Activity Selection: Greedy Methods and Matroids

[1]

1 An Activity-Selection Problem

Suppose we have a set $S = \{a_1, a_2, a_3, ..., a_n\}$ of activities that require exclusive use of a common resource **R**. Each activity has a start time s_i and a finish time f_i such that $0 \leq s_i < f_i < \infty$.

Our objective is to select a subset of S that has maximum size among all such possible subsets such that each activity in the set is **compatible** with each other.

Two activites a_i and a_j are said to be compatible if they do not overlap with each other. If a_i is selected to use **R**, then it takes place in the half-open interval $[s_i,f_i)$, similarly if a_j is selected to use **R** then it takes place in the half-open interval $[s_j,f_j)$. We need the two intervals $[s_i,f_i)$ and $[s_j,f_j)$ to not overlap with each other.

Thus, it is required that either $s_i \geq f_j$ (activity i starts after activity j ends) or $s_j \geq f_i$ (activity j starts after activity i ends)

Tab. 1: Example set S of activities

i	1	2	3	4	5	6	7	8	9	10	11
$\boldsymbol{s_i}$	1	3	0	5	3	5	6	8	8	2	12
$\boldsymbol{f_i}$	4	5	6	7	9	9	10	11	12	14	16

Tab. 1 shows a set S of eleven activities sorted in non-decreasing order of their finish times f_i.

The following lists all possible subsets of S that satisfy the compatibility property:

[1] Notes By Ayush Kumar Shukla & Sudip Roy

[1]	[2]	[3]	[4]	[5]
[6]	[7]	[8]	[9]	[10]
[11]	[1, 4]	[1, 6]	[1, 7]	[1, 8]
[1, 9]	[1, 11]	[2, 4]	[2, 6]	[2, 7]
[2, 8]	[2, 9]	[2, 11]	[3, 7]	[3, 8]
[3, 9]	[3, 11]	[4, 8]	[4, 9]	[4, 11]
[5, 11]	[6, 11]	[7, 11]	[8, 11]	[9, 11]
[1, 4, 8]	[1, 4, 9]	[1, 4, 11]	[1, 6, 11]	[1, 7, 11]
[1, 8, 11]	[1, 9, 11]	[2, 4, 8]	[2, 4, 9]	[2, 4, 11]
[2, 6, 11]	[2, 7, 11]	[2, 8, 11]	[2, 9, 11]	[3, 7, 11]
[3, 8, 11]	[3, 9, 11]	[4, 8, 11]	[4, 9, 11]	[1, 4, 8, 11]
[1, 4, 9, 11]	[2, 4, 8, 11]	[2, 4, 9, 11]		

Abb. 1: Compatible subsets in S

From Abb. 1 it can be inferred that the subsets: [1, 4, 8, 11], [1, 4, 9, 11], [2, 4, 8, 11], [2, 4, 9, 11] are maximal since they all are the largest sized subsets.[2] The objective of the problem is to find such maximum sized subsets of S.

Problems like the **Activity Selection Problem** can be solved using the Greedy Approach due to the presence of certain characteristic properties in the nature of the problem itself.

We shall now discuss the greedy approach for solving this problem.

1.1 Greedy Approach To Activity Selection: What Makes it Greedy Solvable ?

We shall present two properties that the problem under consideration possesses:

1.1.1 The Optimal Substructure of the Activity-Selection Problem

Let us consider S_{ij} to be the set of activities that start after activity i ends, and end before activity j starts. Note that if $i = 1$ and $j = n$, where n is the number of activities, then S_{0n} is the set S itself.

We wish to find a maximum subset of S_{ij} (say A_{ij}) which contains mutually compatible activities.

Suppose that A_{ij} contains some activity a_k ($a_k \in S_{ij}$)

By including a_k in the optimal solution, we are left with two subproblems:
1. finding mutually compatible activites in the set S_{ik} and
2. finding mutually compatible activities in the set S_{kj}

Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$

Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

To obtain the global optimal solution A_{ij}, we need to obtain the optimal solution for it's subproblems A_{ik} and A_{kj}. This is described as the optimal substructure property.

Formally put,

Let P be a problem and $\{P_1, P_2, ..., P_n\}$ be subproblems derieved from P. If solving $\{P_1, P_2, ..., P_n\}$ optimally guarantees an optimal solution for P, then P has the optimal substructure property.

1.1.2 Making the Greedy Choice in Activity-Selection

The principles of Dynamic Programming require us to first solve all the subproblems that can be obtained (in an optimal manner), and combining the results to obtain the solution to the original problem. This bottom-up approach is computationally intensive.

[2]Notes By Ayush Kumar Shukla & Sudip Roy

In the **activity selection problem**, intuition says that in order to get the largest set of compatible activities, we should pick that activity which relinquishes the resource at the earliest, that is, the activity with the least finish time must be selected first, as it leaves the resource free for other resources as fast as possible, such a decision strategy is called **greedy choice**.[3] Note that the **greedy choice** gives the optimal solution to problems that have the **greedy choice property**, thus, it is essential to prove the presence of greedy choice property in the problem before using the greedy strategy to solve it.

We now prove that the **activity selection problem** indeed has the greedy choice property.

Theorem:

Consider any non-empty subproblem S_k and let $a_m \in S_k$ such that a_m has the earliest finish time. Then a_m is included in some maximum-sized subset of mutually compatible activities of s_k

Proof

Let A_k be a maximum sized subsey of mutually compatible activities in S_k. Let $a_j \in A_k$ with the earliest finish time. If $a_j = a_m$, we are done.

Let us assume that $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but replacing a_j with a_m. The activities in A'_k are disjoint since the activities in A_k are disjoint. Now we have $|A'_k| = |A_k|$, both A'_k and A_k are maximal sets of compatible activities since $f_m \leq f_j$, that is, the replaced activity cannot overlap with the second activity in A'_k.

$\therefore a_m$ belongs to a maximum-sized set of compatible activities in S_k ■

[3]Notes By Ayush Kumar Shukla & Sudip Roy

KNAPSACK PROBLEM

It is a classical optimization problem. There are two variants of this problem:

- **0-1 knapsack problem:**
  A thief robbing a store finds n items. The i^{th} item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W. Which items should he take? [We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.]

- **Fractional knapsack problem:**
  In the fractional knapsack problem, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. We can think of an item in the 0-1 knapsack problem as being like a gold ingot and an item in the fractional knapsack problem as more like gold dust.

Both knapsack problems exhibit the optimal-substructure property. Although the problems are similar, we can solve the fractional knapsack problem by a greedy strategy, but we cannot solve the 0-1 problem by such a strategy (0-1 knapsack problem is solved by dynamic programming).

Problem Definition

Given n objects and a knapsack where object i has a weight w_i and the knapsack has a capacity m. If a fraction x_i of object i placed into knapsack, a profit $p_i x_i$ is earned. The objective is to obtain a filling of knapsack maximizing the total profit.

$$\text{Maximize } \sum_{i=1}^{n} p_i x_i \qquad \text{----- (i)}$$

$$\text{subject to } \sum_{i=1}^{n} w_i x_i \leq m \qquad \text{----- (ii)}$$

$$\text{and } 0 \leq x_i \leq 1 \text{ , } 1 \leq i \leq n \qquad \text{----- (iii)}$$

- A feasible solution is any set satisfying (ii) and (iii).
- An optimal solution is a feasible solution which satisfies (i), (ii) and (iii).

Example

Capacity of knapsack (m) = 40

No. of objects (n) = 3

Object (i)	Profit (p_i)	Weight (w_i)
1	25	18
2	24	15
3	15	10

SOLUTION:

1. Calculate the profit/weight for each object.

Object (i)	Profit (p_i)	Weight (w_i)	Profit/Weight (p_i/w_i)
1	25	18	25/18 = 1.3889 ≈1.4
2	24	15	24/15 = 1.6
3	15	10	15/10 = 1.5

Created by Vijoya Guha Thakurta

2. Sort the objects in non-increasing order of profit/weight.

Object (i)	Profit (p_i)	Weight (w_i)	Profit/Weight (p_i / w_i)
1	24	15	1.6
2	15	10	1.5
3	25	18	1.4

3. Determine the fraction of each object to be considered to maximize profit.

   U = m = 40

Object (i)	Profit (p_i)	Weight (w_i)	Profit/Weight (p_i / w_i)	Remaining Capacity (U)	Fraction of i (x_i)	Cumulative Profit (P)
1	24	15	1.6	40	1	24
2	15	10	1.5	40 – 15 = 25	1	24 + 15 = 39
3	25	18	1.4	25 – 10 = 15	15/18 = 0.83	39 + (25 * 0.83) = 59.75

Thus, the maximum profit = 59.75

Algorithm

- Inputs:

m	Capacity of the knapsack
n	No. of objects
p [1: n]	Profit of each object
w [1: n]	Weight of each object

- Outputs:

P	Maximized profit
x [1: n]	Fraction of each object

1. Sort the objects in the non-increasing order such that $p[i] / w[i] \geq p[i+1] / w[i+1]$
2. Initialize $x[i]$ with 0 for $0 \leq i \leq n$ and $U = m$.
3. For $i = 1$ to n
   3.1. If $w[i] > U$ then stop the loop.
   3.2. Assign $x[i] = 1$ and $U = U - w[i]$
4. If $i \leq n$ then $x[i] = U / w[i]$
5. Assign $P = p[i] * x[i]^T$

Complexity Analysis

Any sorting algorithm can be used for sorting in line 1. Thus, we can consider sorting algorithm takes O ($n \log n$) time.

Line 2 takes constant time i.e. O (1) for initialization.

The loop in line 3 runs a maximum of n times, thus all statements inside it will be executed at most n times. Lines 3.1 and 3.2 takes constant time. Thus, the complexity of line 3 is O(n).

Line 4 and 5 both takes constant time.

Thus, time complexity of Greedy knapsack algorithm is O ($n \log n$) + O(n) = O ($n \log n$)

References:

- Introduction to Algorithms (Third Edition), T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, 2009.
- Fundamentals of Computer Algorithms (Second Edition), E. Horowitz, S. Sahni, S. Rajasekaran, 2008.
- Dr. Rajat Kr. Pal, Professor, University of Calcutta for notes on Heaps, and Optimization.

Created by Vijoya Guha Thakurta

2 Matroids: Uses in the Greedy Approach

DEF: A **Matroid** is an ordered pair $M = (S, I)$ satisfying the following conditions:

1. S is a finite set
2. I is a nonempty family of subsets of S, called the independent subsets of S, such that if $B \in I$ and $A \subseteq B$, then $A \in I$. We say that I is **hereditary** if it satisfies this property.
3. If $A \in I, B \in I$, and $|A| < |B|$, then there exists some element $x \in B - A$ such that $A \cup \{x\} \in I$. We say that M satisfies the **exchange property**. In such a case, the element x is called the **extension** of A.

