GREEDY METHOD

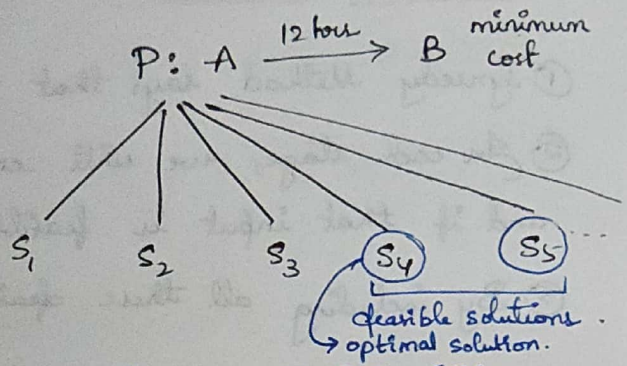
* This method is used for solving optimization problems.
* Optimization problems are those that require either minimum result or maximum result.

Explanation :-

Suppose there is a problem to travel from location A to location B.

You can travel by

walk (S1)
bike (S_2)
car (S_3)
train (S_4)
flight (S_5)
⋮



If you have to travel within 12 hours, then S4 & S5 are feasible solutions.

A solution which is satisfying the condition given in the problem is Feasible solution, though, for a problem, there can be many solutions.

For example, we want journey to complete in minimum cost, then it is minimization problem.

If a problem demands the result should be minimum, then it is minimization problem.

S4 (travel by train) is taking minimum cost. So S4 is optimal solution.

A solution which is both feasible and minimized, ~~then~~ is an Optimal Solution.

So, for any problem, there can be only one Optimal Solution.

Hence, a problem which requires either minimum solution or maximum solution is called as OPTIMIZATION PROBLEM.

Strategies for solving Optimization Problems :-

1. Greedy Method
2. Dynamic Programming
3. Branch and Bound.

GREEDY METHOD :-

1. Greedy Method says that a problem should be solved in stages.
2. In each stage, we will consider one input from a given problem and if that input is feasible, then we will include it in the solution.
3. By including all these feasible inputs, we will get an optimal solution.

General algorithm :-

```
Algorithm Greedy(a, n) {
    for j=1 to n do
        x = Select(a); // selects input everytime from the given problem.
        if Feasible(x) then // if input x is feasible
            Solution = Solution + x; // then include x in solution.
    }
}
```

n = 5

a	a_1	a_2	a_3	a_4	a_5
	1	2	3	4	5

Real-time Example :-

1. Problem: Buying a Car.

1st Approach: Check all the brands and models of cars in your city and buy one.

2nd Approach: You approach greedily. You have your own method of choosing a car ⇒ get the top models and then get well-tested car.

You don't have to check each and every car available in the market

② Problem: Interview.

Selection Procedure is the greedy approach.

Instead of conducting written test, GD, TR, HR for all people, we adopt a procedure where we filter candidates. This is called as greedy approach.

When you follow known methods of solving a problem and you quickly solve it, then that approach is called greedy.

1. KNAPSACK PROBLEM - GREEDY METHOD

* $n = 7$ → no. of objects

$m = 15$ → Bag/Knapsack Capacity is 15

Objects O :	1	2	3	4	5	6	7	
Profits P :	10	5	15	7	6	18	3	→ Profit associated with each object
Weights w :	2	3	5	7	1	4	1	→ weight of object.

Find maximum profit that can be earned in 15 kg knapsack after filling the objects in it.

If the total weights is equal to the capacity of container, then it is well and good.

But if total weight is larger than capacity of container, we need to fill only few goods that don't exceed capacity but earn the max profit possible.

Optimization Problem:- (Maximization Problem) Fill the bag/container in such a way that the profit is maximized.

Hence, we used greedy method.

Constraints:- only ≤ 15 kg.

Solution:- The weights & objects can be divisible.

For ex, object 5 has 4 as weight. We can take only 1, 2, 3 or 4 kgs (like potatoes).

indivisible objects are like 4 kg fridge, we cannot take 1, 2, 3, ~~or~~ 4 kg from it.

So the fraction can be $0 \leq x \leq 1$.

We need to fill the bag with objects that are highest profit by weight.

P:	10	5	15	7	6	18	3
w:	2	3	5	7	1	4	1

18 profit for 4 kgs.

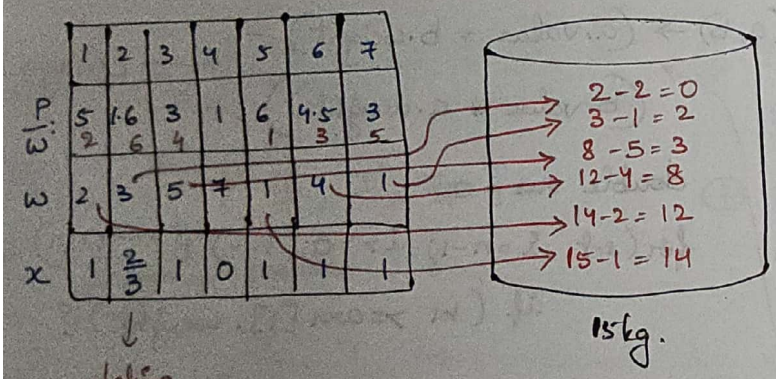
$\frac{18}{4}$ profit for 1 kg

$= 4.5$ for 1 kg.

So take $\frac{P}{w}$ ratio of each object

O:	1	2	3	4	5	6	7
(per kg profit) $\frac{P}{w}$:	5	1.6	3	1	6	4.5	3

Greedy method asks you to select a method by which you fill the bag. Then, asks you to fill the bag one-by-one.



↓ taking only 2kg as capacity of bag has only 2kg left.

Weights taken

$x \times w = 1\times2 + \frac{2}{3}\times3 + 1\times5 + 0\times7 + 1\times1 + 4\times1 + 1\times1$

(or) $\sum x_i w_i$

$= 2+2+5+0+1+4+1$

$= 15$.

