GREEDY TECHNIQUE

The greedy technique suggests constructing a solution to an optimization problem through a sequence of steps, each expanding a partially constructing solution obtained so far, until a complete solution to the problem is reached.
On each step, the choice made must be

- feasible, (ie) it has to satisfy the problem's constraint
- locally optimal (ie) it has to be the best local choice among all feasible choices available on that step.
- irrevocable (ie) once made, it cannot be changed on subsequent steps of the algorithm.

These requirements explain the technique's name: on each step, it suggests a "greedy" grab of the best alternative available in the hope that a sequence of locally optimal choices will yeild a (globally) optimal solution to the entire problem.
Eg: Fractional Knapsack, Activity Selection, Huffman coding etc.

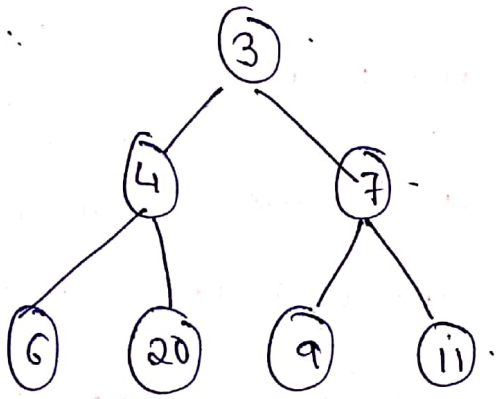
Pros

→ simple
→ easy to code.
→ runs fast.

cons

→ There are very few cases where it will end up in optimal value.

Eg:



Greedy choice would be:

$3 \rightarrow 7 \longrightarrow 11 = 21$.

but $3 \rightarrow 4 \longrightarrow 20 = 27$ } → gives more profit.

Thus, we can conclude that locally optimal greedy choice does not lead to optimal global solution.

① ACTIVITY SELECTION PROBLEM

I Problem:

Given `n' activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can work on a single activity at a time.

II Steps to solve using greedy technique:

Step 1: Sort the activities according to their finish time.
Step 2: Select the first activity from the sorted array and print it.
Step 3: Perform the following for remaining activities in the sorted array:
(a) if the start time of this activity is greater than or equal to finish time of previously selected activity then select this activity and print it.

III Example:

Activity	A1	A2	A_3	A_4	A_5	A_6
start	0	3	1	5	5	8
Finish	6	4	2	9	7	9

Step 1: Sort as per finish time.

Activity	A_3	A_2	A_1	A_5	A_6	A_4
Start	1	3	0	5	8	5
Finish	2	4	6	7	9	9

Step 2 : Print A_3.

$A_3 \rightarrow$

Step 3 : Compare start time of A_2 with finish time of A_3

(i) start (A_2) > finish (A_3).

∴ Print A_2.

$A_3 \rightarrow A_2$.

(ii) Compare start (A_1) with finish (A_2).

start (A_1) < finish (A_2).

∴ Don't print A_1

(iii) Compare start (A_5) with finish (A_2).

start (A_5) > finish (A_2).

∴ print A_5.

$A_3 \rightarrow A_2 \rightarrow A_5$

(iv) Compare . start (A_6) with finish (A_5).

start (A_6) > finish (A_5).

∴ Print A_6

$A_3 \rightarrow A_2 \rightarrow A_5 \rightarrow A_6$ ⟹ solution !

(v) Compare start (A_4) with finish (A_6).

start (A_4) < finish (A_6).

∴ Don't print A_4

Therefore A person can perform at most four activities in this problem.

IV Algorithm for Activity Selection Problem:

```
struct Activity.
{
    char id[5];
    int start;
    int finish;
};

Void ActivitySelection ( Activity A[], int n).
{
    int i, j;
    Activity temp;
    /* sort as per finishing time */
    // step 1.
    for (i=1; i<n; i++).
    {
        for (j=0; j<n-1; j++).
        {
            if (A[j].finish > A[j+1].finish).
                temp = A[j]
                A[j] = A[j+1]
                A[j+1] = temp.
        }
    }
    // step 2.
    printf ("%s %d %d ", A[0].id, A[0].start, A[0].Finish);
    i=0
    for (j=1; j<n; j++).
    { if (A[j].start >= A[i].finish)
        printf ("%s %d %d ", A[j].id, A[j].start, A[j].finish);
        i=j;
}. }
```

sorted algorithm is your choice.

