Computer Engineering - Sem IV

NCMPC 41 : Design and Analysis of Algorithms

Module - 3 : Greedy Method
(6 Hours)

Instructor : Mrs. Lifna C S

Department of Computer Engineering, VESIT, Mumbai

Topics to be covered

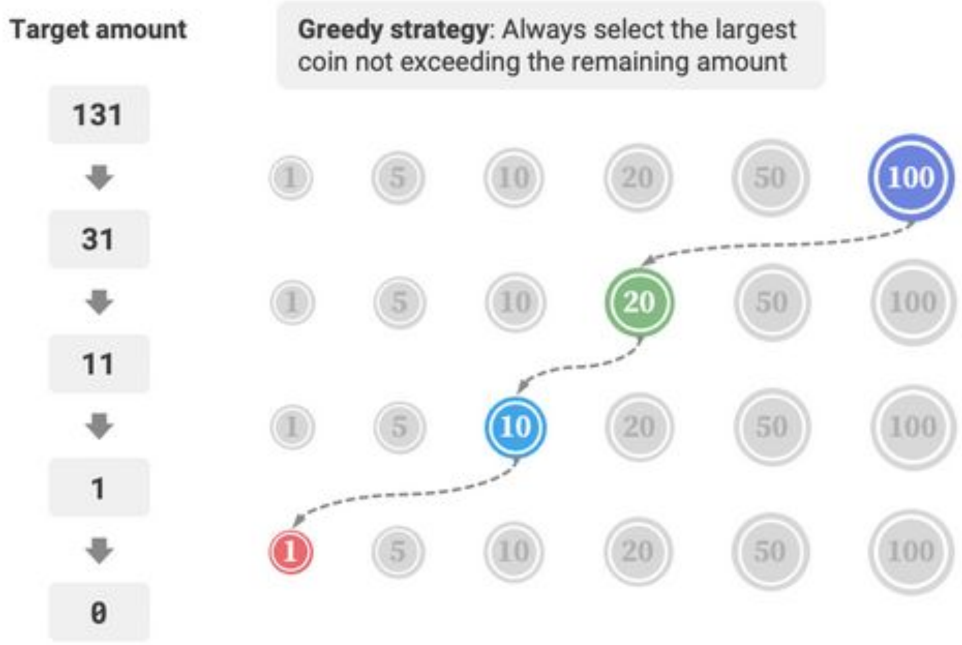
- **General method**
- Fractional Knapsack problem,
- Job sequencing with deadlines,
- Single source shortest path: Dijkstra Algorithm
- Minimum cost spanning trees:
    - Kruskal Algorithm
    - Prim's algorithms

Extra Topics:

- Coin Change Problem
- Disjoint Set Data Structure

Typical Problem - Coin Change

Given n types of coins, where the denomination of the ith type of coin is $coins[i-1]$, and the target amount is amt, with each type of coin available indefinitely, what is the minimum number of coins needed to make up the target amount? If it is not possible to make up the target amount, return -1.

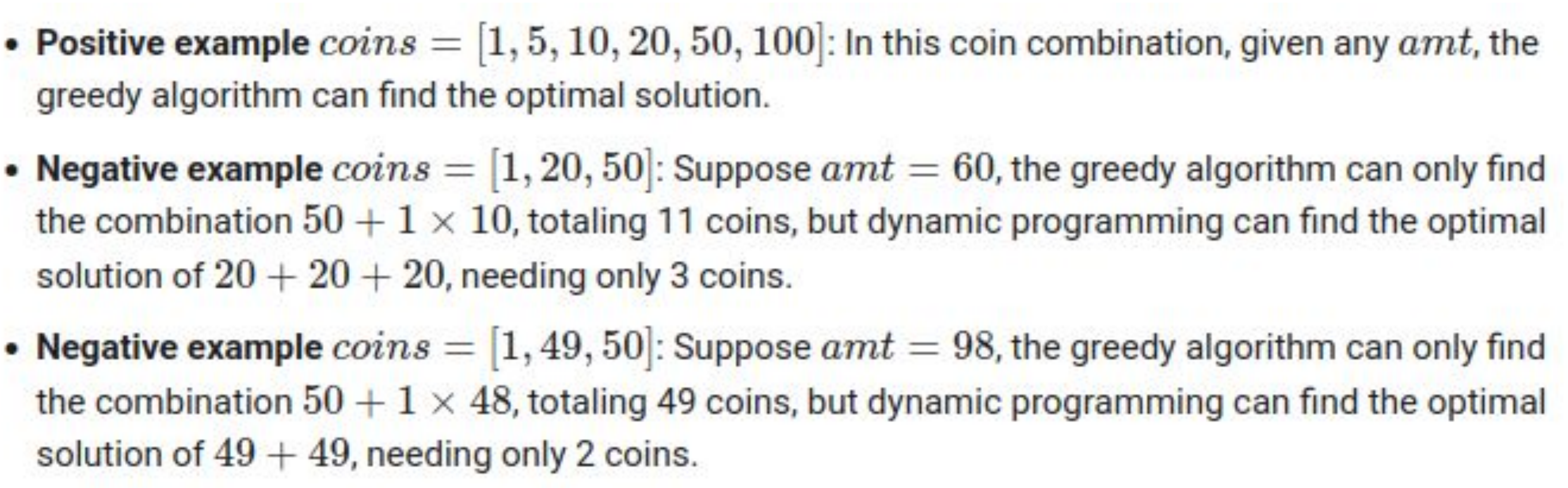


Target amount
Greedy strategy: Always select the largest coin not exceeding the remaining amount
131
31
11
1
0
1
5
10
20
50
100

<u>Approach :</u>
Given the target amount, **we greedily choose the coin that is closest to and not greater than it**, repeatedly following this step until the target amount is met.

Courtesy : Hello algo

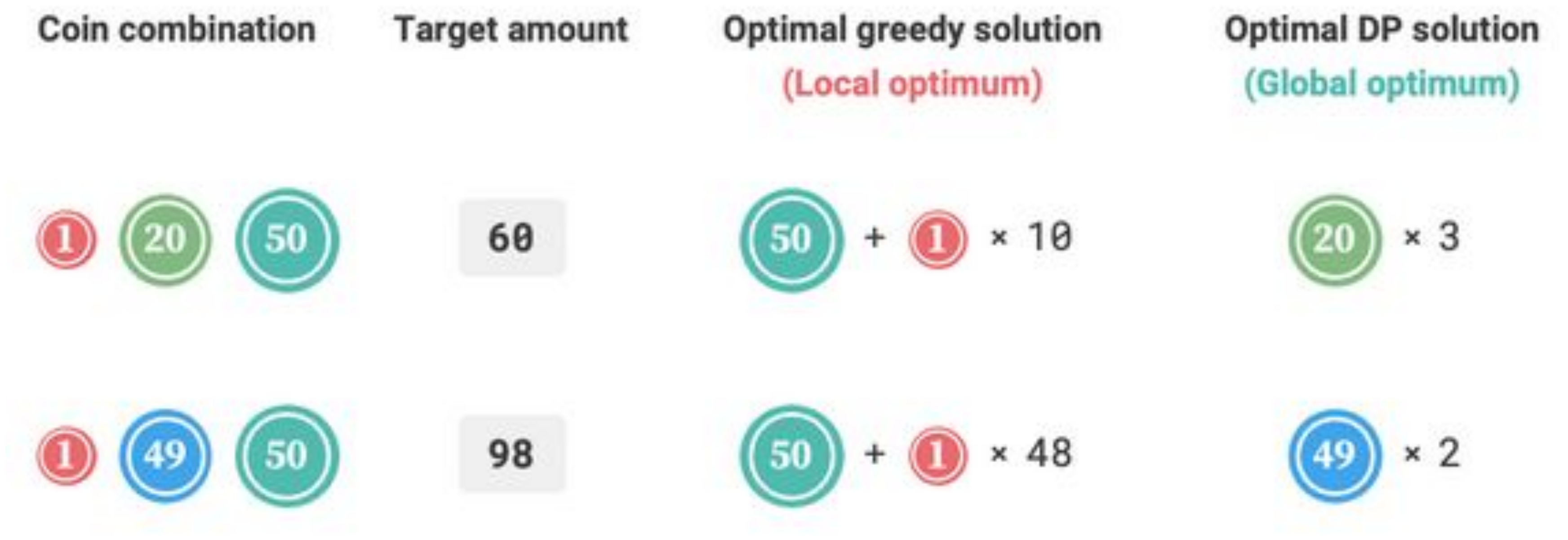
Typical Problem - Coin Change



- **Positive example** $coins = [1, 5, 10, 20, 50, 100]$: In this coin combination, given any amt, the greedy algorithm can find the optimal solution.
- **Negative example** $coins = [1, 20, 50]$: Suppose $amt = 60$, the greedy algorithm can only find the combination $50 + 1 \times 10$, totaling 11 coins, but dynamic programming can find the optimal solution of $20 + 20 + 20$, needing only 3 coins.
- **Negative example** $coins = [1, 49, 50]$: Suppose $amt = 98$, the greedy algorithm can only find the combination $50 + 1 \times 48$, totaling 49 coins, but dynamic programming can find the optimal solution of $49 + 49$, needing only 2 coins.

Courtesy : Hello algo

Typical Problem - Coin Change

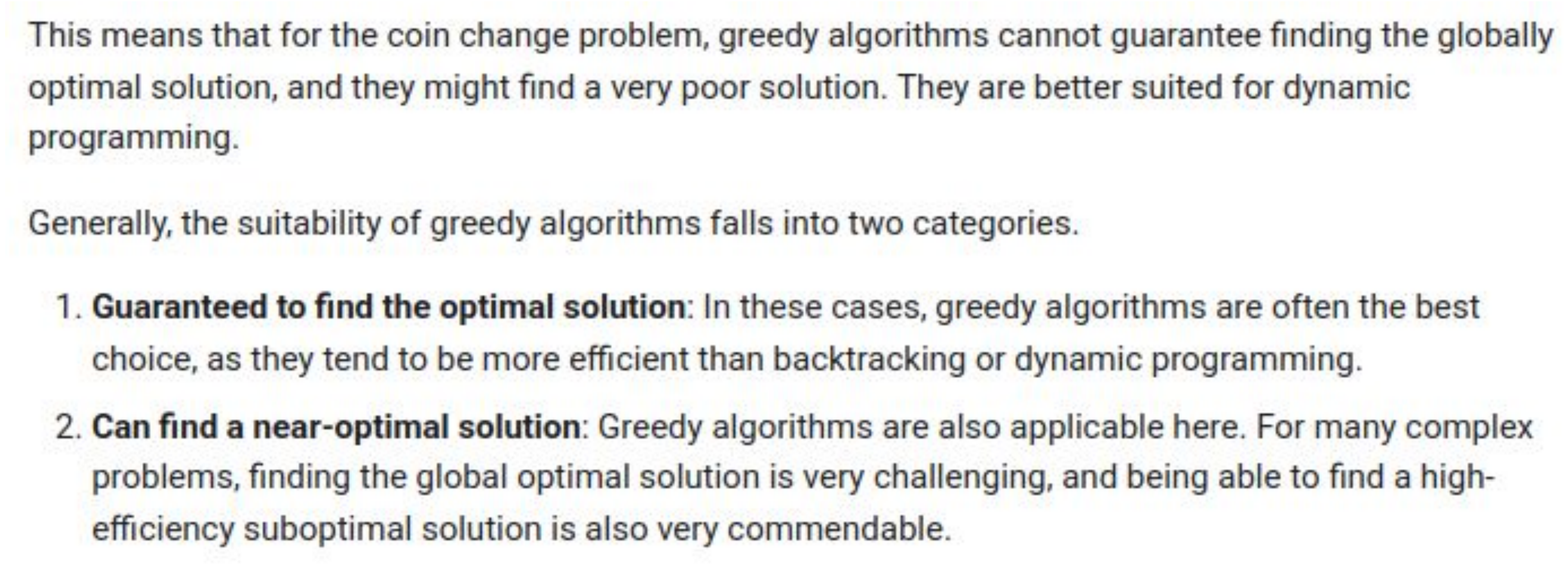


Coin combination	Target amount	Optimal greedy solution (Local optimum)	Optimal DP solution (Global optimum)
1 20 50	60	50 + 1 × 10	20 × 3
1 49 50	98	50 + 1 × 48	49 × 2

Examples where greedy algorithms do not find the optimal solution

Courtesy : Hello algo

Typical Problem - Coin Change



This means that for the coin change problem, greedy algorithms cannot guarantee finding the globally optimal solution, and they might find a very poor solution. They are better suited for dynamic programming.

Generally, the suitability of greedy algorithms falls into two categories.

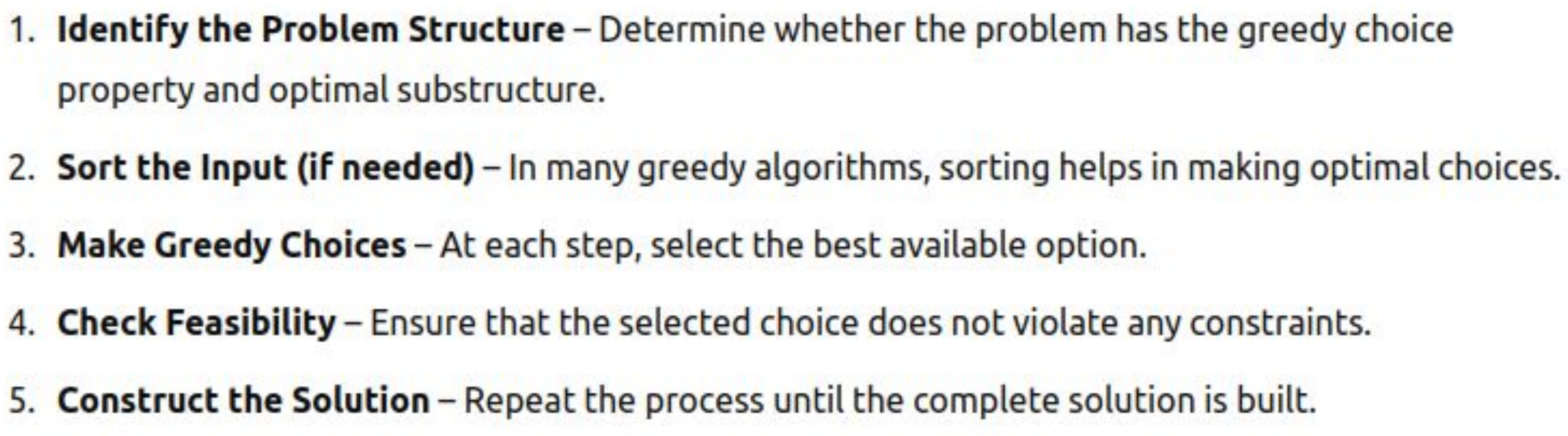
1. **Guaranteed to find the optimal solution**: In these cases, greedy algorithms are often the best choice, as they tend to be more efficient than backtracking or dynamic programming.
2. **Can find a near-optimal solution**: Greedy algorithms are also applicable here. For many complex problems, finding the global optimal solution is very challenging, and being able to find a high-efficiency suboptimal solution is also very commendable.

Courtesy : Hello algo

Greedy Method

- Algorithmic paradigm which makes locally optimal choices at each stage with the hope that these choices will lead to a globally optimal solution.
- **Key Characteristics of the Greedy Method**
  1. **Greedy Choice Property**
     - A globally optimal solution can be reached by making locally optimal choices.
  2. **Optimal Substructure**
     - A problem has an optimal substructure if an optimal solution to the problem contains optimal solutions to its subproblems.

Steps in Greedy Method



1. **Identify the Problem Structure** – Determine whether the problem has the greedy choice property and optimal substructure.
2. **Sort the Input (if needed)** – In many greedy algorithms, sorting helps in making optimal choices.
3. **Make Greedy Choices** – At each step, select the best available option.
4. **Check Feasibility** – Ensure that the selected choice does not violate any constraints.
5. **Construct the Solution** – Repeat the process until the complete solution is built.

Examples for Greedy Method

1. **Activity Selection Problem** – Select the maximum number of activities that don’t overlap.
2. **Huffman Coding** – Builds an optimal prefix-free binary tree for data compression.
3. **Prim’s Algorithm** – Finds the Minimum Spanning Tree (MST) for a graph.
4. **Kruskal’s Algorithm** – Another MST algorithm that sorts edges and picks the smallest ones while avoiding cycles.
5. **Dijkstra’s Algorithm** – Finds the shortest path from a source node to all other nodes in a weighted graph.
6. **Fractional Knapsack Problem** – Maximizes profit by selecting items with the highest value-to-weight ratio.

Pros & Cons of Greedy Method

Advantages of the Greedy Method

- Efficient: Often faster than dynamic programming or brute-force methods.
- Simple to Implement: Usually straightforward and easy to code.
- Optimized for Specific Problems: Works well when the problem exhibits greedy choice property and optimal substructure.

Limitations of the Greedy Method

- Not Always Optimal: Greedy algorithms may fail for problems lacking the greedy choice property.
- Local Optimum vs. Global Optimum: Greedy methods may get stuck in local optima.
- No Backtracking: Once a choice is made, it is not reconsidered, which can lead to incorrect results in some cases.

Topics to be covered

- General method
- **Fractional Knapsack problem,**
- Job sequencing with deadlines,
- Single source shortest path: Dijkstra Algorithm
- Minimum cost spanning trees:
    - Kruskal Algorithm
    - Prim’s algorithms

Extra Topics:

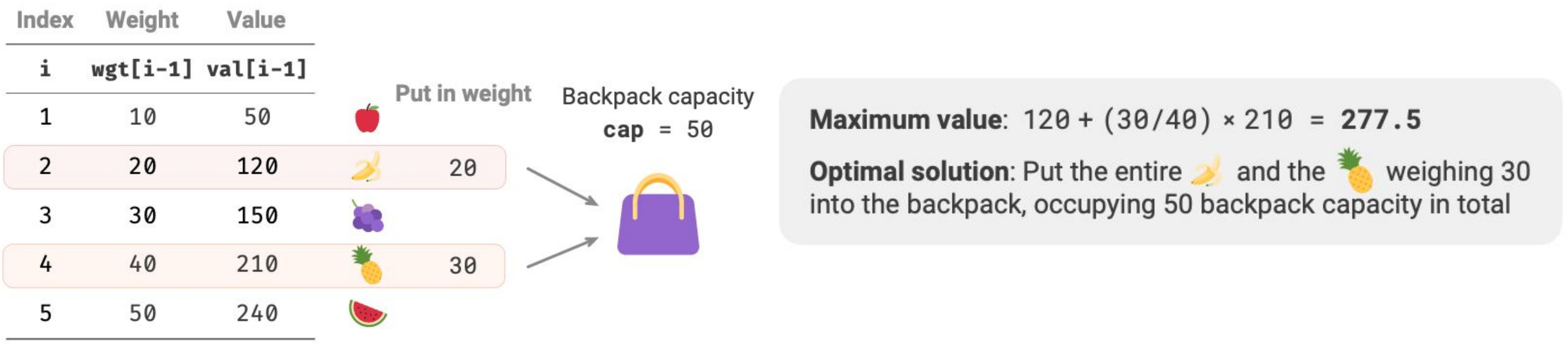
- Coin Change Problem
- Disjoint Set Data Structure

Fractional Knapsack Problem

Given:

- `n` items, each with a weight `w[i]` and value `v[i]`
- A knapsack with maximum weight capacity `W`

We need to **maximize the total value** while ensuring that the total weight does not exceed `W`. If necessary, we can take fractional parts of items.



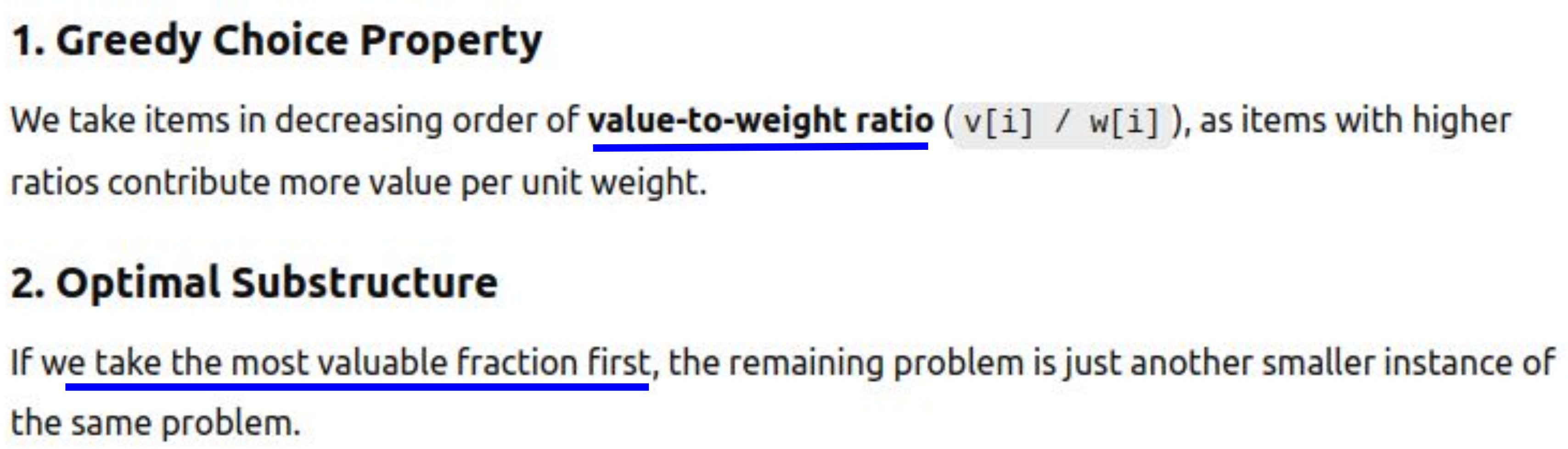
Index	Weight	Value		Put in weight
i	wgt[i-1]	val[i-1]		
1	10	50	🍎	
2	20	120	🍌	20
3	30	150	🍇	
4	40	210	🍍	30
5	50	240	🍉	

Maximum value: 120 + (30/40) × 210 = **277.5**

Optimal solution: Put the entire 🍌 and the 🍍 weighing 30 into the backpack, occupying 50 backpack capacity in total

Courtesy : Hello algo

Fractional Knapsack Problem - Greedy Method



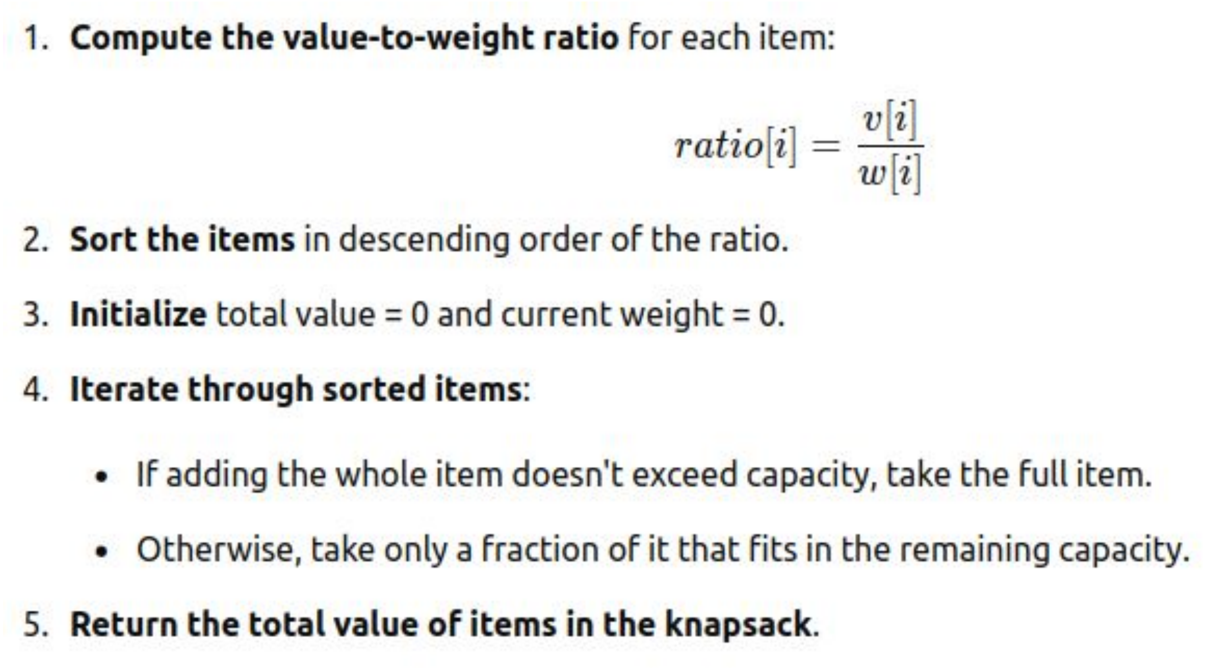
1. Greedy Choice Property

We take items in decreasing order of **value-to-weight ratio** (`v[i] / w[i]`), as items with higher ratios contribute more value per unit weight.

2. Optimal Substructure

If we take the most valuable fraction first, the remaining problem is just another smaller instance of the same problem.

Fractional Knapsack Problem - Greedy Method



1. **Compute the value-to-weight ratio** for each item:

$$ratio[i] = \frac{v[i]}{w[i]}$$

2. **Sort the items** in descending order of the ratio.
3. **Initialize** total value = 0 and current weight = 0.
4. **Iterate through sorted items**:
    - If adding the whole item doesn't exceed capacity, take the full item.
    - Otherwise, take only a fraction of it that fits in the remaining capacity.
5. **Return the total value of items in the knapsack**.