Profits :-

$x \times P = 1\times10 + \frac{2}{3}\times5 + 1\times15 + 0\times7 + 1\times6 + 1\times18 + 1\times3$

(or) $\sum x_i P_i$

$= 10 + 3.33 + 15 + 0 + 6 + 18 + 3$

$= 55.33$ (Max Profit).

Constraint :-

$\sum x_i w_i \leq n$ ✓✓ Total weight shouldn't exceed n is satisfied.

Objective :-

$\sum x_i p_i$ should be maximum

There is also 0/1 Knapsack problem → it has indivisible objects.

Program :-

```
fractionalKnapsack(int N, int W, int[] values, int[] weights) {
    double[] profitPerWeight =
        computeProfitPerWeight(values, weights);
```

```
double fractionalKnapsack(int W, Item arr[], int n) {
    Arrays.sort(arr, (a, b) -> (a.value * b.weight) -
        (b.value * a.weight));                          // O(n log n)
    double res = 0;
    for (int i = n-1; i >= 0; i--) {                    // O(n)
        if (W >= arr[i].weight) {
            W -= arr[i].weight;
            res += arr[i].value;
        } else {
            res += ((double) W / (double) arr[i].weight) * (double) arr[i].value;
            W = 0;
            break;
        }
    }
    return res;
}
```

$\frac{a.value}{a.weight} > \frac{b.value}{b.weight}$

$\Rightarrow \frac{a.value}{a.weight} - \frac{b.value}{b.weight} > 0$

$\Rightarrow a.value * b.weight - b.value * a.weight > 0.$

```
computeProfitPerWeight(int[] values, int[] weights) {
    double[] vals = new double[values.length];
    for (int i=0; i < values.length; i++) {
        vals[i] = (double)(values[i]) / (double)(weights[i]);
    }
    return vals;
}
```

TC :- $O(n \log n + n)$

2. JOB SEQUENCING WITH DEADLINES - GREEDY METHOD

$n = 5$

Jobs	J_1	J_2	J_3	J_4	J_5
Profits	20	15	10	5	1 → notice the sorted order of profits
Deadlines	2	2	1	3	3

Assumption :- There is one machine and each job needs 1 unit of time. Find the sequence of jobs that needs to be finished in their deadlines such that the profit is maximised.

This is a maximisation problem that can be solved by greedy method.

Explanation :-

Say, we have 3 slots (as the maximum hours in deadlines is 3).

0 $\underline{J_2}$ 1 $\underline{J_1}$ 2 $\underline{J_4}$ 3.
9am 10am 11am 12pm.

$\{ J_2, J_1, J_4 \}$

Sequences possible →

J_2, J_1, J_4

J_1, J_2, J_4

as J_1 & J_2 can wait till 2

Max profit = $J_2 + J_1 + J_4$

= 15 + 20 + 5

= 35 + 5

= 40.

① J_1 can wait till 2.
So put J_1 in 1-2

② J_2 can wait till 2.
But J_1 is in 1-2
So put J_2 in 0-1

③ J_3 can wait till 1.
But J_2 has max profit than J_3 and already occupied 0-1

④ J_4 can wait till 3
J_4 profit > J_5 profit
put J_4 in 2-3.

Solution :-

Job Consider	Slot assign	Solution	Profit
—	—	$\emptyset$	0
J1	[1,2]	J_1	20
J2	[0,1] [1,2]	J_1, J_2	20+15
J3 ×	[0,1] [1,2]	J_1, J_2	20+15
J4	[0,1] [1,2] [2,3]	J_1, J_2, J_4	20+15+5
J5 ×	[0,1] [1,2] [2,3]	J_1, J_2, J_4	40

n = 7

Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profits	35	30	25	20	15	12	5 → sorted
Deadlines	3	4	4	2	3	1	2

0 $\underline{J_4}$ 1 $\underline{J_3}$ 2 $\underline{J_1}$ 3 $\underline{J_2}$ 4.
20 25 35 30 = 110

```
class Job {
    int id, profit, deadline;
    Job(int x, int y, int z) {
        this.id = x;
        this.deadline = y;
        this.profit = z;
    }
}
```

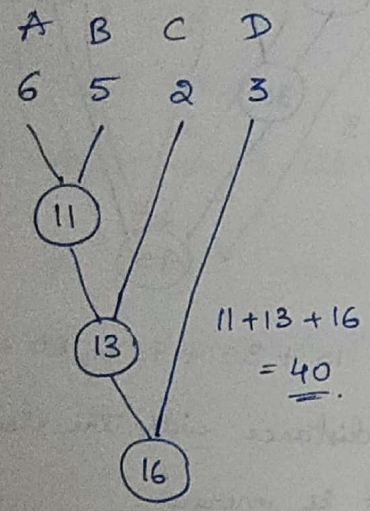
Program :-

```
//return no of Jobs and maximum profit.
int[] jobScheduling(Job arr[], int n) {
    Arrays.sort(arr, (a,b) -> (b.profit - a.profit));
    int maxDeadline = 0
    for(int i=0; i<n; i++) {
        maxDeadline = Math.max(arr[i].deadline, maxDeadline);
    }
    boolean[] slotBooked = new boolean[maxDeadline+1];
    int totalProfit = 0, noOfJobs = 0;
    for(int i=0; i<n; i++) {
        for(int j=arr[i].deadline; j>0; j--) {
            if(!slotBooked[j]) {
                slotBooked[j] = true;
                totalProfit += arr[i].profit;
                noOfJobs++;
                break;
            }
        }
```

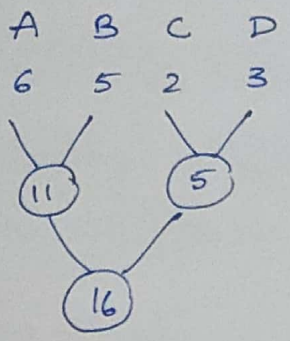
3. OPTIMAL MERGE PATTERN - GREEDY METHOD

If you can solve this problem using greedy method, you can also solve Huffman coding.