V Time complexity:

It takes:

$O(n \log n)$: if the input activities are not sorted.

$O(n)$: if the input activities are all sorted.

② FRACTIONAL KNAPSACK.

I Problem:

Given weights and values of 'n' items, we need to put these items in a knapsack (container) of capacity 'W' to get the maximum total value (ie) max profit) in the knapsack.

II Steps to solve using greedy technique:

Step 1: Calculate the ratio (~~value~~ profit/weight) for each item.
Step 2: Sort items based on ratio. (highest to lowest).
Step 3: Take the item with highest ratio and add them until we can't add next (upcoming) item as whole.
Step 4: At the end add the next item as much as we can (fraction) without exceeding the capacity (W) of the bag.

III Example:

$W = 15$

objects.	1	2	3	4	5	6	7.
profit	10	5	15	7	6	18	3.
weights.	2	3	5	7	1	4	1
Step1: P/w	5	1.66	3	1	6	4.5	3

Step 2:

objects	5	1	6	3	~~7~~	2	4
profit	6	10	18	15	3	5	7
weights	1	2	4	5	1	3	7
P/w.	6.	5	4.5	3	3	1.66	1.

Step 3:

→ Take the first item in the sorted list as it has max P/w_i value

Capacity = 15 units

Weight of item 5 = 1 units.

∴ 15 − 1 = 14 units space left in the knapsack

[item 5 is taken completely]

→ take the next best P/w_i value associated item.

Capacity = 15 units.

Current weight inside the knapsack = 1 unit

Space left to fill = 14 units.

weight of item 1 = 2 units.

∴ 14 − 2 = 12 units space left in the knapsack.

[item 1 taken completely]

Continuing in the same manner

12 − (4) = 8 units. [item 6 is taken completely] → weight of item 6.

8 − (5) = 3 units [item 3 is taken completely] → weight of item 3

3 − (1) = 2 units. [item 7 is taken completely] → weight of item 7

→ take the next best P/w_i value associated item (ie) item 2

Total quantity of 'item 2' = 3 units.

Space left in the knapsack = 2 units.

Hence only 2 units of 'item 2' are filled into the knapsack.

[item 2 is not taken completely, only 2/3 units is taken]

Let 'X' represent the set of how much quantity of each item is filled in the knapsack.

value '1' → entire quantity of the item is selected.

'0' → the item is not selected.

fraction → a portion of the total quantity of the item is selected.

∴ X

x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	2/3	1	0	1	1	1

→ solution

Total weight of the knapsack (selected items in the): $\sum x_i w_i$

$= (1 \times 2) + (2/3 \times 3) + (1 \times 5) + (0 \times 7) + (1 \times 1) + (1 \times 4) + (1 \times 1)$

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

$= 15 \approx$ to the capacity of the knapsack !!

Total profit obtained from the knapsack $= \sum x_i P_i$

$= (1 \times 10) + (2/3 \times 5) + (1 \times 15) + (0 \times 7) + (1 \times 6) + (1 \times 18) + (1 \times 3)$

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

$= 55.33$ → solution

IV Algorithm for Fractional Knapsack Problem:

```
Greedy Knapsack( w[1.....n], p[1.....n], capacity)
    for i = 1 to n.
    do
        x[i] = 0.
        weight = 0.
    for i = 1 to n.
        if weight + w[i] ≤ capacity then.
            x[i] = 1.
            weight = weight + w[i]
        else.
            x[i] = (capacity - weight) / w[i]
            weight = capacity.
            break;
    return x.
```

V Time complexity of fractional knapsack.

Time complexity = Sorting time + selection time.

merge sort ← $n \log n$ + n.

$= n \log n$.

$O(n \log n)$

③ HUFFMAN CODING ALGORITHM

I Problem:

Huffman coding is a lossless data compression algorithm.
The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.
The most frequent character get the smallest code and the least frequent character gets the largest code.

II Terminologies:

- A Huffman tree is a binary tree that minimizes the weighted path length from the root to the leaves of predefined weights. The most important application of Huffman trees are Huffman codes.
- A Huffman code is an optimal prefix-free variable-length encoding scheme that assigns bit strings to characters based on their frequencies in a given text.
- The variable-length codes assigned to input characters are called Prefix-codes or codewords.
- Two major parts in Huffman coding.
  → To build a Huffman tree from input characters
  → To traverse the Huffman tree and assign codes.

