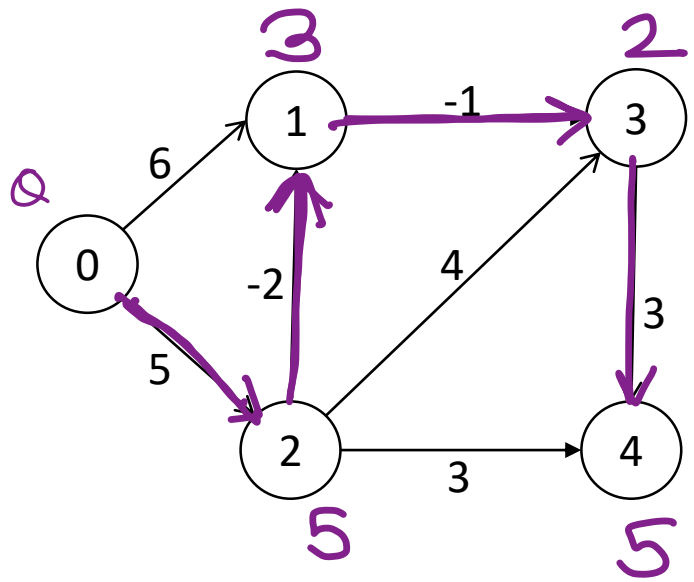






Bellman Ford Algorithm

	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



E V

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

O(VE)

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).

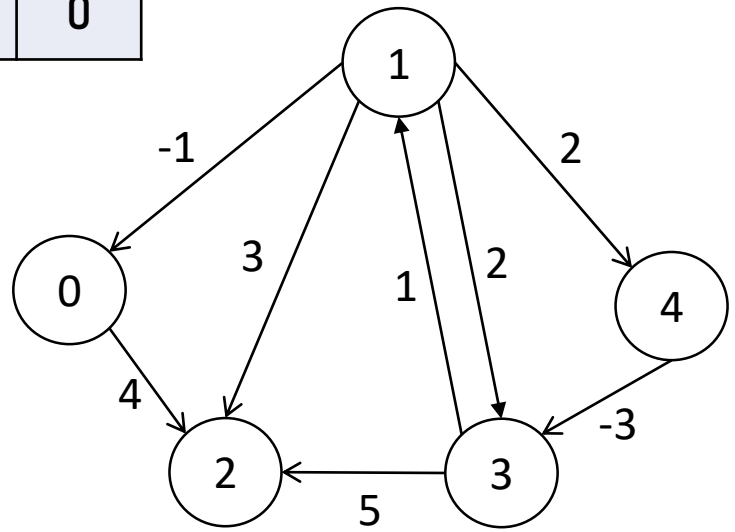
	0	1	2	3	4
0	0	∞	4	∞	∞
1	-1	0	3	2	2
2	∞	∞	0	∞	∞
3	∞	1	5	0	∞
4	∞	∞	∞	-3	0

Bellaman T.C. for one vertex : O(VE)

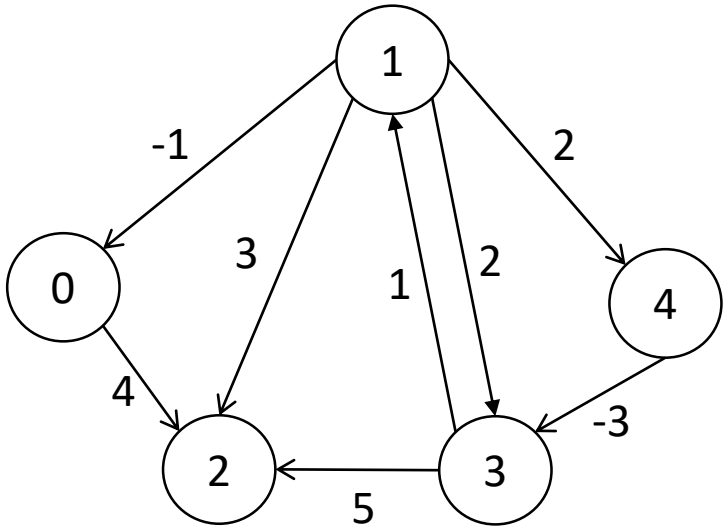
Bellaman for all vertices : O(V * VE)
↳ $O(V^2E)$