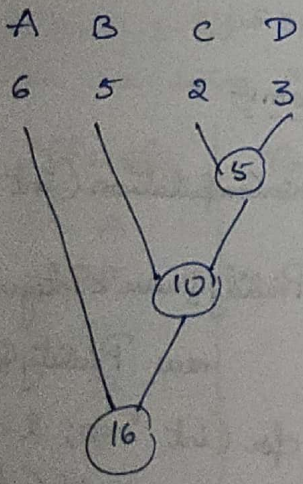
A B C D
6 5 2 3



11+13+16
= 40.



11+5+16 = 32.



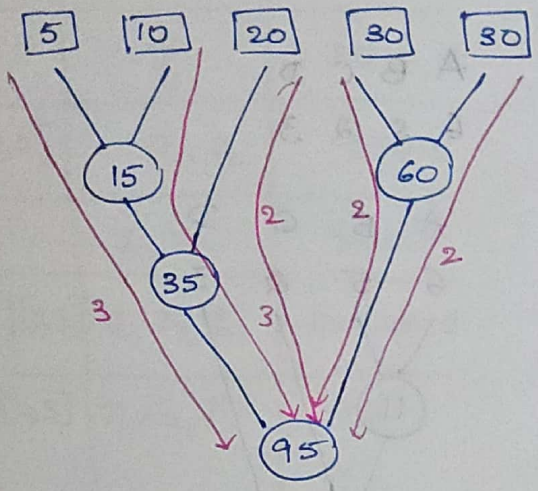
16+10+5 = 31.

The observation here is merge pattern will be more optimal if we merge a pair of small size lists. This is the greedy method we should follow.

Example :-

Files →	x_1	x_2	x_3	x_4	x_5
Sizes →	20	30	10	5	30

Increasing order of sizes →



Total cost : 15 + 35 + 95 + 60 = 205.

Find total cost by multiplying distance with the element. This says how many times the element is merged.

$5 \times 3 + 10 \times 3 + 20 \times 2 + 30 \times 2 + 30 \times 2$

$= 15 + 30 + 40 + 60 + 60$

$= 205.$

$\sum d_i * x_i$

Program :-

```
int minComputation (int n, int[] arr){
    PriorityQueue<Integer> pq =
        new PriorityQueue<>();
    for (int i=0; i<n; i++) {
        pq.add(arr[i]);
    }
    int count = 0;
n —— while (pq.size() > 1) {
        int temp = pq.poll() + pq.poll();
        count += temp;
logn —— pq.add(temp);
    }
    return count;
}
```

O(nlogn).

×

4. HUFFMAN CODING - GREEDY METHOD

Huffman coding is a compression technique. It is used for reducing the size of data or message.

To store files - compressed data.

To transfer data - compressed data

Message -> BCCABBDDAECCBBAEDDCC.

length = 20.

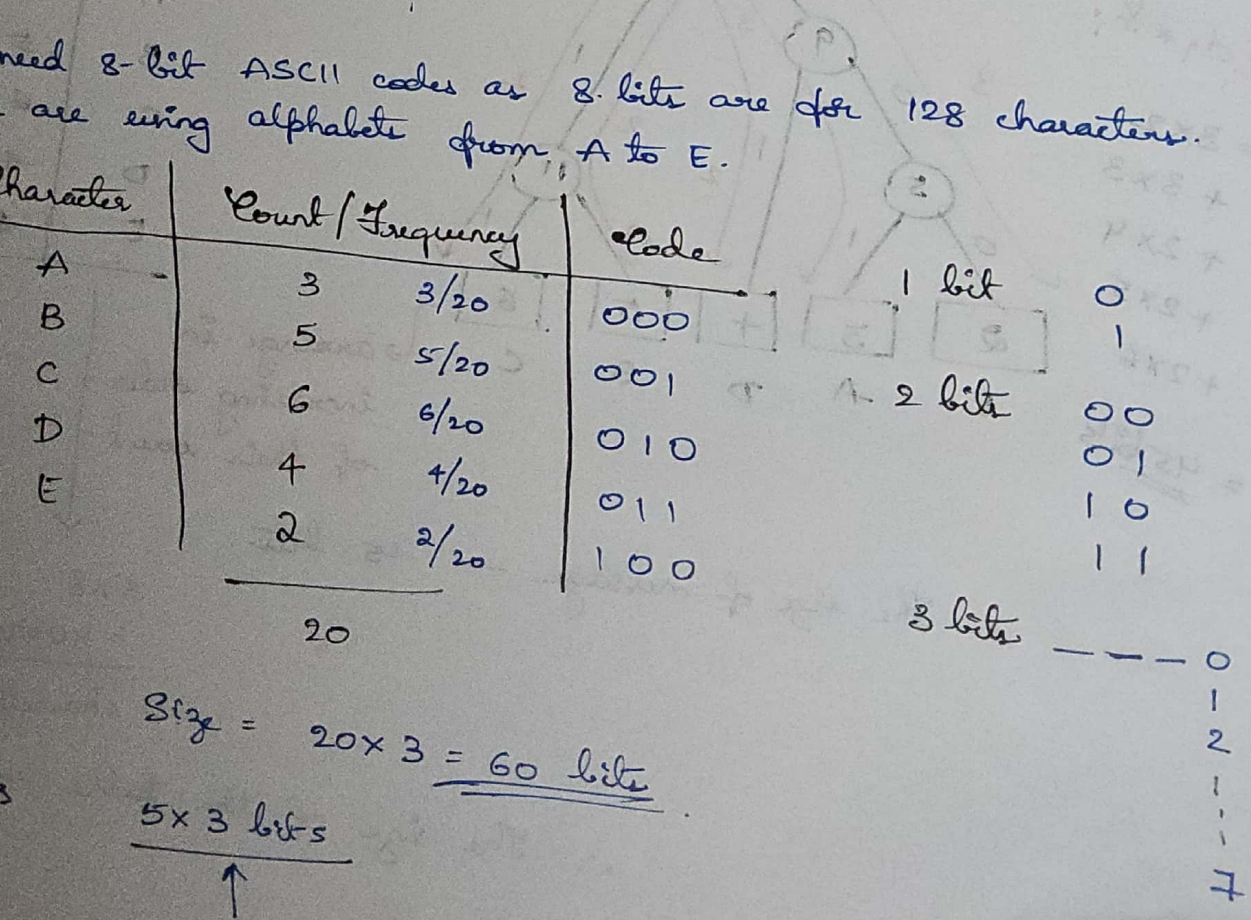
ASCII — 8-bit.