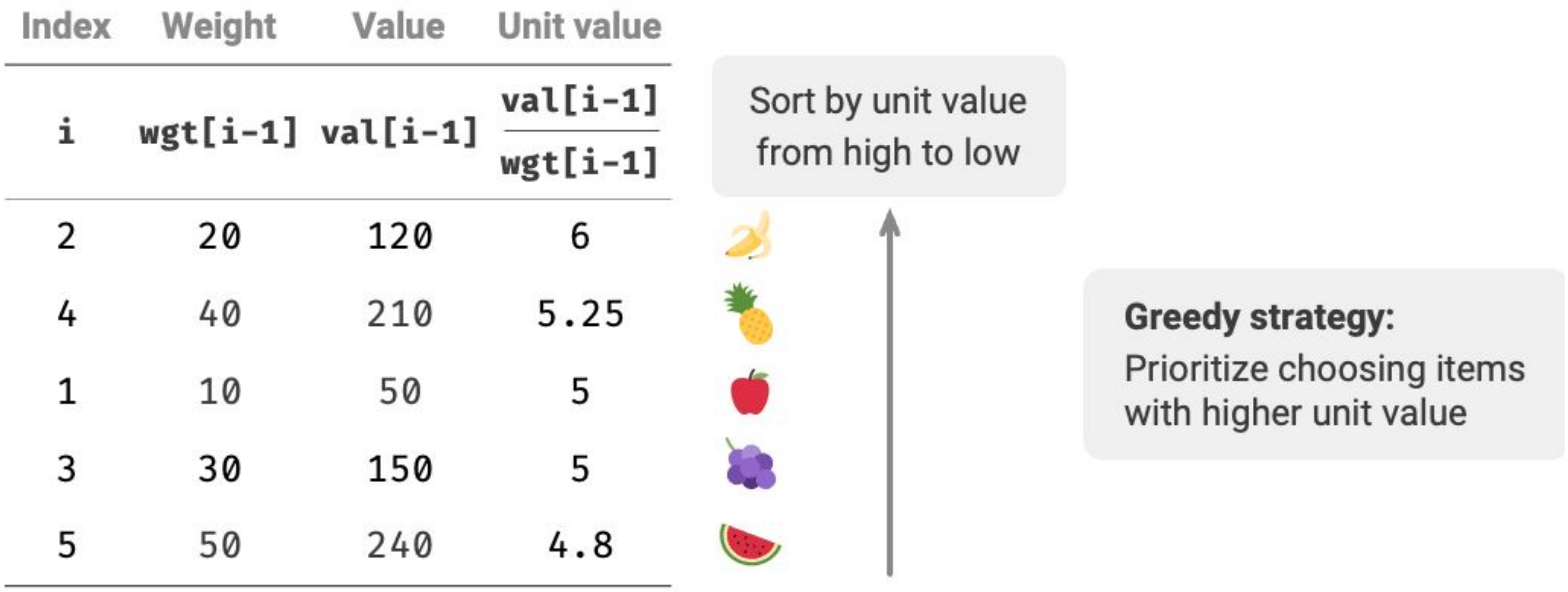
Fractional Knapsack Problem

Index	Weight	Value	Unit value	
i	wgt[i-1]	val[i-1]	$\frac{val[i-1]}{wgt[i-1]}$	
1	10	50	5	🍎
2	20	120	6	🍌
3	30	150	5	🍇
4	40	210	5.25	🍍
5	50	240	4.8	🍉

Weight of the item		Unit value of the item		Increased value
w	×	$\frac{val[i-1]}{wgt[i-1]}$	=	👜

Courtesy : Hello algo

Fractional Knapsack Problem



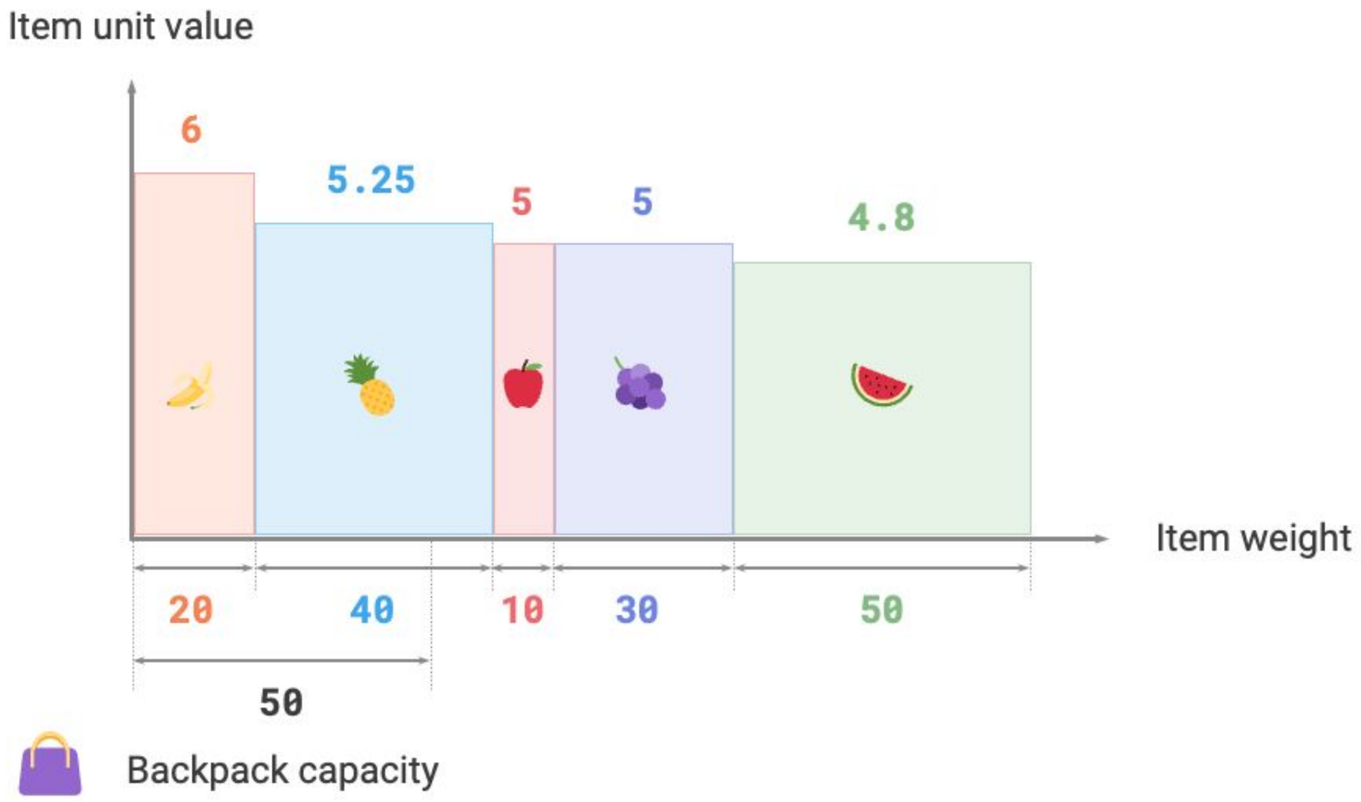
Index	Weight	Value	Unit value
i	wgt[i-1]	val[i-1]	$\frac{\text{val[i-1]}}{\text{wgt[i-1]}}$
2	20	120	6
4	40	210	5.25
1	10	50	5
3	30	150	5
5	50	240	4.8

Sort by unit value from high to low

Greedy strategy:
Prioritize choosing items with higher unit value

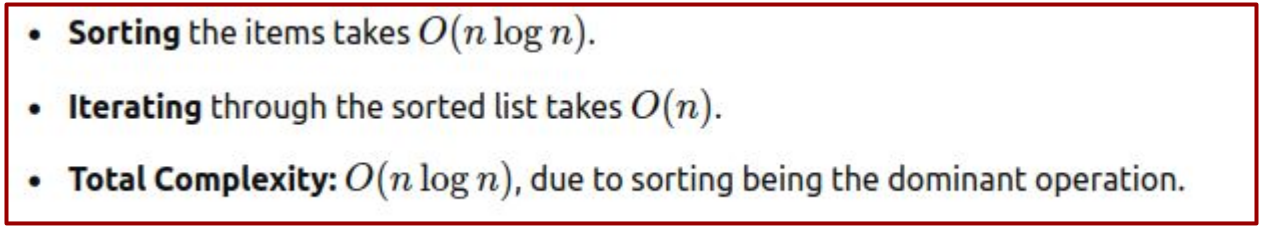
Courtesy : Hello algo

Fractional Knapsack Problem



Courtesy : Hello algo

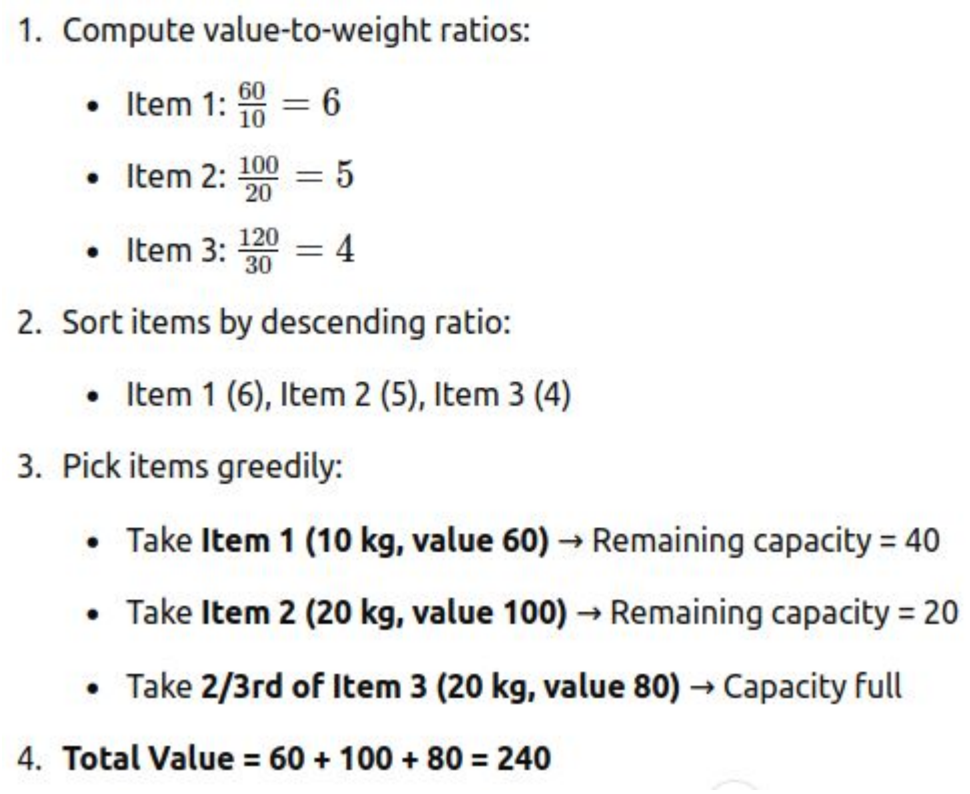
Fractional Knapsack Problem - Time Complexity Analysis



- **Sorting** the items takes $O(n \log n)$.
- **Iterating** through the sorted list takes $O(n)$.
- **Total Complexity:** $O(n \log n)$, due to sorting being the dominant operation.

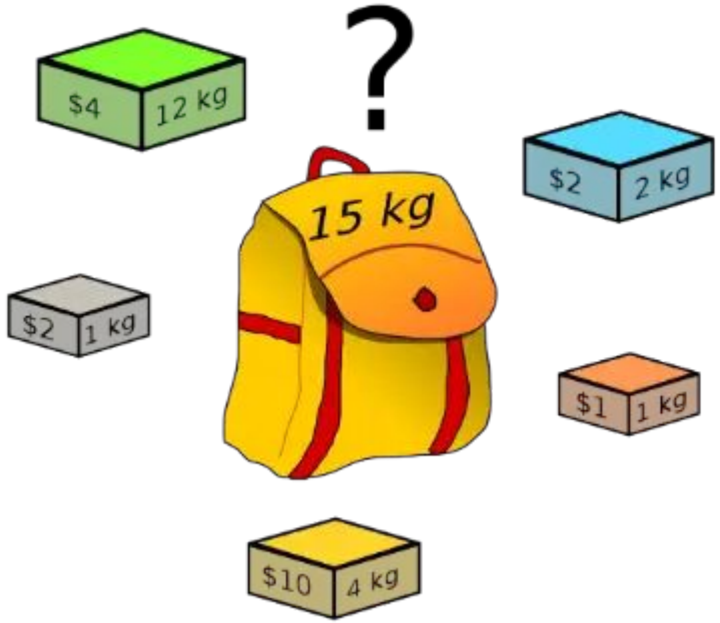
Fractional Knapsack Problem - 1

Items:	1	2	3
Weights:	10	20	30
Values:	60	100	120
Capacity:	50		



1. Compute value-to-weight ratios:
   - Item 1: $\frac{60}{10} = 6$
   - Item 2: $\frac{100}{20} = 5$
   - Item 3: $\frac{120}{30} = 4$
2. Sort items by descending ratio:
   - Item 1 (6), Item 2 (5), Item 3 (4)
3. Pick items greedily:
   - Take **Item 1 (10 kg, value 60)** → Remaining capacity = 40
   - Take **Item 2 (20 kg, value 100)** → Remaining capacity = 20
   - Take **2/3rd of Item 3 (20 kg, value 80)** → Capacity full
4. **Total Value = 60 + 100 + 80 = 240**

Fractional Knapsack Problems



Knapsack Problem

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

Topics to be covered

- General method
- Fractional Knapsack problem,
- **Job sequencing with deadlines,**
- Single source shortest path: Dijkstra Algorithm
- Minimum cost spanning trees:
  - Kruskal Algorithm
  - Prim's algorithms

Extra Topics:

- Coin Change Problem
- Disjoint Set Data Structure

Job sequencing with Deadlines

Given:

- `n` jobs, each with:
    - **Profit (p[i])**
    - **Deadline (d[i])** (max time slot to complete the job)
    - **Job ID (i)**
- Only **one job can be scheduled per time unit**.

Find:

- The sequence of jobs that maximizes the total profit while ensuring that **no job is scheduled beyond its deadline**.

Job sequencing with Deadlines

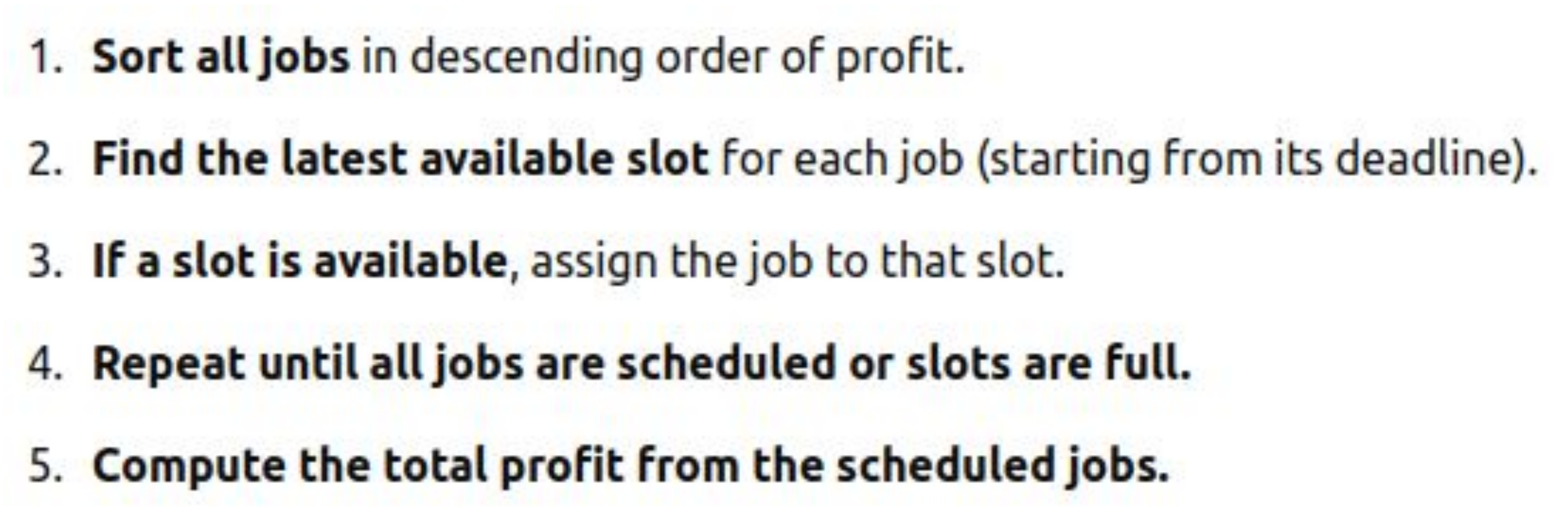
Suppose you are a content writer and have 4 articles to write. Each article takes a different amount of time to complete and has different payments. You want to earn the maximum amount by writing the articles.

So which article will you write first? You have two options:

- **Article with maximum payment** - In this case, the article can take a longer duration to complete but by writing this article, you may miss some other articles which can be completed in a shorter interval of time, and it can be unprofitable to you if the sum of the payments from the missed articles is greater than the payment of this single article which you are doing.

- **The article which takes less time**: In this case, it may happen that you have written multiple articles in the given time but there could be the possibility that if you would have chosen the article with maximum payment, you could have earned more, even by writing a single article.

Job sequencing with Deadlines



1. **Sort all jobs** in descending order of profit.
2. **Find the latest available slot** for each job (starting from its deadline).
3. **If a slot is available**, assign the job to that slot.
4. **Repeat until all jobs are scheduled or slots are full.**
5. **Compute the total profit from the scheduled jobs.**

Greedy Choice Property : Choosing the highest profit job first

Optimal Substructure : Solving smaller scheduling problems optimally

Job sequencing with Deadlines - Complexity Analysis

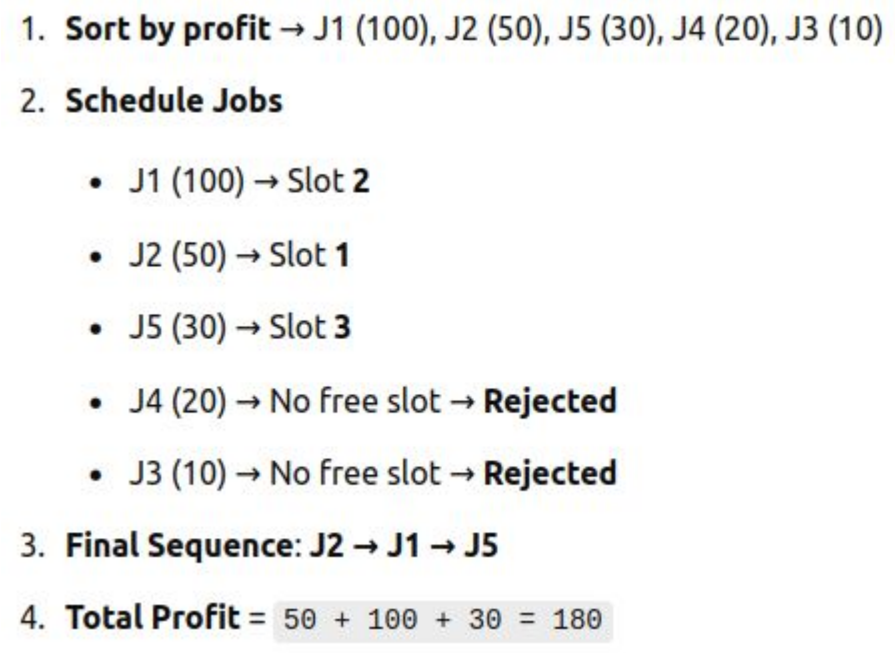
Time Complexity Analysis :

1. **Sorting the jobs** – $O(n \log n)$
2. **Placing jobs in slots** – $O(n^2)$ (in worst case)

Space Complexity : **O(n)**

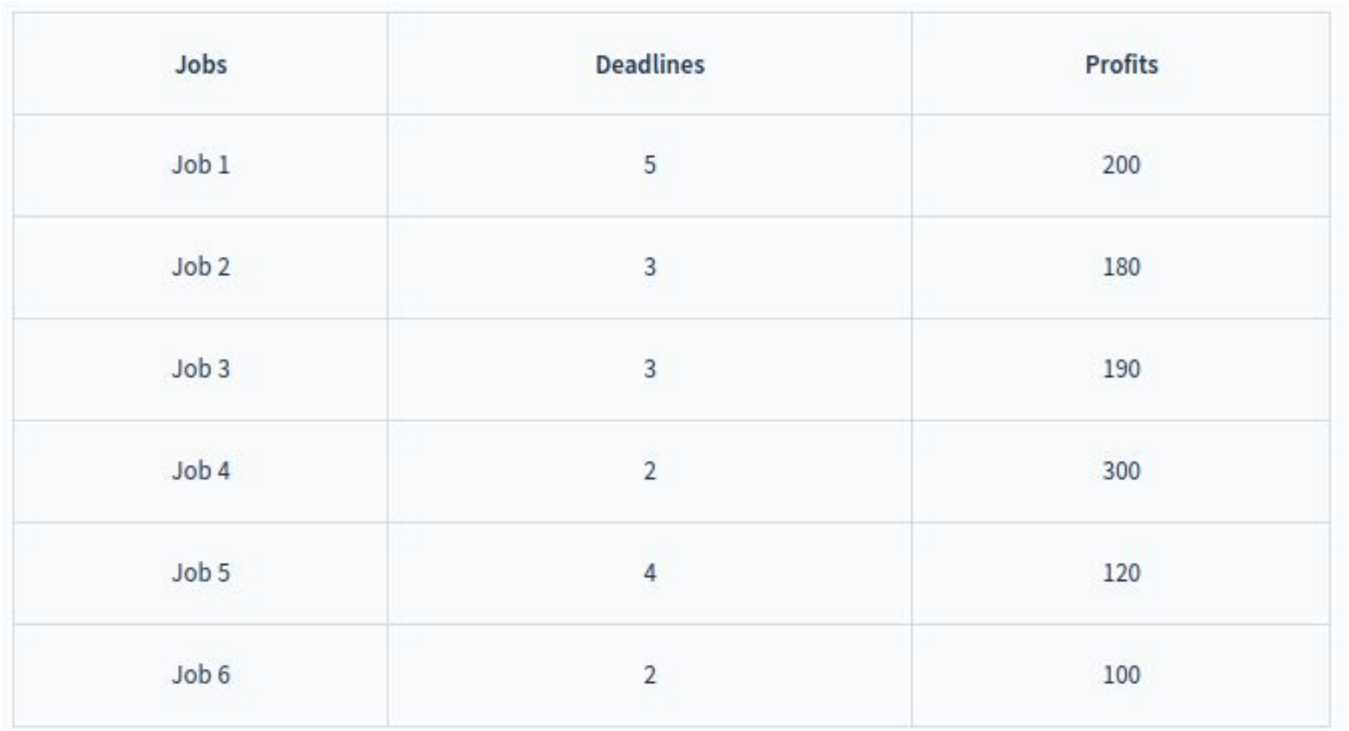
Job sequencing with Deadlines - Example 1

Jobs:	J1	J2	J3	J4	J5
Profits:	100	50	10	20	30
Deadlines:	2	1	2	1	3



1. **Sort by profit** → J1 (100), J2 (50), J5 (30), J4 (20), J3 (10)
2. **Schedule Jobs**
   - J1 (100) → Slot **2**
   - J2 (50) → Slot **1**
   - J5 (30) → Slot **3**
   - J4 (20) → No free slot → **Rejected**
   - J3 (10) → No free slot → **Rejected**
3. **Final Sequence**: **J2 → J1 → J5**
4. **Total Profit** = `50 + 100 + 30 = 180`

Job sequencing with Deadlines - Example 2

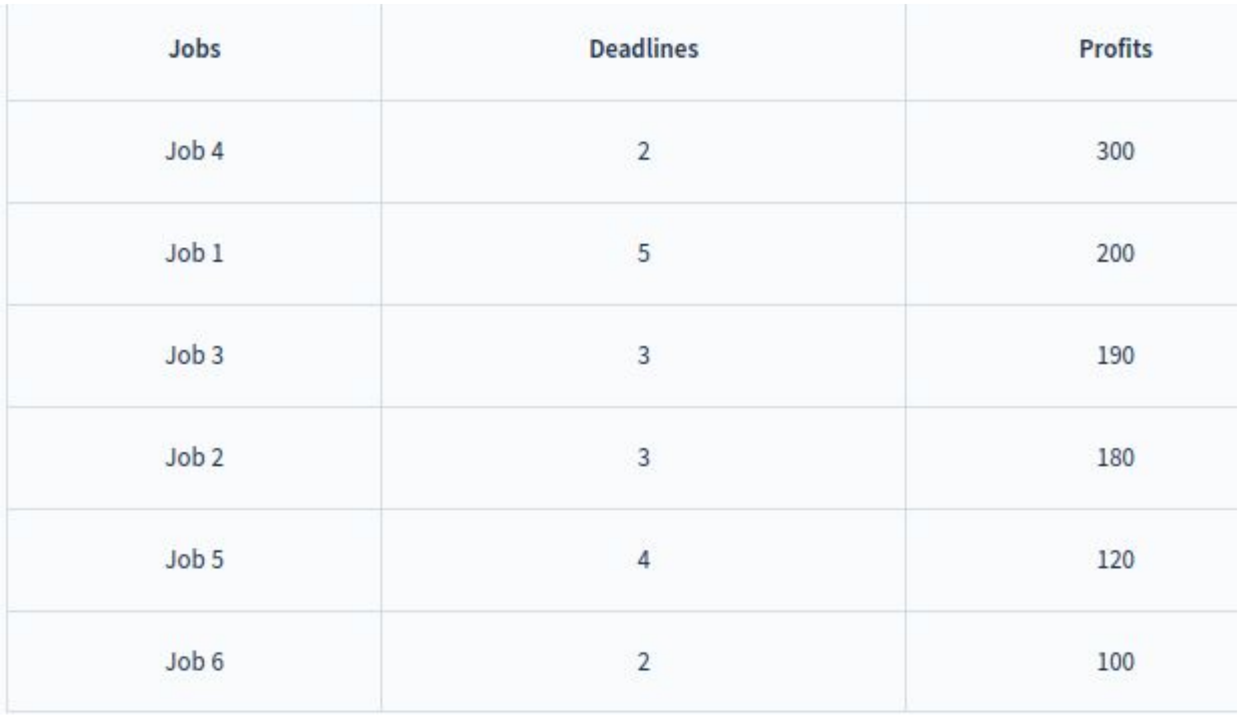


Jobs	Deadlines	Profits
Job 1	5	200
Job 2	3	180
Job 3	3	190
Job 4	2	300
Job 5	4	120
Job 6	2	100

Courtesy : Scaler.com

Job sequencing with Deadlines - Example 2

Step 1: Sort the jobs in decreasing order of their profit



Jobs	Deadlines	Profits
Job 4	2	300
Job 1	5	200
Job 3	3	190
Job 2	3	180
Job 5	4	120
Job 6	2	100

Courtesy : Scaler.com

Job sequencing with Deadlines - Example 2

Step 2: Prepare the Gantt Chart with maximum deadline as 5.

Step 3 : Pick the jobs one by one as presented in step, 1, and place them on the Gantt chart as far as possible from 0.

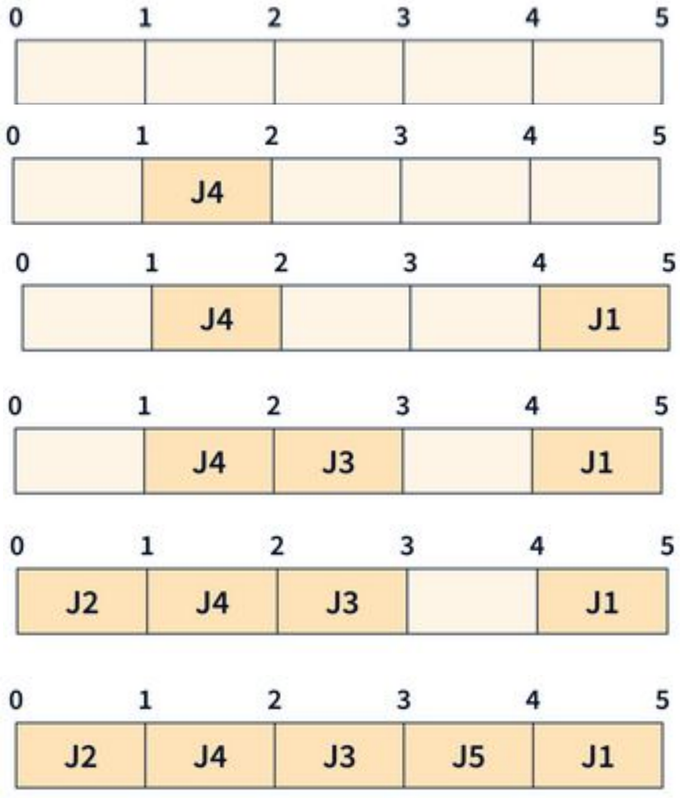
- Pick Job 4. Its deadline is 2. Place the job in the empty slot available just before the deadline.
- pick Job 1. Its deadline is 5. Place the job in the empty slot available just before the deadline.
- pick Job 3. Its deadline is 3. Place the job in the empty slot available just before the deadline.
- pick Job 2. Its deadline is 3. Second and third slots are already filled. So place job 2 on the next available free slot farthest from 0, i.e first slot.
- pick Job 5. Its deadline is 4. Place the job in the first empty slot before the deadline,i.e fourth slot.
- pick Job 6. Its deadline is 2. Since, no such slot is available, hence Job 6 can not be completed.