III. Steps to build Huffman tree using greedy technique.

Step 1: Create a Leaf node for each unique character and build a min heap of all leaf nodes.

Step 2: Extract two nodes with the minimum frequency from the min heap.

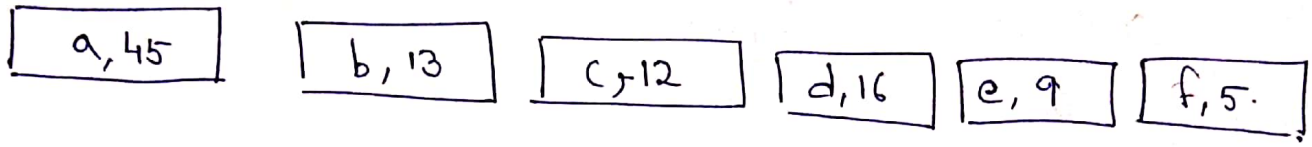
Step 3: Create a new internal node with frequency equal to the sum of the two nodes frequencies.
Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

Step 4: Repeat step 2 and step 3 until the heap contains only only one node. The remaining node is the root node and the tree is complete.

IV Examples:

character	frequency
a	45
b	13
c	12
d	16
e	9
f	5

Step 1: Assigning leaf nodes to each input character.

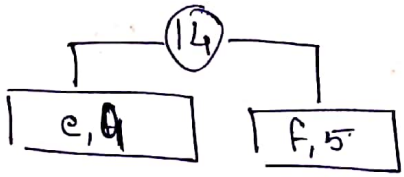


Step 2:

Least frequencies = e, f.

:

Creating new internal node with frequency (e+f) =



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of the tree with 3 elements.

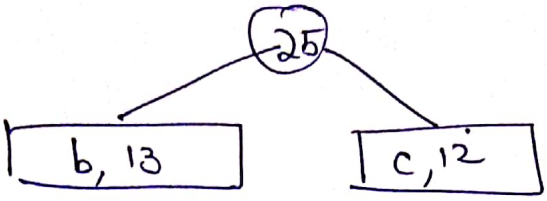
character	frequency
a	45
b	13.
c	12.
d	16.
internal node	14.

Step 3: Extract two minimum frequency nodes from heap.

Add a new internal node

Least frequency nodes : b, c.

Creating new internal node : b+c ⇒ 13 + 12 = 25



character	frequency
a	45
internal node	25 ⇒ 13 + 12
d	16
internal node.	14

Repeating steps 2 and step 3 for the rest of the heap.

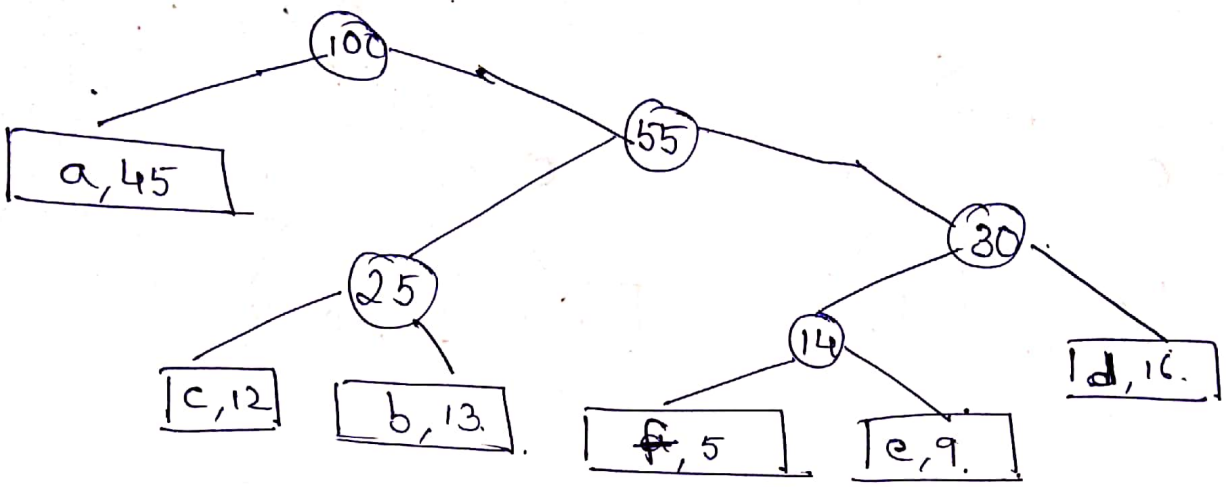
character	frequency
a	45
internal node	25
internal node	30 ⇒ 16 + 14