2.1 Weighted Undirected Graphs as Weighted Matroids

Theorem:

If $G = (V, E)$ is an indirected graph, then $M_G = (S_G, I_G)$ is a matroid.[4]

Proof:

Note: For any set of edges $E \subseteq S_G$, if $E \in I_G$, it implies that E does not form a cycle. This is so because the set I_G contains independent subsets of S_G. If E indeed formed a cycle, by the **hereditary property** *each set $E' \subset E$ would also be in I_G. Some $e \in E'$ would not contain cycles and still be independent, which leads to both cyclic and acyclic set of edges being deemed independent in our system, which is contradictory. To evade this contradiction, only acyclic set of edges are allowed to be included as independent subsets of S_G.*

1. $S_G = E$ is a finite set *(Condition 1 from Definition of Matroid)*
2. I_G is hereditary since the subset of a forest is also a forest (Condition 2 from Definition of Matroid)
3. Now we show that M_G satisfies the exchange property.

   Let $G_A = (V, A)$ and $G_B = (V, B)$ are forests of G and that $|B| > |A|$.

   We claim that a forest $F = (V_F, E_F)$ contains exactly $|V_F| - |E_F|$ trees.. To prove this, suppose that F contains t trees, where the i^{th} tree contains v_i vertices and e_i edges. Then, we have,

$$|E_F| = \sum_{i=1}^{t}(e_i) = \sum_{i=1}^{t}((v_i - 1)) = \sum_{i=1}^{t}(v_i) - t = |V_F| - t$$

$$\therefore t = |V_F| - |E_F|$$

   So, G_A contains $|V| - A$ trees and G_B contains $|V| - B$ trees.

   Since $|B| > |A|$, G_B has fewer trees than G_A, this implies that there exists two trees t_a and t_b in G_B such that an edge (u,v) connects t_a and t_b. This edges *(u,v)* can be safely added to G_A without creating a cycle. Thus, the exchange property is satisfied. ■

   A weighted undirected graph is represented by a weighted matroid, which contains a weight function ω that assigns a strictly positive weight $\omega(x)$ to each element $x \in S$.

Most problems of interest require us to find the **optimal subset** of a weighted matroid, in which the goal is to find an indepenent set $A \in I$ such that $\omega(A)$ is maximized.

The following two theorems establish Matroids as a generalization tool in solving problems using the greedy approach:

[4]Notes By Ayush Kumar Shukla & Sudip Roy

***Theorem**: (Matroids exhibit the greedy-choice property)*
Suppose that $M = (S, I)$ is a weighted matroid with weight function ω and that S *is sorted into monotonically decreasing order by weight. Let* x *be the first element of S such that $\{x\}$ is independent, if any such* x *exists. If* x *exists, then there exists an optimal subset* A *of* S *that contains* x.

***Theorem**: (Matroids exhibit the optimal substructure property)*
Let x be the first element of S chosen by the greedy method for the weighted matroid $M = (S, I)$. The remaining problem of finding a maximum-weight independent subset containing x reduces to finding a maximum-weight independent subset of the weighted matroid $M' = (S', I')$ where,

- $S' = \{y \in S : \{x, y\} \in I\}$
- $I' = \{B \subseteq S - \{x\} : B \cup \{x\} \in I\}$

and the weight function for M' is the weight function for M, restricted to S'.[5]

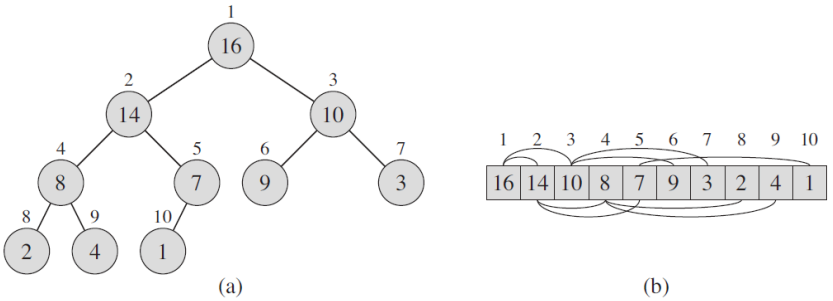
---

[5]Notes By Ayush Kumar Shukla & Sudip Roy

Complexity Analysis of Heap Operations

A heap (Min or Max) when stored in an Array 'Heap', has the following indexing:-

PARENT[i] is at Heap[⌊i/2 ⌋] index
LEFT[i] is at Heap[2i] index
RIGHT[i] is at Heap[2i+1] index.



(1) Heapify(Heap, i) **[applicable for both Min/Max Heaps]**

Pseudocode for Min-Heapify.

MIN-HEAPIFY(Heap, i) // Max-Heapify will have the same pseudocode but only the comparisons will differ

1. l = LEFT[i]
2. r = RIGHT[i]
3. smallest = i
4. if l ≤ Heap.length and Heap[l] < Heap[i]
5. smallest = l
6. if r ≤ Heap.length and Heap[r] < Heap[smallest]
7. smallest = r
8. if smallest ≠ i
9. exchange Heap[i] with Heap[smallest]
10. MIN-HEAPIFY(Heap, smallest)

Heapify assumes the LEFT[i] (left sub-tree) and RIGHT[i] (right subtree) of Heap[i] are heaps. Now maybe Heap[i] is violating the heap property, so it checks if either the LEFT[i] or RIGHT[i] is supposed to be in the position of Heap[i]. If so then they are exchanged and the one we exchange (left or right child), we perform Heapify on that.

So fixing the relationship between Heap[i], LEFT[i] and RIGHT[i] takes constant time, but we need to call Heapify recursively on either of the subtrees (left or right subtree).

Now we know heaps are filled in the leaf level, from left to right. So the left subtree is filled first then we move to the right subtree. This and assuming that the fixes (i.e. Heapify calls) occur only on the left subtree till leaf level, (a worse case considering all the recursive call occurs and when the bottom level is exactly half-full), the time taken becomes

T(n) ≤ T(total nodes in left subtree) + constant

Lines 1 to 9 take constant time and Line 10 calls Heapify recursively so it takes T(total nodes in left subtree) time.

Now the total nodes in the left subtree of a heap i.e. having its leaf level exactly half filled is

Total number of nodes (n) = 1 + N_L + N_R [N_L: Nodes in left sub-tree, N_R: Nodes in right subtree]

Now $N_L = 2N_R + 1$ [for a tree whose only the bottom level is exactly half-full]

$\Rightarrow N_R = \frac{N_L - 1}{2}$

So, $n = 1 + N_L + \frac{N_L - 1}{2} \Rightarrow n = \frac{1}{2} + \frac{3}{2}N_L \Rightarrow 2n = 1 + 3N_L$

$N_L = \frac{(2n-1)}{3} \Rightarrow \mathbf{N_L = \frac{2n}{3} - \frac{1}{3}}$ **So there can be at most $\frac{2n}{3}$ in the Left Subtree [i.e. floor($\frac{2n}{3}$)]**

So the time complexity can be found by solving the recurrence relation

$T(n) \leq T(\frac{2n}{3}) + \Theta(1)$

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

So, the Master Theorem can be used to solve the recurrence.

a = 1, b = $\frac{3}{2}$, So a $\geq$ 1, b > 1 [$\frac{2}{3}$n = $\frac{n}{b}$ => b = $\frac{3}{2}$]

Now we need to check if f(n) = $\Theta(1)$

=> f(n) = $\Theta(n^{log_{3/2}1})$ = $\Theta(n^0)$ = $\Theta(1)$ [$log_{3/2}1 = 0$]

So we can apply Case-2 of Master Theorem to solve the recurrence

T(n) $\leq$ $\Theta(n^{log_{3/2}1}$. log n)
=> T(n) $\leq$ Θ(log n)
=> T(n) = O(log n) [as the worse case (upper bound) occurs when T(n) = Θ(log n)]

So the complexity of **Heapify is O(log n) or O(h)**

Note that 'n' is not the total number of nodes in the entire tree, rather it is the total number of nodes in the sub-tree on whose root node, Heapify was called.

(2) BUILD-HEAP(A) **[applicable for creating both Min/Max Heaps]**

Pseudocode for Build-Min-Heap.

BUILD-MIN-HEAP(A)

1. Heap = A
2. for i = [Heap.length/2] down to 1
3. MIN-HEAPIFY(Heap, i)

Build-Heap converts an array 'A' into a Heap (Min or Max Heap).

The last parent node is at index i = $\frac{n}{2}$ (n is Heap.length), so we call Heapify from this node till index i = 1.

Now as we call Heapify, we can observe that the nodes closer to the leaf level take much less time than the ones closer to the root. Also, once Heapify is called on a node, its subtrees are already heaps.

Therefore, the complexity of Heapify is O(h) or O(log n') [n' is the total number of nodes in the sub-tree on which Heapify was called] as explained earlier, which means we need to sum up all the Heapify operations with varying heights to get our tighter bound analysis.

A heap having n elements will have ceil($\frac{n}{2^{h+1}}$) nodes at height h.

Reason:

Let H be the height of the entire heap.

A heap with a total n elements, at height 0 (in the leaf level) will have at most $ceil(\frac{n}{2})$ elements, so 2^H = ceil($\frac{n}{2}$)

The number of nodes at height 'h' in a binary tree with total height 'H' is $2^{H-h} = \frac{2^H}{2^h} = \frac{ceil(\frac{n}{2})}{2^h}$

Therefore, the number of nodes at height 'h' in a binary tree with total nodes n is ceil($\frac{n}{2^{h+1}}$)

Now, there are ceil($\frac{n}{2^{h+1}}$) nodes for which we call Heapify where $0 \le h \le H$ [H = log n]

Therefore the total time is $\sum_{h=0}^{log\ n} ceil(\frac{n}{2^{h+1}})$ * O(h) => O ($\sum_{h=0}^{log\ n} \frac{n}{2^{h+1}}$ * h)

$Solving \sum_{h=0}^{log\ n} \frac{n}{2^{h+1}}$ * h

=> $\sum_{h=0}^{log\ n} \frac{n}{2.2^h}$ * h

=> $\frac{n}{2}$ ($\sum_{h=0}^{log\ n} \frac{h}{2^h}$)

For large values of n, where n → ∞ then log n → ∞

=> $\frac{n}{2}$ ($\sum_{h=0}^{\infty}$ h * (½)^h) [using this formula: $\sum_{k=0}^{\infty} k.x^k = \frac{x}{(1-x)^2}$]

=> $\frac{n}{2} * \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}$

=> $\frac{n}{2}$ * 2

=> n

So the time complexity of **BUILD-HEAP is O(n)**

Min-Priority Queue using Binary Heaps

Priority Queue is a data structure for maintaining a set S of elements each having a key value. There can be a Max or a Min priority Queue. Prim's, Huffman Encoding etc. require the use of min-priority queues. Now implementing a Min-Priority Queue using a binary Min-Heap can make our algorithms work faster. The methods commonly used to implement Min-Priority queue are **MINIMUM**(Queue), **DECREASE-KEY**(Queue, i, newValue), **INSERT**(Queue, newElem), **EXTRACT-MIN**(Queue).