Most optimal sequence of jobs : **Job 2, Job 4, Job 3, Job 5, Job 1.**

Maximum profit earned = **Profit (Job 2 + Job 4 + Job 3 + Job 5 + Job 1)**

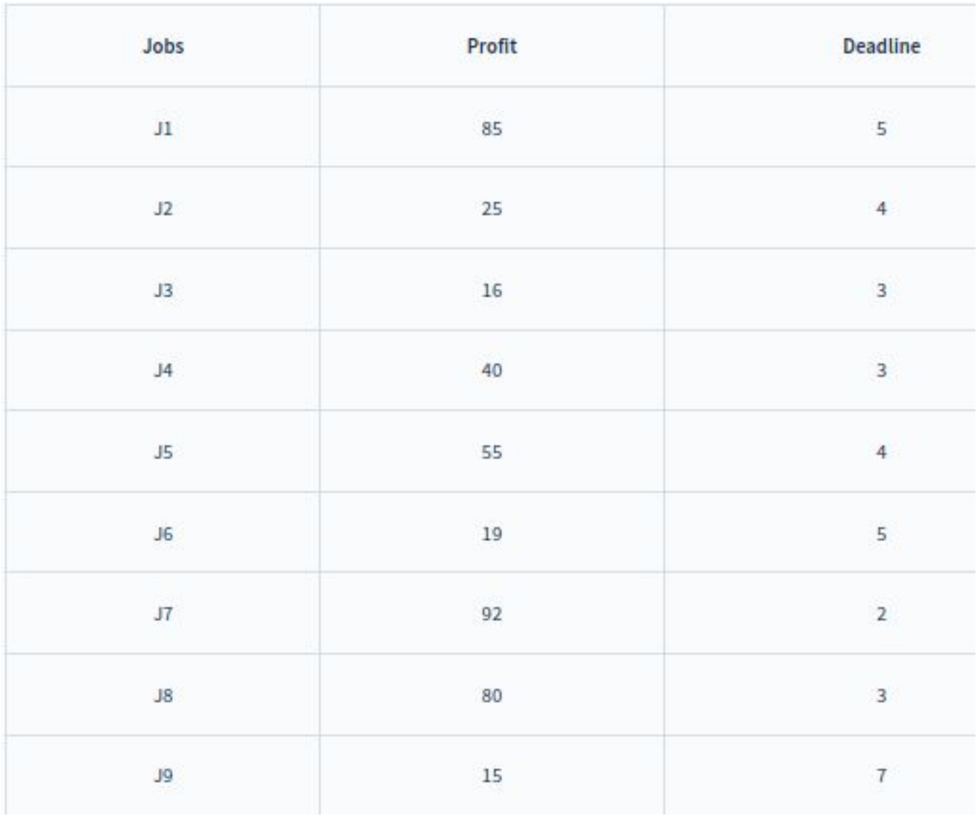
= 180+300+190+120+200

= **990**



Courtesy : Scaler.com

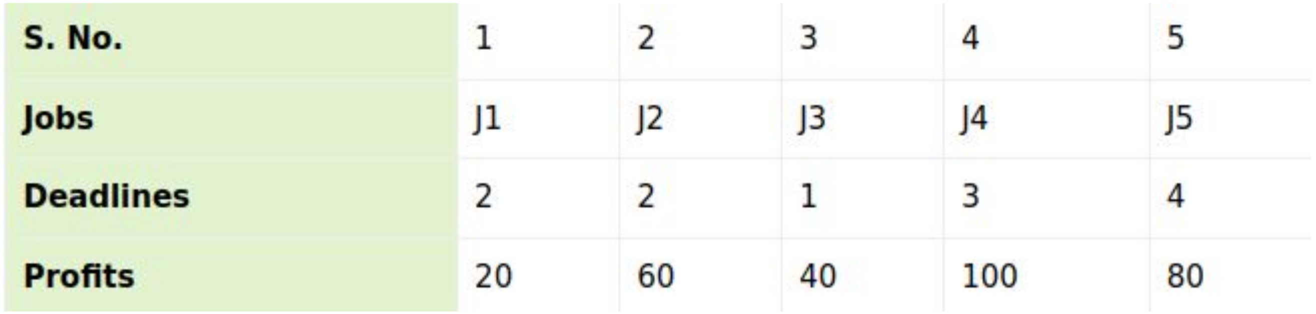
Job sequencing with Deadlines - Practice Problem 1



Jobs	Profit	Deadline
J1	85	5
J2	25	4
J3	16	3
J4	40	3
J5	55	4
J6	19	5
J7	92	2
J8	80	3
J9	15	7

Courtesy : Scaler.com

Job sequencing with Deadlines - Practice Problem 2

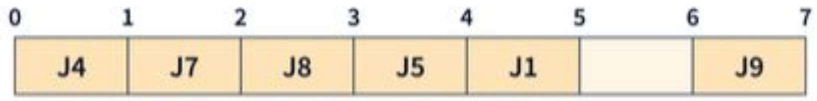


S. No.	1	2	3	4	5
Jobs	J1	J2	J3	J4	J5
Deadlines	2	2	1	3	4
Profits	20	60	40	100	80

Courtesy : Scaler.com

Job sequencing with Deadlines - Practice Problem 1 (Soln)

Jobs	Profit	Deadline
J1	85	5
J2	25	4
J3	16	3
J4	40	3
J5	55	4
J6	19	5
J7	92	2
J8	80	3
J9	15	7



Optimal sequence : **J4, J7, J8, J5, J1, J9.**

Maximum profit earned = 40+92+80+55+85+15

= 367

Courtesy : Scaler.com

Topics to be covered

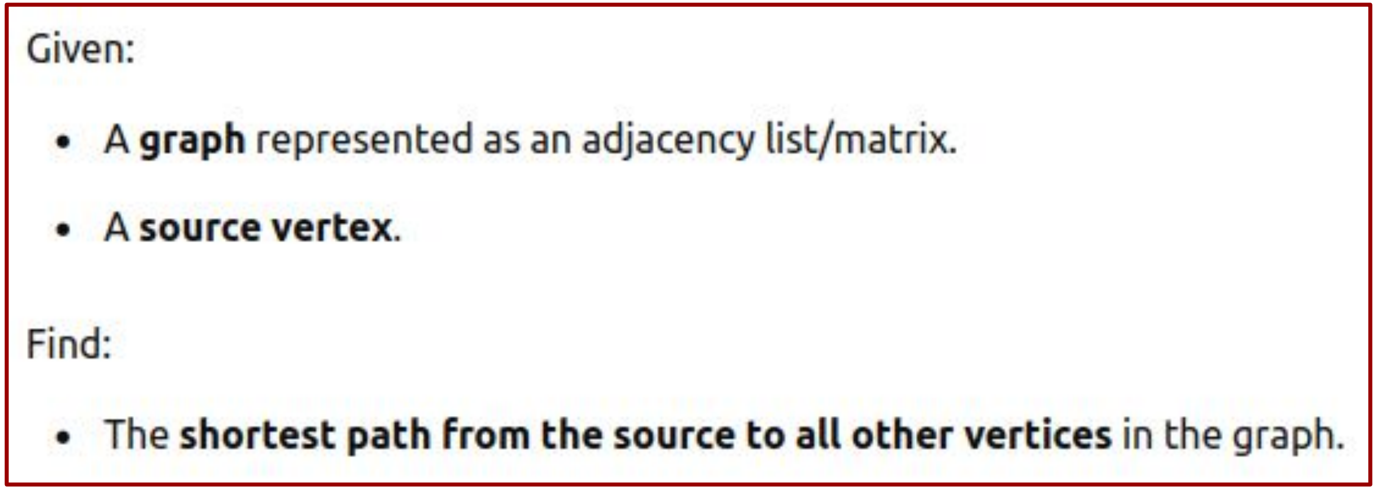
- General method
- Fractional Knapsack problem,
- Job sequencing with deadlines,
- **Single source shortest path: Dijkstra Algorithm**
- Minimum cost spanning trees:
    - Kruskal Algorithm
    - Prim’s algorithm

Extra Topics:

- Coin Change Problem
- Disjoint Set Data Structure

Single source shortest path: Dijkstra Algorithm

- To find the **single-source shortest path** in a graph with **non-negative edge weights**.
- widely used in
    - **Google Maps & GPS Navigation** – Finding shortest travel routes.
    - **Network Routing Protocols** – Used in OSPF (Open Shortest Path First).
    - *AI Pathfinding (A)** – Used in games for optimal movement.



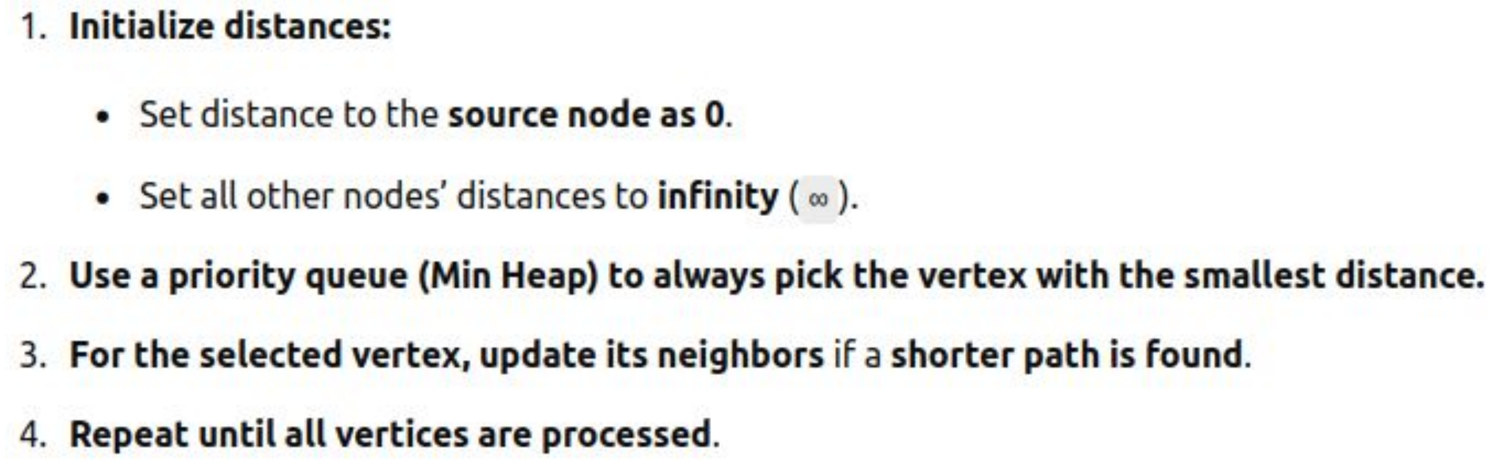
Given:

- A **graph** represented as an adjacency list/matrix.
- A **source vertex**.

Find:

- The **shortest path from the source to all other vertices** in the graph.

Single source shortest path: Dijkstra Algorithm

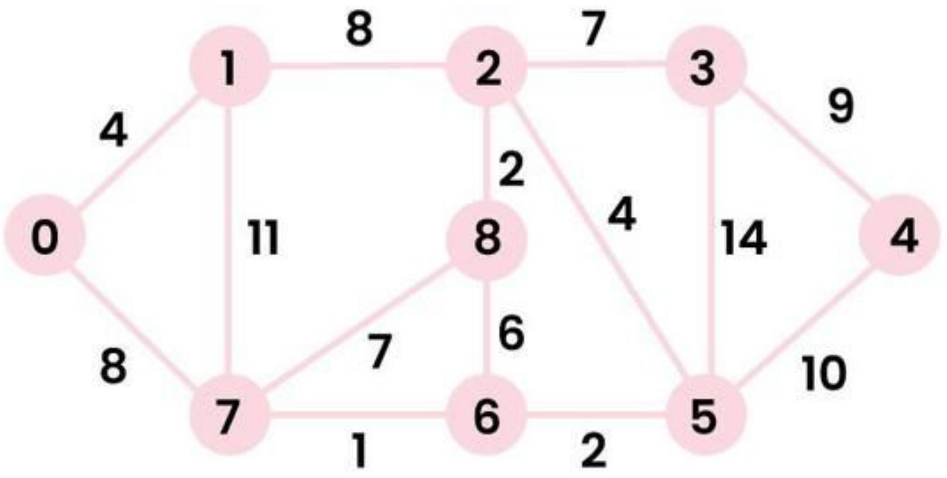


1. **Initialize distances:**
   - Set distance to the **source node as 0**.
   - Set all other nodes' distances to **infinity** (∞).
2. **Use a priority queue (Min Heap) to always pick the vertex with the smallest distance.**
3. **For the selected vertex, update its neighbors** if a **shorter path is found**.
4. **Repeat until all vertices are processed.**

1. **Greedy Choice Property :** At each step, select the **node with the smallest known distance** and updates neighbors.

2. **Optimal Substructure :** The shortest path to a node is **built from the shortest paths to its predecessor nodes.**

Single source shortest path: Dijkstra Algorithm



Target	Min. distance	Path
1	4	0 → 1
2	12	0 → 1→ 2
3	19	0 → 1→ 2 → 3
4	21	0 → 7 → 6 → 5 → 4
5	11	0 → 7 → 6 → 5
6	9	0 → 7 → 6
7	8	0 → 7
8	14	0 → 1 → 2 → 8

Single source shortest path: Dijkstra Algorithm

Dijkstra's Algorithm using Adjacency Matrix :

The idea is to generate a ***SPT (shortest path tree)*** with a given source as a root. Maintain an Adjacency Matrix with two sets,

- one set contains vertices included in the shortest-path tree,
- other set includes vertices not yet included in the shortest-path tree.

At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.

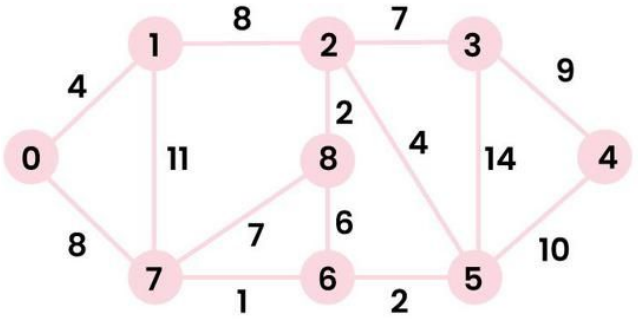
Single source shortest path: Dijkstra Algorithm (Adjacency Matrix)

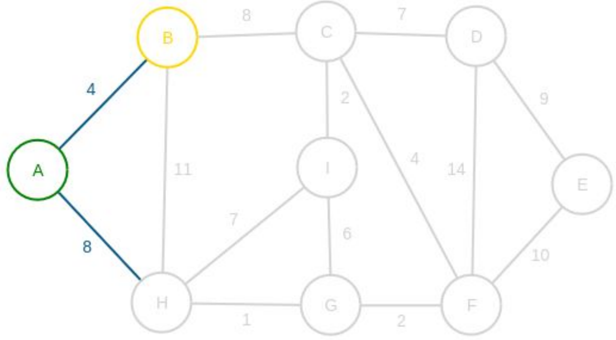
1. Create a set **sptSet** (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as **INFINITE** . Assign the distance value as 0 for the source vertex so that it is picked first.
3. While **sptSet** doesn’t include all vertices
    - Pick a vertex **u** that is not there in **sptSet** and has a minimum distance value.
    - Include u to **sptSet** .
    - Then update the distance value of all adjacent vertices of **u** .
        - To update the distance values, iterate through all adjacent vertices.
        - For every adjacent vertex **v,** if the sum of the distance value of **u** (from source) and weight of edge **u-v** , is less than the distance value of **v** , then update the distance value of **v** .

Note: We use a boolean array **sptSet[]** to represent the set of vertices included in **SPT** . If a value **sptSet[v]** is true, then vertex v is included in **SPT** , otherwise not. Array **dist[]** is used to store the shortest distance values of all vertices.

Single source shortest path: Dijkstra Algorithm (Adjacency Matrix)

Vertex	sptSet	Adjacent vertices	dist
--	{}	--	[0, INF, INF, INF, INF, INF, INF, INF, INF]
0	{0}	1, 7	[0, **4**, INF, INF, INF, INF, INF, **8**, INF]

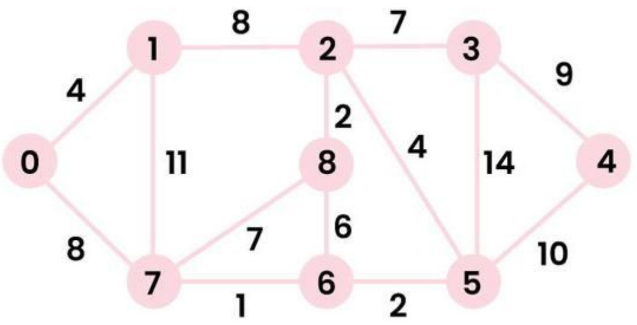


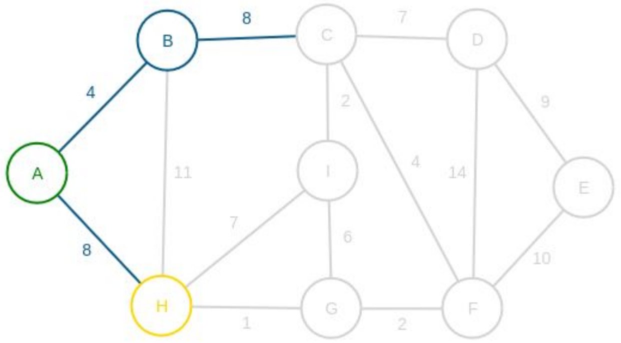


Courtesy : Dijkstra Algo. Visualization

Single source shortest path: Dijkstra Algorithm (Adjacency Matrix)

Vertex	sptSet	Adjacent vertices	dist
--	*{}*	--	[0, INF, INF, INF, INF, INF, INF, INF, INF]
0	{0}	1, 7	[0, **4**, INF, INF, INF, INF, INF, **8**, INF]
1	{0,1}	0, 2, 7	[0, **4**, **12**, INF, INF, INF, INF, **8**, INF]

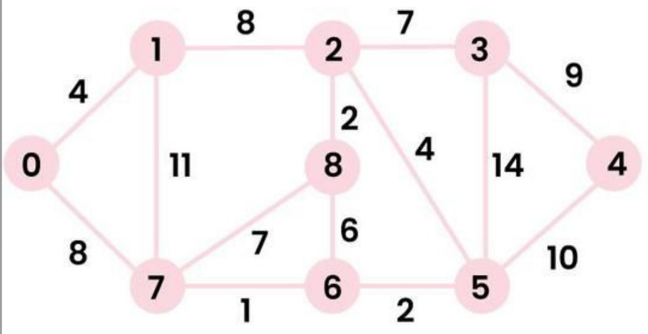


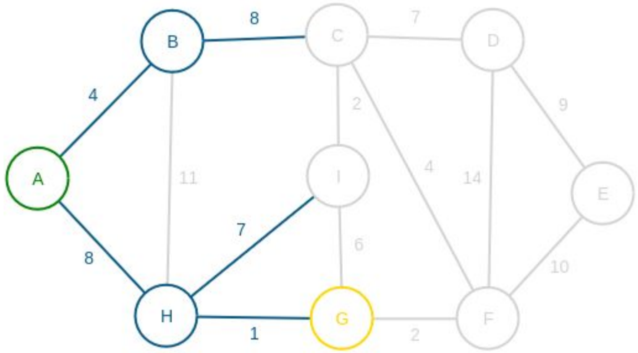


Courtesy : Dijkstra Algo. Visualization

Single source shortest path: Dijkstra Algorithm (Adjacency Matrix)

Vertex	sptSet	Adjacent vertices	dist
--	*{}*	--	[0, INF, INF, INF, INF, INF, INF, INF, INF]
0	{0}	1, 7	[0, **4**, INF, INF, INF, INF, INF, **8**, INF]
1	{0,1}	0, 2, 7	[0, **4**, **12**, INF, INF, INF, INF, **8**, INF]
7	{0,1,7}	0, 1, 6, 8	[0, **4**, **12**, INF, INF, INF, **9**, **8**, **15**]

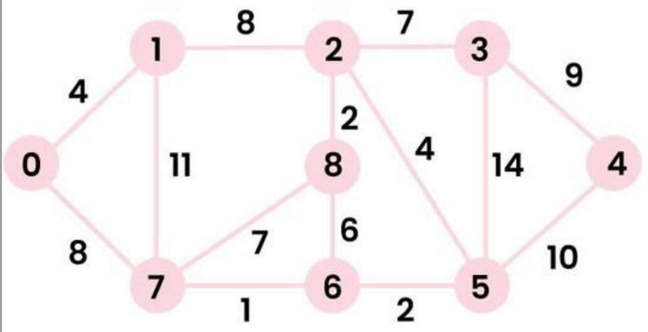


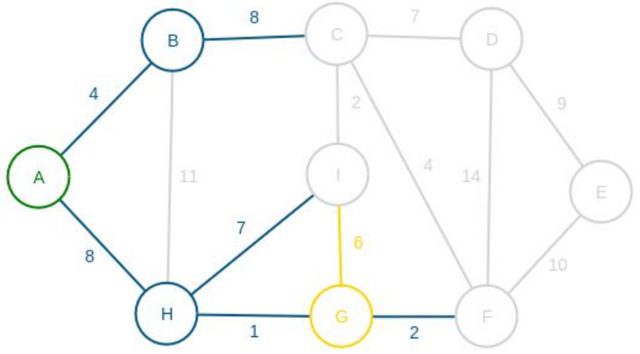


Courtesy : Dijkstra Algo. Visualization

Single source shortest path: Dijkstra Algorithm (Adjacency Matrix)

Vertex	sptSet	Adjacent vertices	dist
--	{}	--	[0, INF, INF, INF, INF, INF, INF, INF, INF]
0	{0}	1, 7	[0, **4**, INF, INF, INF, INF, INF, **8**, INF]
1	{0,1}	0, 2, 7	[0, **4**, **12**, INF, INF, INF, INF, **8**, INF]
7	{0,1,7}	0, 1, 6, 8	[0, **4**, **12**, INF, INF, INF, **9**, **8**, **15**]
6	{0,1,6,7}	5, 7, 8	[0, **4**, **12**, INF, INF, **11**, **9**, **8**, **15**]

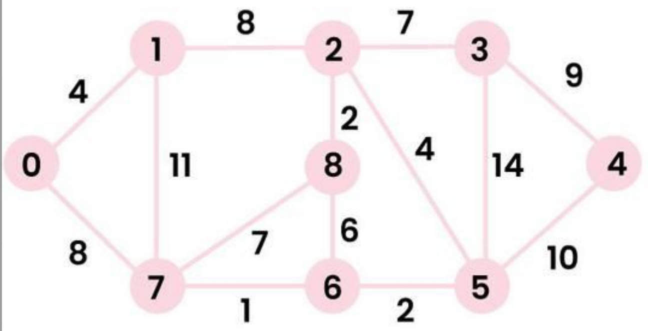


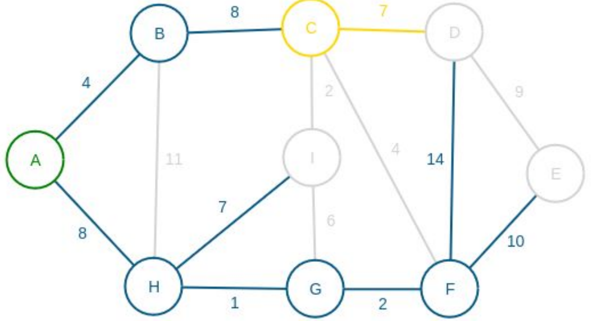


Courtesy : Dijkstra Algo. Visualization

Single source shortest path: Dijkstra Algorithm (Adjacency Matrix)

Vertex	sptSet	Adjacent vertices	dist
--	{}	--	[0, INF, INF, INF, INF, INF, INF, INF, INF]
0	{0}	1, 7	[0, **4**, INF, INF, INF, INF, INF, **8**, INF]
1	{0,1}	0, 2, 7	[0, **4**, **12**, INF, INF, INF, INF, **8**, INF]
7	{0,1,7}	0, 1, 6, 8	[0, **4**, **12**, INF, INF, INF, **9**, **8**, **15**]
6	{0,1,6,7}	5, 7, 8	[0, **4**, **12**, INF, INF, **11**, **9**, **8**, **15**]
5	{0,1,5,6,7}	2, 3, 4, 6	[0, **4**, **12**, **25 21**, **11**, **9**, **8**, **15**]

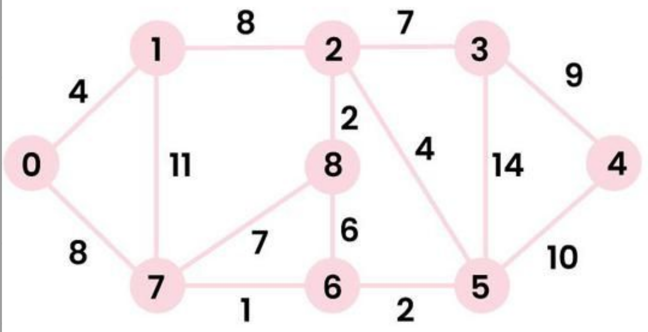


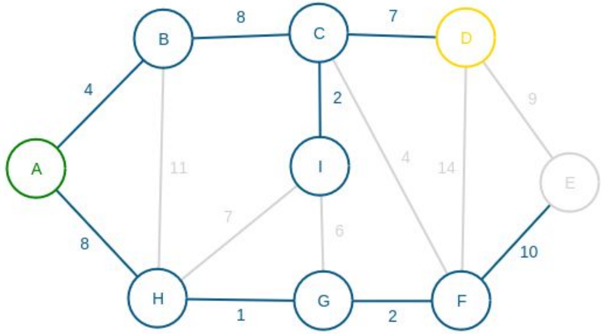


Courtesy : Dijkstra Algo. Visualization

Single source shortest path: Dijkstra Algorithm (Adjacency Matrix)

Vertex	sptSet	Adjacent vertices	dist
--	{}	--	[0, INF, INF, INF, INF, INF, INF, INF, INF]
0	{0}	1, 7	[0, **4**, INF, INF, INF, INF, INF, **8**, INF]
1	{0,1}	0, 2, 7	[0, **4**, **12**, INF, INF, INF, INF, **8**, INF]
7	{0,1,7}	0, 1, 6, 8	[0, **4**, **12**, INF, INF, INF, **9**, **8**, **15**]
6	{0,1,6,7}	5, 7, 8	[0, **4**, **12**, INF, INF, **11**, **9**, **8**, **15**]
5	{0,1,5,6,7}	2, 3, 4, 6	[0, **4**, **12**, **25 21**, **11**, **9**, **8**, **15**]
2	{0,1,2,5,6,7}	1. 3, 5, 8	[0, **4**, **12**, **19**, **21**, **11**, **9**, **8**, **14**]

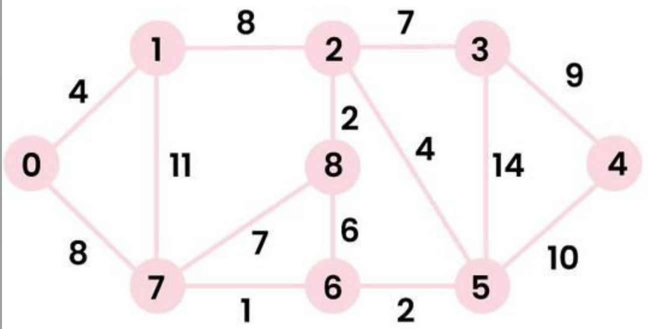


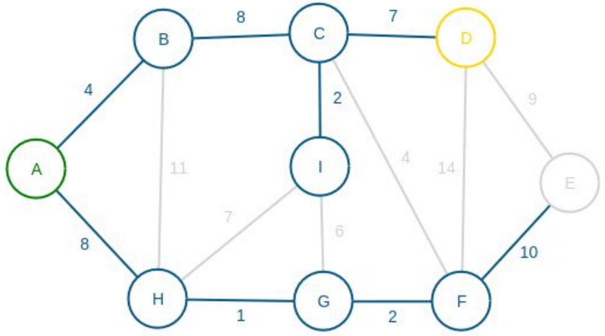


Courtesy : Dijkstra Algo. Visualization

Single source shortest path: Dijkstra Algorithm (Adjacency Matrix)

Vertex	sptSet	Adjacent vertices	dist
--	{}	--	[0, INF, INF, INF, INF, INF, INF, INF, INF]
0	{0}	1, 7	[0, **4**, INF, INF, INF, INF, INF, **8**, INF]
1	{0,1}	0, 2, 7	[0, **4**, **12**, INF, INF, INF, INF, **8**, INF]
7	{0,1,7}	0, 1, 6, 8	[0, **4**, **12**, INF, INF, INF, **9**, **8**, **15**]
6	{0,1,6,7}	5, 7, 8	[0, **4**, **12**, INF, INF, **11**, **9**, **8**, **15**]
5	{0,1,5,6,7}	2, 3, 4, 6	[0, **4**, **12**, **25 21**, **11**, **9**, **8**, **15**]
2	{0,1,2,5,6,7}	1. 3, 5, 8	[0, **4**, **12**, **19**, **21**, **11**, **9**, **8**, **14**]
8	{0,1,2,5,6,7,8}	2, 6	[0, **4**, **12**, **19**, **21**, **11**, **9**, **8**, **14**]

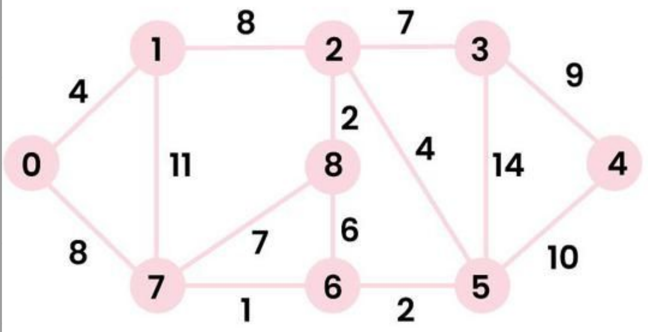


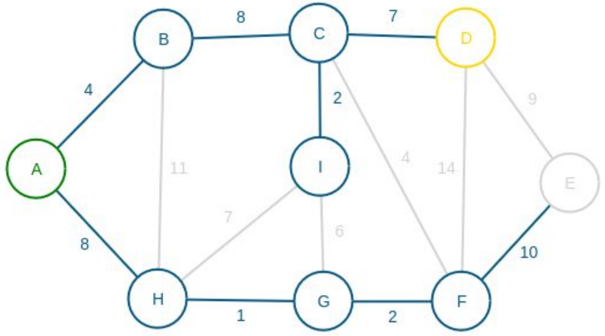


Courtesy : Dijkstra Algo. Visualization

Single source shortest path: Dijkstra Algorithm (Adjacency Matrix)

Vertex	sptSet	Adjacent vertices	dist
--	{}	--	[0, INF, INF, INF, INF, INF, INF, INF, INF]
0	{0}	1, 7	[0, **4**, INF, INF, INF, INF, INF, **8**, INF]
1	{0,1}	0, 2, 7	[0, **4**, **12**, INF, INF, INF, INF, **8**, INF]
7	{0,1,7}	0, 1, 6, 8	[0, **4**, **12**, INF, INF, INF, **9**, **8**, **15**]
6	{0,1,6,7}	5, 7, 8	[0, **4**, **12**, INF, INF, **11**, **9**, **8**, **15**]
5	{0,1,5,6,7}	2, 3, 4, 6	[0, **4**, **12**, **25 21**, **11**, **9**, **8**, **15**]
2	{0,1,2,5,6,7}	1. 3, 5, 8	[0, **4**, **12**, **19, 21**, **11**, **9**, **8**, **14**]
8	{0,1,2,5,6,7,8}	2, 6	[0, **4**, **12**, **19, 21**, **11**, **9**, **8**, **14**]
3	{0,1,2,3,5,6,7,8}	2, 4, 5	[0, **4**, **12**, **19, 21**, **11**, **9**, **8**, **14**]