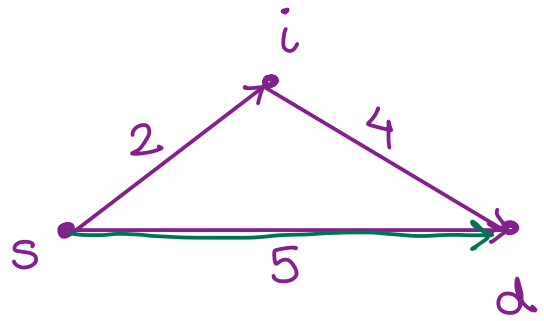
Dijkstra T.C. for one vertex → $O(V \log V)$

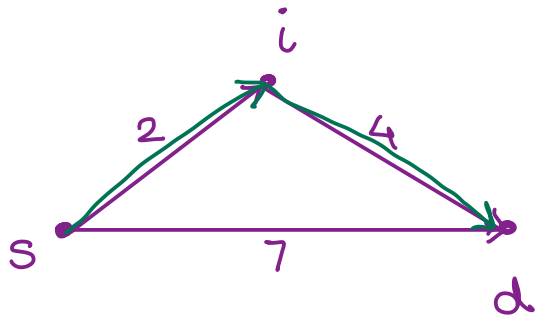
Dijkstra for all vertices → $O(V * V \log V)$
$O(V^2 \log V)$
↳ using Dijkstra for all pair shortest path.



Warshall Floyd Algorithm → O(V³)



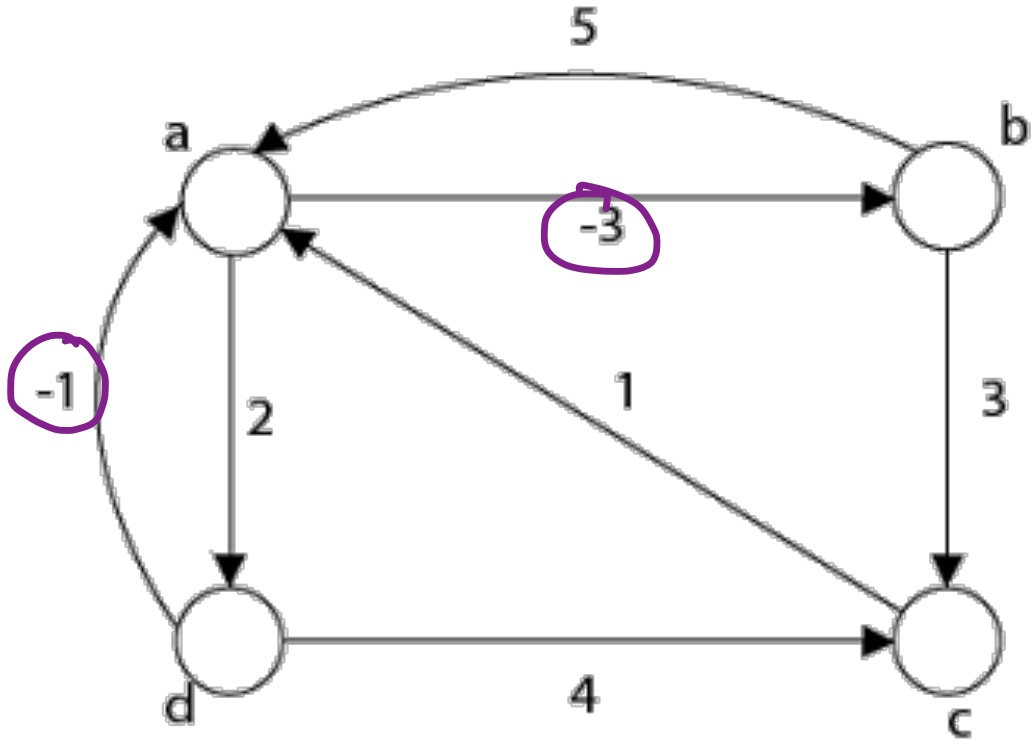




Johnson's Algorithm → All pair shortest Path algo.