character	frequency
a	45
internal node.	55 ⇒ 25 + 30

character	frequency.
root-node	100

$\Longrightarrow$ 45 + 55

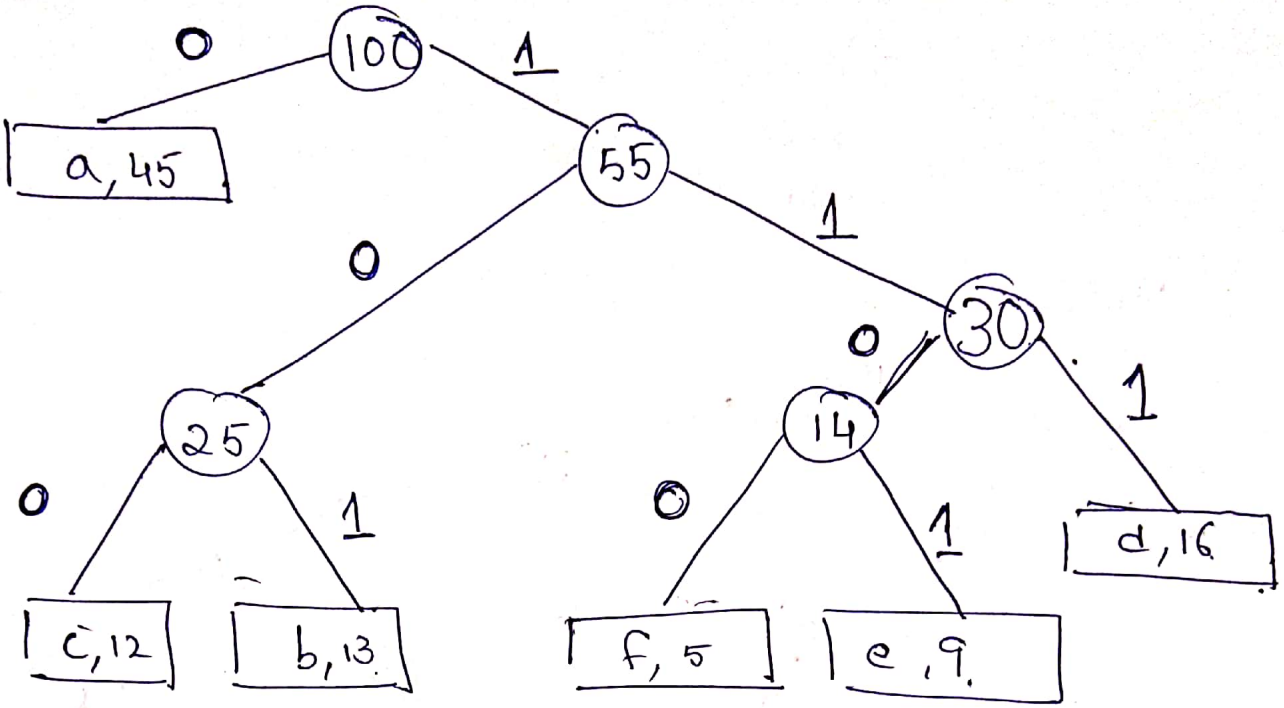
Huffman tree



NOTE: IN THE HUFFMAN TREE, ALL NODES TO THE LEFT ARE LESSER THAN THE NODES IN THE RIGHT HAND SIDE [Binary Search tree]

Steps to print codes from Huffman tree:

- Traverse the tree formed starting from the root.
- While moving to the left child, write 0 to the array.
- While moving to the right child, write 1 to the array.
- Print the array when a leaf node is encountered.



character	code-word.
a	0.
b	101.
c	100.
d	111.
e	1101
f	1100.

VI Time Complexity.

Time complexity = O(nlogn).

(4) KRUSKAL'S ALGORITHM (To build minimum cost spanning tree)

I Termithologies.

(a) Minimum spanning tree. (MST).

→ Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices in the graph.

→ MST, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree.

→ the weight/cost of the spanning tree = sum of costs of each edge of the spanning tree.

(b) minimum number of edges in a spanning tree for a graph with 'V' vertices is $(V-1)$ edges.