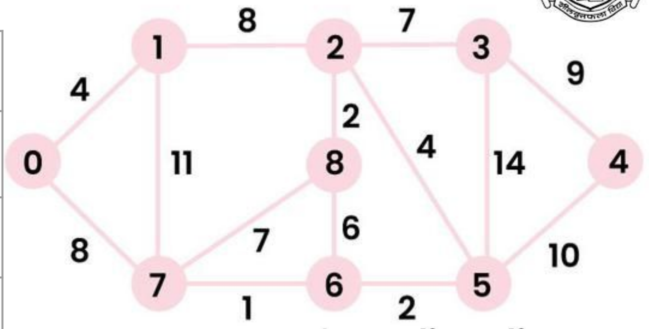


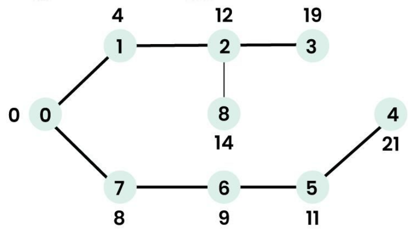


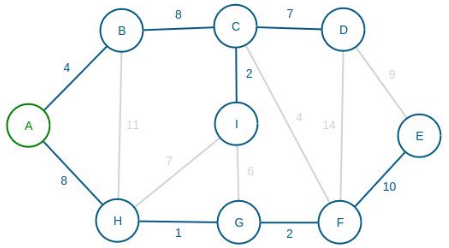
Courtesy : Dijkstra Algo. Visualization

Single source shortest path: Dijkstra Algorithm (Adjacency Matrix)

Vertex	sptSet	Adjacent vertices	dist
--	{}	--	[0, INF, INF, INF, INF, INF, INF, INF, INF]
0	{0}	1, 7	[0, 4, INF, INF, INF, INF, INF, 8, INF]
1	{0,1}	0, 2, 7	[0, 4, 12, INF, INF, INF, INF, 8, INF]
7	{0,1,7}	0, 1, 6, 8	[0, 4, 12, INF, INF, INF, 9, 8, 15]
6	{0,1,6,7}	5, 7, 8	[0, 4, 12, INF, INF, 11, 9, 8, 15]
5	{0,1,5,6,7}	2, 3, 4, 6	[0, 4, 12, 25 21, 11, 9, 8, 15]
2	{0,1,2,5,6,7}	1. 3, 5, 8	[0, 4, 12, 19, 21, 11, 9, 8, 14]
8	{0,1,2,5,6,7,8}	2, 6	[0, 4, 12, 19, 21, 11, 9, 8, 14]
3	{0,1,2,3,5,6,7,8}	2, 4, 5	[0, 4, 12, 19, 21, 11, 9, 8, 14]
4	{0,1,2,3,4,5,6,7,8}	3, 5	[0, 4, 12, 19, 21, 11, 9, 8, 14]

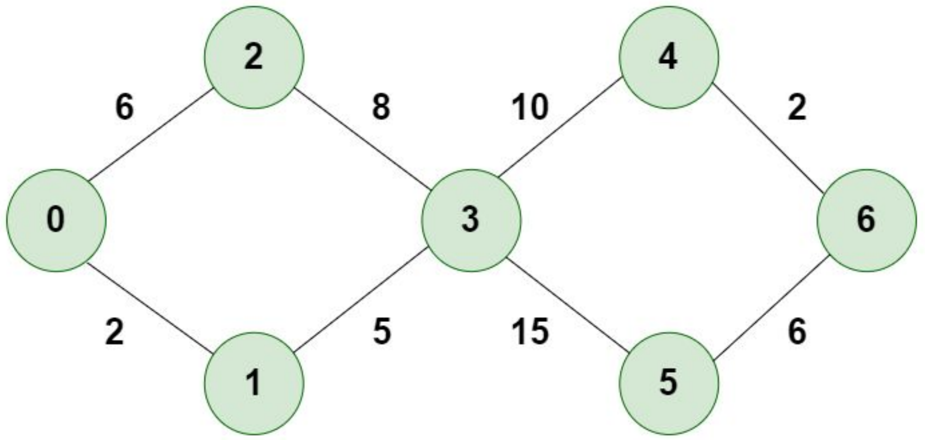


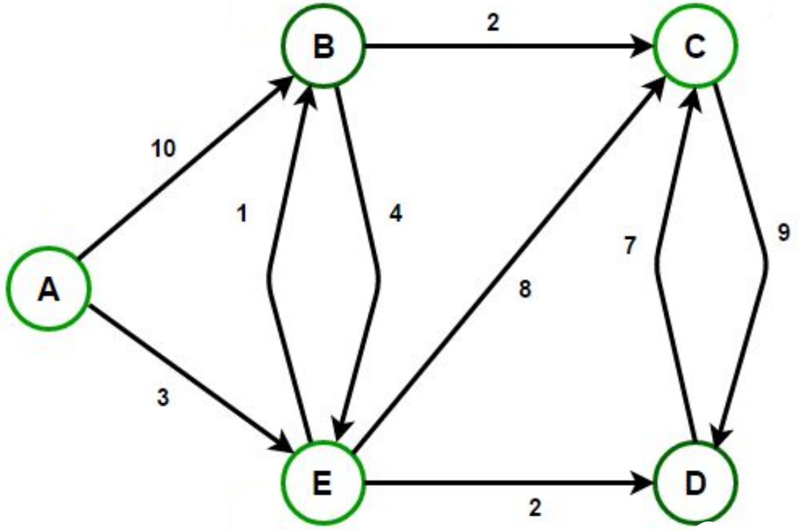




Courtesy : Dijkstra Algo. Visualization

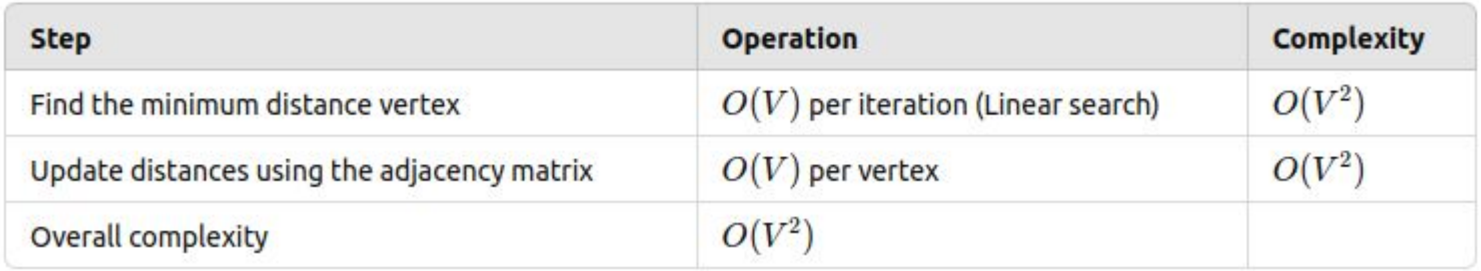
Single source shortest path: Dijkstra Algorithm Problems





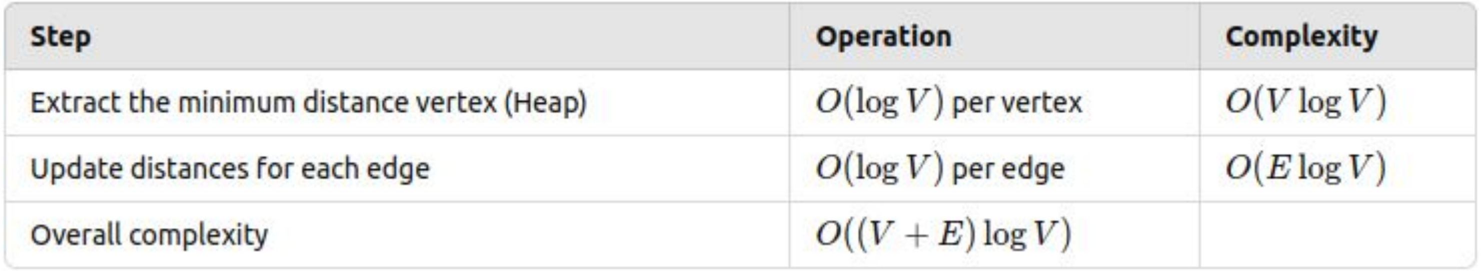
Single source shortest path: Dijkstra Algorithm Analysis

Using Adjacency Matrix :



Step	Operation	Complexity
Find the minimum distance vertex	$O(V)$ per iteration (Linear search)	$O(V^2)$
Update distances using the adjacency matrix	$O(V)$ per vertex	$O(V^2)$
Overall complexity	$O(V^2)$	

Using Adjacency List :



Step	Operation	Complexity
Extract the minimum distance vertex (Heap)	$O(\log V)$ per vertex	$O(V \log V)$
Update distances for each edge	$O(\log V)$ per edge	$O(E \log V)$
Overall complexity	$O((V + E) \log V)$	

Topics to be covered

- General method
- Fractional Knapsack problem,
- Job sequencing with deadlines,
- Single source shortest path: Dijkstra Algorithm
- **Minimum cost spanning trees:**
    - Kruskal Algorithm
    - Prim’s algorithm

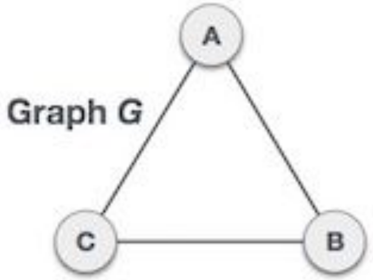
Extra Topics:

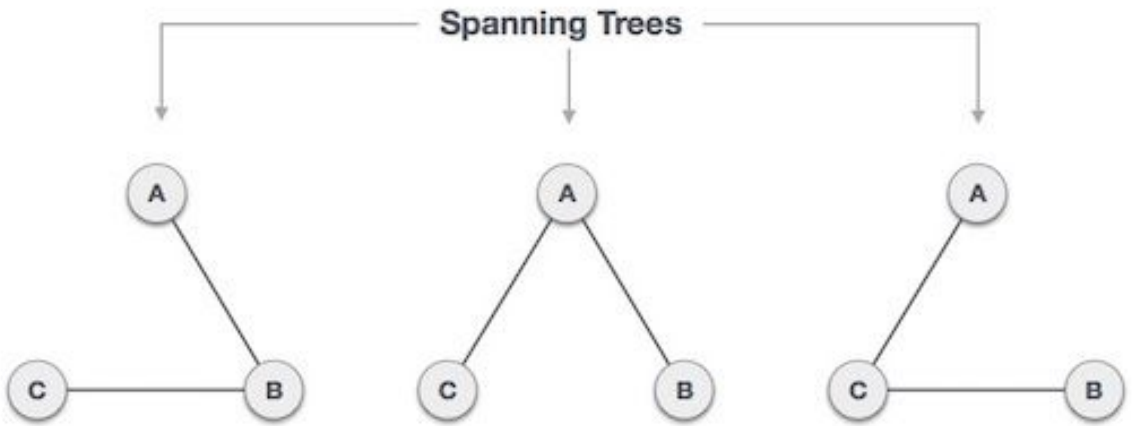
- Coin Change Problem
- Disjoint Set Data Structure

Spanning Tree

A **Spanning Tree** of a graph is a **subgraph** that:

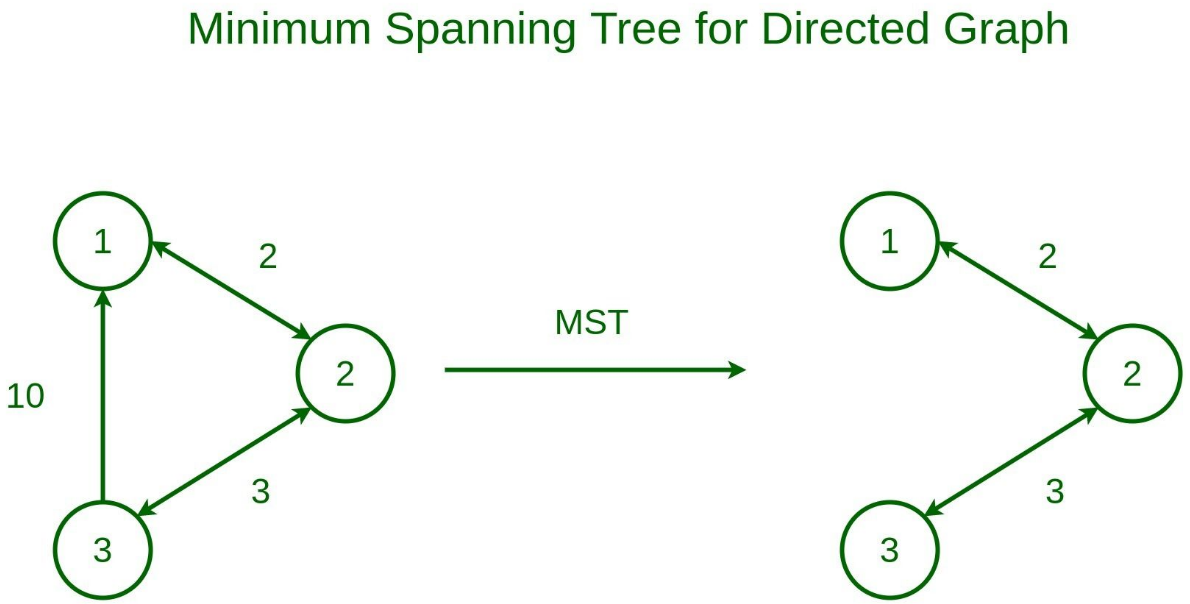
- Contains **all vertices**.
- Is **connected**.
- Has **no cycles**.
- Has **exactly** $V - 1$ **edges**, where V is the number of vertices.





Minimum Spanning Tree

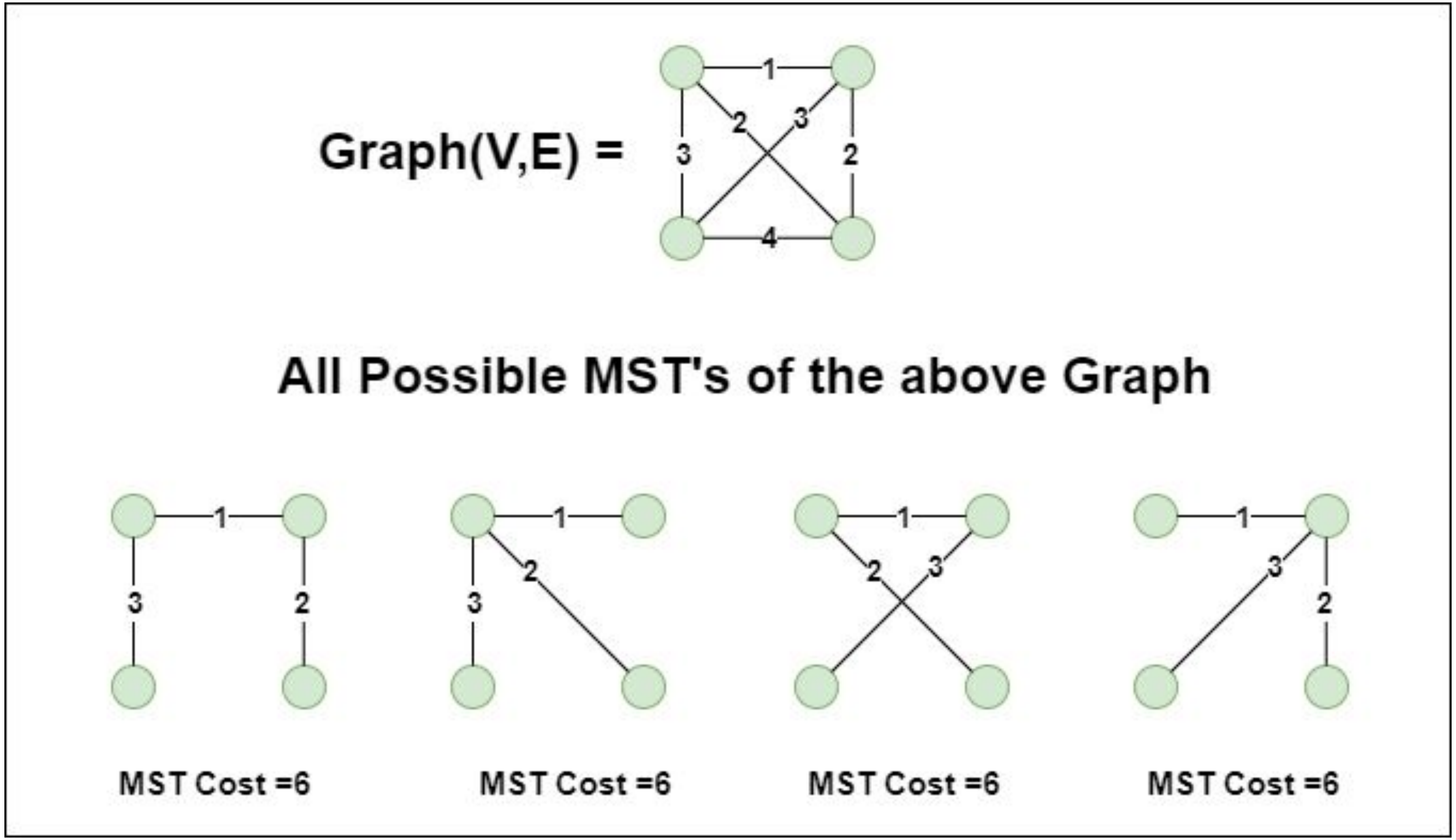
- Spanning tree with the **smallest possible total edge weight**.



Minimum Spanning Tree - Properties

1. **Connects all the vertices** in the graph, ensuring that there is a path between any pair of nodes.
2. A**cyclic** ⇒ contains no cycles. This property ensures that it remains a tree and not a graph with loops.
3. An **MST** with **V** vertices will have exactly **V - 1** edges
4. **Optimal** for minimizing the total edge weight, but it may not necessarily be unique.
5. **Cut property**
   - If you take any cut (a partition of the vertices into two sets) in the original graph and consider the minimum-weight edge that crosses the cut, that edge is part of the **MST**.

Minimum Spanning Tree - may not be unique



Minimum Spanning Tree (MST)

Applications of MST:

✅ **Network Design** – Laying out cables, road networks, power grids.

✅ **Cluster Analysis** – Grouping data points in machine learning.

✅ **Computer Vision** – Image segmentation.

Algorithms to Find MST

1. **Kruskal's MST Algorithm**
2. **Prim's MST Algorithm**
3. Boruvka's Algorithm
4. Reverse-Delete Algorithm

Topics to be covered

- General method
- Fractional Knapsack problem,
- Job sequencing with deadlines,
- Single source shortest path: Dijkstra Algorithm
- **Minimum cost spanning trees:**
  - **Kruskal Algorithm**
  - Prim’s algorithm

Extra Topics:

- Coin Change Problem
- Disjoint Set Data Structure

Minimum Spanning Tree (MST) - Kruskal's Algorithm

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

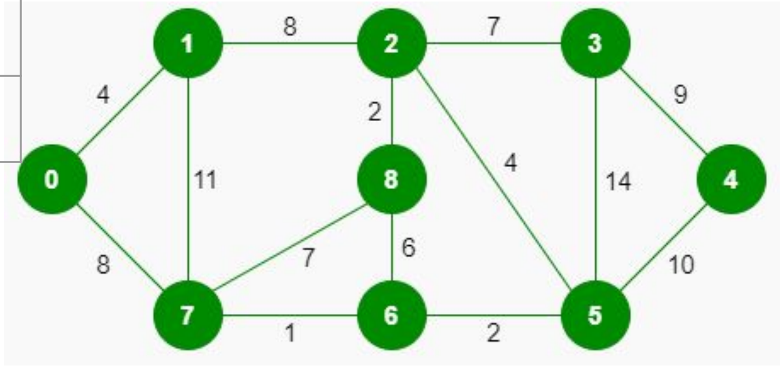
Greedy Choice Property: At every step, pick the **minimum-weight edge** without forming a cycle.

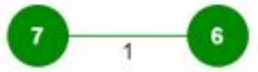
Optimal Substructure Property: The MST for the entire graph **builds upon the MSTs of its subgraphs**.

Courtesy : Prof. Galles - Kruskal's Algo

Minimum Spanning Tree (MST) - Kruskal's Algorithm Example

Selected Edge	EdgeSet	VertexSet	Cost
(7,6)	{(7,6)}	{7,6}	1

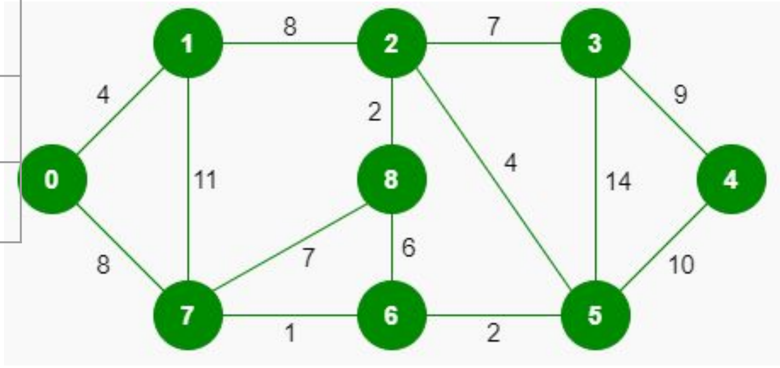


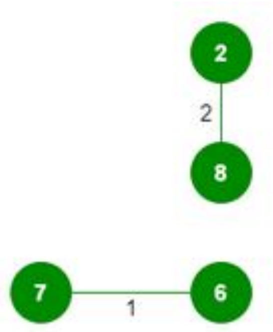


Courtesy : Prof. Galles - Kruskal's Algo

Minimum Spanning Tree (MST) - Kruskal's Algorithm Example

Selected Edge	EdgeSet	VertexSet	Cost
(7,6)	{(7,6)}	{6,7}	1
(2,8)	{(7,6),(2,8)}	{2,6,7,8}	3

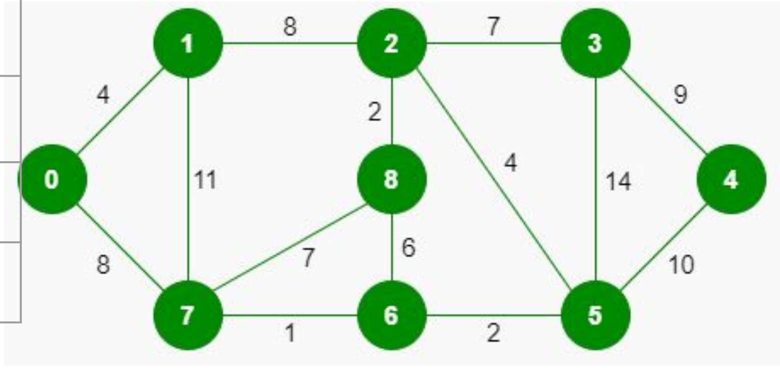


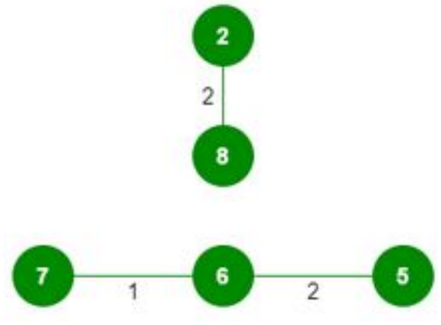


Courtesy : Prof. Galles - Kruskal's Algo

Minimum Spanning Tree (MST) - Kruskal's Algorithm Example

Selected Edge	EdgeSet	VertexSet	Cost
(7,6)	{(7,6)}	{6,7}	1
(2,8)	{(7,6),(2,8)}	{2,6,7,8}	3
(6,5)	{(7,6),(2,8),(6,5)}	{2,5,6,7,8}	5

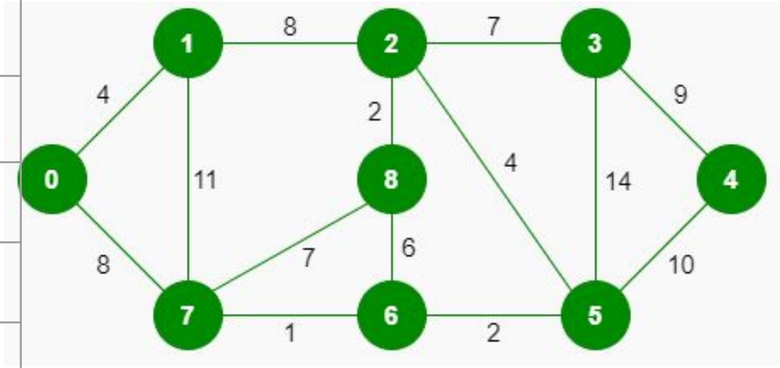


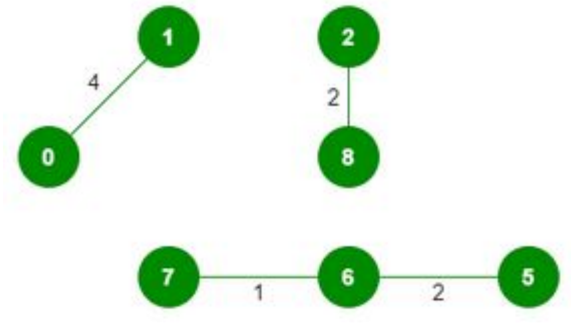


Courtesy : Prof. Galles - Kruskal's Algo

Minimum Spanning Tree (MST) - Kruskal's Algorithm Example

Selected Edge	EdgeSet	VertexSet	Cost
(7,6)	{(7,6)}	{6,7}	1
(2,8)	{(7,6),(2,8)}	{2,6,7,8}	3
(6,5)	{(7,6),(2,8),(6,5)}	{2,5,6,7,8}	5
(0,1)	{(7,6),(2,8),(6,5),(0,1)}	{0,1,2,5,6,7,8}	9

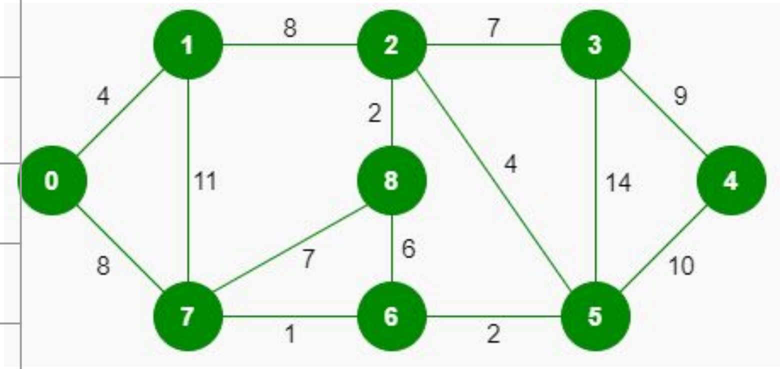


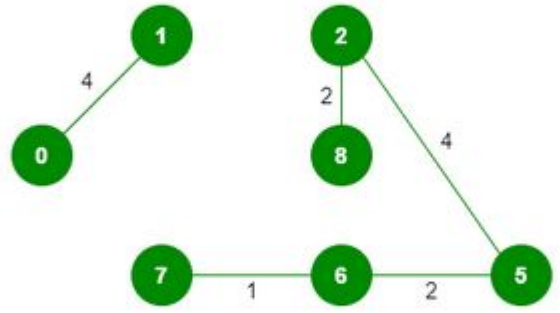


Courtesy : Prof. Galles - Kruskal's Algo

Minimum Spanning Tree (MST) - Kruskal's Algorithm Example

Selected Edge	EdgeSet	VertexSet	Cost
(7,6)	{(7,6)}	{6,7}	1
(2,8)	{(7,6),(2,8)}	{2,6,7,8}	3
(6,5)	{(7,6),(2,8),(6,5)}	{2,5,6,7,8}	5
(0,1)	{(7,6),(2,8),(6,5),(0,1)}	{0,1,2,5,6,7,8}	9
(2,5)	{(7,6),(2,8),(6,5),(0,1),(2,5)}	{0,1,2,5,6,7,8}	13

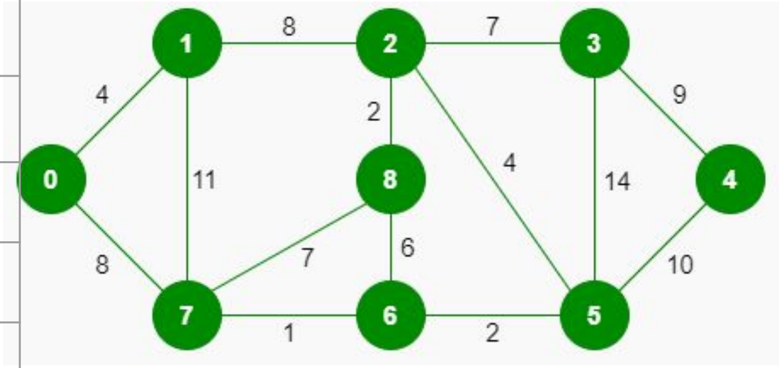


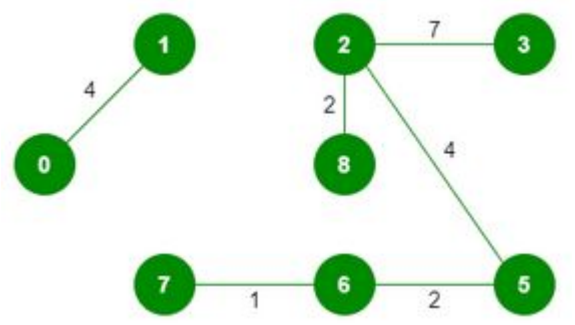


Courtesy : Prof. Galles - Kruskal's Algo

Minimum Spanning Tree (MST) - Kruskal's Algorithm Example

Selected Edge	EdgeSet	VertexSet	Cost
(7,6)	{(7,6)}	{6,7}	1
(2,8)	{(7,6),(2,8)}	{2,6,7,8}	3
(6,5)	{(7,6),(2,8),(6,5)}	{2,5,6,7,8}	5
(0,1)	{(7,6),(2,8),(6,5),(0,1)}	{0,1,2,5,6,7,8}	9
(2,5)	{(7,6),(2,8),(6,5),(0,1),(2,5)}	{0,1,2,5,6,7,8}	13
(2,3)	{(7,6),(2,8),(6,5),(0,1),(2,5),(2,3)}	{0,1,2,3,5,6,7,8}	20