A	65	01000001
B	66	01000010
C	67	⋮
D	68	
E	69	

Size = 8 × 20 = 160 bits.

We do not need 8-bit ASCII codes as 8 bits are for 128 characters. But here we are using alphabets from A to E.

Character	Count / Frequency		Code
A	3	3/20	000
B	5	5/20	001
C	6	6/20	010
D	4	4/20	011
E	2	2/20	100
	20		



Size = 20 × 3 = 60 bits.

5 × 8 bits → characters

5 × 3 bits → codes

We need to send the message as well as the table for encoding.

Therefore total size = 60 + 55 = 115 bits

Instead of passing a 20 character message of size 160 bits, we can use above approach to reduce size to 115 bits.

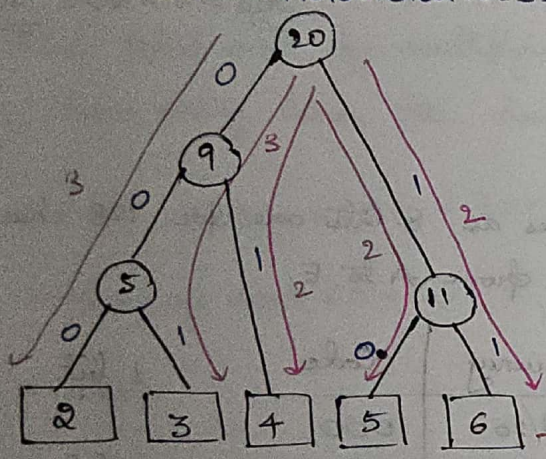
Huffman Coding

Instead of using fixed size, use variable size.

If more repetitive characters are there, they can be coded with small size code.

Message → BCCABBDDAECCBBAEDDCC

10 11 11 001 10 10 01 01 001 - - - - -



$\sum d_i * f_i$

= 3×2 + 3×3 + 2×4 + 2×5 + 2×6

= 45 bits

char	count	code	
A	3	001	3×3=9
B	5	10	2×5=10
C	6	11	2×6=12
D	4	01	2×4=8
E	2	000	3×2=6

45 bits

→ arrange in increasing order of their count.

∴ The size of message — 45 bits.

5×8 = 40 bits / characters + 12 bits / code

∴ The size of the table — 52 bits.

45 + 52 = 97 bits.

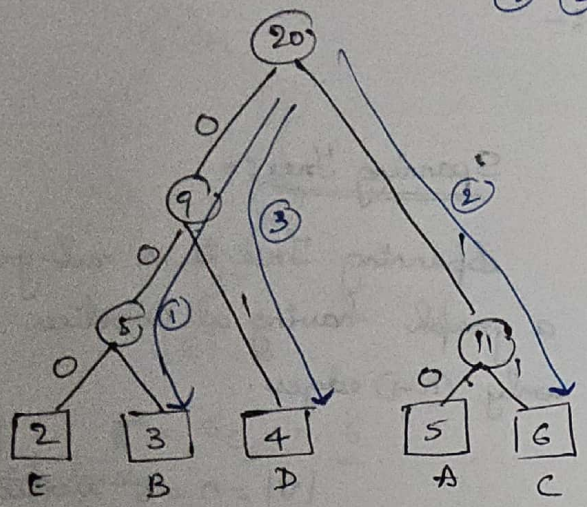
Decoding :-

B C C D A C C B D A B C C D E A A

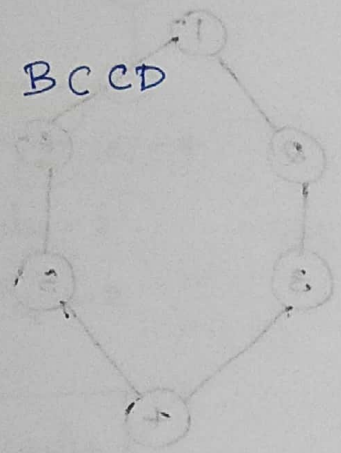
001 11 11 01 10 11 11 001 01 10 001 11 11 01 ...

① ② ③

E D A

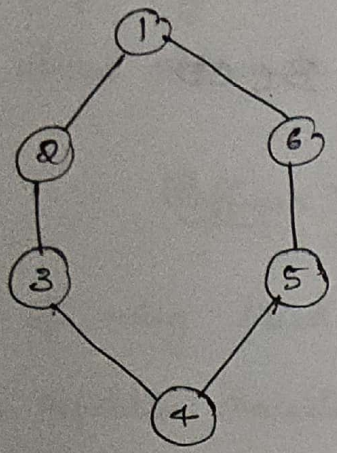


BCCD



5. PRIMS AND KRUSKALS ALGORITHMS

Minimum Cost Spanning Tree :-



$G = (V, E)$

$V = \{1, 2, 3, 4, 5, 6\}$

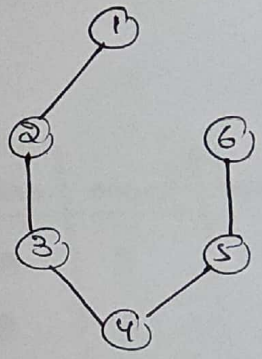
$E = \{(1,2), (2,3), (3,4), \ldots\}$

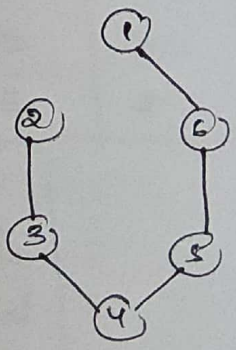
Spanning Tree :-

Spanning Tree is a sub-graph of a graph having all vertices but only $(n-1)$ edges.

$|V| = n = 6$ vertices

$|V| - 1 = 5$ edges.