- Time complexity of Warshall Floyd is $O(V^3)$.
- 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:

$$O(VE + V^2 \log V).$$ < $O(V^3)$

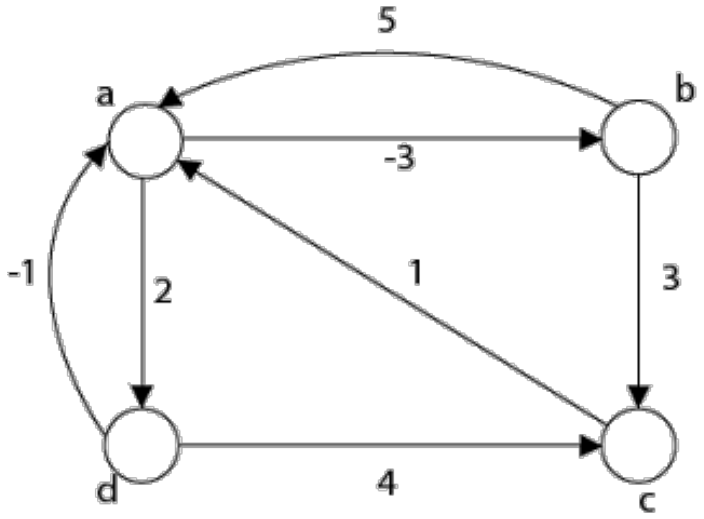


Johnson's Algorithm

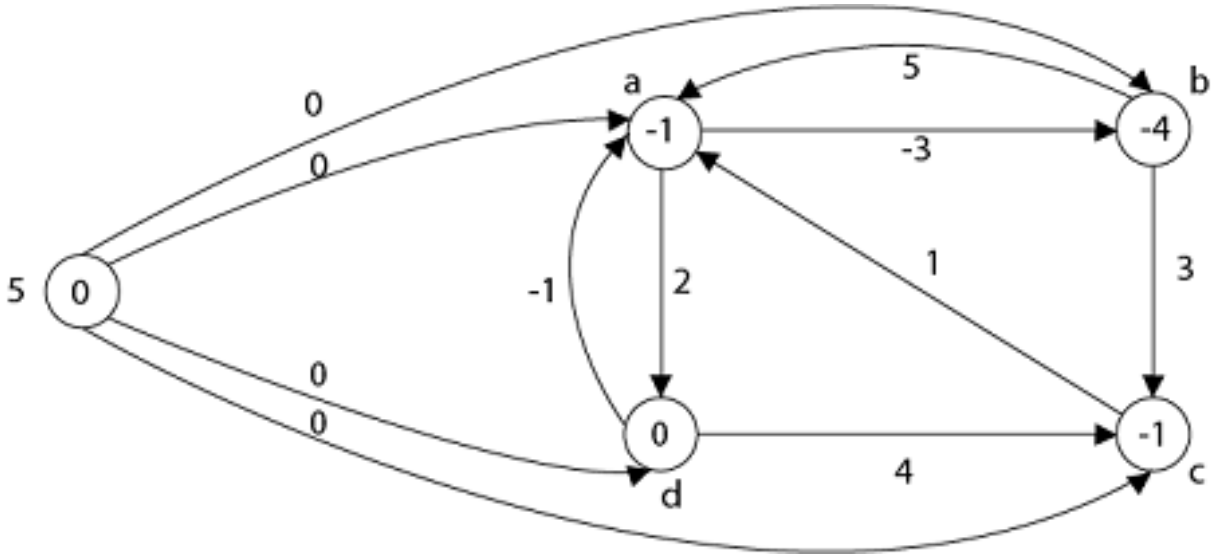
1. Add a vertex (s) into a graph and add edges from it all other vertices, with weight 0.
2. Find shortest distance of all vertices from (s) using Bellman Ford algorithm.