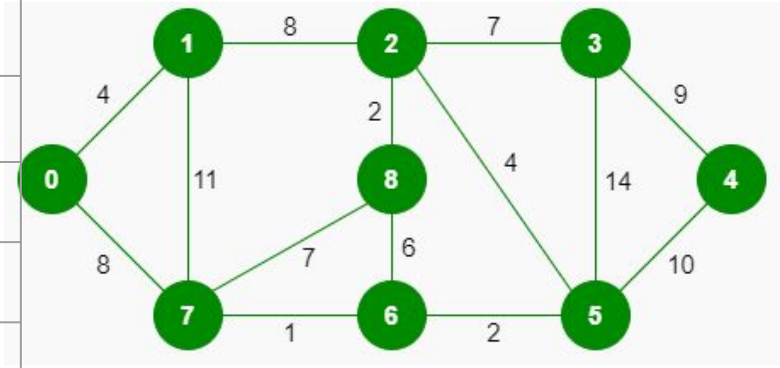


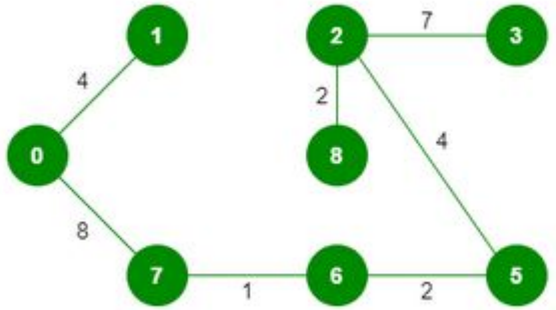


Courtesy : Prof. Galles - Kruskal's Algo

Minimum Spanning Tree (MST) - Kruskal's Algorithm Example

Selected Edge	EdgeSet	VertexSet	Cost
(7,6)	{(7,6)}	{6,7}	1
(2,8)	{(7,6),(2,8)}	{2,6,7,8}	3
(6,5)	{(7,6),(2,8),(6,5)}	{2,5,6,7,8}	5
(0,1)	{(7,6),(2,8),(6,5),(0,1)}	{0,1,2,5,6,7,8}	9
(2,5)	{(7,6),(2,8),(6,5),(0,1),(2,5)}	{0,1,2,5,6,7,8}	13
(2,3)	{(7,6),(2,8),(6,5),(0,1),(2,5),(2,3)}	{0,1,2,3,5,6,7,8}	20
(0,7)	{(7,6),(2,8),(6,5),(0,1),(2,5),(2,3),(0,7)}	{0,1,2,3,5,6,7,8}	28

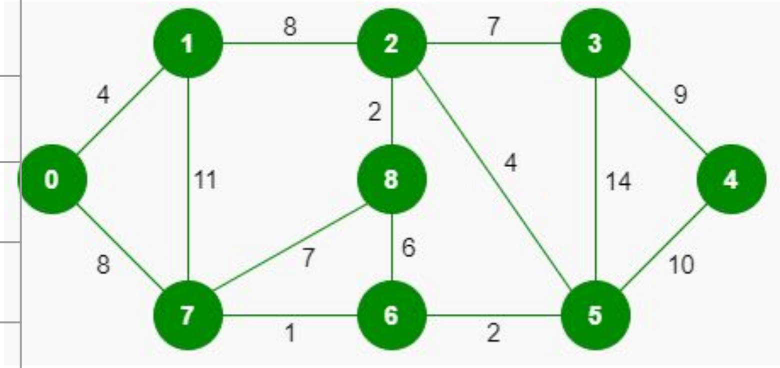
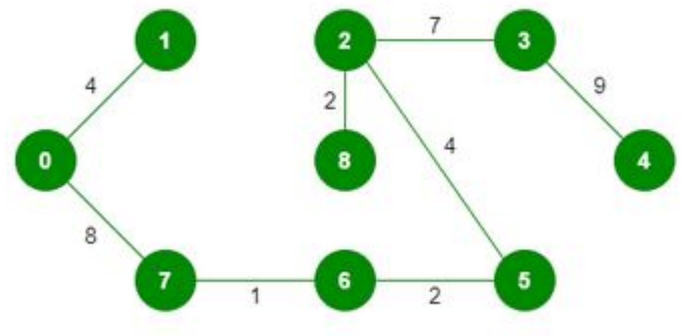




Courtesy : Prof. Galles - Kruskal's Algo

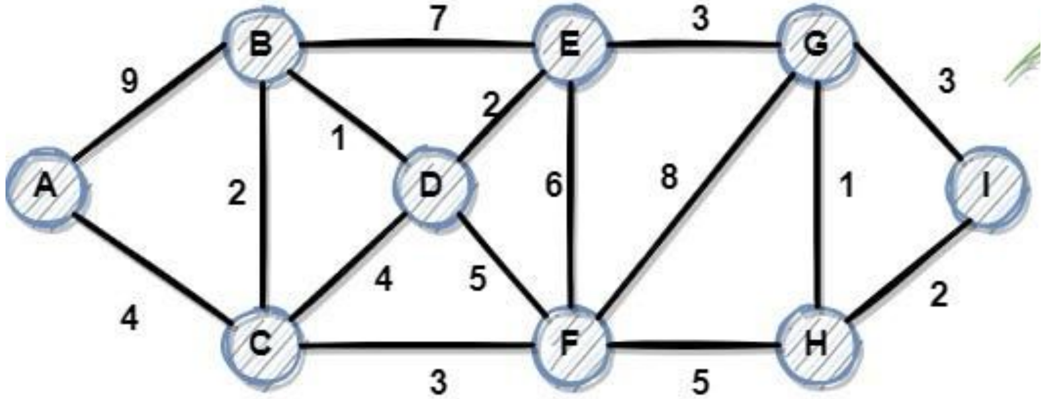
Minimum Spanning Tree (MST) - Kruskal's Algorithm Example

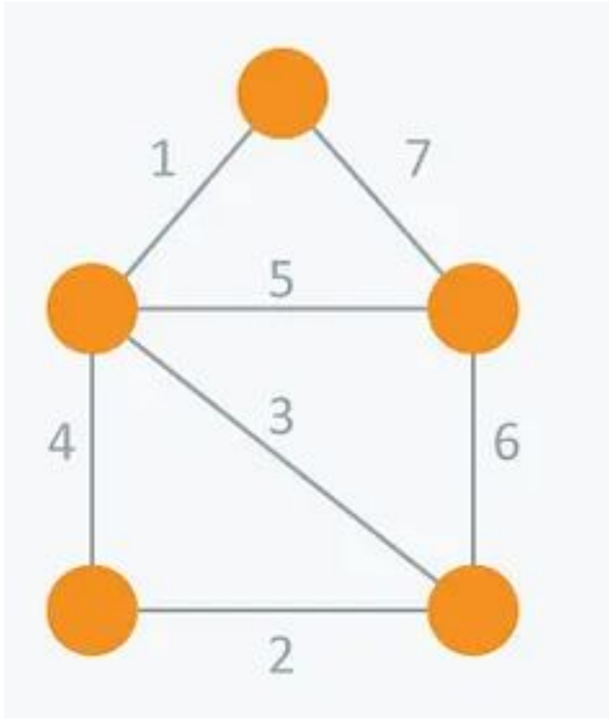
Selected Edge	EdgeSet	VertexSet	Cost
(7,6)	{(7,6)}	{6,7}	1
(2,8)	{(7,6),(2,8)}	{2,6,7,8}	3
(6,5)	{(7,6),(2,8),(6,5)}	{2,5,6,7,8}	5
(0,1)	{(7,6),(2,8),(6,5),(0,1)}	{0,1,2,5,6,7,8}	9
(2,5)	{(7,6),(2,8),(6,5),(0,1),(2,5)}	{0,1,2,5,6,7,8}	13
(2,3)	{(7,6),(2,8),(6,5),(0,1),(2,5),(2,3)}	{0,1,2,3,5,6,7,8}	20
(0,7)	{(7,6),(2,8),(6,5),(0,1),(2,5),(2,3),(0,7)}	{0,1,2,3,5,6,7,8}	28
(3,4)	{(7,6),(2,8),(6,5),(0,1),(2,5),(2,3),(0,7),(3,4)}	{0,1,2,3,4,5,6,7,8}	**37**

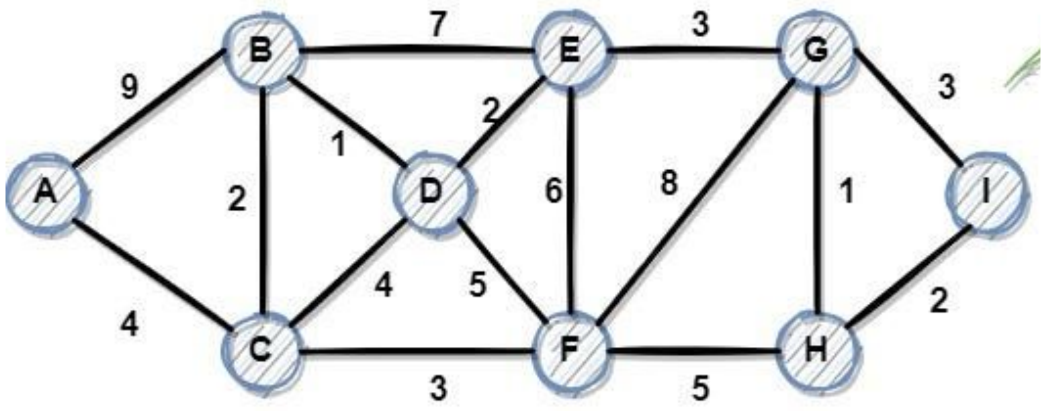
Courtesy : Prof. Galles - Kruskal's Algo

Minimum Spanning Tree (MST) - Kruskal's Algorithm Problems

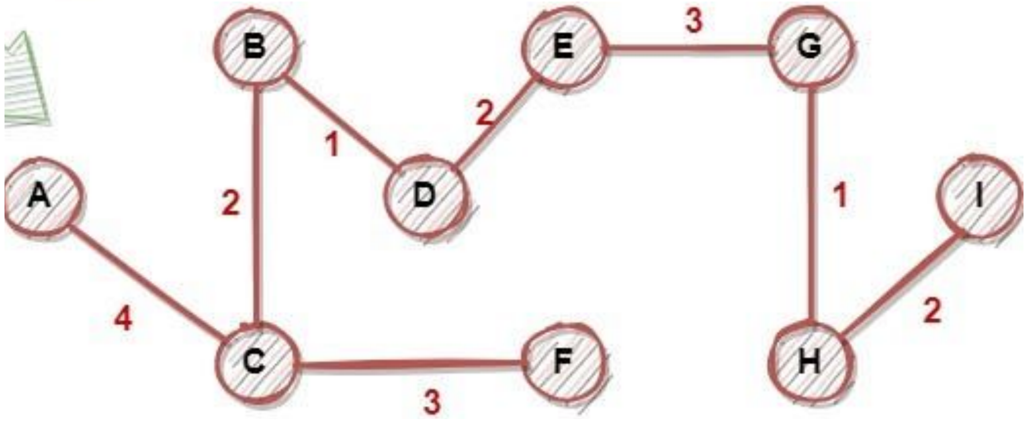




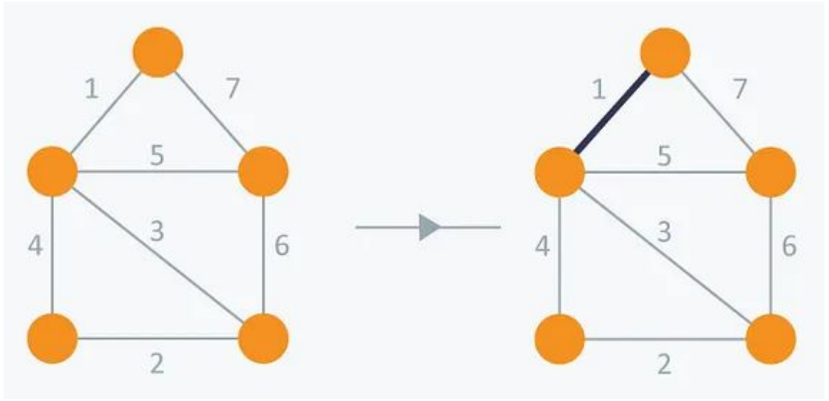
Minimum Spanning Tree (MST) - Kruskal's Algorithm Problems

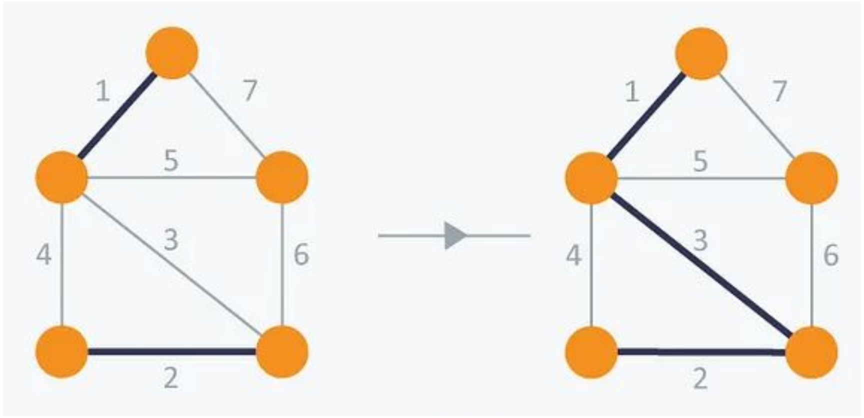


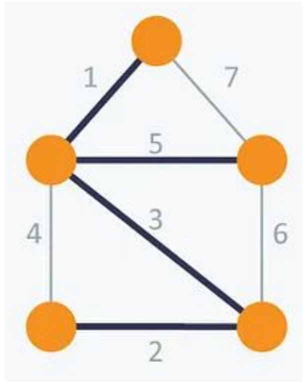
Minimum Cost = **18**



Minimum Spanning Tree (MST) - Kruskal's Algorithm Problems

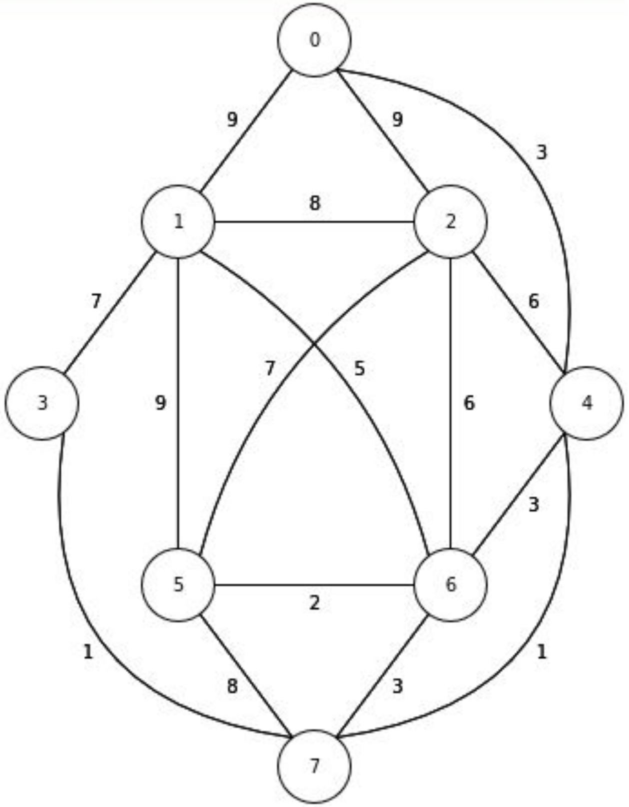


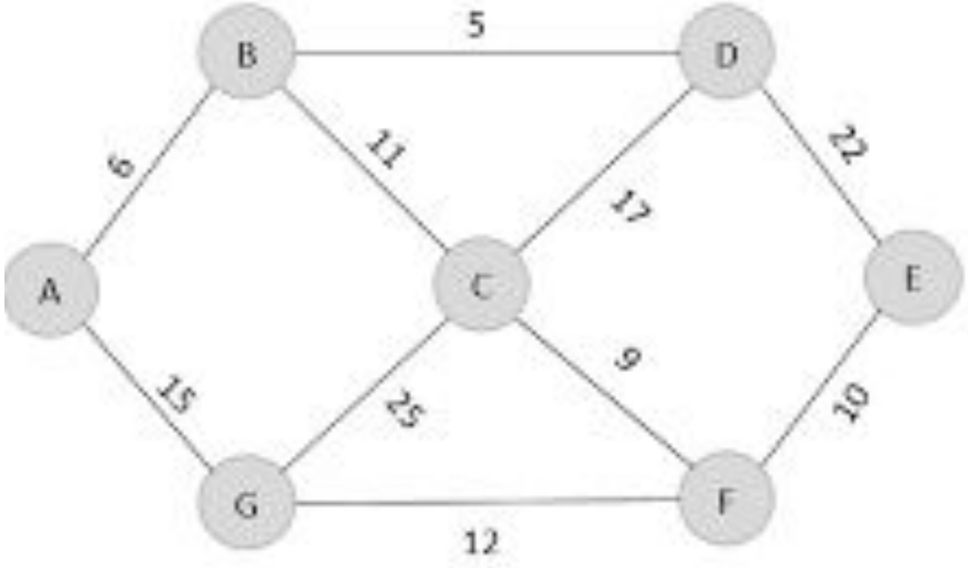




Minimum Cost = **11**

Minimum Spanning Tree (MST) - Kruskal's Algorithm Problems





Disjoint Sets (Data Structure)

Parameter	Joint Set	Disjoint Set
Definition	Sets which have common elements i.e., intersection of the set is not an empty set.	Sets which do not have common elements i.e., intersection of the set is an empty set
Condition	$A \cap B \neq \phi$	$A \cap B = \phi$
Example	If X = {a, b} and Y = {b, c} then, **$X \cap Y = \{b\}$** X and Y are joint sets.	If X = {a, d} and Y = {b, c} then, **$X \cap Y = \phi$** X and Y are disjoint sets.

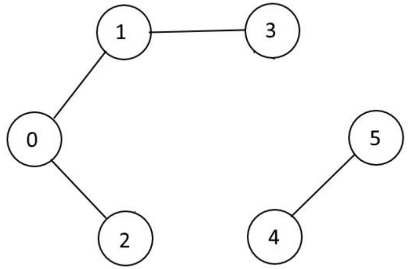
Disjoint Sets (Data Structure)

Disjoint Set (Union-Find)

- data structure that **keeps track of a partition of elements into disjoint sets**.
- a collection of **non-overlapping** sets.
- Each set has a **representative (or leader)** element.
- **Union-Find** provides two key operations:
  1. **Find(x):** Determines the **representative** of the set containing xxx.
  2. **Union(x, y):** Merges the sets containing xxx and yyy.

Applications :

- **Kruskal's Algorithm**,
- **Network connectivity problems**,
- **Image processing**.



Disjoint Sets

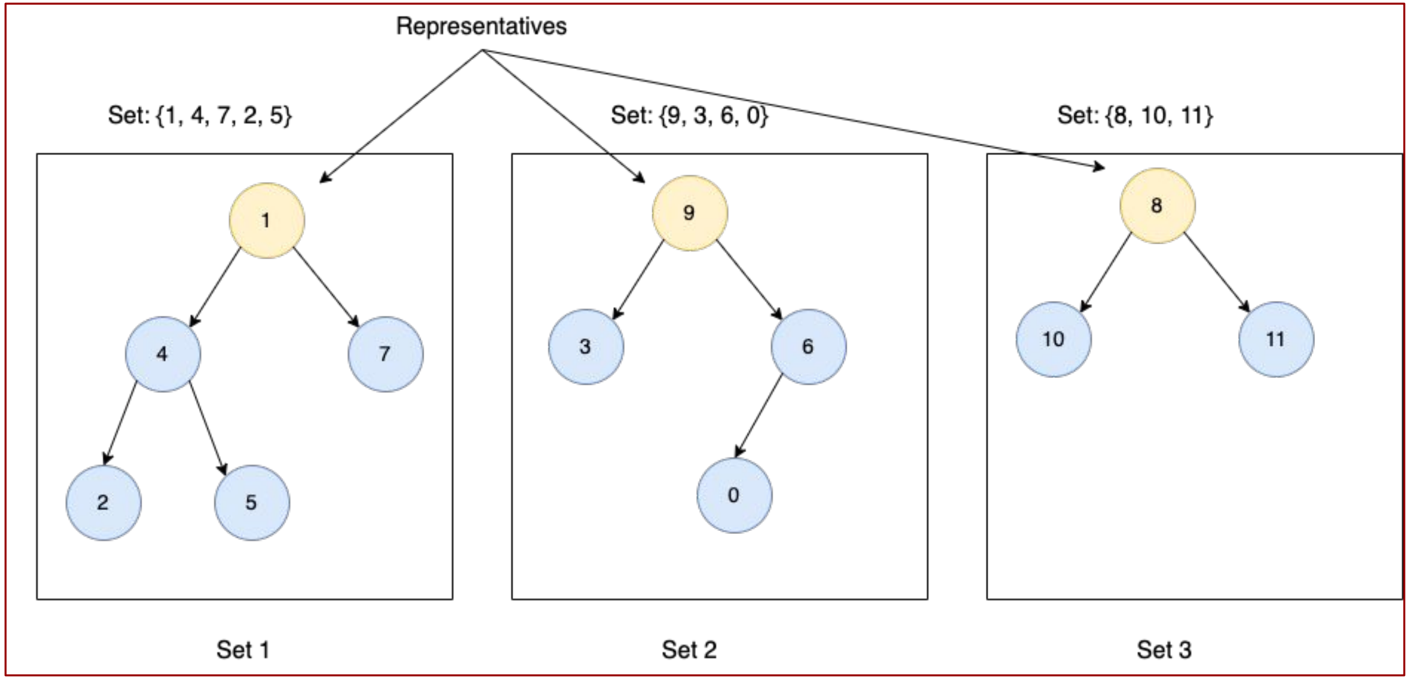
Set Id: 0, elements: [0, 1, 2, 3]

Set Id: 4, elements: [4, 5]

Courtesy : Tutorialhorizon.com

Disjoint Sets (Data Structure)

- Internally, disjoint set is nothing but a collection of trees.
- Hence we call it a Forest.
- Every disjoint set is represented as an independent Tree.
- Each set is represented by one element (The Root of the tree.)



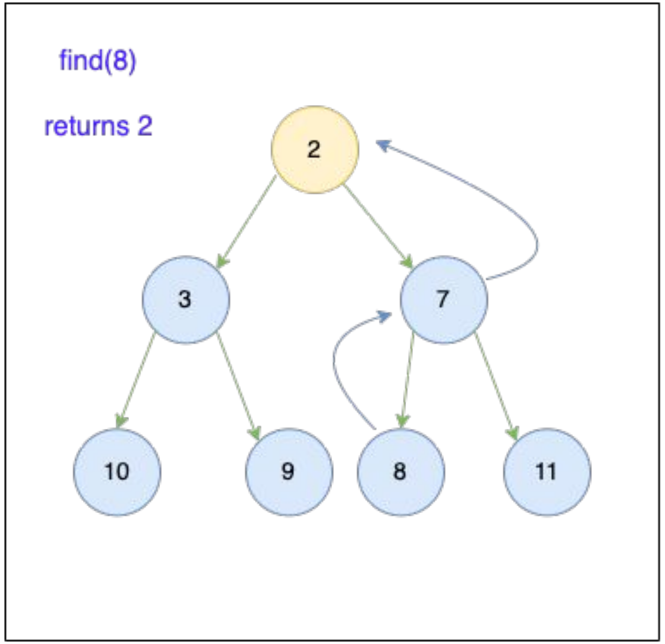
Courtesy : [Sourav Prateek - LinkedIn]

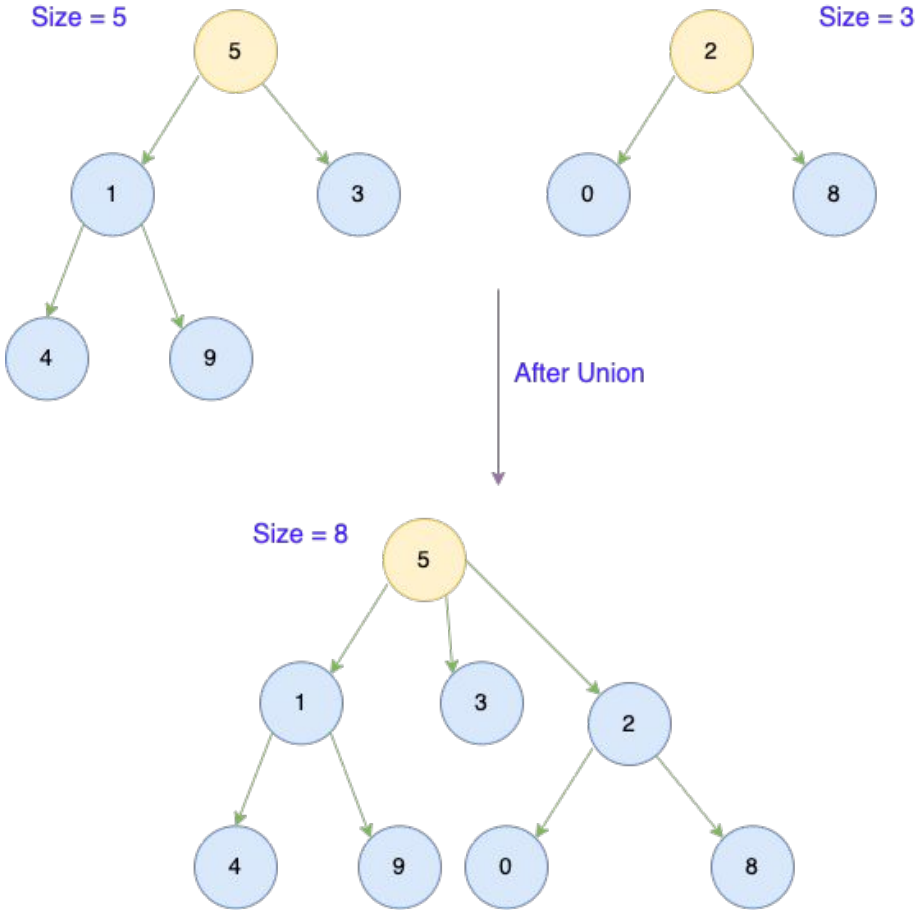
Disjoint Sets (Data Structure)

1. **MakeSet(X)**:
   - Constructs a new set by creating a new element with a parent pointer to itself.
   - The parent pointer to itself indicates that the element is the representative member of its own set.
   - Time complexity : **O(1)**
2. **Find(X):**
   - follows the chain of parent pointers from x upwards through the tree until an element is reached whose parent is itself.
   - This element is the root of the tree.
   - Root is the representative member of the set to which x belongs, and may be x itself.
3. **Union**(x,y):
   - Uses Find to determine the roots of the trees x and y belong to.
   - If the roots are distinct, the trees are combined by attaching the root of one to the root of the other.
   - If this is done naively, such as by always making x a child of y, the height of the trees can grow as O(n).