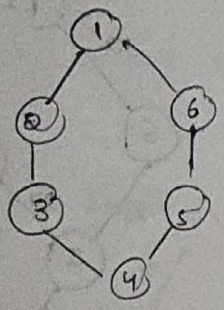
$S \subseteq G$

$S = (V', E')$

$V' = V$

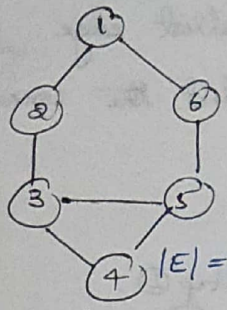
$|E| = |V| - 1.$

How many ways to generate Spanning Trees?



$|E| = 6$

no. of spanning trees $= {}^{6}C_{5} = 6$.



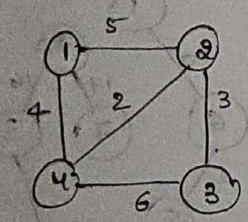
$|E| = 7$

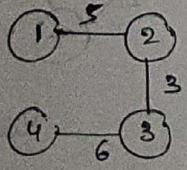
cycles = 2.

${}^{7}C_{5} = 2$

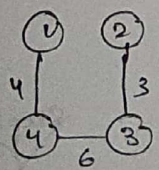
$$|E| C_{|V|-1} - \text{no. of cycles}$$

Weighted graph:-

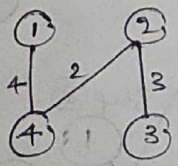




cost = 14



cost = 13



cost = 9.

There can be different spanning trees possible and the costs can be different. We need to find spanning tree of minimum cost known as Minimum Cost Spanning Tree.

The method to select MCST is to check all spanning trees and get the minimum cost one. But the spanning trees can be too lengthy. So, there are greedy methods available to find MCST for a weighted graph.

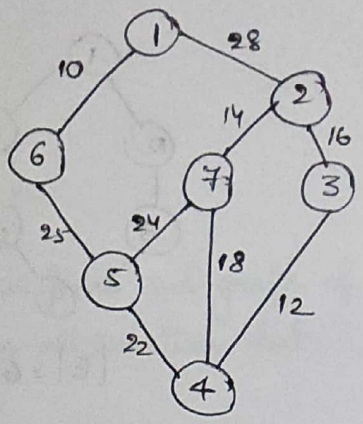
1. Prim's algorithm
2. Kruskal's algorithm

Prim's Algorithm

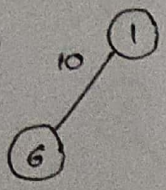
This follows greedy approach.

① Select the smallest edge

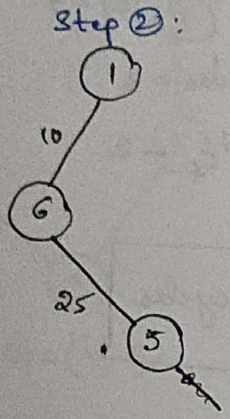
② Then always select the connected smallest edge.



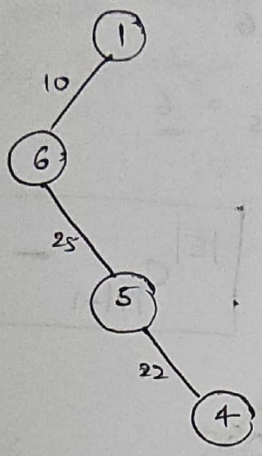
Step ①:



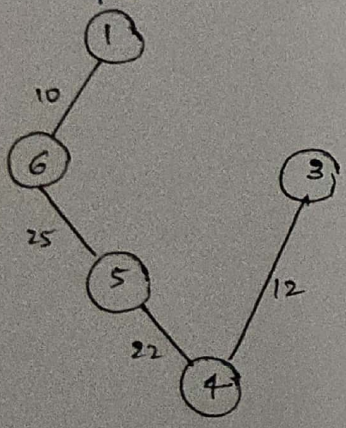
Step ②:



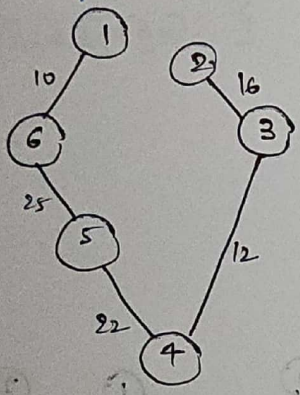
Step ③:



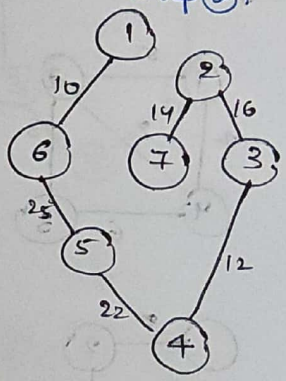
Step ④:



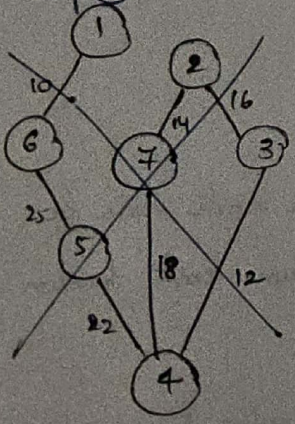
Step ⑤:



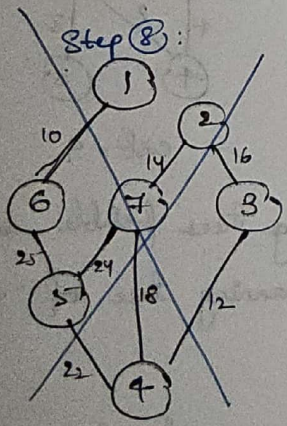
Step ⑥:



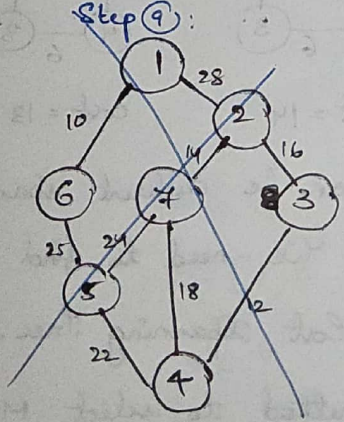
Step ⑦:



Step ⑧:

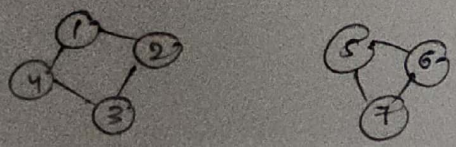


Step ⑨:



We should stop at step ⑥. Because, the algorithm should stop when all nodes are visited.