(3) MINIMUM(Queue)

Pseudocode for Minimum

```
MINIMUM(Queue)
1. return Queue[1]
```

Line 1 takes a constant time to execute. The running time complexity is **O(1)**.

(4) DECREASE-KEY(Queue, i, newValue)

Decrease-Key changes the value of a node 'i' in the Priority Queue with the given lower value iff it is smaller than the current value. Then if required it rebuilds the Heap accordingly.

Pseudocode for Decrease-Key

```
DECREASE-KEY(Queue, i, newValue)
1. if newValue > Queue[i]
2.         error "new key is larger than current key"
3. Queue[i] = newValue
4. while i > 1 and Queue[ PARENT[i] ] > Queue[i]
5.         exchange Queue[i] with Queue[ PARENT[i] ]
6.         i = PARENT[i]
```

Lines 1 to 3 take constant time $i.e.\ O(1)$. The maximum number of exchanges that can occur is equal to the height of the Queue (Min-Heap). Therefore the running time for a Queue (Min-Heap) of length n is **O(log n)** [n is the total nodes in the heap (min-priority queue)].

(5) INSERT(Queue, newElem)

Insert adds a new element into the queue. This new element is added as a leaf node such that the structure of the Heap is retained (i.e. a complete binary tree). Now the insertion of this new element can violate the Heap property, so we use DECREASE-KEY or INCREASE-KEY methods to insert.

Pseudocode for Insert

```
INSERT(Queue, newElem)
1. Queue.length = Queue.length + 1
2. Queue[Queue.length] = ∞
3. DECREASE-KEY(Queue, Queue.length, newElem)
```

Lines 1 and 2 take constant time i.e. $O(1)$. Line 3 takes O(log n) as explained earlier. So the running time complexity for inserting a new element is **O(log n)** [n is the total nodes in the heap (min-priority queue)].

(6) EXTRACT-MIN(Queue)

Extract Min will remove and return the node with the minimum key from the Queue (Min-Heap). It does this by first removing the root node from the Heap, and then replacing the empty root with the rightmost leaf node.

Pseudocode for Extract-Min

```
EXTRACT-MIN(Queue)
1. if Queue.length < 1
2.         error “underflow”
3. min = Queue[1]
4. Queue[1] = Queue[ Queue.length ]
5. Queue.length = Queue.length - 1
6. MIN-HEAPIFY(Queue, 1)
```

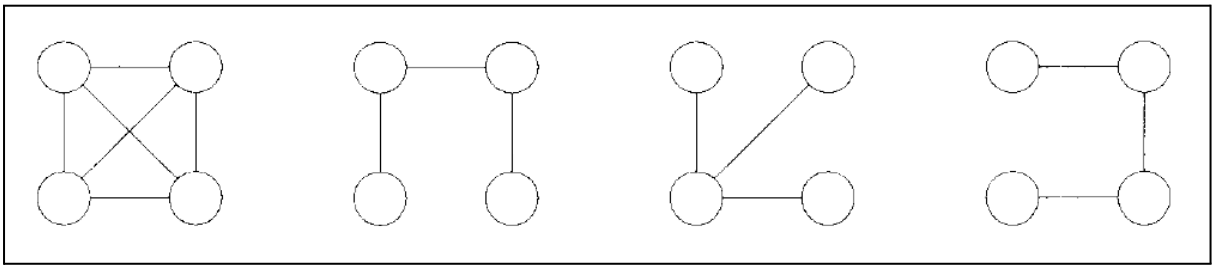
Lines 1 to 5 take constant time i.e. O(1). Line 6 takes O(log n) as explained earlier. So the total running time complexity of extracting a minimum element from a queue is **O(log n)** [n is the total nodes in the heap (min-priority queue)].

Spanning Tree

Definition: Let G = (V, E) be an undirected connected graph. A sub-graph G' = (V, E') of G is a spanning tree of G iff G' is a tree.

Observation / Properties:
1. A spanning tree is a minimal subgraph G' of G such that V(G') = V(G) and G' is connected (no components in G').
2. The spanning tree of a connected Graph G of |V| vertices and |E| edges, its spanning tree has |V| vertices but |V|-1 edges. So the spanning tree cannot have any cycles.

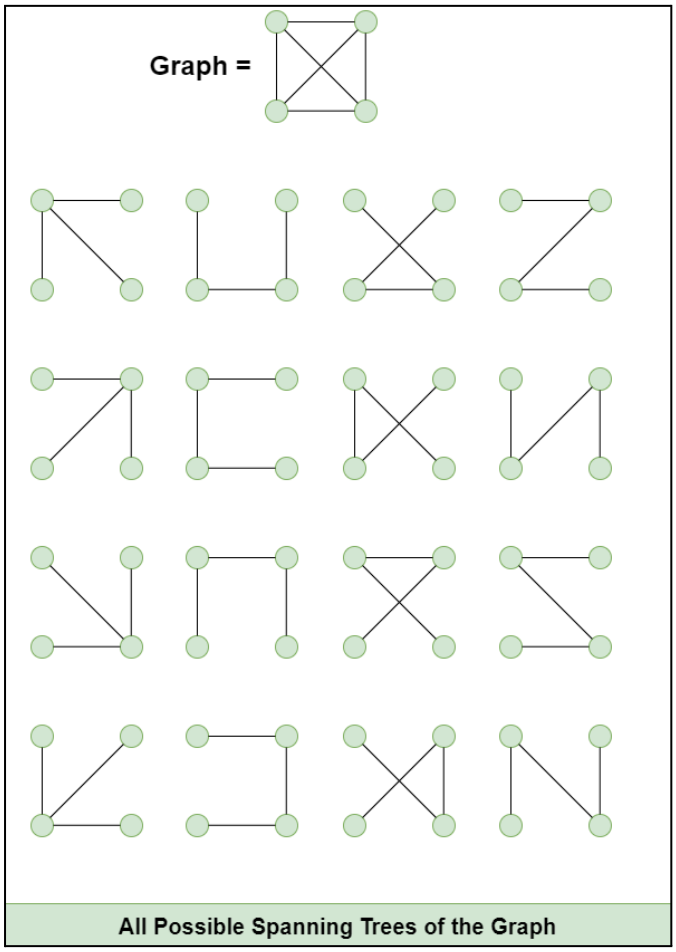
Example:

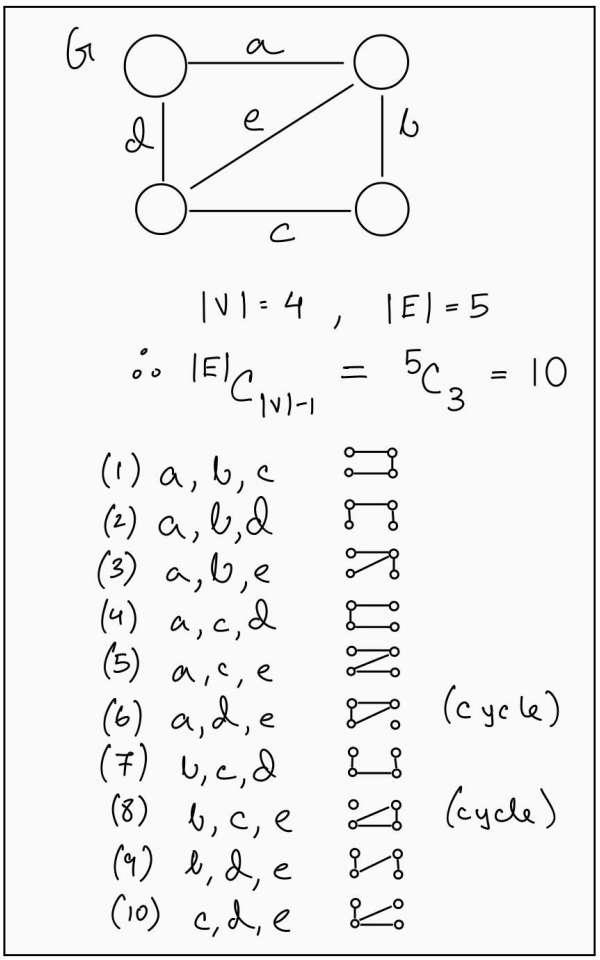


An undirected graph and its three spanning trees

Total number of possible spanning trees.

For a graph G, we choose |V|-1 edges among all the |E| number of edges. So, from a total of |E| edges, we select |V|-1 edges to create all our possible spanning trees.

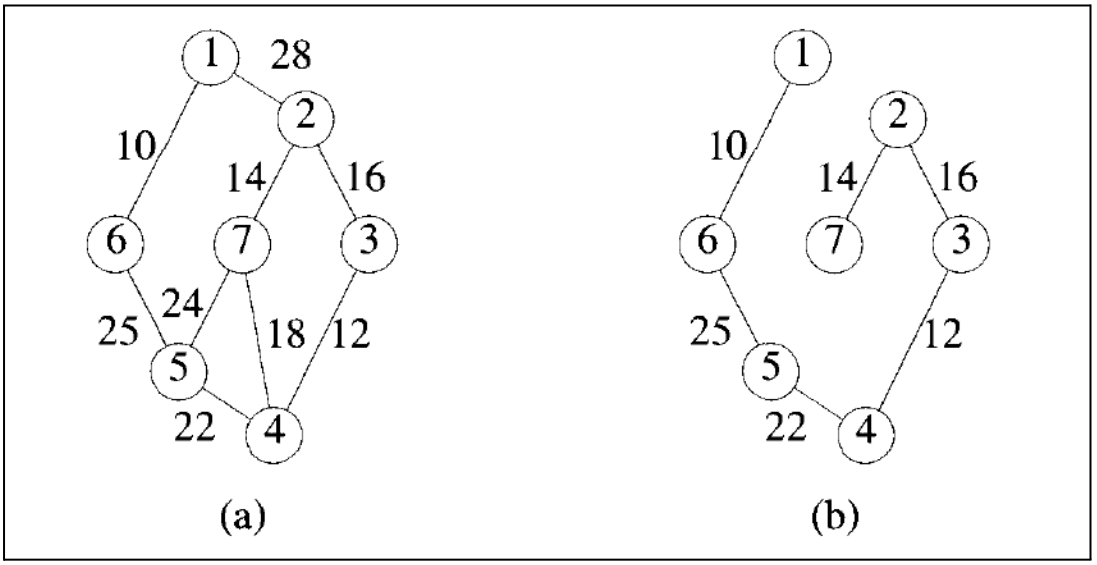




So there can be a total of ${}^{|E|}C_{|V|-1}$ ways we can select the edges but the ones that make spanning trees are simply **${}^{|E|}C_{|V|-1}$ - number of cycles**.