   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

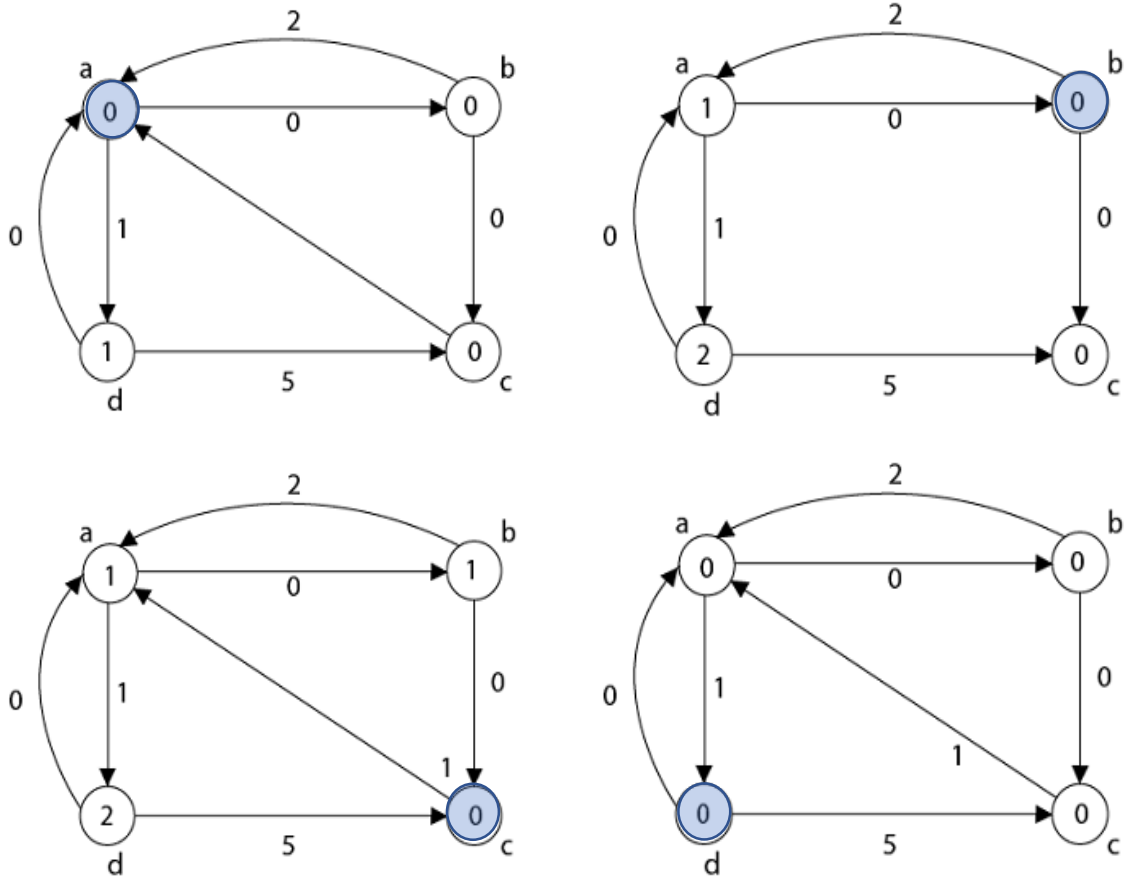


Johnson's Algorithm

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 shortest path.

- 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.

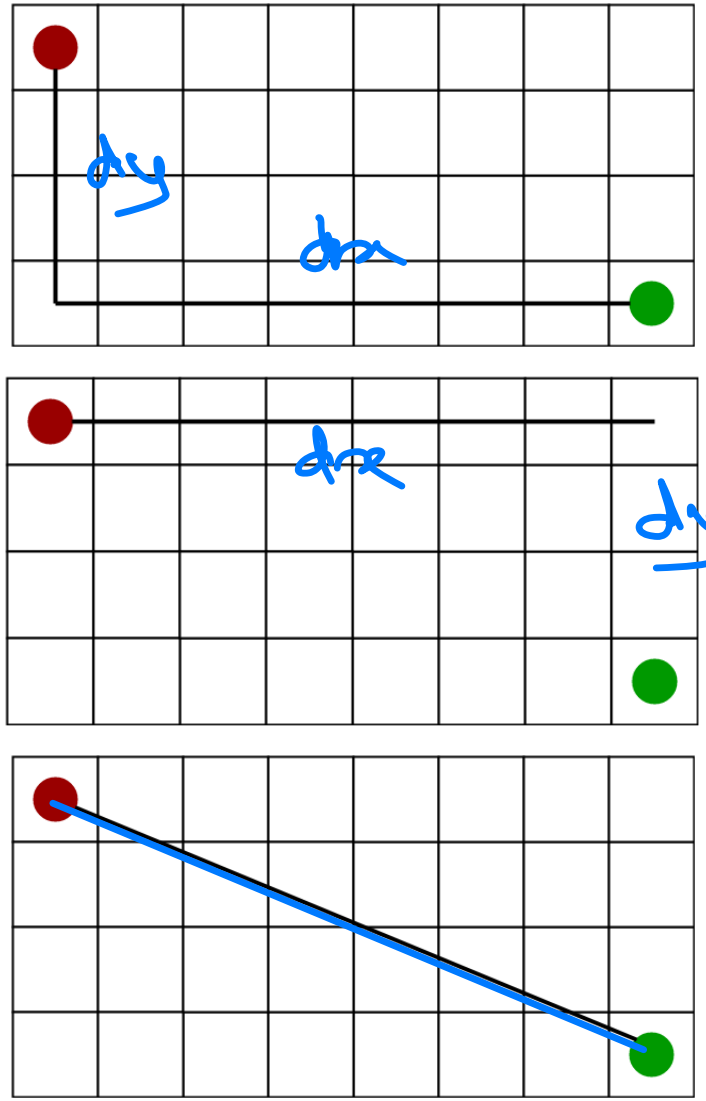
g(v)