For unconnected graph, Prim's cannot find MST. It can only find for one component of unconnected graph as the smallest edge is chosen if connected only.

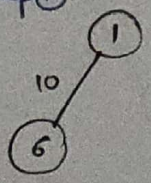


Kruskal's Algorithm

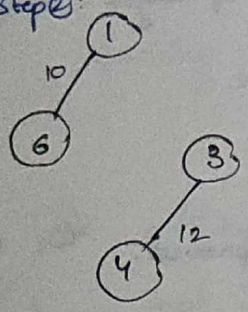
This follows Greedy approach.

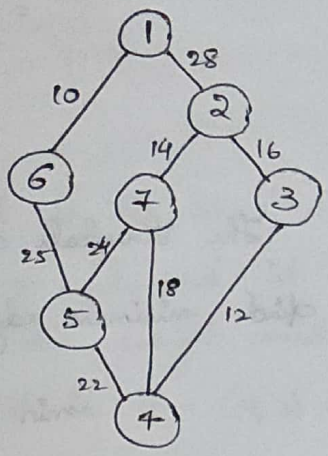
① Always select the smallest edge.

Step①:

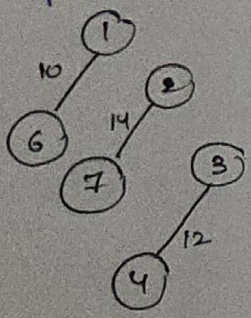


Step②:

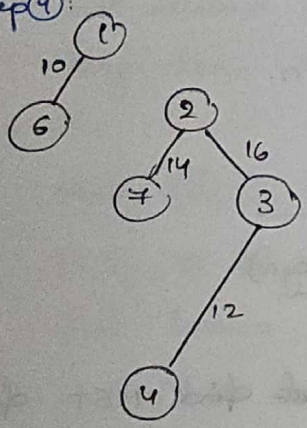




Step③:

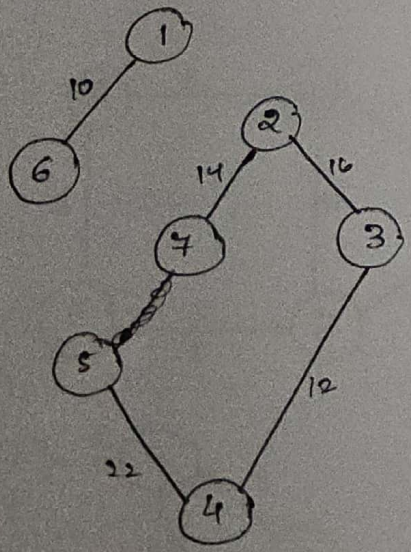


Step④:



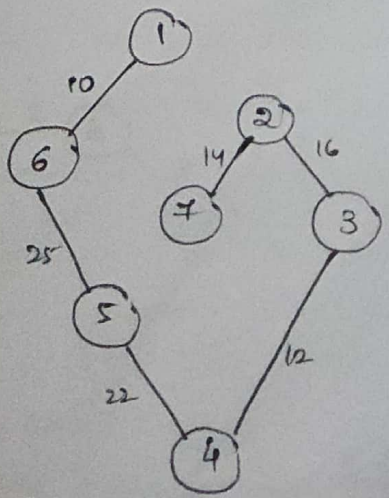
Next edge is (7) —18— (4) but including this forms a cycle. Kruskal allows not to form a cycle.

Step⑤:



Next edge is (7) —24— (5). Don't include as it forms a cycle.

Step⑥:



Cost = 99.

It always selects minimum edges. So, it selects totally |V|-1 edges.

$\Theta(|V||E|)$. → for each vertex, it selects min edge.

$\Theta(n.e)$ — $n \rightarrow$ no. of vertices.

$= \underline{\Theta(n^2)}$. ← If $n == e$ — $e \rightarrow$ no. of edges.

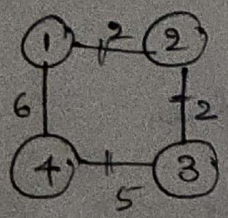
The Kruskal's algorithm can be improved. We can use "Min Heap" to find minimum edge.

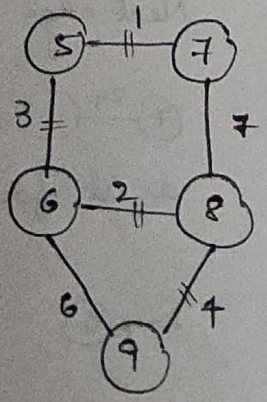
∴ min heap ⇒ $\log n$.

no. of vertices ⇒ n

$\underline{\Theta(n \log n)}$

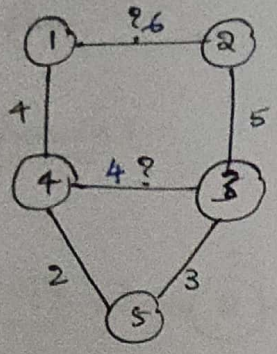
The Kruskal's algorithm ~~cannot~~ can find MST for entire graph if unconnected but can find MST for each component.





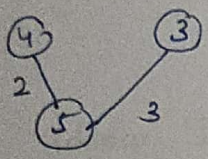
Missing Edges

8) Given a graph
What could be the minimum possible value for the missing edges?



If we hide the unknown edges, it makes the spanning tree of a graph.