Minimum-Cost Spanning Tree

In practical situations, edges have associated weights/costs, so the Spanning Tree with the minimum cost among the others is the *Minimum-Cost Spanning Tree*.



A graph and its minimum cost spanning tree

Obtaining Minimum-Cost Spanning Tree

For graph G, there could be multiple spanning trees of their respective cost *(Feasible Solutions)*.
Now we want the spanning tree with the minimum cost amongst others *(Minimization)*.

So we must choose the minimum (best solution according to our problem) among the Feasible solutions.

There are two Greedy Algorithms to find out MST

1. Prim's Algorithm (published by Robert C. Prim in 1957)
2. Kruskal's Algorithm (published by Joseph Kruskal in 1956)

Prim's Algorithm:

Prim's algorithm begins from a vertex either provided or selected lexicographically. It continues to add edges having minimum cost where one vertex of the edge is a part of the tree and the other isn't.

Input: A weighted connected graph G(V, E) with V vertices and E edges.
Output: A minimum cost spanning tree T with |V| vertices and |V|-1 edges.

Step 1: Choose a vertex 'v_0' lexicographically from set V. Add vertex 'v_0' into tree T.

$T := \{ \{v_0\}, \Phi \}$

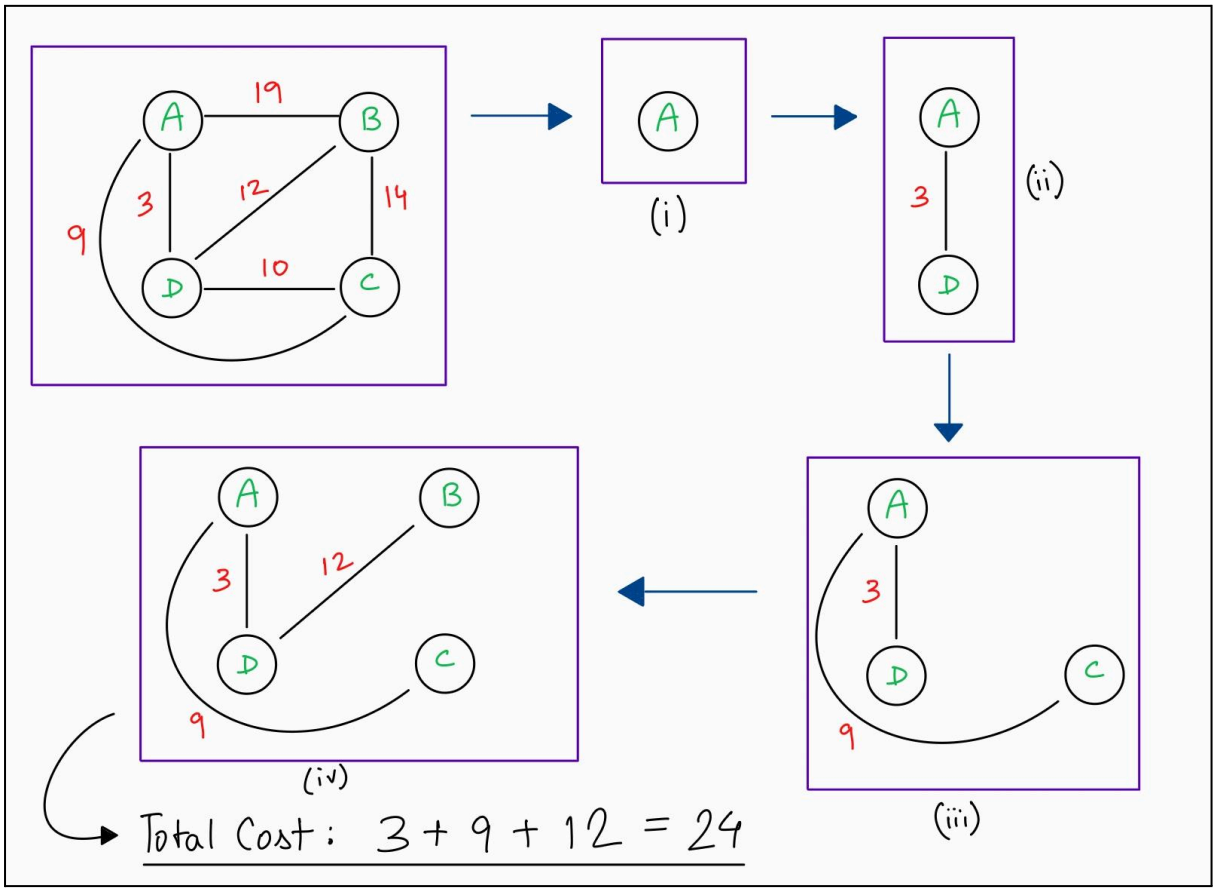
Step 2: Find an edge $(i, j) \in E$ such that vertex 'i' is already present in T and vertex 'j' is not yet included, and the cost of (i, j) is minimum or equal among all the edges (k, l) such that 'k' is a vertex included in T and 'l' is not. Add this vertex 'j' and edge (i, j) to the growing spanning tree T.

$T := T \cup \{ \{j\}, \{i, j\} \}$

Step 3: Repeat step **(2)** until all the vertices from set V are added to the tree T [i.e V' = V where $V \in G$, $V' \in T$].

Step 4: Evaluate the total cost of T.

Prim's Dry Run:



We are given an weighted-connected graph G(V, E), where V = {A, B, C, D} and E = { {A, B}, {A, D}, {B, C}, {B, D}, {C, D} }. So, |V| = 4 and |E| = 6. The MST T(V', E') has |V'| = |V| and |E'| = |V|-1.

Now we begin the process of creating the MST using Prim's algorithm.

- First, we select vertex 'A' lexicographically from set V and add it into T [i.e. step (i)].
  T = { {A}, Φ }

- Then we choose edge {A, D} as vertex 'A' is a part of T and vertex 'D' isn't. This and the fact that edge {A, D} has the minimum cost amongst edges { {A, B}, {A, D}, {A, C} }, we add this edge {A, D} and vertex 'D' into T. [i.e. step (ii)].
  T = { {A, D}, { {A, D} } }

- Then we choose edge {A, C} as vertex 'A' is a part of T and vertex 'C' isn't. This and the fact that edge {A, C} has the minimum cost amongst edges { {A, B}, {A, C}, {D, B}, {D, C} }, we add this edge {A, C} and vertex 'C' into T. [i.e. step (iii)]. (*Note we did not consider the edge {A, D} as both vertices 'A' and 'D' were part of T)*
  T = { {A, D, C}, { {A, D}, {A, C} } }

- Then we choose edge {D, B} as vertex 'D' is a part of T and vertex 'B' isn't. This and the fact that edge {D, B} has the minimum or equal cost amongst edges { {A, B}, {D, B}, {C, B} }, we add this edge {D, B} and vertex 'B' into T. [i.e. step (iv)]. (*Note we did not consider the edges {{A, D}, {A, C}} as vertices 'A', 'C' and 'D' were part of T)*
  T = { {A, D, C, B}, { {A, D}, {A, C} {D, B} } }

- Total cost = cost {A, D} + cost {A, C} + cost {D, B}
  = 3 + 9 + 12 = **24**

If we select the right edge faster, we can implement Prim's algorithm efficiently. This can be achieved using a min-priority queue implemented as a binary min-heap. The graph is stored inside an Adjacency List. Here is pseudocode using a min-priority queue.

Prim's Pseudocode

G: Given Graph (V, E).
root: root node of generated MST.
π: parent of any vertex 'v' ∈ MST.V
status: stores the status of a vertex, either unvisited or processed (i.e. included in MST).
Queue: Min-Priority Queue, implemented using a binary min-heap.
EXTRACT-MIN(Queue): Remove and return the vertex having minimum key value (i.e. delete root node from min-heap).
DECREASE-KEY(Queue, v, cost(u, v)): Decrements the Vertex's key value. This might violate the min-heap property, so we traverse from this node(vertex) towards the root to find a proper place for this node.

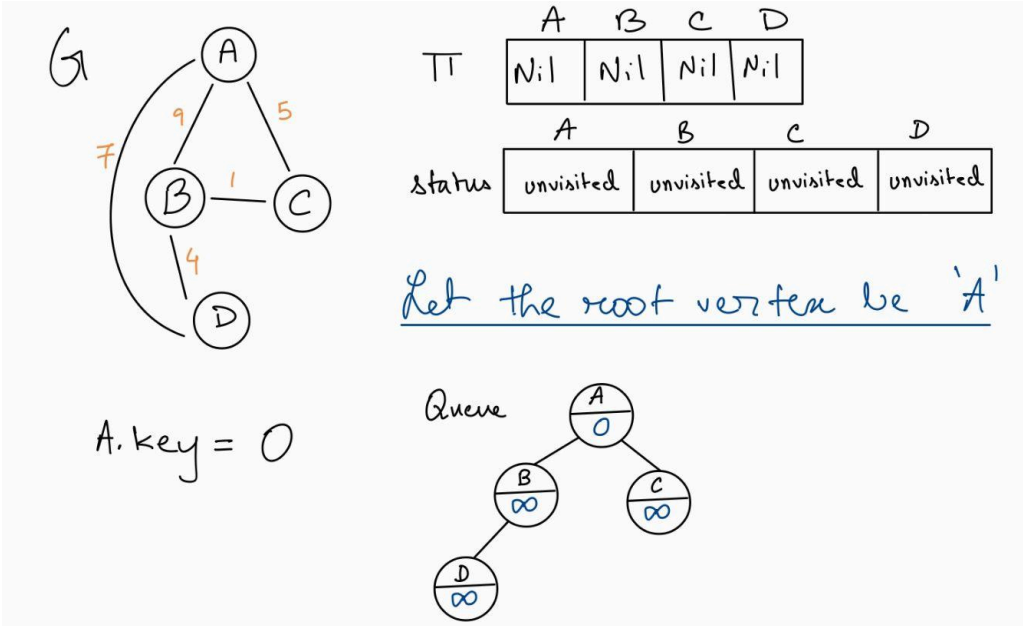
MST-PRIM(G, root)

```
1.      for each u ∈ G.V
2.          u.key = infinity
3.          u.π = NIL
4.          u.status = unvisited
5.      root.key = 0
6.      Queue = G.V
7.      while(Queue ≠ ∅)
8.          u = EXTRACT-MIN(Queue)
9.          for each v ∈ G.Adj[u]
10.             if v.status = unvisited and cost(u, v) < v.key
11.                 v.π = u
12.                 DECREASE-KEY(Queue, v, cost(u, v))
13.         u.status = processed
```

So the parent π stores the MST upon the termination of the algorithm. For any vertex 'v' ∈ V, v.π is its parent, and that edge's cost is cost(v.π, v).