(c) the total number of edges in a graph with 'E' edges and 'V' vertices is.

$$E <= V^2$$

I Problem:

To build a spanning tree with the least cost of edges. (minimum cost spanning tree) (MST).

Algorithms to build minimum cost spanning trees:

(a) Kruskal's algorithm.

(b) Prim's Algorithm

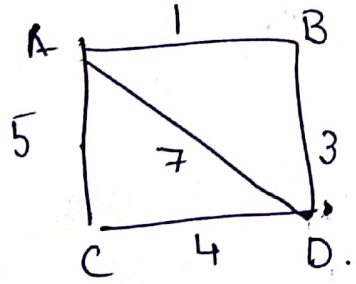
II Steps to finding MST using Kruskal's Algorithm.

Step 1 : Sort all the edges in non-decreasing order of their weights.

Step 2 : Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

Step 3 : Repeat step 2 until there are (V-1) edges in the spanning tree.

III Example:



Step 1 : Sorting of edges.

AB = 1

BD = 3

CD = 4

CA = 5

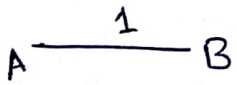
AD = 7

V-1 ⇒ 4-1

= 3 edge required.

Step 2 : Pick AB, forms a cycle? No

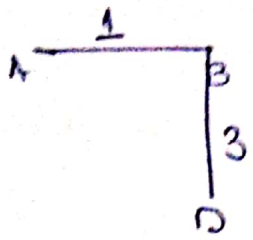
Hence include in MST



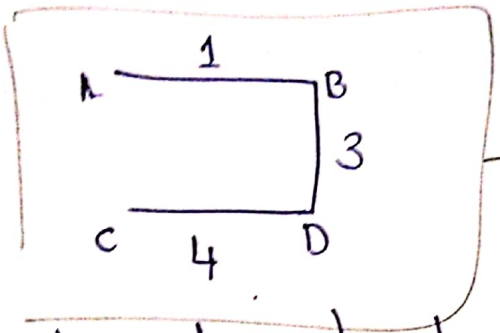
Step 3 : Pick next least cost edge.

BD, forms a cycle? No.

Hence, include in MST.



Step 4: Pick next edge CD, forms a cycle? No.
Hence include in MST.



Total cost:
1 + 3 + 4 = 8.

→ solution MST.

As three edges have already been selected (V-1 edges) the rest of the edges are rejected.

NOTE: More problems are done in class.

IV' Algorithm for MST using Kruskal's:

```
MST_Kruskal (G, w).
{
    A = 0.                                  ——→ O(1)
    for each vertex V ∈ G,V.                }
        MAKE_SET(V)                         } → O(V)
    Sort the edges of G into non decreasing order of  } → O(ElogE)
                                                wt 'w'
    for each edge (u,v) ∈ G,E.              }
        if FIND_SET(u) ≠ FIND_SET(v)        } → O(VlogV)
            A = A ∪ {(u,v)}.                }
            UNION (u,v).
    return A
}.
```

V. Time complexity :

Time complexity : sorting edges + iteration through all edge + apply find-union

$$= O(E \log E) + O(E \log V).$$

So complexity is $\boxed{O(E \log E) \text{ or } O(E \log V)}$.

because E can be atmost $O(V^2)$, so $O(\log V)$ is same as $O(\log E)$

PRIM'S ALGORITHM

I Working of Prim's Algorithm

The idea behind prim's algorithm is every new edge that is added to the MST should be connected to any one of the already existing edges in the MST. Also the new edge minimum weight edge and should not form a cycle.

II Steps to building MST using Prim's algorithm

Step 1: Create a set mstSet that keeps track of vertices already included in MST.

Step 2: Assign a key value as 0 for the first vertex so that Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as '0' for the first vertex so that it is picked first.

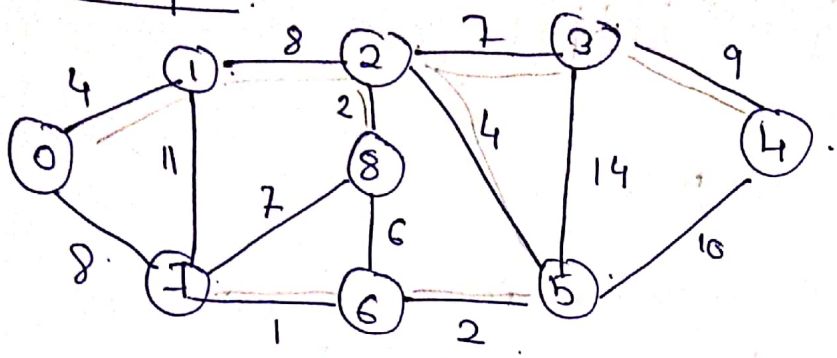
Step 3: While mstSet doesn't include all vertices.