	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

h(v)

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

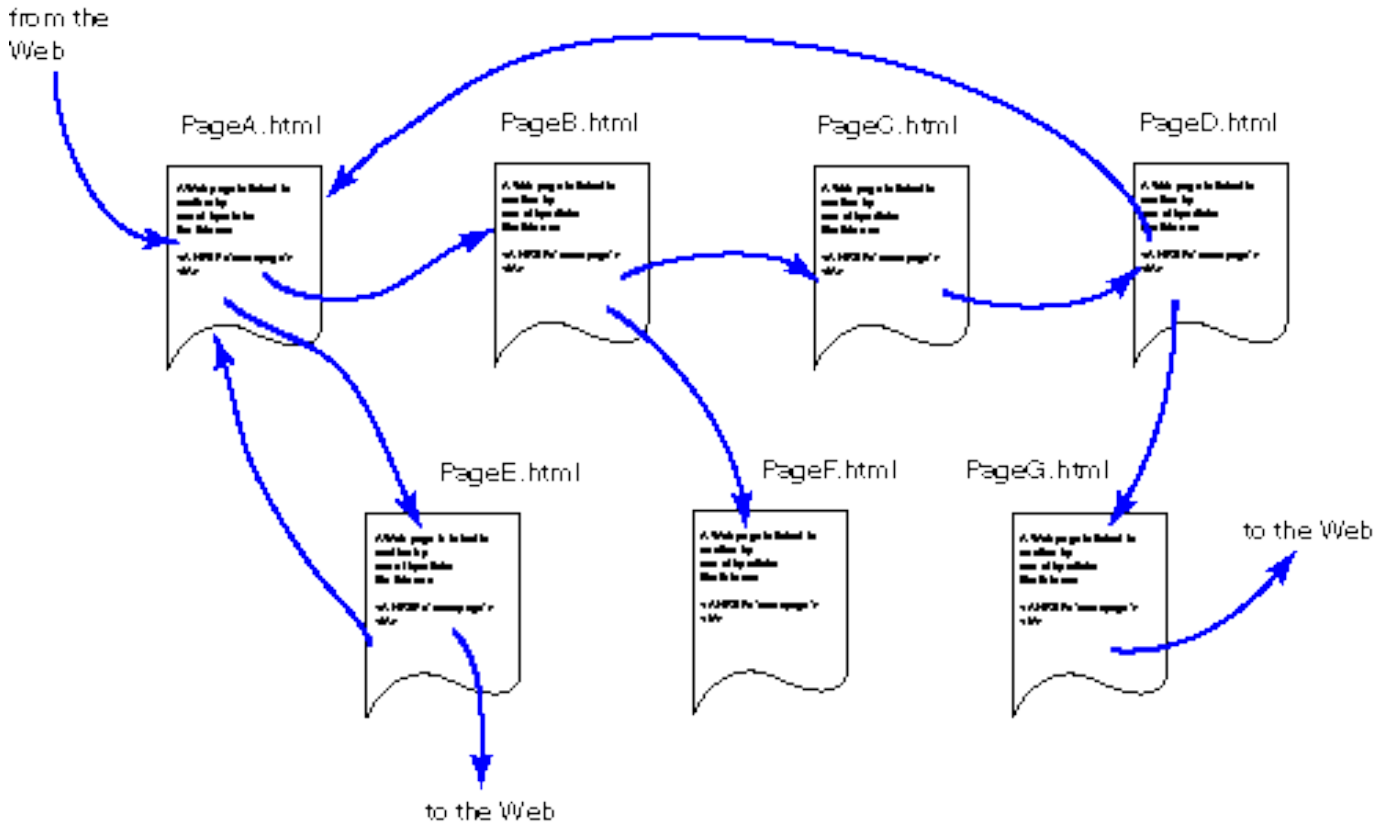
- Graph represents flow of computation/tasks. It is used for resource planning and scheduling. MST algorithms are used for resource conservation. DAG are used for scheduling in Spark or Tez.