Courtesy : Tutorialhorizon.com

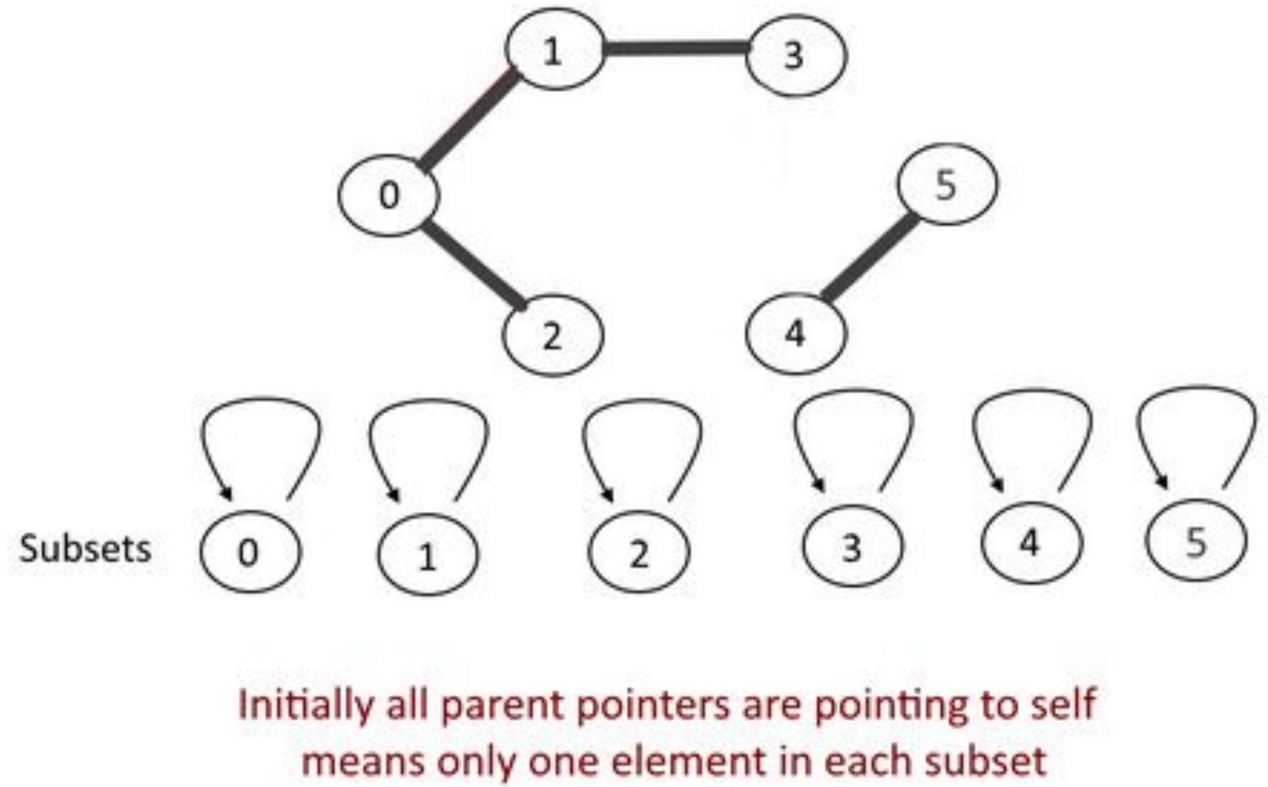
Disjoint Sets (Data Structure)





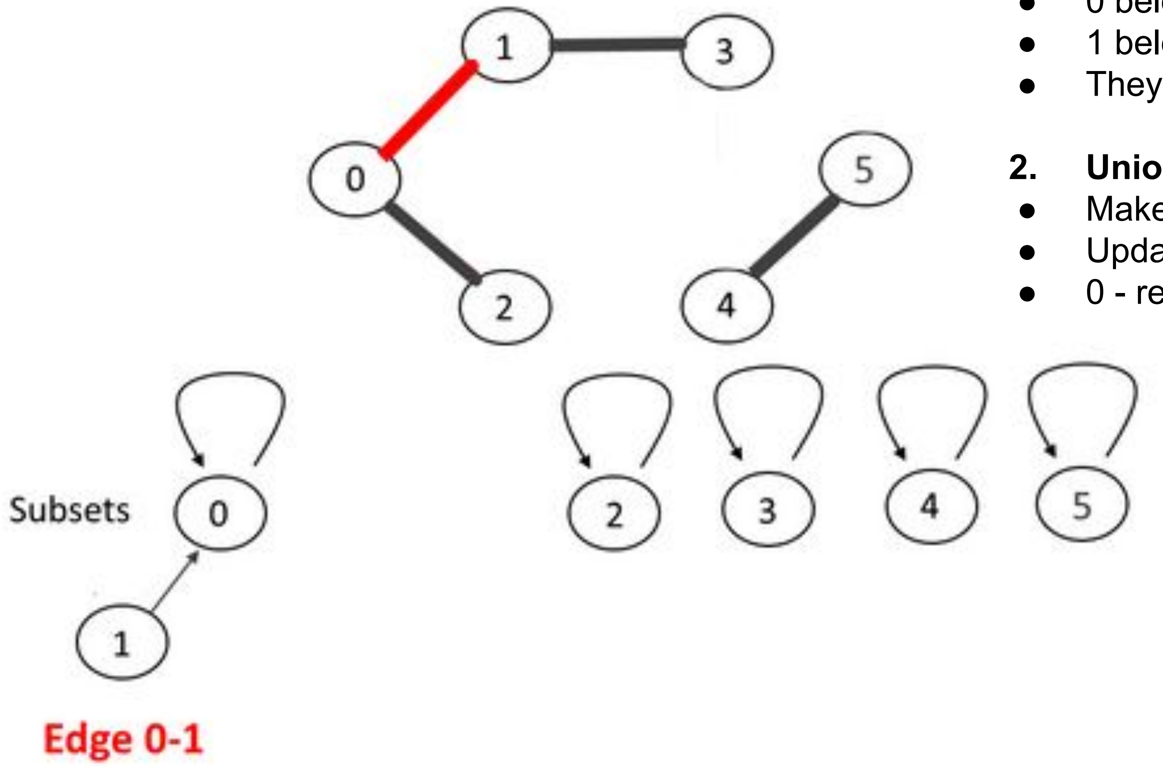
Courtesy : Sourav Prateek - LinkedIn

Disjoint Sets (Data Structure) Example for a given Graph


1
3
0
5
2
4
Subsets
0
1
2
3
4
5
Initially all parent pointers are pointing to self
means only one element in each subset

Courtesy : Tutorialhorizon.com

Disjoint Sets (Data Structure)

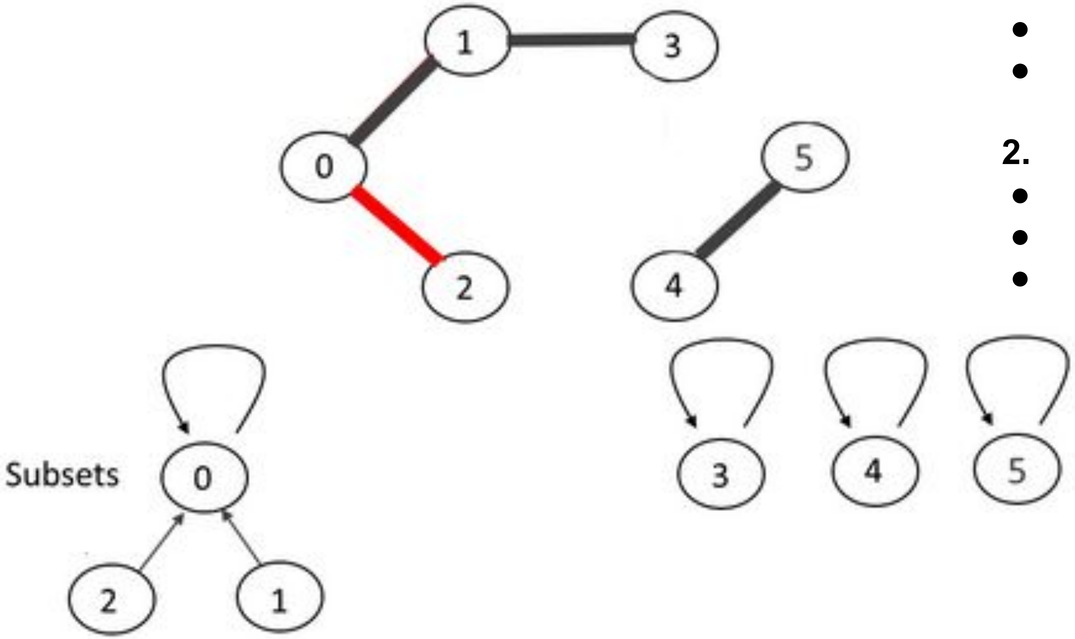


1. **Find:**
- 0 belongs to subset {0}
- 1 belongs to subset {1}
- They are in different subsets.

2. **Union**
- Make 0 the parent of 1
- Update the set {0,1}
- 0 - representative member as 0 is the parent to itself

Courtesy : Tutorialhorizon.com

Disjoint Sets (Data Structure)



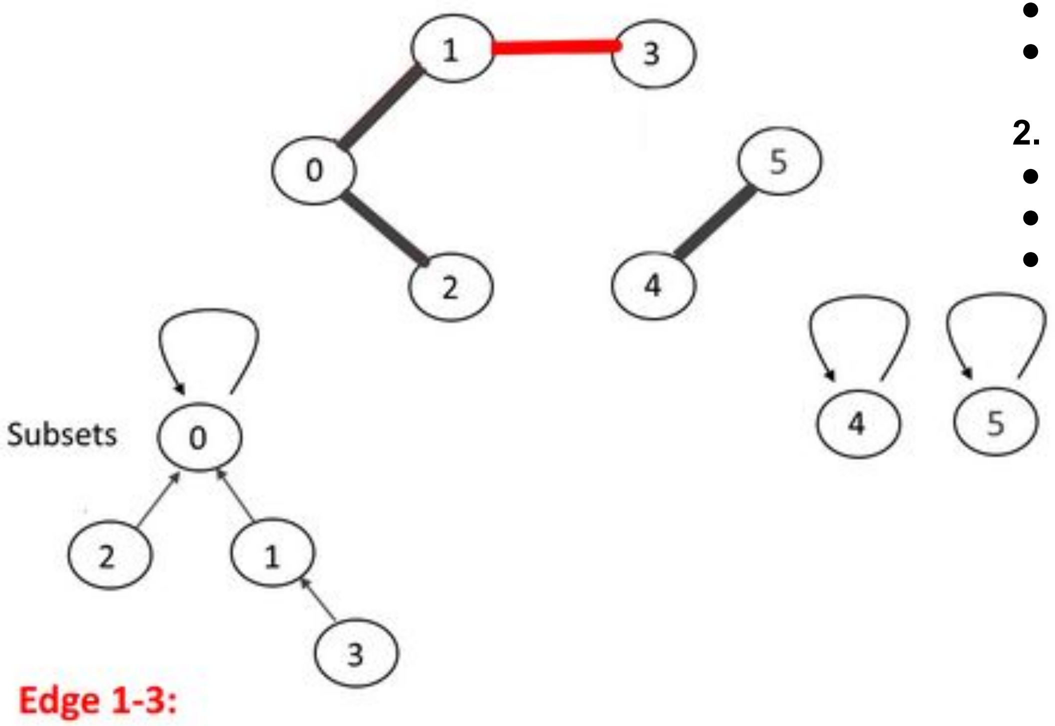
1. **Find:**
- 0 belongs to subset {0,1}
- 2 belongs to subset {2}
- They are in different subsets.

2. **Union**
- Make 0 the parent of 2
- Update the set {0,1,2}
- 0 - representative member as 0 is the parent to itself

Edge 0-2:

Courtesy : Tutorialhorizon.com

Disjoint Sets (Data Structure)

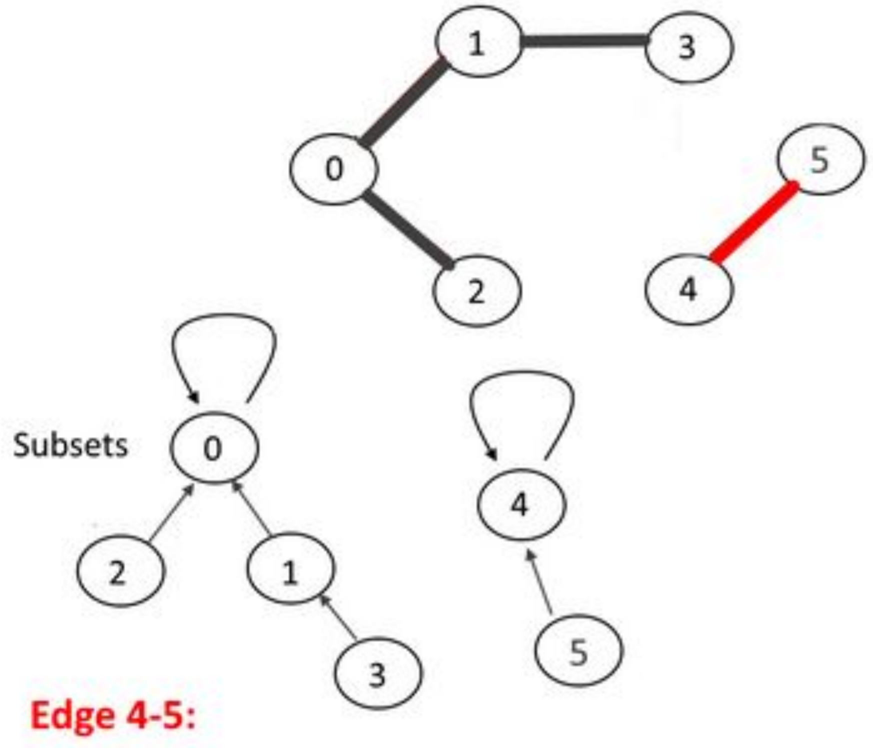


1. **Find:**
- 1 belongs to subset {0,1,2}
- 3 belongs to subset {3}
- They are in different subsets.

2. **Union**
- Make 1 the parent of 3
- Update the set {0,1,2,3}
- 0 - representative member as 0 is the parent to itself

Courtesy : Tutorialhorizon.com

Disjoint Sets (Data Structure)



1. **Find:**
- 4 belongs to subset {4}
- 5 belongs to subset {5}
- They are in different subsets.

2. **Union**
- Make 4 the parent of 5
- Update the set {4,5}
- 4 - representative member as 4 is the parent to itself

Courtesy : Tutorialhorizon.com

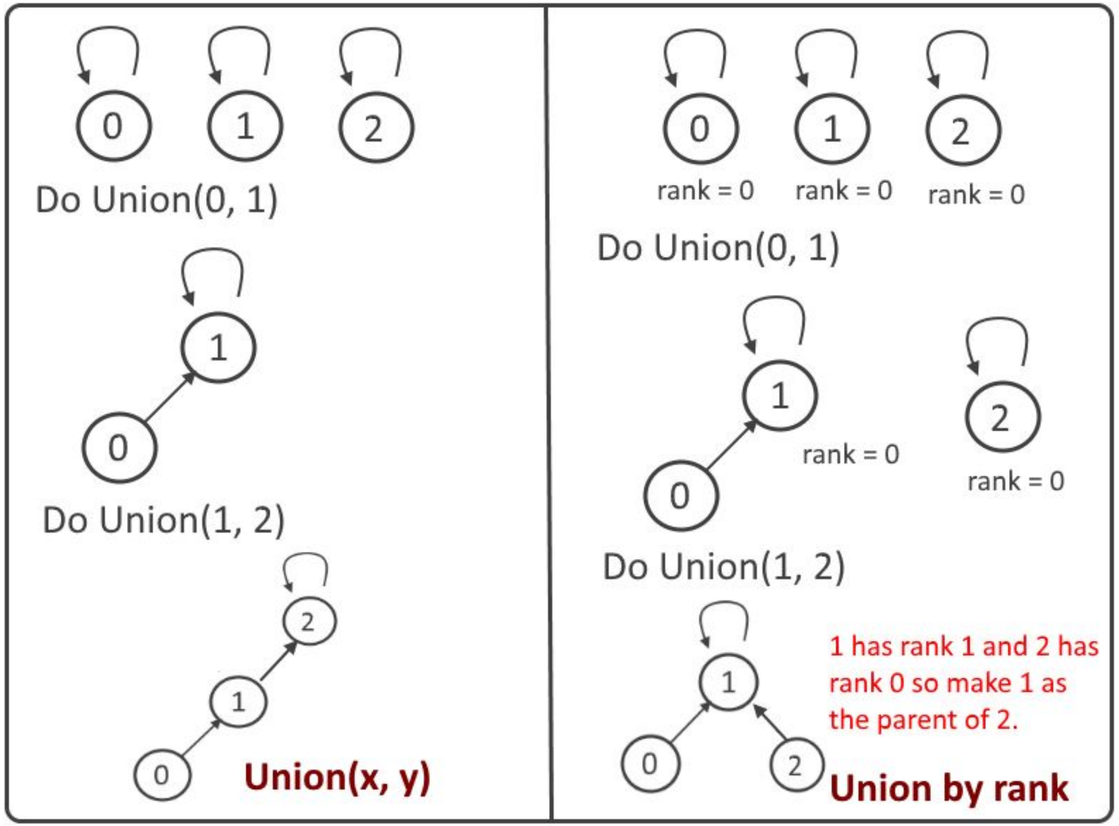
Disjoint Sets (Data Structure) Optimization

Method - 1 : **Union by Rank**

- Each element is associated with a rank.
- Initially a set has one element and a rank of zero.
- Attache the shorter tree to the root of the taller tree.
- Union of two sets
    1. Both trees have the same rank - the resulting set's rank is one larger
    2. Both trees have the different ranks - the resulting set's rank is the larger of the two.
- Worst case complexity: **O(log N)**

Courtesy : Tutorialhorizon.com

Disjoint Sets (Data Structure) Optimization



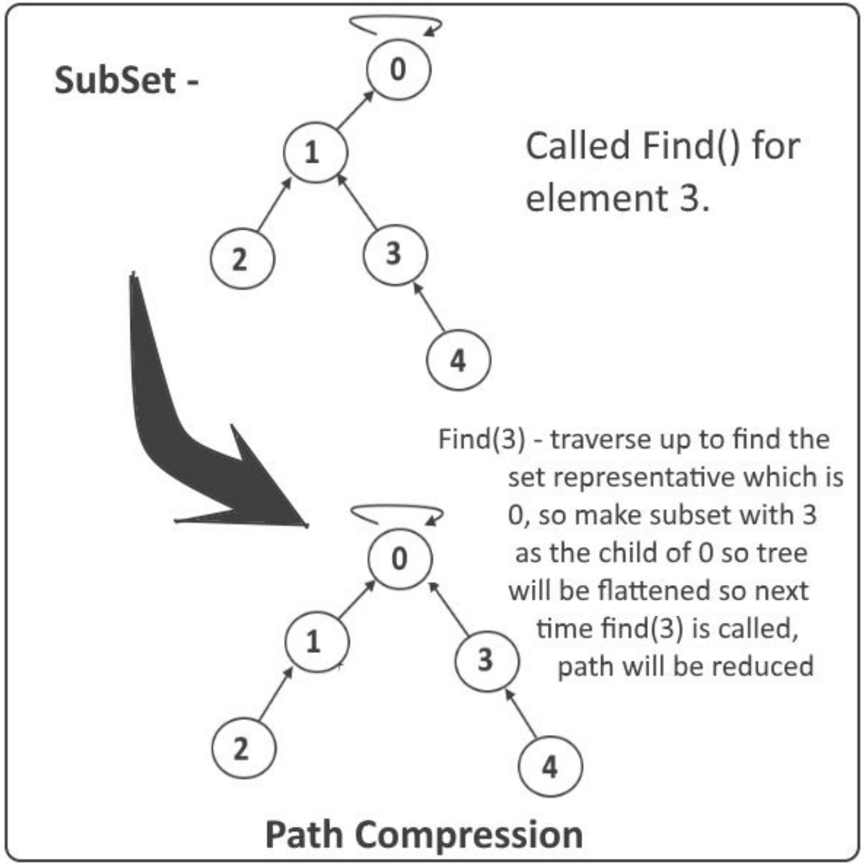
Courtesy : Tutorialhorizon.com

Disjoint Sets (Data Structure) Optimization

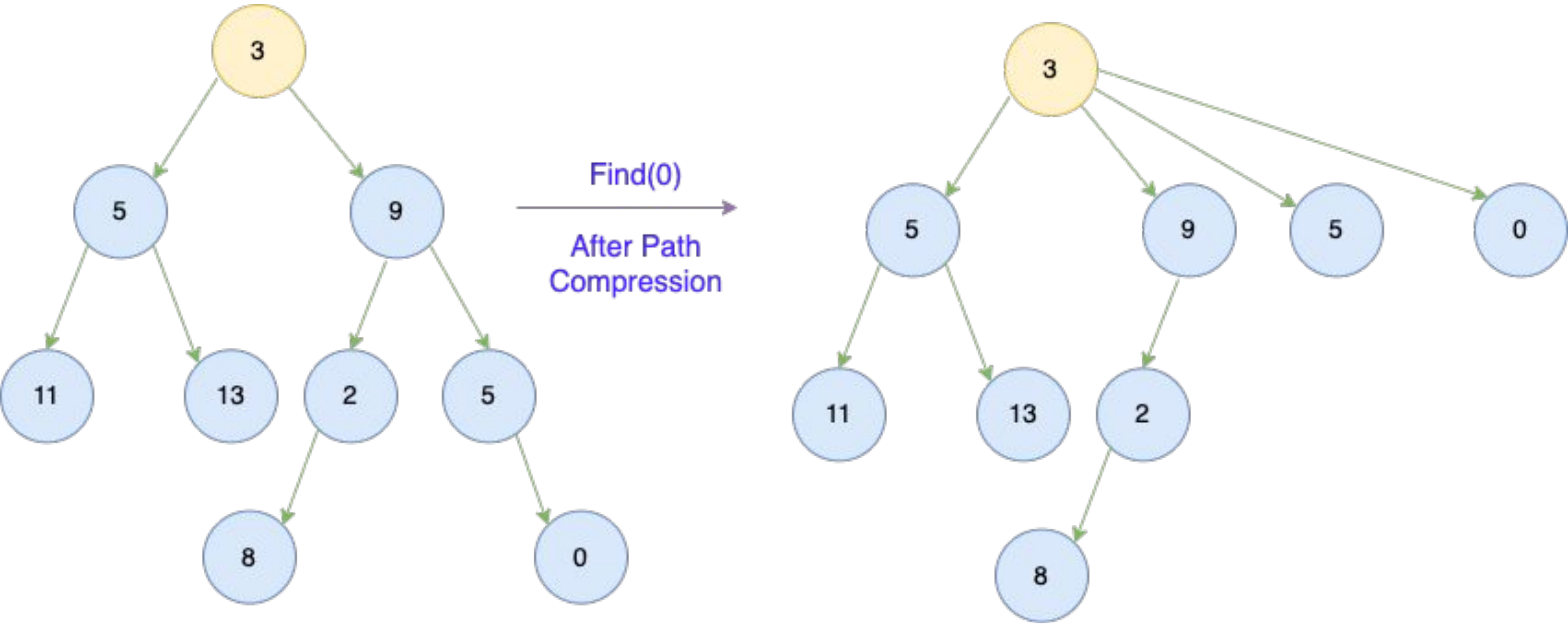
Method - 2 : **Path by compression**

- Way of flattening the structure of the tree whenever **Find** is used on it.
- Each element visited on the way to a root is part of the same set, all of these visited elements can be reattached directly to the root.
- Resulting tree is much flatter
- It speeds up future operations
  (on these elements, and on those referencing them)

Courtesy : Tutorialhorizon.com


SubSet -
0
1
2
3
4
Called Find() for element 3.
Find(3) - traverse up to find the set representative which is 0, so make subset with 3 as the child of 0 so tree will be flattened so next time find(3) is called, path will be reduced
0
1
3
2
4
Path Compression

Disjoint Sets (Data Structure) Optimization

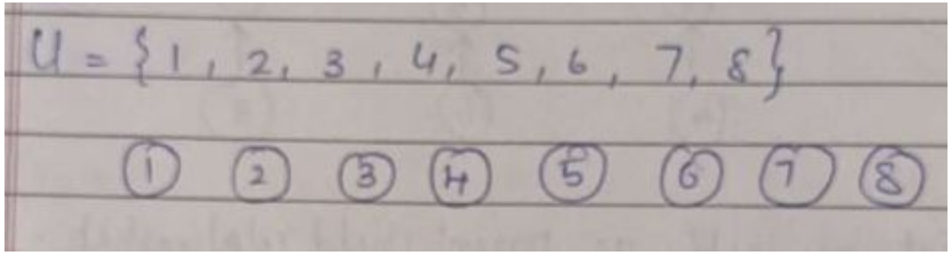


Courtesy : Sourav Prateek - LinkedIn

Disjoint Sets (Data Structure) Array representation

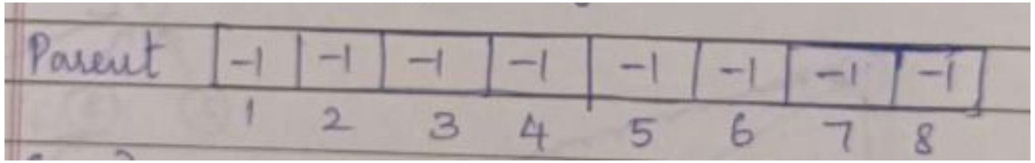
Assumption :

- Each element in **U** are independent sets

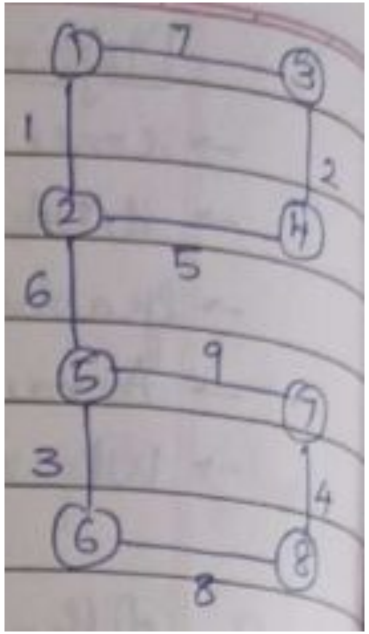


U = {1, 2, 3, 4, 5, 6, 7, 8}

- Parent element is maintained for these sets which is initialized to -1 (Each set is a parent to itself)

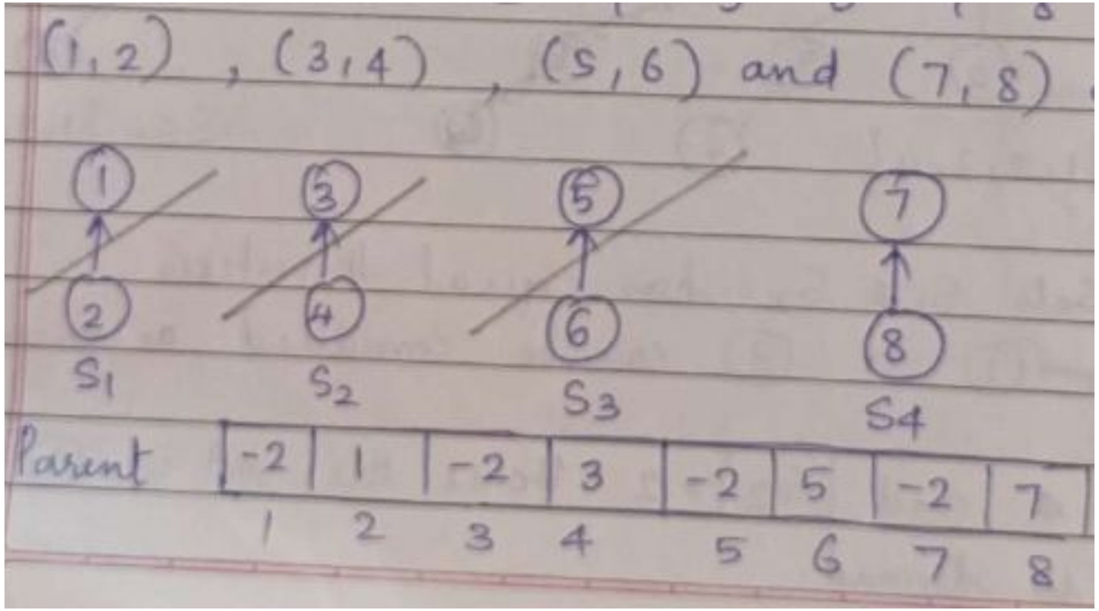


Parent	-1	-1	-1	-1	-1	-1	-1	-1
	1	2	3	4	5	6	7	8



Disjoint Sets (Data Structure) Array representation

- Select the edges and form sets S_1, S_2, S_3 and S_4


(1, 2), (3, 4), (5, 6) and (7, 8)

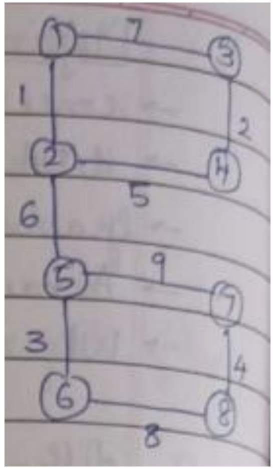
Parent	-2	1	-2	3	-2	5	-2	7
	1	2	3	4	5	6	7	8