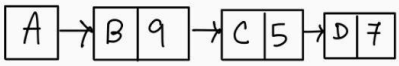
Here is a dry run of the above pseudocode

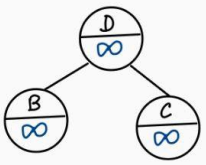
First, we create the parent, status and Queue (i.e. lines 1-6).



Iteration 1

u = EXTRACT-MIN(Queue) ⇒ u = A





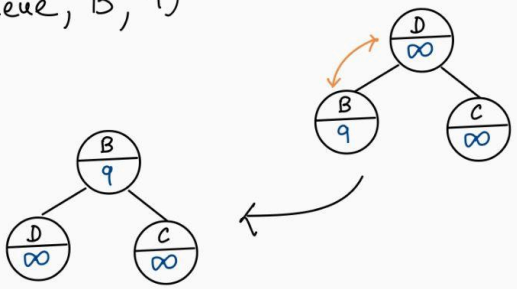
for vertex 'B'

B.status = unvisited and cost(A, B) < B.key (9 < ∞)

B.π = A

DECREASE-KEY(Queue, B, 9)

π	A	B	C	D
	Nil	A	Nil	Nil



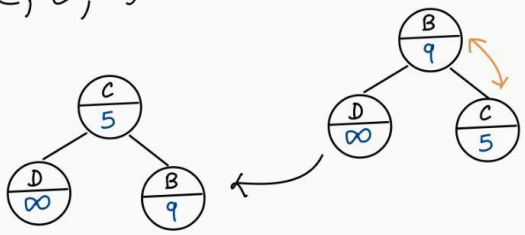
for vertex 'C'

C.status = unvisited and cost(A, C) < C.key (5 < ∞)

C.π = A

DECREASE-KEY(Queue, C, 5)

π	A	B	C	D
	Nil	A	A	Nil



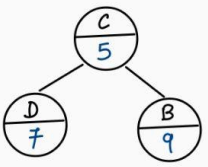
for vertex 'D'

D.status = unvisited and cost(A, D) < D.key (7 < ∞)

D.π = A

DECREASE-KEY(Queue, D, 7)

π	A	B	C	D
	Nil	A	A	A

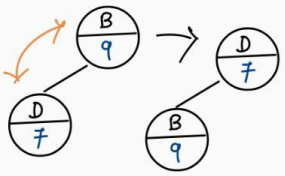


A.status = processed

status	A	B	C	D
	processed	unvisited	unvisited	unvisited

Iteration 2

u = EXTRACT-MIN(Queue) ⇒ u = C



for vertex 'A'

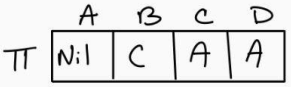
A.status = processed, so skip

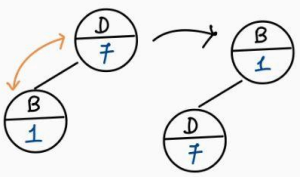
for vertex 'B'

B.status = unvisited and cost(C, B) < B.key (1 < 9)

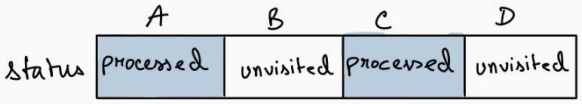
B.π = C

DECREASE-KEY(Queue, B, 1)



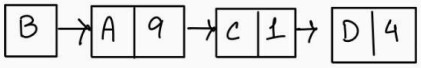


C.status = processed



Iteration 3

u = EXTRACT-MIN(Queue) ⇒ u = B



for vertex 'A'

A.status = processed, so skip

for vertex 'C'

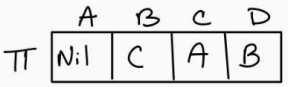
C.status = processed, so skip

for vertex 'D'

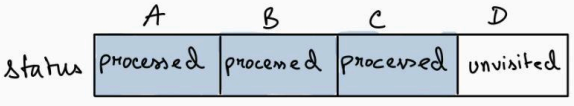
D.status = unvisited and cost(B, D) < D.key (4 < 7)

D.π = B

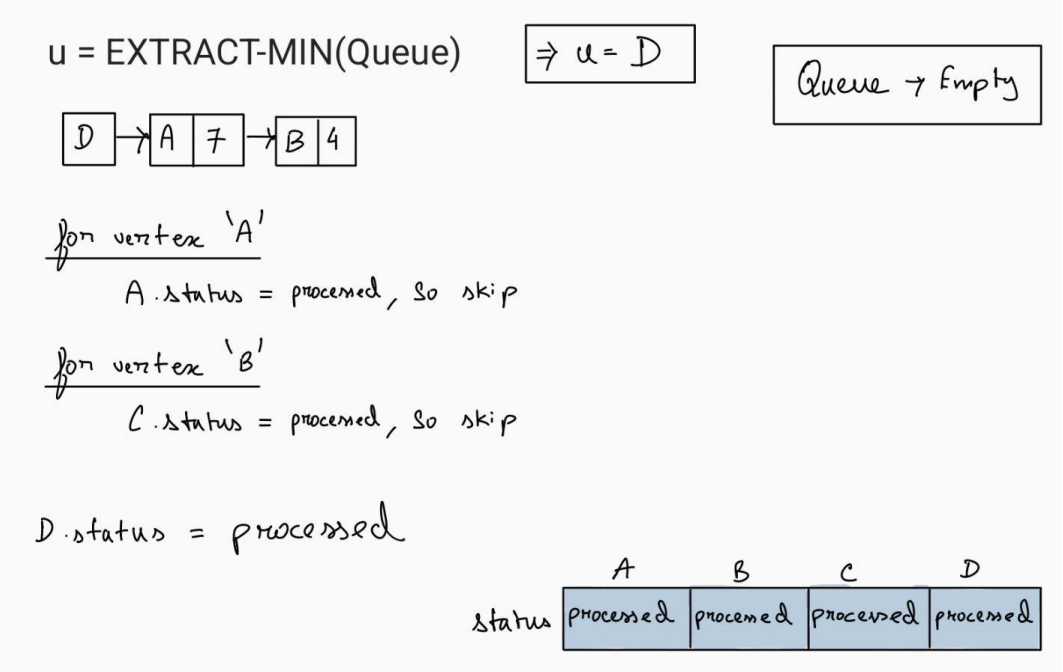
DECREASE-KEY(Queue, D, 4)



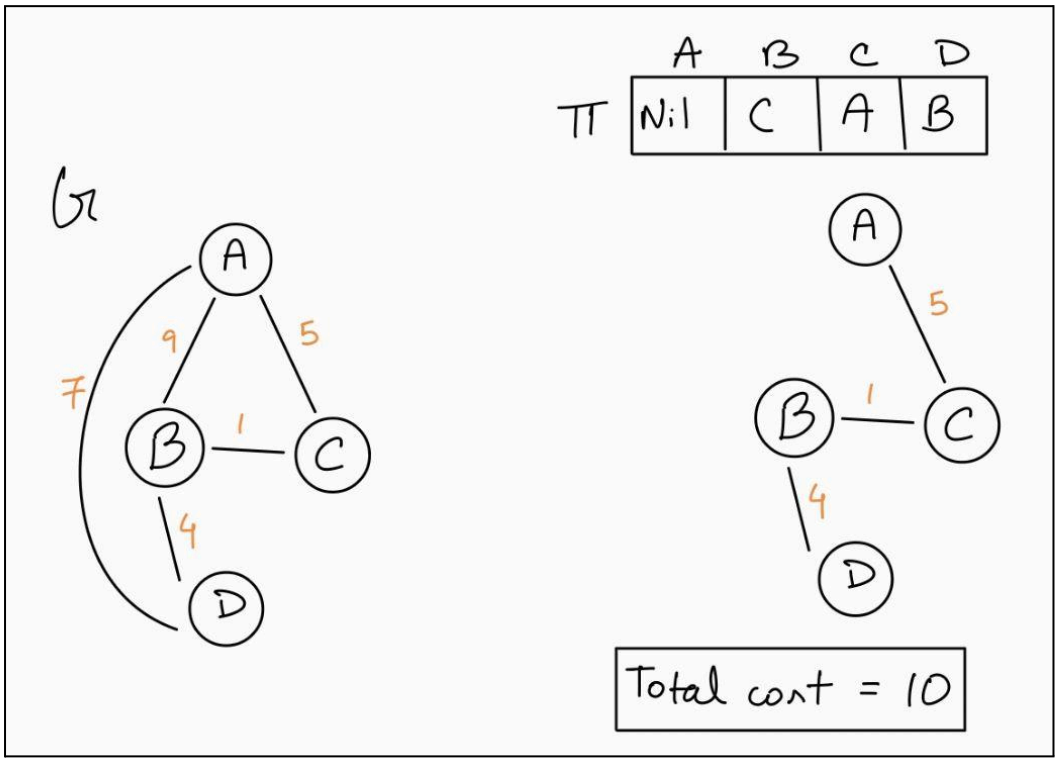
B.status = processed



Iteration 4



The algorithm terminates as the Queue has become empty.



The MST has been created of Graph G with a total cost of 10.

Time complexity analysis of the above Prim's pseudocode.

Line 1: The for loop iterates for |V| times. So everything inside this loop will iterate |V| times their own time.

Line 2: Marking the key of vertex 'u' takes constant time. So this has a complexity of **O(|V|)**.
Line 3: Marking the parent (π) of vertex 'v' takes constant time. So this has a complexity of **O(|V|)**.
Line 4: Marking the status of vertex 'v' takes constant time. So this has a complexity of **O(|V|)**.

Line 5: Marking the key of the root vertex 'root' takes constant time, so **O(1)**.

Line 6: Building the Min-Priority queue using Binary Min-Heap takes **O(|V|)** time.

Line 7: The while loop will iterate for |V| times. So everything inside it will iterate |V| times their own time.

Line 8: Extracting the Minimum from the priority queue will take **O(|V| * log |V|)** time.
Line 9: The for loop will execute for a total of 2|E| times (The adjacency list stores each edge twice. for undirected graphs). So everything inside it will iterate 2|E| times their own time.

Line 10: Checking the status of if vertex 'v' and the comparison of cost(u, v) with v.key can be done in constant time. So this has a complexity of **O(|E|)**.
Line 11: Marking the parent (π) of vertex 'v' also takes constant time. So this line has a complexity of **O(|E|)**.
Line 12: Updating the key of vertex 'v' with cost(u, v) using the DECREASE-KEY operation takes **O(|E| * log |V|)** time.
Line 13: Marking the status of vertex 'v' takes constant time. So this has a complexity of **O(|V|)**.