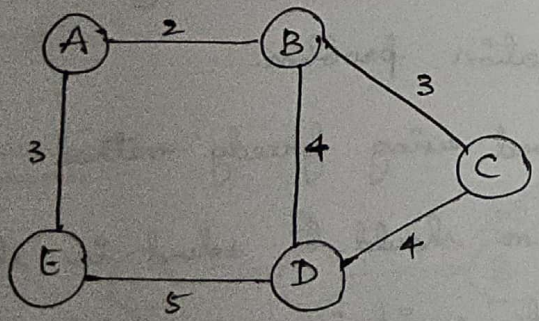
5).



(4,5), (5,3) are included in the ST. (4,3) not included as it forms a cycle and it is not selected first because it's weight could be larger than 2 & 3. So it is 4 (4,3) ⇒ 4.

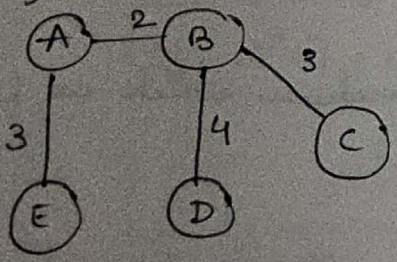
Next, (1,4), (3,2) are taken with weights 4, 5.
(1,2) if taken forms a cycle and it is not taken first because it's value could be greater than 4,5.
∴ (1,2) ⇒ 6.

Example



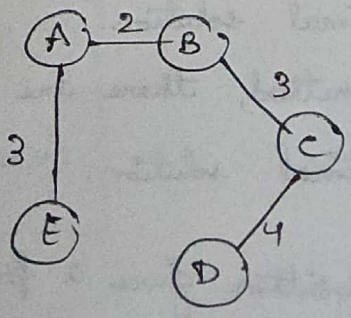
Find MST.

Sol).



5 vertices
4 edges

2+3+3+4 = 12.



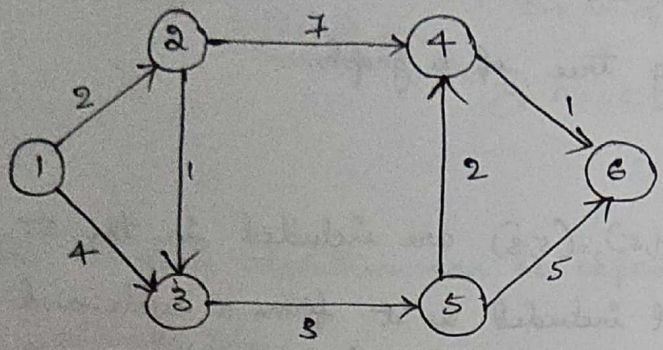
2+3+3+4 = 12.

There can be two MSTs possible but there is only one solution 12.

6. DIJKSTA ALGORITHM -

SINGLE SOURCE SHORTEST PATH -

GREEDY METHOD



If a weighted graph is given, then we have to find shortest path from some starting vertex to all other vertices.

Suppose, we select source as ①, then we will find shortest paths to all other vertices whether by direct path or via other vertices.

We can select any other vertex as source vertex.

If we have to find shortest path, it is a minimization problem. A minimization problem is an optimization problem.

Optimization problems can be solved using greedy method.

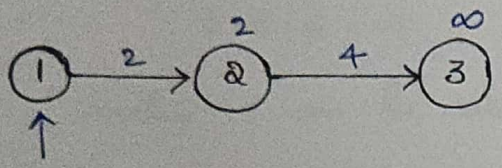
Greedy method says that a problem should be solved in stages by taking one step at a time and considering one input at a time, to get the optimal solution.

In greedy method, there are pre-defined procedures which we follow to get the optimal solution.

Dijkstra Algorithm gives a procedure to get optimal solution i.e., shortest path.

Dijkstra Algorithm can work on directed as well as undirected graphs.

Approach :-

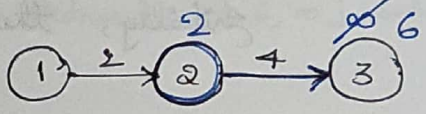


From ①, there is direct path to ②, so distance is 2.

From ① to ③, there is no direct path, so distance is ∞.

If we follow Dijkstra's algorithm, we will be selecting the shortest path vertex which is ②.

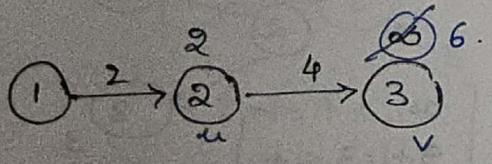
Once you select, we will check if any shortest path from that vertex.



There is shortest path from vertex ② and the '∞' can be changed to '6' (2+4) which means there is a shortest path from ① → ③ as 6.

This process of updating the shortest path to other vertices is called "relaxation".

Relaxation :-



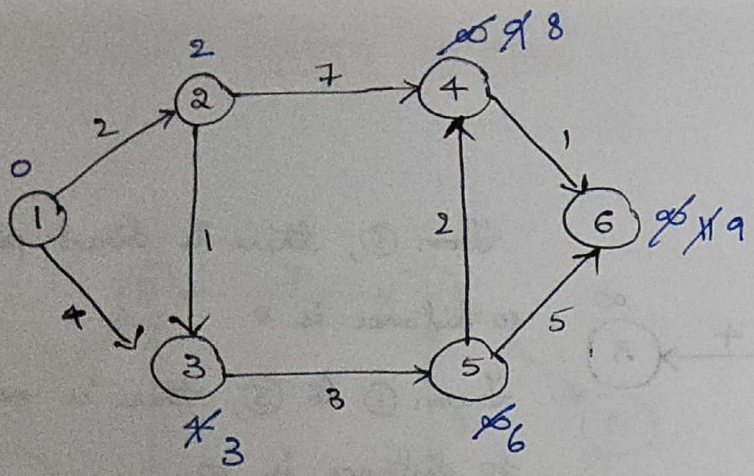
$2+4 < \infty$.

$6 < \infty$

```
if( d[u] + c(u,v) < d[v])

        d[v] = d[u] + c(u,v).
```

Example :-



Solution :-

① Label distances from src ① to all vertices.