(a) Pick a vertex 'u' which is not there in mstSet and has minimum key value.

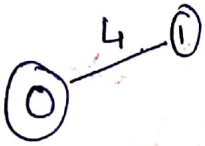
(b) Include u to mstSet.

(c) Update key value of all adjacent vertices of 'u'. To update the key values, iterate through all adjacent vertices. For every adjacent vertex 'v', if weight of edge u-v is less than the previous key value of v, update the key value as weight of 'u-v'.

III Example:



1) Start from node 0, pick least cost edge which is to node 1 with cost 4.



2) Now check from both node '0' and node '1' (out of all edge possibilities) which is the least cost edge.

Edge Possibilities from node '0' and node '1' :

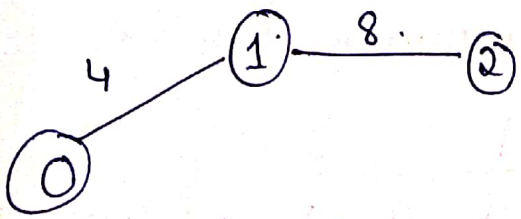
01 ⇒ already included.

07 ⇒ 8 → since they are same cost either edge can be select, your choice.

17 ⇒ 11.

12 ⇒ 8 → least cost

so select this edge and include in MST

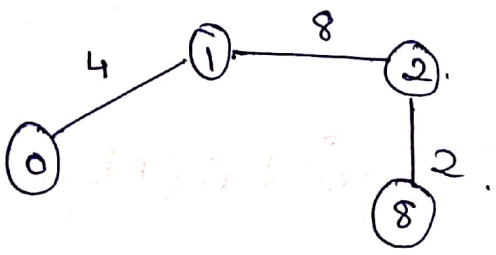


3). Now check from node `0', node `1' and node `2' which is least cost edge.

Edge Possibilites from node `0', node `1' and node `2':

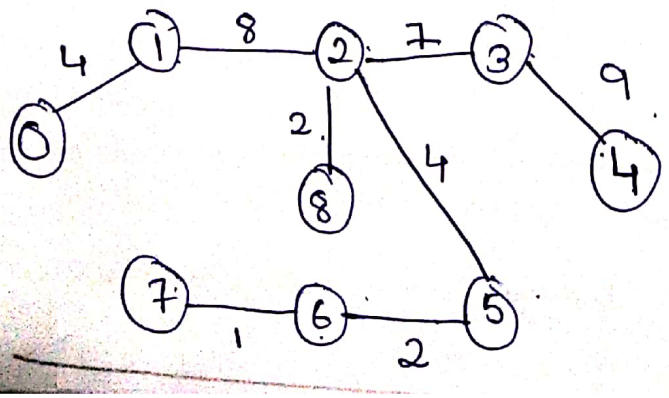
01 => 4
12 => 8.
07 => 8.
17 => 11.
28 => 2. ——→ least cost edge so select this edge and include in MST.
23 => 7.
25 => 7.

∴



continuing until all vertices are include in the MST in the same manner, we get the final MST.

solution ←



Total cost:

4 + 8 + 7 + 9 + 2 + 4 + 1 + 2

= 37.

IV Algorithm for Prim's:

```
Prim's()
{
    T = 0
    U = {1}.
    while (U ≠ V).
        Let (u, v) be the lowest cost edge such that u ∈ U
                                                and v ∈ V - U

        T = T ∪ {(u, v)}.
        U = V ∪ {v}.
}
```

V Time complexity.

Time complexity = $O(v \log v) + O(E \log v)$.

$= O(V \log v + E \log V)$.

$\approx \boxed{O(E \log v)}$.

↓

because E can be almost $O(v^2)$,

so $O(\log V)$ is same as $O(\log E)$.

NOTE: For both Kruskal's & Prim's

Time complexity = edge traversal + merge operation + sorting time + find-component operation