Summing up all the time complexities, we get

= $|V| + |V| + |V| + 1 + |V| + |V| \log |V| + 2|E| + 2|E| + 2|E| \log |V| + |V|$
= $1 + 5|V| + 4|E| + |V| \log |V| + 2|E| \log |V|$
= **O(|E| log |V|)**

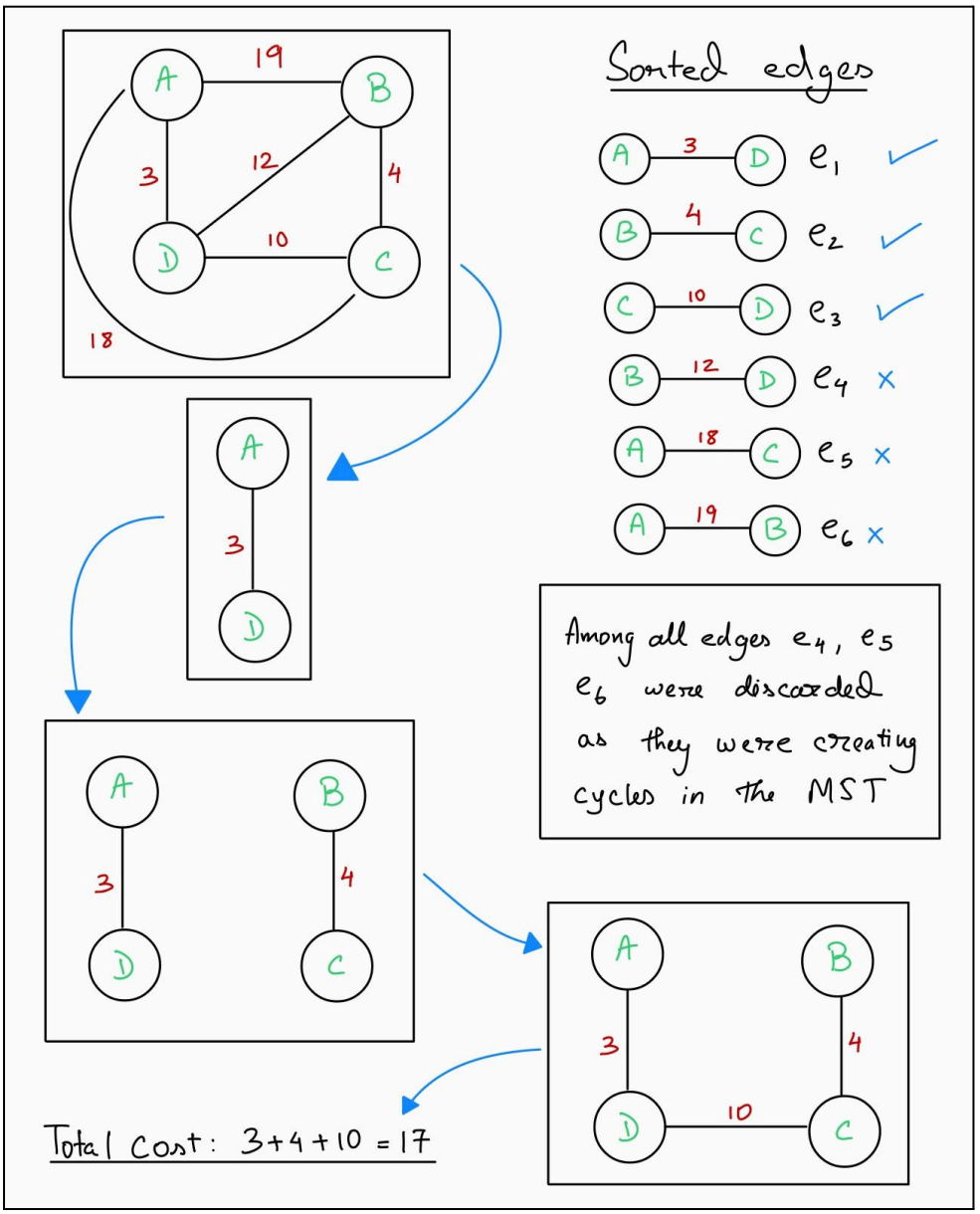
Kruskal's Algorithm

Kruskal's algorithm sorts all the edges of the graph into a non-decreasing order based on their cost, and then it selects and adds a safe edge from the sorted list which will not create a cycle in the growing forest. The algorithm begins with all vertices of the input graph as an individual disconnected component and then it builds the minimum cost spanning tree.

Input: A weighted connected graph G(V, E) with V vertices and E edges.
Output: A minimum cost spanning tree T with |V| vertices and |V|-1 edges.

Step 1: Add all the vertices v_i into the MST T. So T has |V| components. [$v_i \in V$ and $1 \leq i \leq |V|$]
Step 2: Sort all the edges e_i based on their cost in non-decreasing order. [$e_i \in E$ and $1 \leq i \leq |E|$]
Step 3: Choose the minimum cost edge e_i from the list such that it only connects two disconnected components in the MST T. Add it to the MST T.
Step 4: Repeat step (**3**) for all the edges e ∈ E.
Step 5: Evaluate the total cost of T.

Kruskal's Dry Run



Kruskal's pseudocode

To make Kruskal's algorithm work faster, we will store the Graph in an adjacency list, sorting edges based on their cost must be done using sorting algorithms like Merge-Sort, and we will use a disjoint data set structure to detect cycles quickly.

G: The input Graph G(V, E)
T: It stores the set of edges

MST-KRUSKAL(G)

```
1.  T = ϕ
2.  for each vertex v ∈ G.V
3.          MAKE-SET(v)
4.  sort all the edges of G.E into non-decreasing order based on their cost
5.  for each edge (u, v) ∈ G.E, taken in non-decreasing order by cost
6.          if FIND-SET(u) ≠ FIND-SET(v)
7.                  T = T ∪ {(u, v)}
8.                  UNION(u, v)
9.  return T
```

So set A stores all the edges of the MST. Each edge (u, v) ∈ A has a cost(u, v).

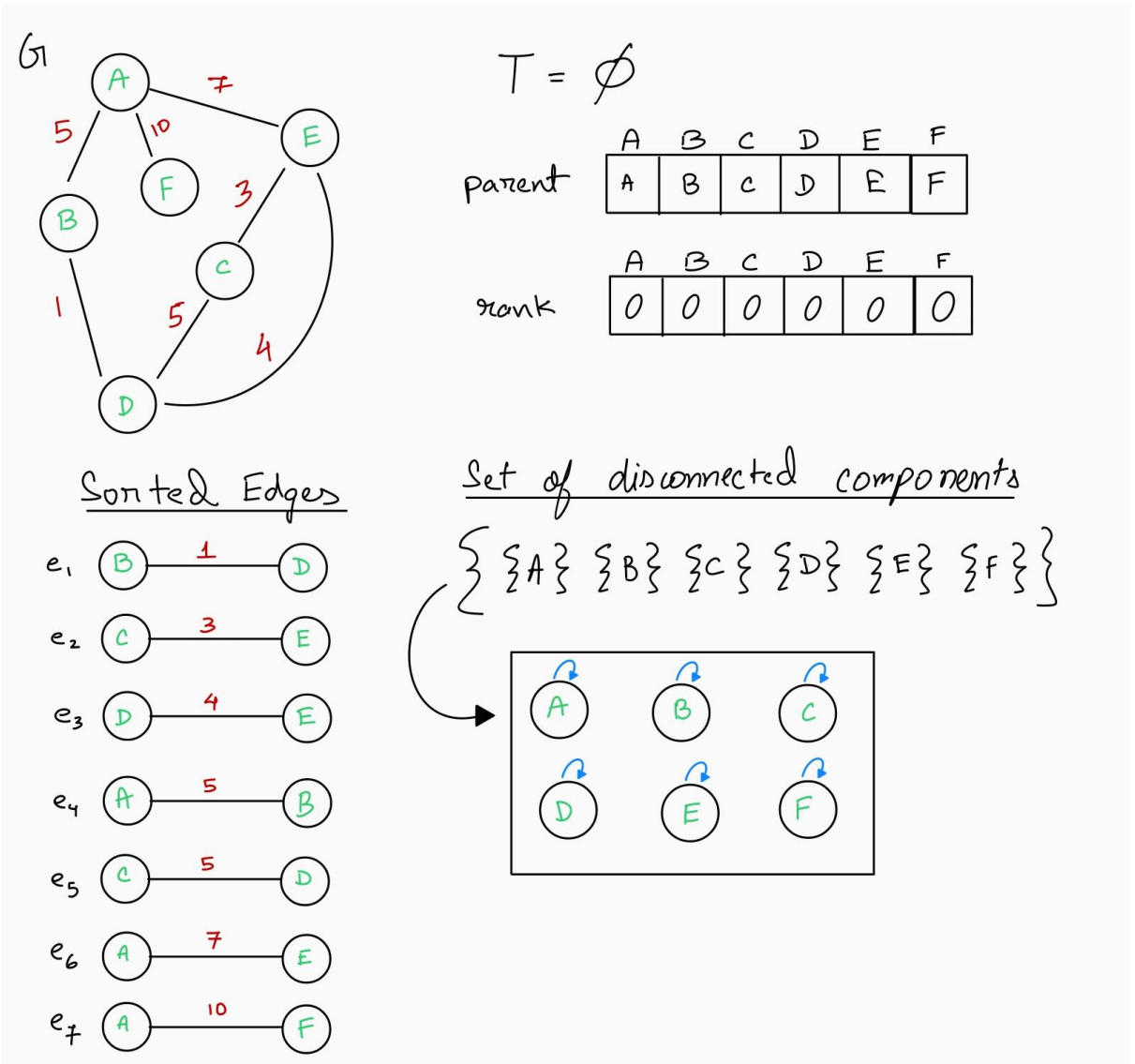
```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2       y.p = x
3  else x.p = y
4       if x.rank == y.rank
5            y.rank = y.rank + 1

FIND-SET(x)
1  if x ≠ x.p
2       x.p = FIND-SET(x.p)
3  return x.p
```

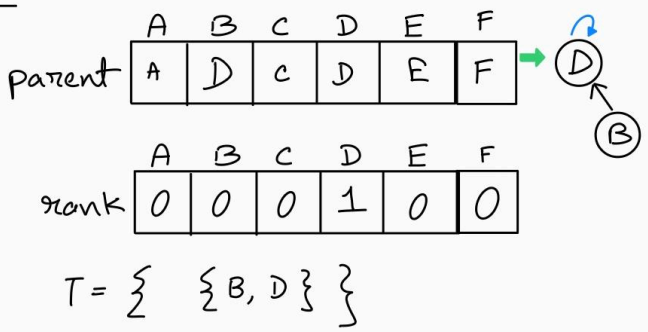
Dry Run of the above Kruskal's pseudocode