Initially, the d[2] = 2

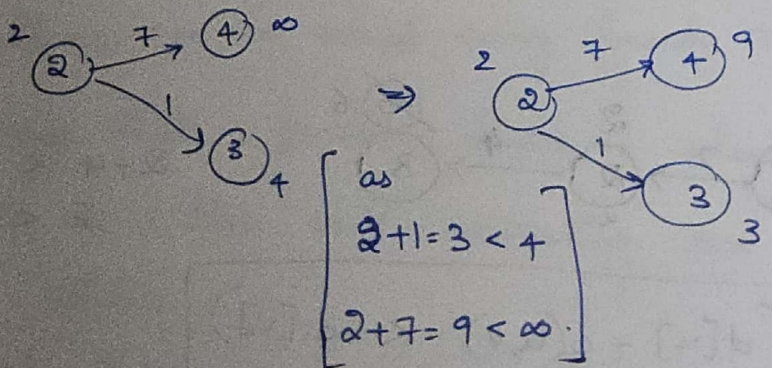
d[1] = 0

d[3] = 4

d[4] = ∞
d[5] = ∞
d[6] = ∞
} (as there is no direct path from src)

② Select smallest path vertex among [2, 4, ∞, ∞, ∞] which is ② as d[2] = 2

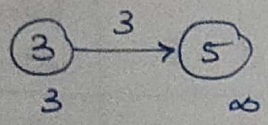
Check which vertices are connected.



Repeat same step for other vertices.

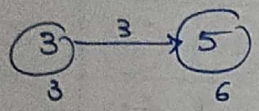
↳ Select among [3, 9, ∞, ∞] ⇒ (3)

The vertex that is connected is (5)

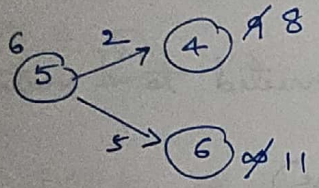


3+3 = 6 < ∞

vertex 5 can be relaxed.



↳ Select vertices among (4), (5), (6) with distances [9, 6, ∞] which is smallest. (5) is smallest

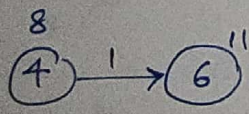


6 + 2 = 8 < 9

6 + 5 = 11 < ∞.

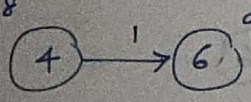
Relax (4) & (6).

↳ Select vertices among (4), (6) ⇒ [8, 11] ⇒ (4) is smallest.



8 + 1 = 9 < 11

vertex (6) can be relaxed.



So, finally the shortest paths to all vertices from (1) are:

v	d[v]
2	2
3	3
4	8
5	6
6	9

Analysis of Time Complexity :-

① It is finding shortest path vertex each time.

So number of vertices, $n = |V|$.

② It is relaxing all connected vertices.

There may be n vertices connected to a vertex.

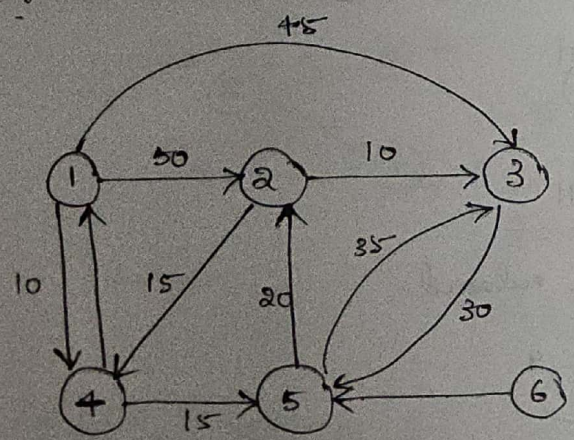
So $n \times n = n^2$ (worst case)

$\Theta(|V|^2)$

or

$\Theta(n^2)$.

Example :-



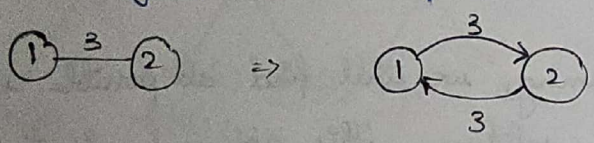
In this example, we will use Dijkstra's algorithm to solve Single Source Shortest Path algorithm.

Starting vertex (1)

Selected vertex	2	3	4	5	6
4	50	45	(10)	∞	∞
5	50	45	(10)	(25)	∞
2	(45)	45	(10)	(25)	∞
3	(45)	(45)	(10)	(25)	∞
6	(45)	(45)	(10)	(25)	(∞)

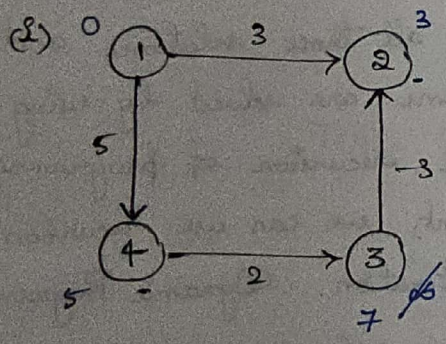
We don't relax already selected vertices.

For undirected graph, we can represent as directed as follows.



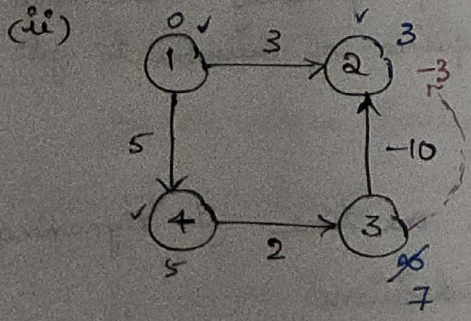
Drawback of Dijkstra Algorithm :-

(i)



The Dijkstra Algorithm works here even when there is negative edge.

(ii)



Since (2) is already visited, we do not relax (2) with -3 according to Dijkstra. But we have traversed the other way instead of Dijkstra, we could have updated 3 on node (2) to -3.

So, Dijkstra Algorithm may & may not work for negative edges. There is Bellman-Ford algorithm belongs to DP which solves Single Source Shortest Path problem.