2’s parent is 1 & 1 has 2 nodes ⇒ -2

4’s parent is 3 & 3 has 2 nodes ⇒ -2

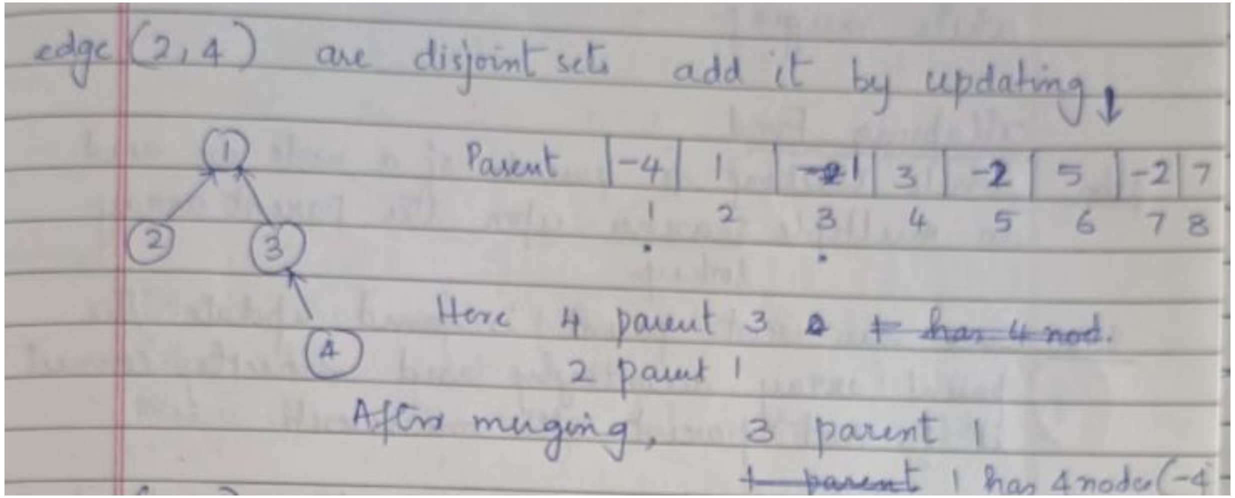
6’s parent is 5 & 5 has 2 nodes ⇒ -2

8’s parent is 7 & 7 has 2 nodes ⇒ -2



Disjoint Sets (Data Structure) Array representation

- Select the edge (2,4)

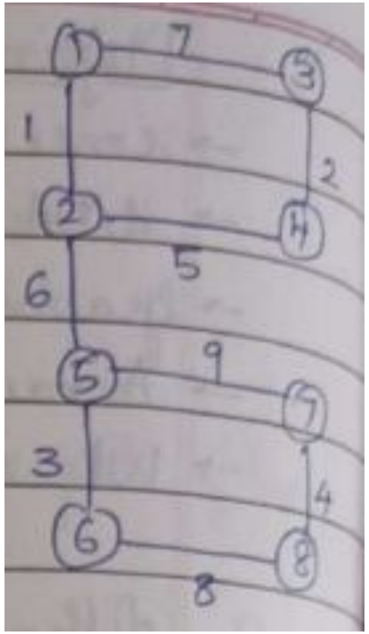


edge (2, 4) are disjoint sets add it by updating ↓

Parent	-4	1	~~-2~~ 1	3	-2	5	-2	7
	1	2	3	4	5	6	7	8

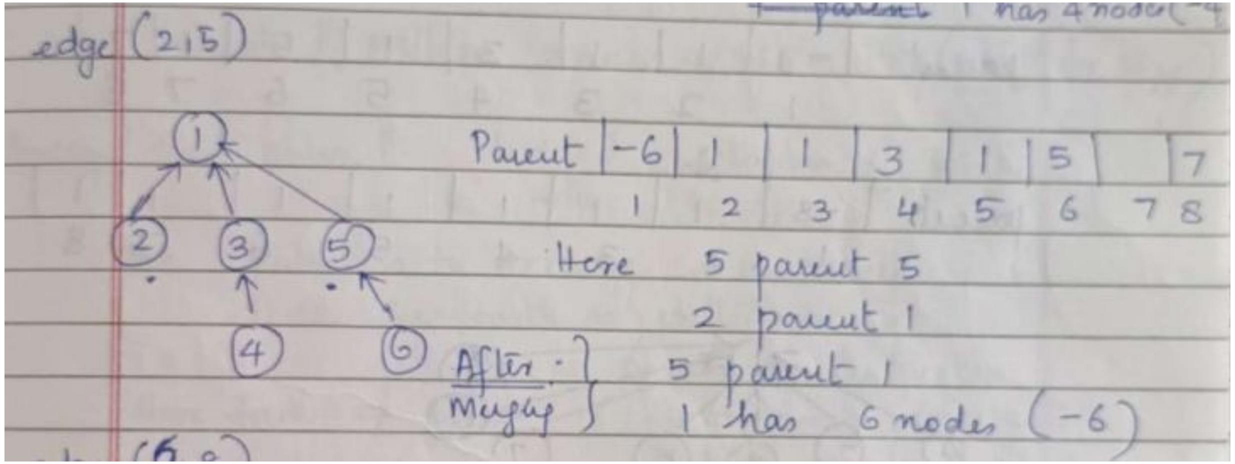
$$\begin{tikzcd} & 1 & \\ 2 \arrow[ur] & & 3 \arrow[ul] \\ & & 4 \arrow[u] \end{tikzcd}$$

Here 4 parent 3 ~~&~~ ~~1 has 4 nod.~~
2 parent 1
After merging, 3 parent 1
~~1 parent~~ 1 has 4 node (-4



Disjoint Sets (Data Structure) Array representation

- Select the edge (2,5)

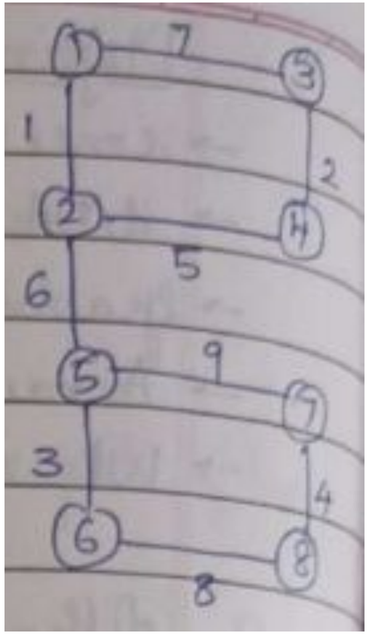


edge (2,5)

Parent	-6	1	1	3	1	5		7
	1	2	3	4	5	6	7	8

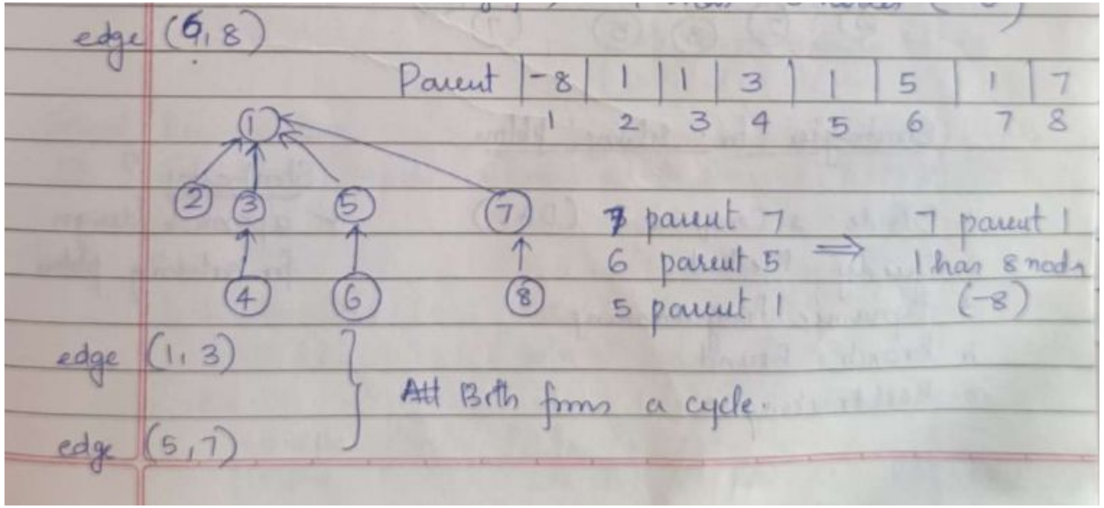
Here 5 parent 5
2 parent 1

After Merging } 5 parent 1
1 has 6 nodes (-6)



Disjoint Sets (Data Structure) Array representation

- Select the edge (2,5)



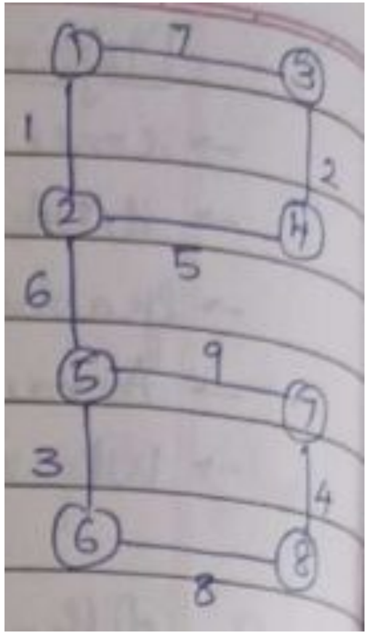
edge (6, 8)

Parent	-8	1	1	3	1	5	1	7
	1	2	3	4	5	6	7	8

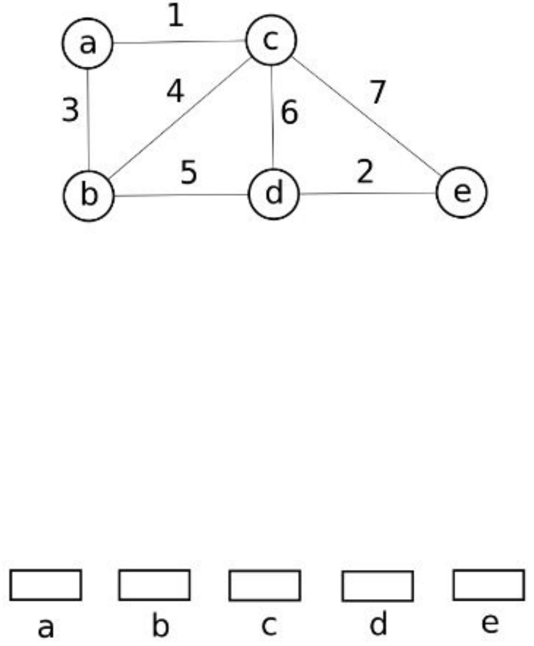
8 parent 7
6 parent 5
5 parent 1

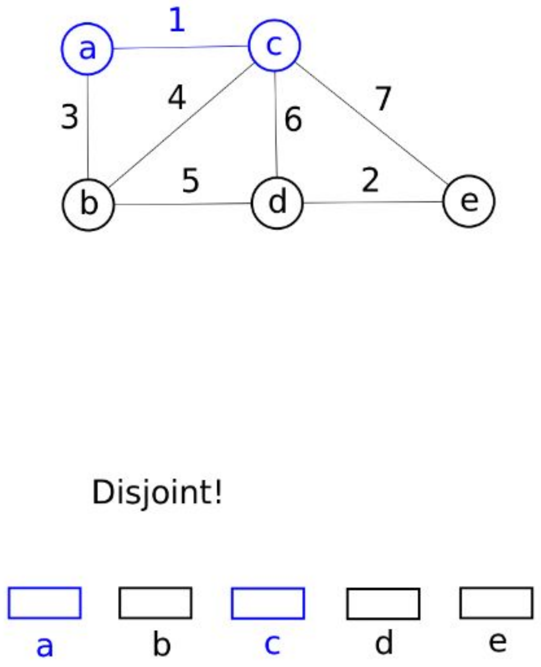
⟹ 7 parent 1
1 has 8 nodes
(-8)

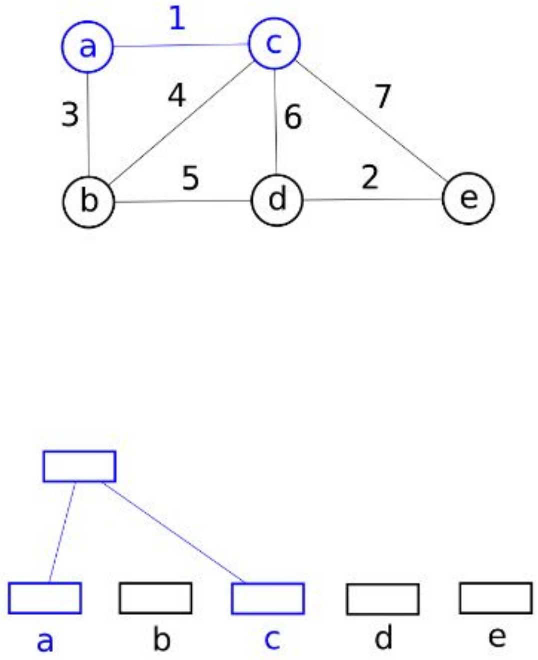
edge (1, 3)
edge (5, 7)
} ~~Att~~ Both forms a cycle.



MST Implementation of Kruskal's Algorithm using Disjoint Sets

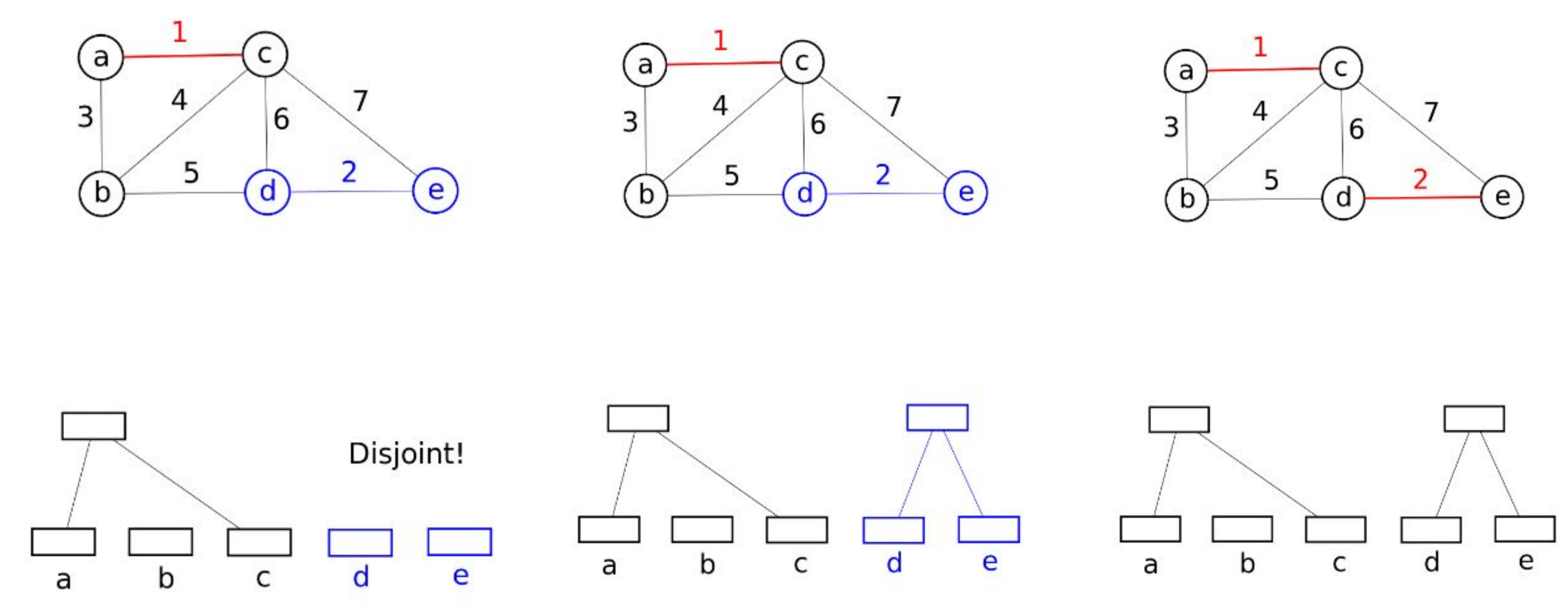






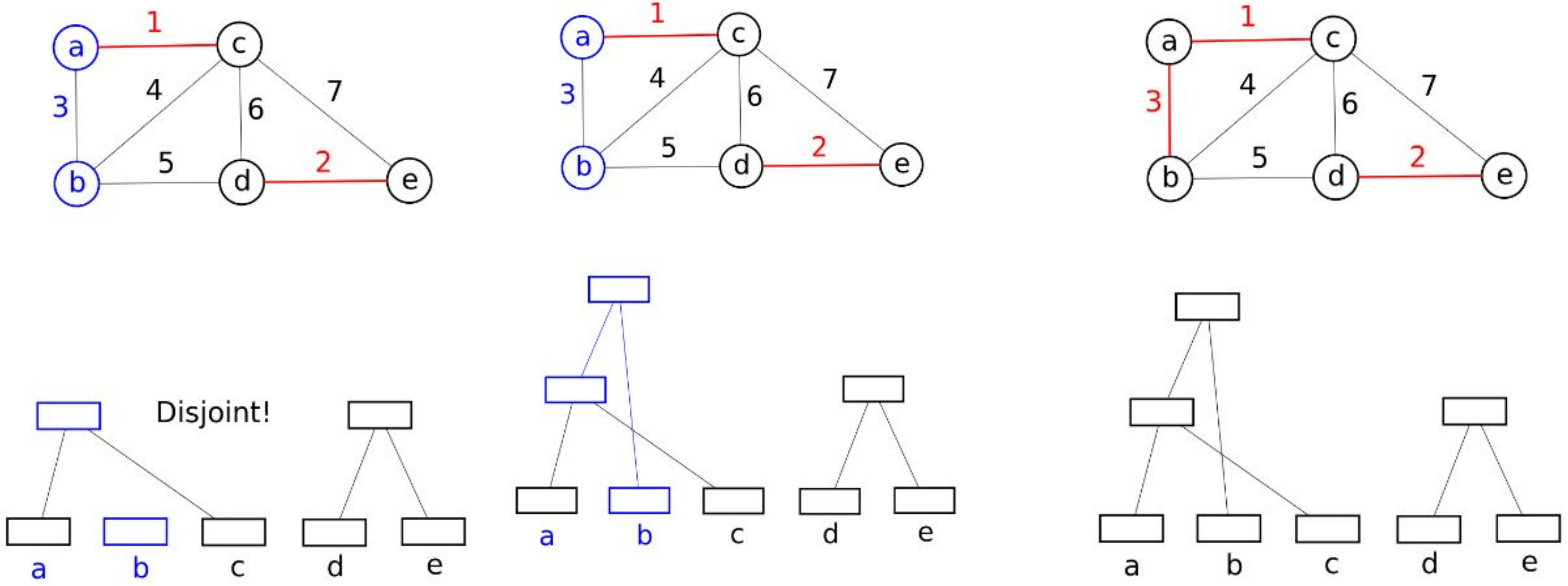
Courtesy : Wikipedia

MST Implementation of Kruskal's Algorithm - Disjoint Sets Array rep.



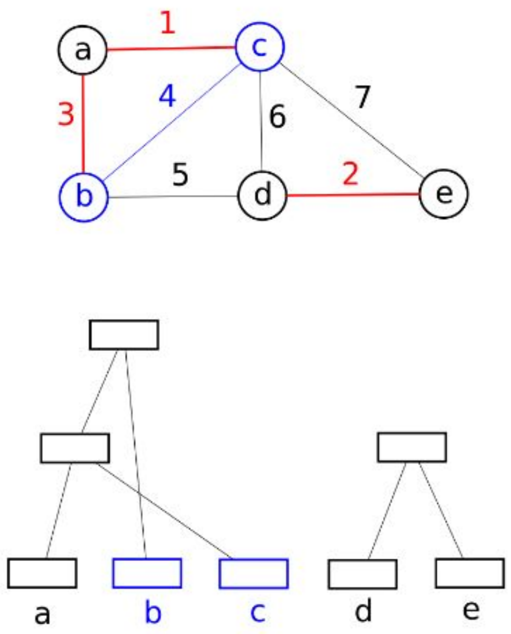
Courtesy : Wikipedia

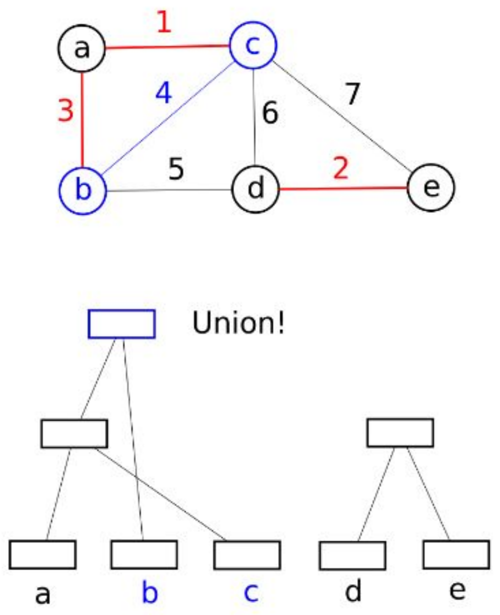
MST Implementation of Kruskal's Algorithm - Disjoint Sets Array rep.

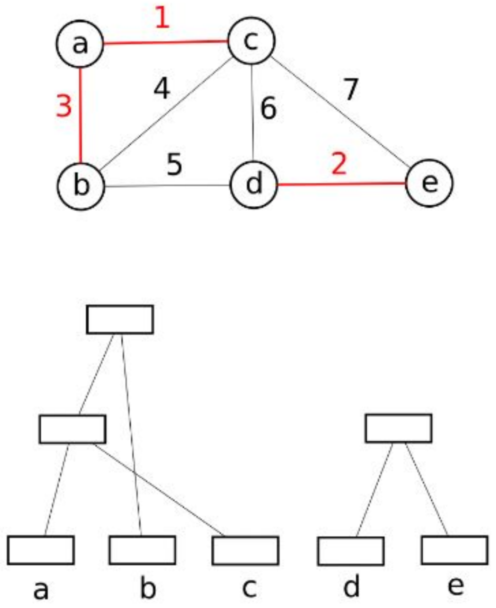


Courtesy : Wikipedia

MST Implementation of Kruskal's Algorithm - Disjoint Sets Array rep.

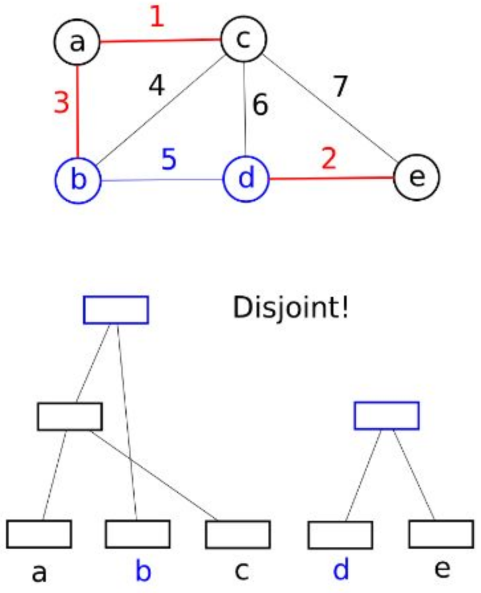


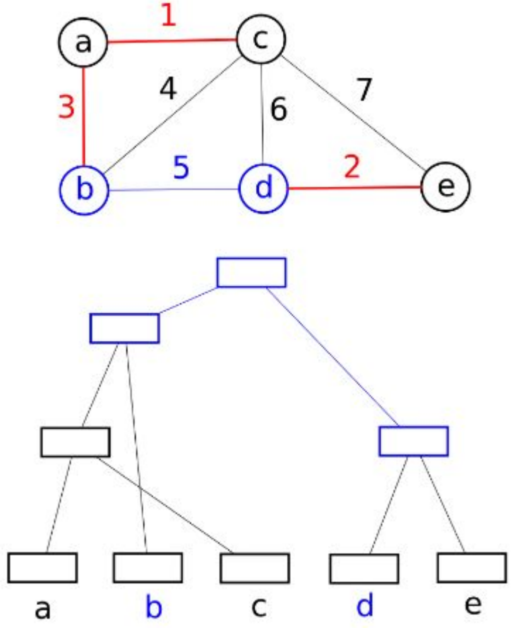


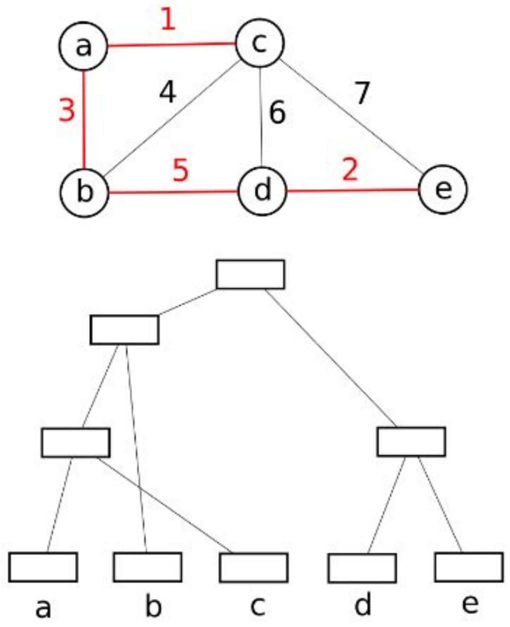


Courtesy : Wikipedia

MST Implementation of Kruskal's Algorithm - Disjoint Sets Array rep.







Courtesy : Wikipedia

Minimum Spanning Tree (MST) - Kruskal's Algorithm Analysis

Time Complexity: O(E * log E) or O(E * log V)

- Sorting of edges takes O(E * logE) time.
- Iterate through all edges and **apply the find-union algorithm** ⇒ at most O(logV) time.
- Overall complexity is O(E * logE + E * logV) time.

Auxiliary Space: O(V + E)

1. **Edge List Storage** → O(E)
2. **Disjoint Set Data Structure (Union-Find)** → O(V)
3. **MST Storage** → O(V)

Topics to be covered

- General method
- Fractional Knapsack problem,
- Job sequencing with deadlines,
- Single source shortest path: Dijkstra Algorithm
- **Minimum cost spanning trees:**
    - Kruskal Algorithm
    - **Prim’s algorithm**

Extra Topics:

- Coin Change Problem
- Disjoint Set Data Structure

Minimum Spanning Tree (MST) - Prim's Algorithm

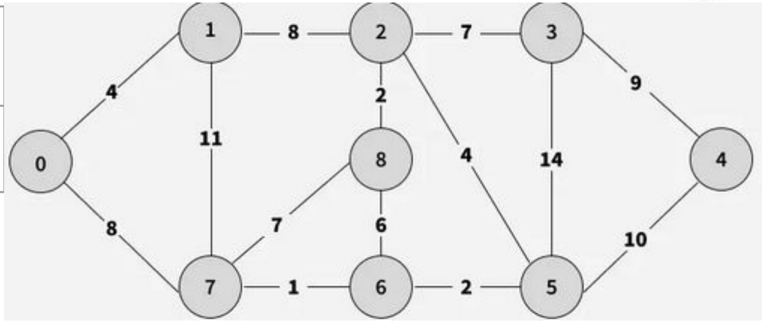
1. **Initialize**: Start with any arbitrary vertex and mark it as part of the MST.
2. **Select the minimum edge**: Among all edges connecting MST vertices to non-MST vertices, pick the edge with the smallest weight.
3. **Include the new vertex**: Add the selected edge and its connected vertex to the MST.
4. **Repeat**: Continue until all vertices are included in the MST.

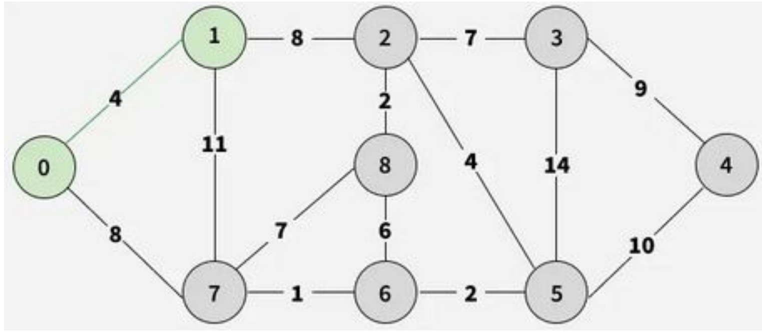
Greedy Choice Property: At every step, pick the **minimum-weight edge** that connects a vertex in the MST to a vertex outside the MST. ensuring a locally optimal choice.

Optimal Substructure Property: **Any subset of an MST is also an MST for the subset of nodes,** meaning the problem can be broken into smaller optimal subproblems.