Edge e_1 (B, D) selected

Iteration : 1

FIND-SET(B) → B
FIND-SET(D) → D
So B ≠ D
T = T ∪ {(B, D)}
UNION (B, D)

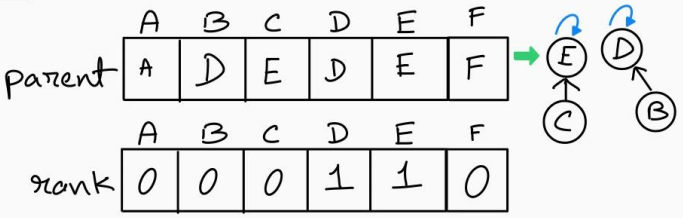


T = { {B, D} }

Edge e_2 (C, E) selected

Iteration : 2

FIND-SET(C) → C
FIND-SET(E) → E
So C ≠ E
T = T ∪ {(C, E)}
UNION (C, E)

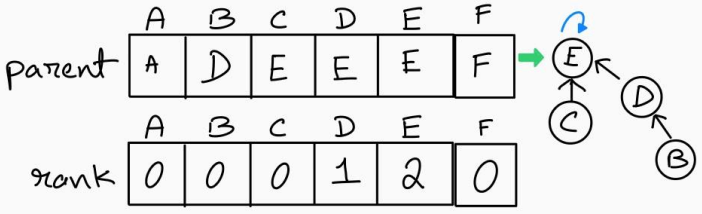


T = { {B, D}, {C, E} }

Edge e_3 (D, E) selected

Iteration : 3

FIND-SET(D) → D
FIND-SET(E) → E
So D ≠ E
T = T ∪ {(D, E)}
UNION (D, E)

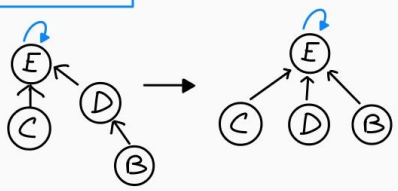


T = { {B, D}, {C, E}, {D, E} }

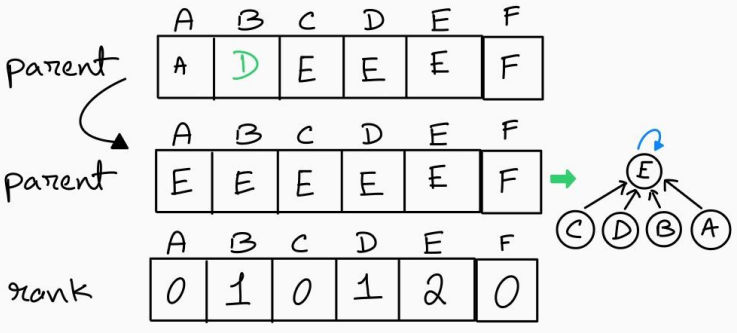
Edge e_4 (A, B) selected

Iteration : 4

FIND-SET(A) → A
FIND-SET(B) → E



So A ≠ E
T = T ∪ {(A, B)}
UNION (A, B)



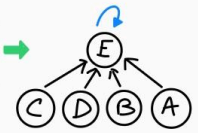
T = { {B, D}, {C, E}, {D, E}, {A, B} }

Edge e_5 (C, D) selected

Iteration : 5

FIND-SET(C) → E
FIND-SET(D) → E
So E ≠ E is false

	A	B	C	D	E	F
parent	E	E	E	E	E	F
rank	0	1	0	1	2	0



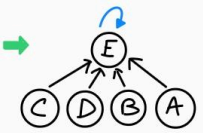
$T = \{ \{B, D\}, \{C, E\}, \{D, E\}, \{A, B\} \}$

Edge e_6 (A, E) selected

Iteration : 6

FIND-SET(A) → E
FIND-SET(E) → E
So E ≠ E is false

	A	B	C	D	E	F
parent	E	E	E	E	E	F
rank	0	1	0	1	2	0



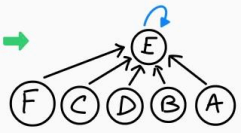
$T = \{ \{B, D\}, \{C, E\}, \{D, E\}, \{A, B\} \}$

Edge e_7 (A, F) selected

Iteration : 7

FIND-SET(A) → E
FIND-SET(F) → F
So E ≠ F
T = T ∪ {(A, F)}
UNION (A, F)

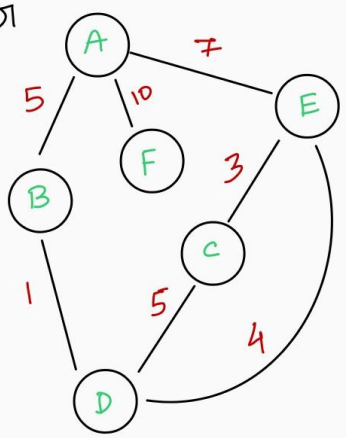
	A	B	C	D	E	F
parent	E	E	E	E	E	E
rank	0	1	0	1	2	0



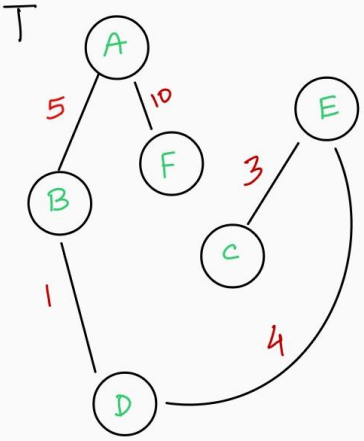
$T = \{ \{B, D\}, \{C, E\}, \{D, E\}, \{A, B\}, \{A, F\} \}$

$T = \{ \{B, D\}, \{C, E\}, \{D, E\}, \{A, B\}, \{A, F\} \}$

G



T



Total Cost = 1 + 3 + 4 + 5 + 10 = 23

Time complexity analysis of the above Kruskal's pseudocode

Line 1: Initializing set A takes constant time so **O(1)**.
Line 2: The for loop iterates for |V| times. So everything inside it will iterate |V| times their own time.
    Line 3: MAKE-SET(v) operation takes constant time so a total of **O(|V|).**
Line 4: Sorting all the edges using the Merge Sort algorithm will take **O(|E| log |E|)** time.
Line 5: The for loop iterates for |E| times. So everything inside it will iterate |E| times of their own time.
    Line 6: The two FIND-SET operations will take 2 * α(|V|) time, so a total of **O(|E| * α(|V|)).**
    Line 7: Adding edge (u, v) to set A takes constant time so a total of **O(|E|)**.
    Line 8. The UNION operation will take 2 * α(|V|) time, so a total of **O(|E| * α(|V|)).**
Line 9: Returning the set A also take constant time so **O(1)**.

[*α(n) is a very slow-growing function. For large values of n, α(n) is very very small, almost a constant****]***

Summing up all the time complexities, we get

= 1 + |V| + |E| log |E| + |E|.α(|V|) + |E| + |E|.α(|V|) + 1
= |E| log |E| + 2.|E|.α(|V|) + |E| + |V| + 2
= O(|E| log |E|)

Now a vertex 'v' can be at most connected to |V|-1 vertices. So |E| = |V|*(|V|-1), then O(|E|) = O($|V|^2$).
So, O(|E| log |E|) becomes O(|E| log $|V|^2$)

= |E| log $|V|^2$
= |E| * 2 * log |V|
= 2 * |E| log |V|

Therefore, Kruskal's algorithm has the same running time complexity as Prim's algorithm, ***O(|E| log |V|).***

Note: For sparse graphs |E| << |V|, then the running time complexity is O(|E| log |E|) therefore much more efficient.

Real-Life Applications of MST:

1. Circuit Design: In Very Large Scale Integration (VLSI) design, MSTs help minimise the total length of wires connecting different components on a chip.

2. Computer Network Design: Creating cost-efficient Local Area Networks (LANs) or other communication networks between multiple computers while minimising the number of cables or connections.

3. Water Supply Systems: Designing efficient water pipeline systems in cities or irrigation projects to minimise construction costs.

There are many more implementations of MST in our world. MSTs provide an optimal solution to connect entities with minimal cost, making them incredibly useful in scenarios involving efficiency, resource management, and cost savings.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). The MIT Press.

Dr. Rajat Kr. Pal, Professor, University of Calcutta for notes on Heaps and Disjoint Data Set Structure.

HUFFMAN CODING

In computer science and information theory, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression.

The process of finding or using such a code is Huffman coding, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

Example:

Suppose we have a message M of length 100 that we want to encode. Let us assume that only 6 characters appear in the message. The count of the occurrence i.e. the frequency of each character in M be as follows:

Character	*a*	*b*	*c*	*d*	*e*	*f*
Frequency	45	13	12	16	9	5

Ways to encode the data:

➢ FIXED LENGTH CODE:

To convert the message M to bits, we will need a combination of minimum 3bits to uniquely identify each character as there are 6 characters in total.

a	000
b	001
c	010
d	011
e	100
f	101

Therefore, total no. of bits required to represent the message M = 100 * 3 bits = 300 bits.

❖ Another way could be to represent each character by their ASCII value. Thus, the ASCII value of each character would take 8bits.
Total no. of bits = 100 * 8 bits = 800 bits.

➢ VARIABLE LENGTH CODE

In this type of coding, frequent characters are given shorter codewords and infrequent characters long codewords.

Now since each character will be represented by varying no. of bits, it might become ambiguous to decode the encoded data.

EXAMPLE: X= “abcaab”

Encoding X using non-prefix codes,

Say the characters in X are represented in bits as follows:

a	0
b	01
c	10

Encoded message X’ = 001100001

The encoding could be done very easily but when we try to decode the encoded data X' ambiguity arises as we do not have a fixed no. bits representing a character.

However, if we consider only codes in which no codeword is a prefix of some other codeword, i.e. **prefix codes**, then decoding becomes unambiguous.

Encoding X using prefix codes,

Say the characters in X are represented in bits as follows:

a	0
b	10
c	11

Encoded message X' = 010110010

The encoding could be done very easily and we can also decode the encoded data X' without ambiguity.

Identify the initial codeword, translate it back to the original character and continue this till the end. Since, we have no codeword with the same prefix there is no ambiguity while decoding.

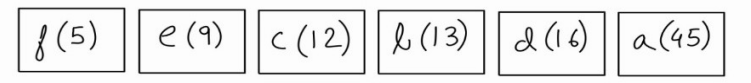
Huffman code is a variable length prefix code.