- 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.

Graph applications

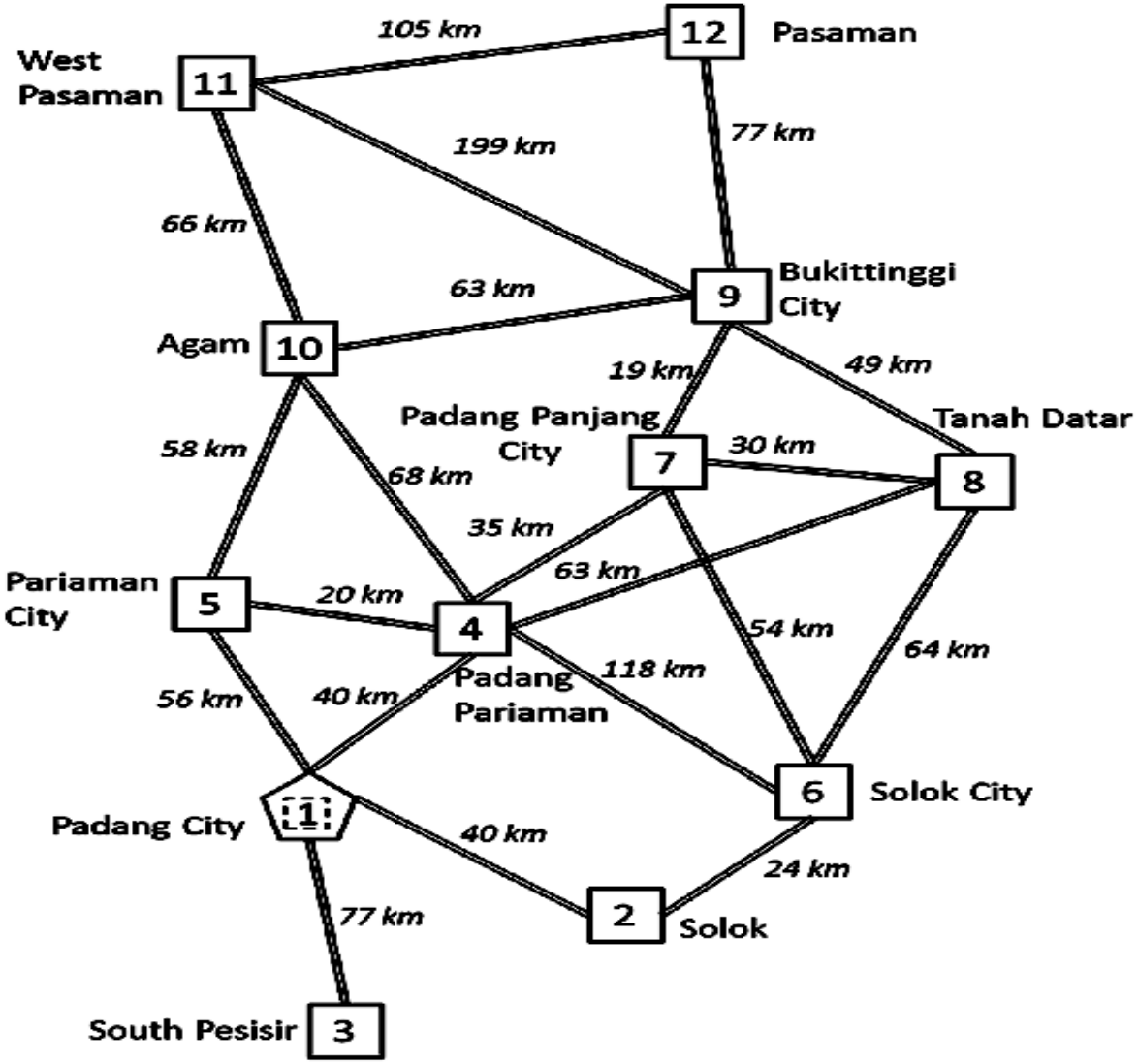
- In world wide web, web pages are like vertices; while links represents edges. This concept can be used at multiple places.
  - Making sitemap
  - Downloading website or resources
  - Developing web crawlers
  - Google page-rank algorithm



PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

Graph applications

- 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.



Thank you!

Nilesh Ghule <Nilesh@sunbeaminfo.com>