Minimum Spanning Tree (MST) - Prim's Algorithm Example

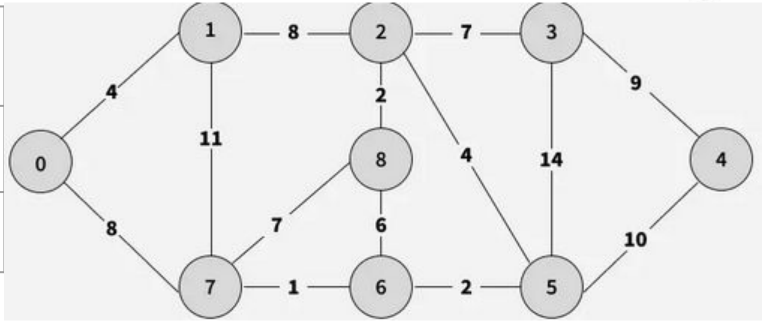
Current vertex	EdgeSet	Selected Edge	VertexSet	Cost
0	{(0,1),(0,7)}	(0,1)	{0,1}	4

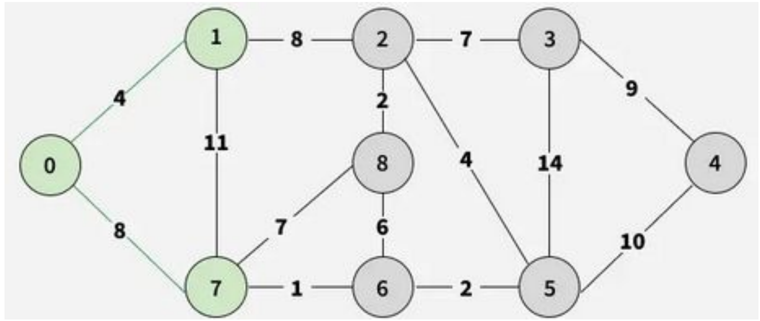




Minimum Spanning Tree (MST) - Prim's Algorithm Example

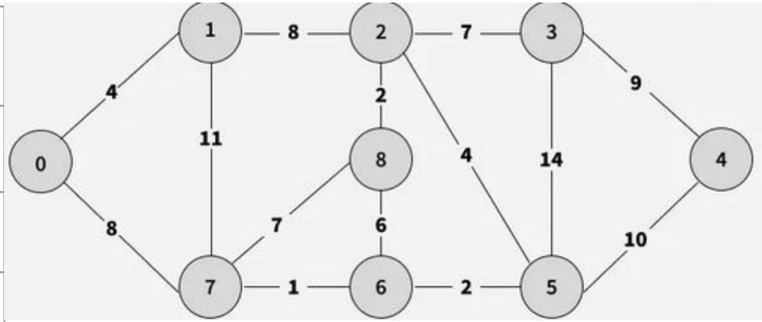
Current vertex	EdgeSet	Selected Edge	VertexSet	Cost
0	{(0,1),(0,7)}	(0,1)	{0,1}	4
1	{(0,7),(1,2),(1,7)}	(0,7)	{0,1, 7}	12

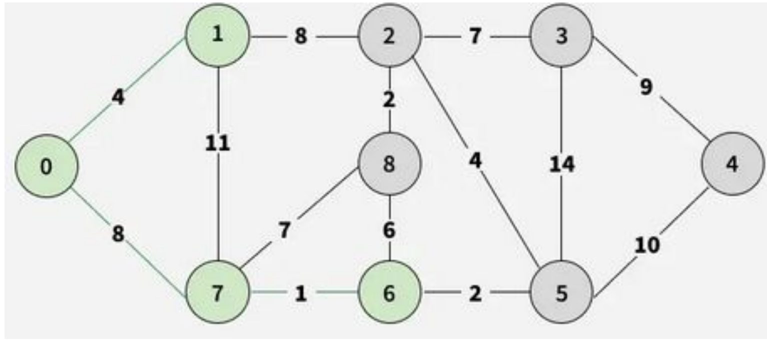




Minimum Spanning Tree (MST) - Prim's Algorithm Example

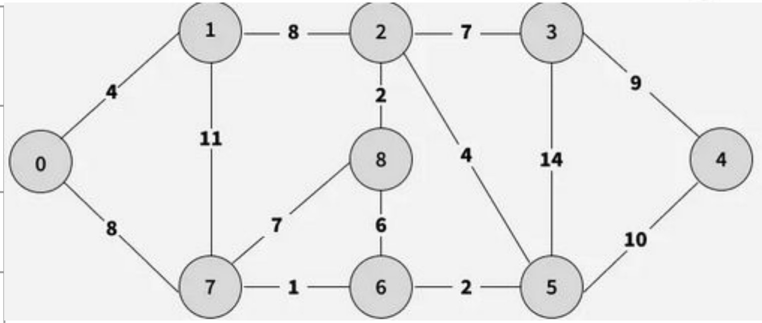
Current vertex	EdgeSet	Selected Edge	VertexSet	Cost
0	{(0,1),(0,7)}	(0,1)	{0,1}	4
1	{(0,7),(1,2),(1,7)}	(0,7)	{0,1, 7}	12
7	{(1,2),(1,7),(7,6),(7,8)}	(7,6)	{0,1,6,7}	13

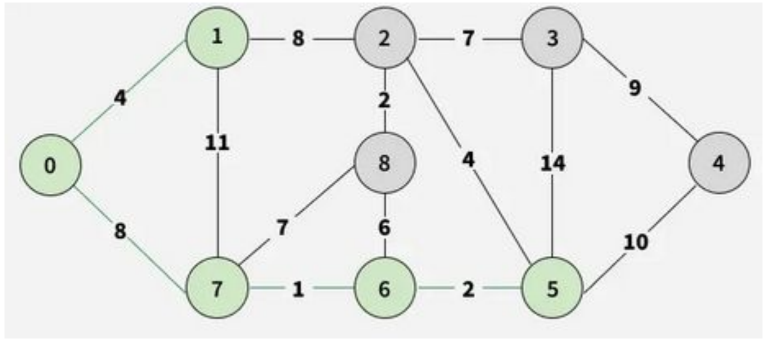




Minimum Spanning Tree (MST) - Prim's Algorithm Example

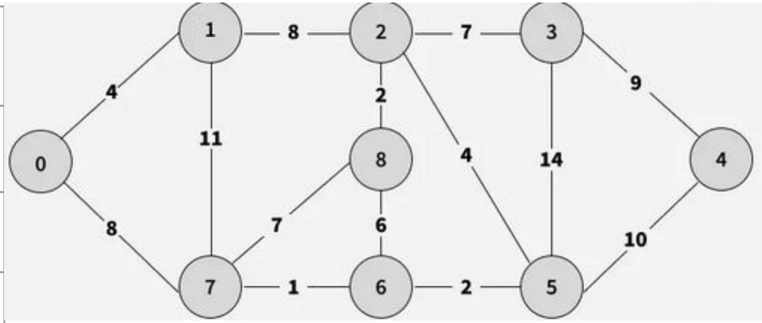
Current vertex	EdgeSet	Selected Edge	VertexSet	Cost
0	{(0,1),(0,7)}	(0,1)	{0,1}	4
1	{(0,7),(1,2),(1,7)}	(0,7)	{0,1, 7}	12
7	{(1,2),(1,7),(7,6),(7,8)}	(7,6)	{0,1,6,7}	13
6	{(1,2),(1,7),(6,5),(6,8),(7,8)}	(6,5)	{0,1,5,6,7}	15

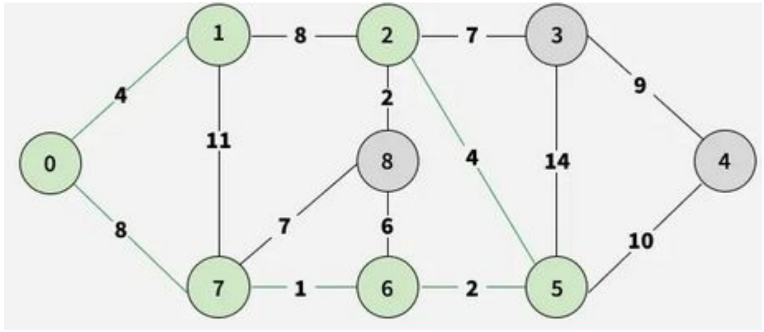




Minimum Spanning Tree (MST) - Prim's Algorithm Example

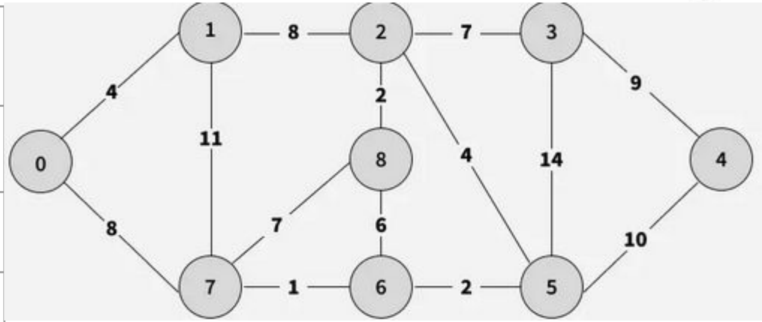
Current vertex	EdgeSet	Selected Edge	VertexSet	Cost
0	{(0,1),(0,7)}	(0,1)	{0,1}	4
1	{(0,7),(1,2),(1,7)}	(0,7)	{0,1, 7}	12
7	{(1,2),(1,7),(7,6),(7,8)}	(7,6)	{0,1,6,7}	13
6	{(1,2),(1,7),(6,5),(6,8),(7,8)}	(6,5)	{0,1,5,6,7}	15
5	{(1,2),(1,7),(5,2),(5,3),(5,4),(6,8),(7,8)}	(5,2)	{0,1,2,5,6,7}	19

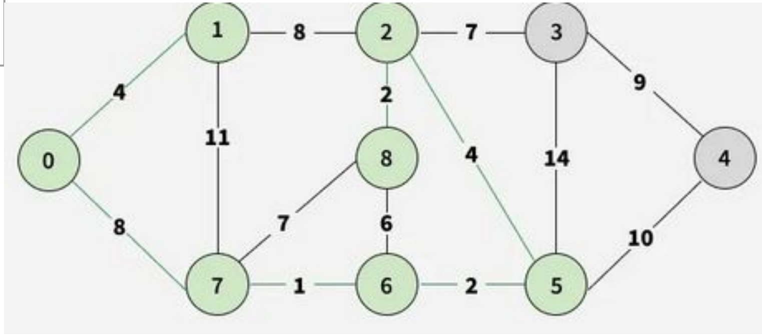




Minimum Spanning Tree (MST) - Prim's Algorithm Example

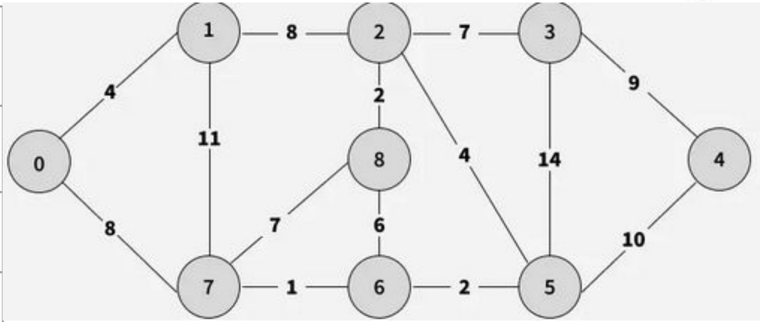
Current vertex	EdgeSet	Selected Edge	VertexSet	Cost
0	{(0,1),(0,7)}	(0,1)	{0,1}	4
1	{(0,7),(1,2),(1,7)}	(0,7)	{0,1, 7}	12
7	{(1,2),(1,7),(7,6),(7,8)}	(7,6)	{0,1,6,7}	13
6	{(1,2),(1,7),(6,5),(6,8),(7,8)}	(6,5)	{0,1,5,6,7}	15
5	{(1,2),(1,7),(5,2),(5,3),(5,4),(6,8),(7,8)}	(5,2)	{0,1,2,5,6,7}	19
2	{(1,2),(1,7),(2,3),(2,8),(5,3),(5,4),(6,8),(7,8)}	(2,8)	{0,1,2,5,6,7,8}	21

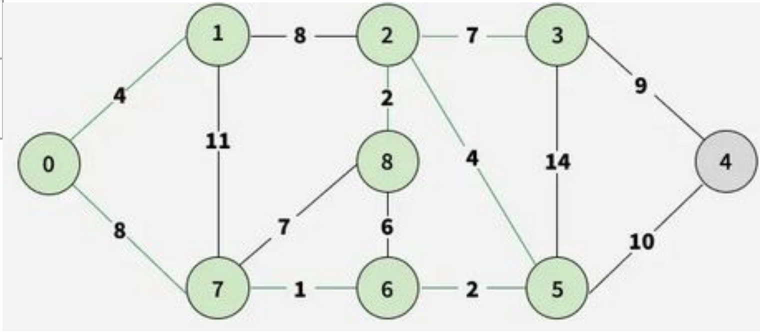




Minimum Spanning Tree (MST) - Prim's Algorithm Example

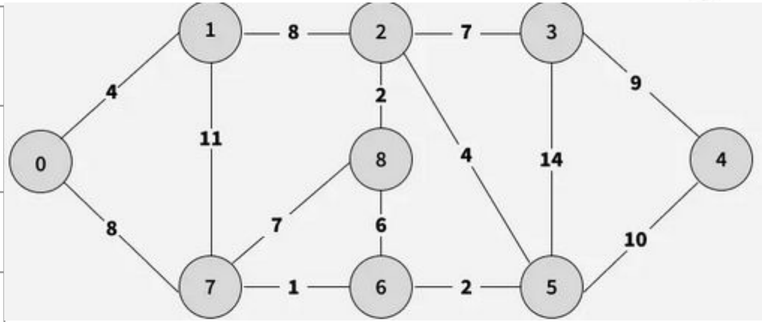
Current vertex	EdgeSet	Selected Edge	VertexSet	Cost
0	{(0,1),(0,7)}	(0,1)	{0,1}	4
1	{(0,7),(1,2),(1,7)}	(0,7)	{0,1, 7}	12
7	{(1,2),(1,7),(7,6),(7,8)}	(7,6)	{0,1,6,7}	13
6	{(1,2),(1,7),(6,5),(6,8),(7,8)}	(6,5)	{0,1,5,6,7}	15
5	{(1,2),(1,7),(5,2),(5,3),(5,4),(6,8),(7,8)}	(5,2)	{0,1,2,5,6,7}	19
2	{(1,2),(1,7),(2,3),(2,8),(5,3),(5,4),(6,8),(7,8)}	(2,8)	{0,1,2,5,6,7,8}	21
2	{(1,2),(1,7),(2,3),(5,3),(5,4),(6,8),(7,8)}	(2,3)	{0,1,2,3,5,6,7,8}	28

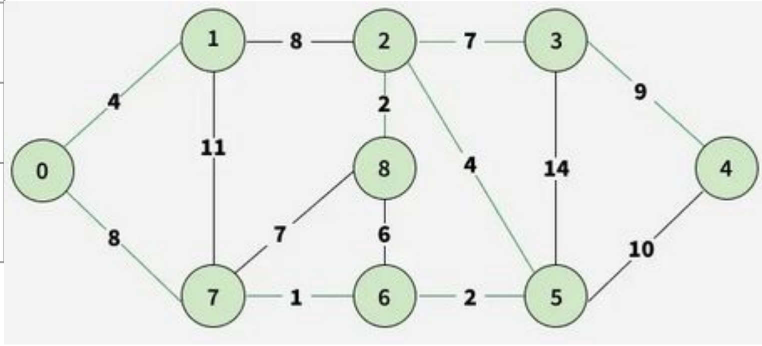




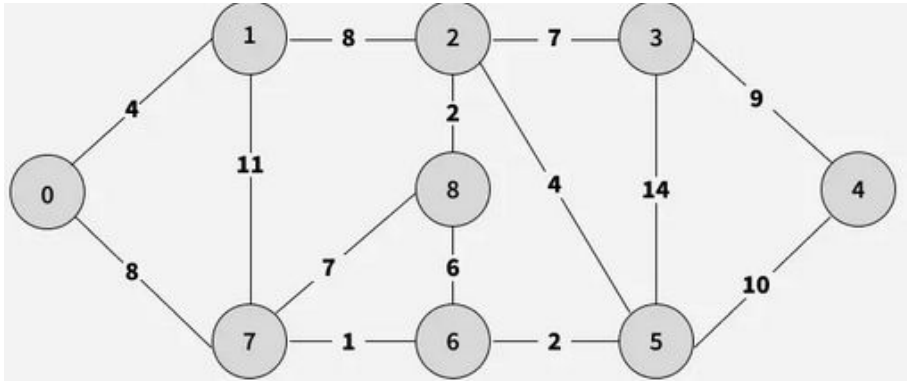
Minimum Spanning Tree (MST) - Prim's Algorithm Example

Current vertex	EdgeSet	Selected Edge	VertexSet	Cost
0	{(0,1),(0,7)}	(0,1)	{0,1}	4
1	{(0,7),(1,2),(1,7)}	(0,7)	{0,1, 7}	12
7	{(1,2),(1,7),(7,6),(7,8)}	(7,6)	{0,1,6,7}	13
6	{(1,2),(1,7),(6,5),(6,8),(7,8)}	(6,5)	{0,1,5,6,7}	15
5	{(1,2),(1,7),(5,2),(5,3),(5,4),(6,8),(7,8)}	(5,2)	{0,1,2,5,6,7}	19
2	{(1,2),(1,7),(2,3),(2,8),(5,3),(5,4),(6,8),(7,8)}	(2,8)	{0,1,2,5,6,7,8}	21
2	{(1,2),(1,7),(2,3),(5,3),(5,4),(6,8),(7,8)}	(2,3)	{0,1,2,3,5,6,7,8}	28
3	{(1,2),(1,7),(3,4),(3,5),(5,3),(5,4),(6,8),(7,8)}	(3,4)	{0,1,2,3,4,5,6,7,8}	37

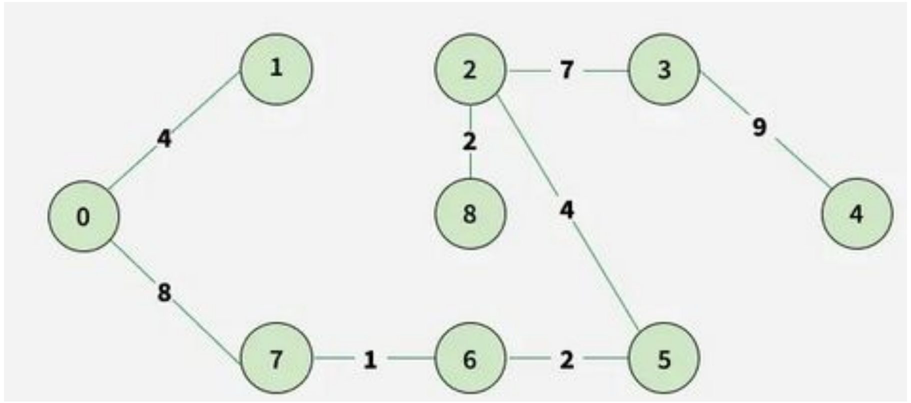


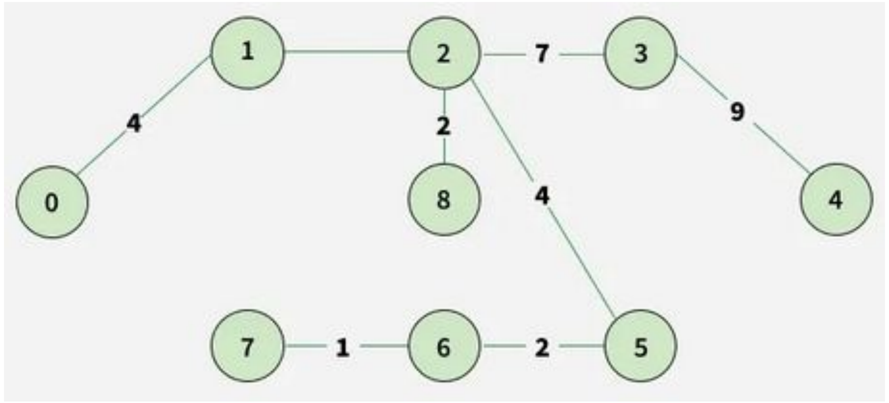


Minimum Spanning Tree (MST) - Prim's Algorithm Example

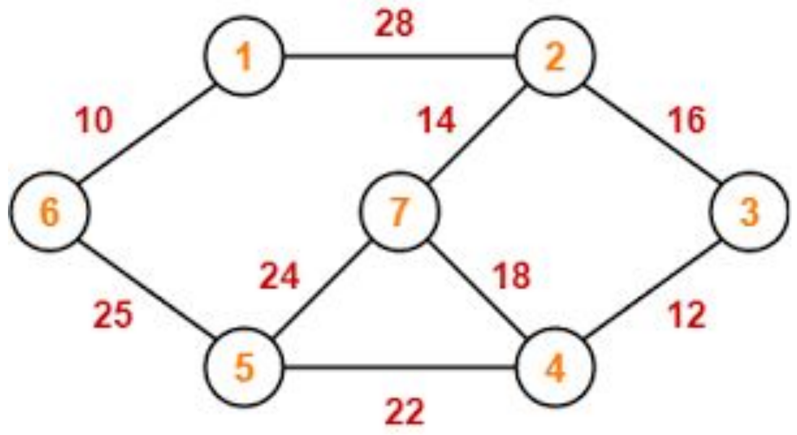


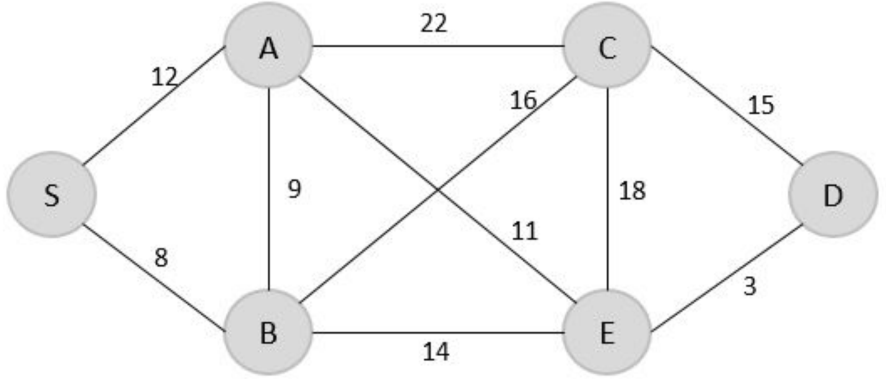
Alternative MST’s (MST = **37**)



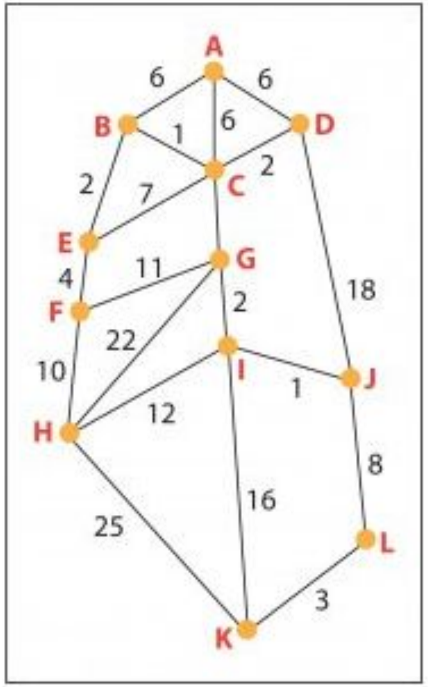


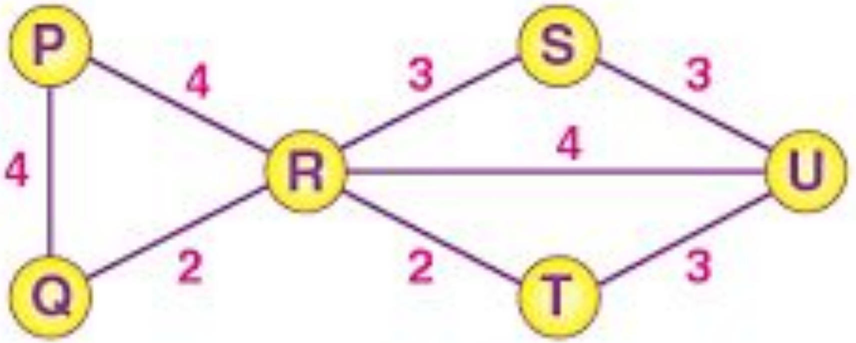
Minimum Spanning Tree (MST) - Prim's Algorithm Problems



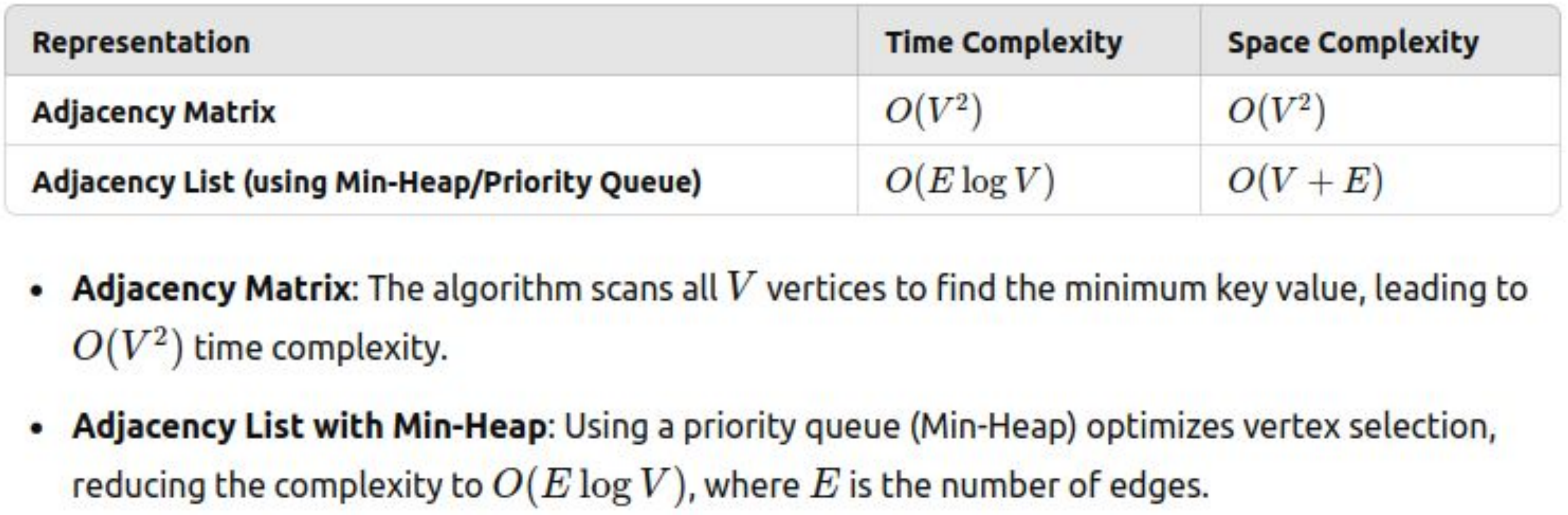


Minimum Spanning Tree (MST) - Prim's Algorithm Problems





Minimum Spanning Tree (MST) - Prim's Algorithm Analysis

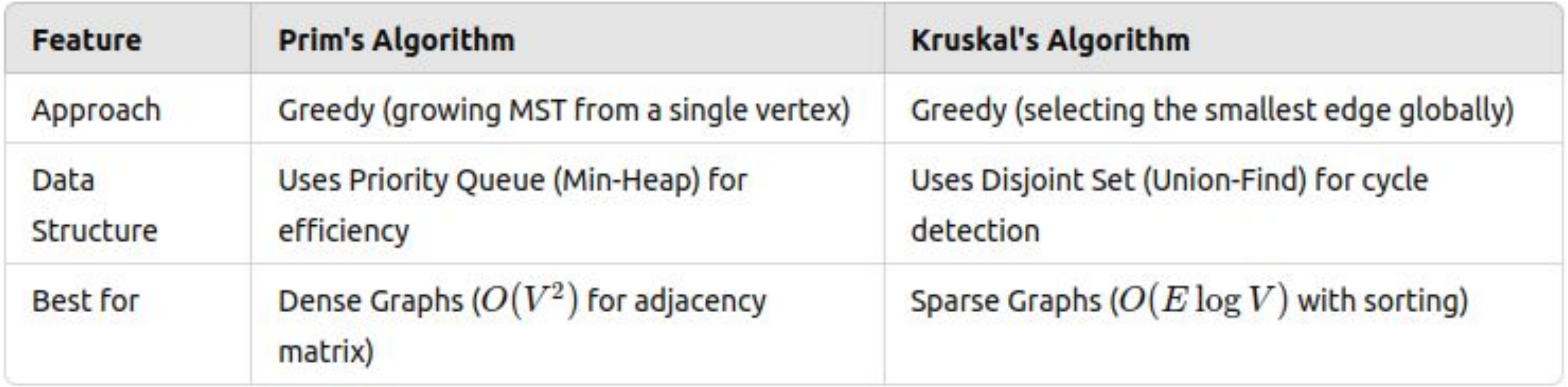


Representation	Time Complexity	Space Complexity
Adjacency Matrix	$O(V^2)$	$O(V^2)$
Adjacency List (using Min-Heap/Priority Queue)	$O(E \log V)$	$O(V + E)$

- **Adjacency Matrix**: The algorithm scans all V vertices to find the minimum key value, leading to $O(V^2)$ time complexity.
- **Adjacency List with Min-Heap**: Using a priority queue (Min-Heap) optimizes vertex selection, reducing the complexity to $O(E \log V)$, where E is the number of edges.

Courtesy : Prof. Galles - Kruskal's Algo

Prim's Algorithm Vs Kruskal’s Algorithm



Feature	Prim's Algorithm	Kruskal's Algorithm
Approach	Greedy (growing MST from a single vertex)	Greedy (selecting the smallest edge globally)
Data Structure	Uses Priority Queue (Min-Heap) for efficiency	Uses Disjoint Set (Union-Find) for cycle detection
Best for	Dense Graphs ($O(V^2)$ for adjacency matrix)	Sparse Graphs ($O(E \log V)$ with sorting)