CONSTRUCTING HUFFMAN CODE

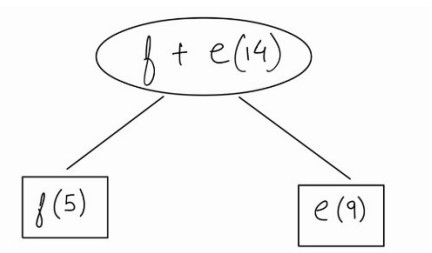
1. Create a list containing all the characters, in the data, sorted in non-decreasing order of their frequencies.
2. Extract the first 2 elements from the sorted list i.e. elements with the lowest frequencies (say ***x*** and ***y***). Create a node ***z*** of a binary tree ***T*** such that left child of ***z*** is ***x*** and the right child of ***z*** is ***y***. The frequency stored in ***z*** will the sum of frequencies of ***x*** and ***y***. Now, ***x*** and ***y*** will be treated as a single unit ***z***.
3. Add this node ***z*** to the existing list in sorted order.
4. Repeat step 2 and 3 until the list is empty.
5. Once the final Huffman tree T is created, assign 0 to all its left edges and 1 to all the right edges of the tree. Huffman tree will have the characters as its leaves.
6. The path from root to leaf node (i.e. a character) will give the codeword for that character. The 0 and 1 in the path are concatenated as they are traversed to get the codeword for that character and stored in a conversion table.

Constructing Huffman code for the example message M

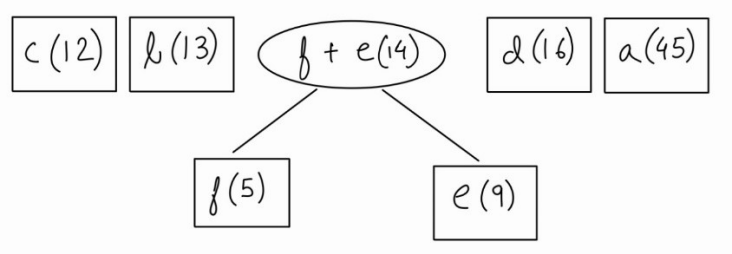
1. Create the initial sorted list.



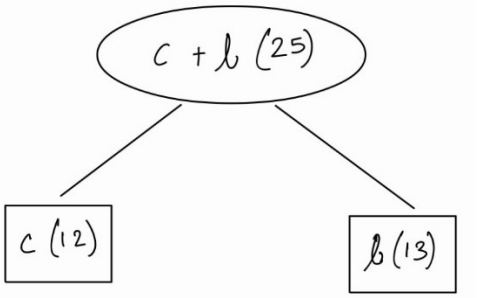
2. Extract f(5) and e(9) from the list and create a node f+e(14).



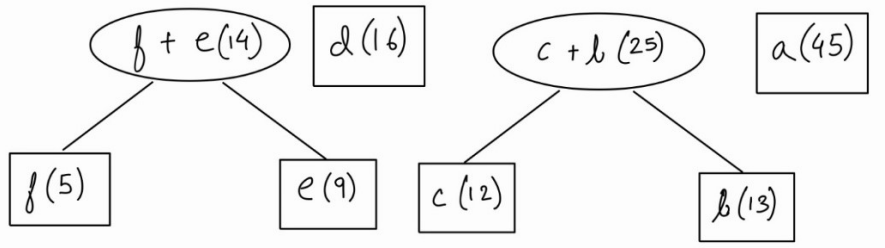
3. Add node f+e(14) to the list.



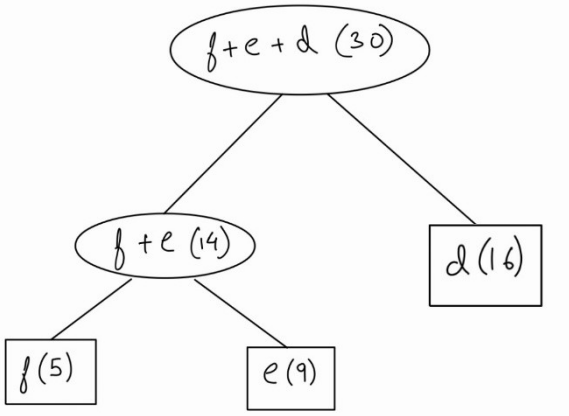
4. Extract c(12) and b(13) from the list and create a node c+b(25).



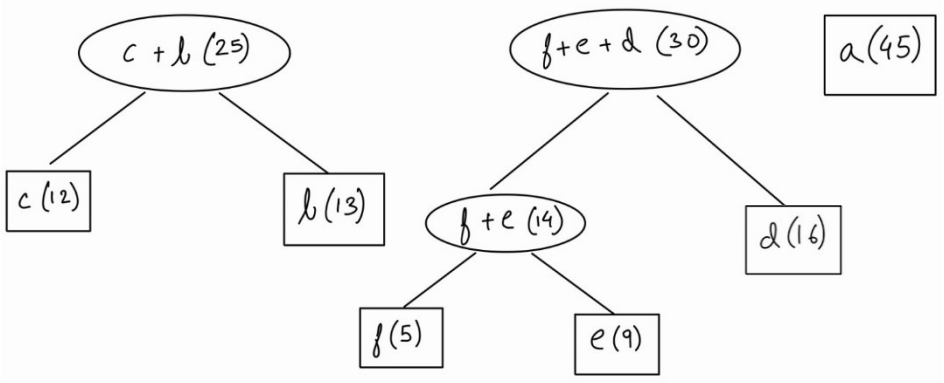
5. Add node c+b(25) to the list.



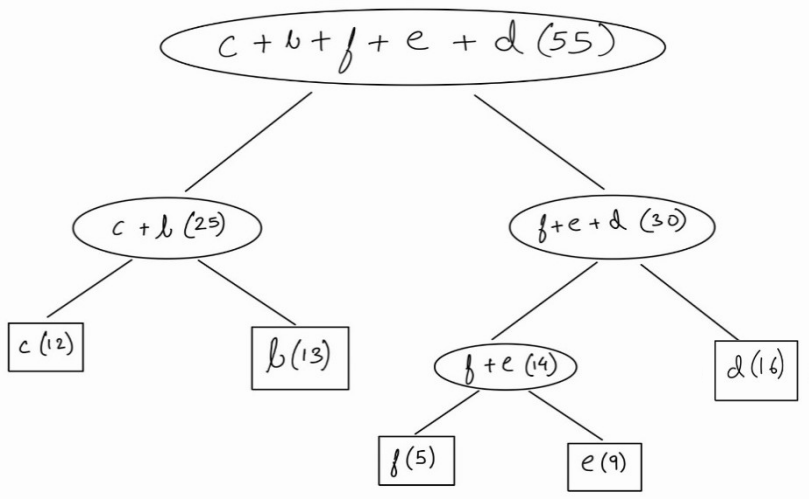
6. Extract f+e(14) and d(16) from the list and create a node f+e+d(30).



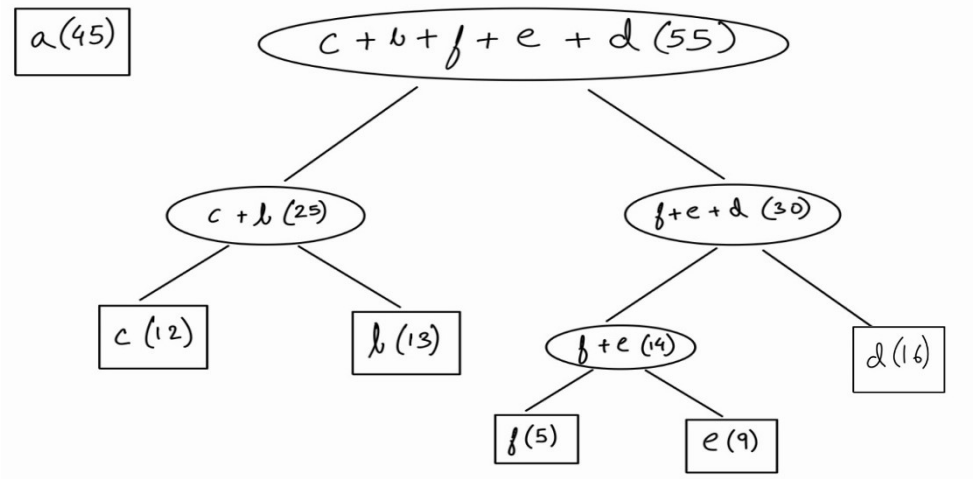
7. Add node f+e+d(30) to the list.



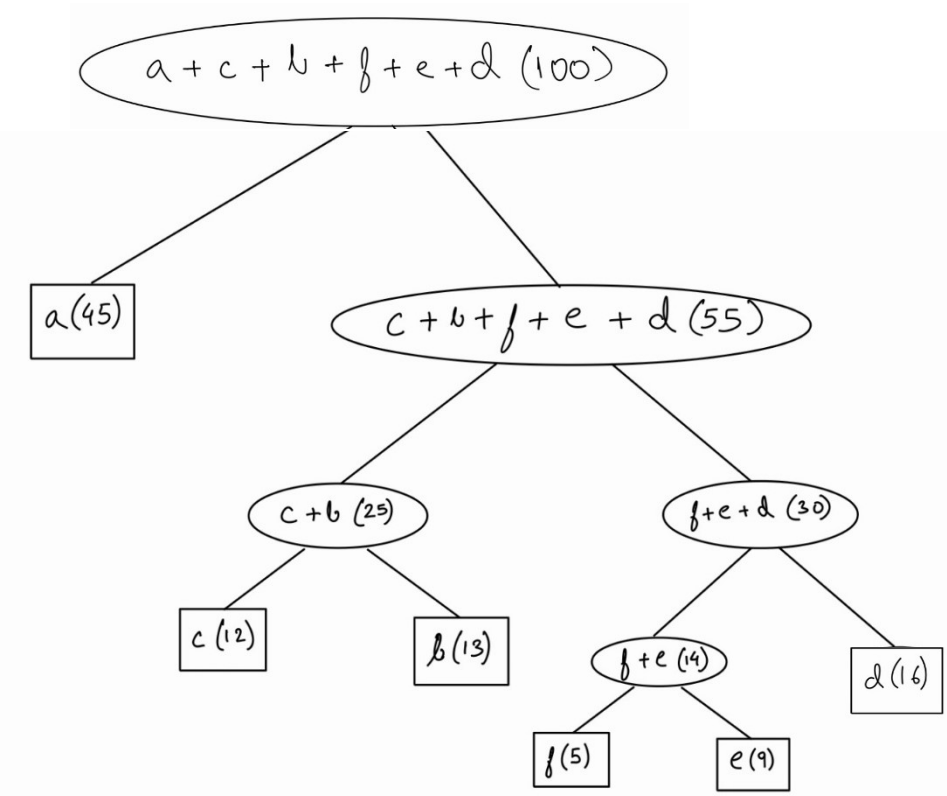
8. Extract c+b(25) and f+e+d(30) from the list and create a node c+b+f+e+d(55).



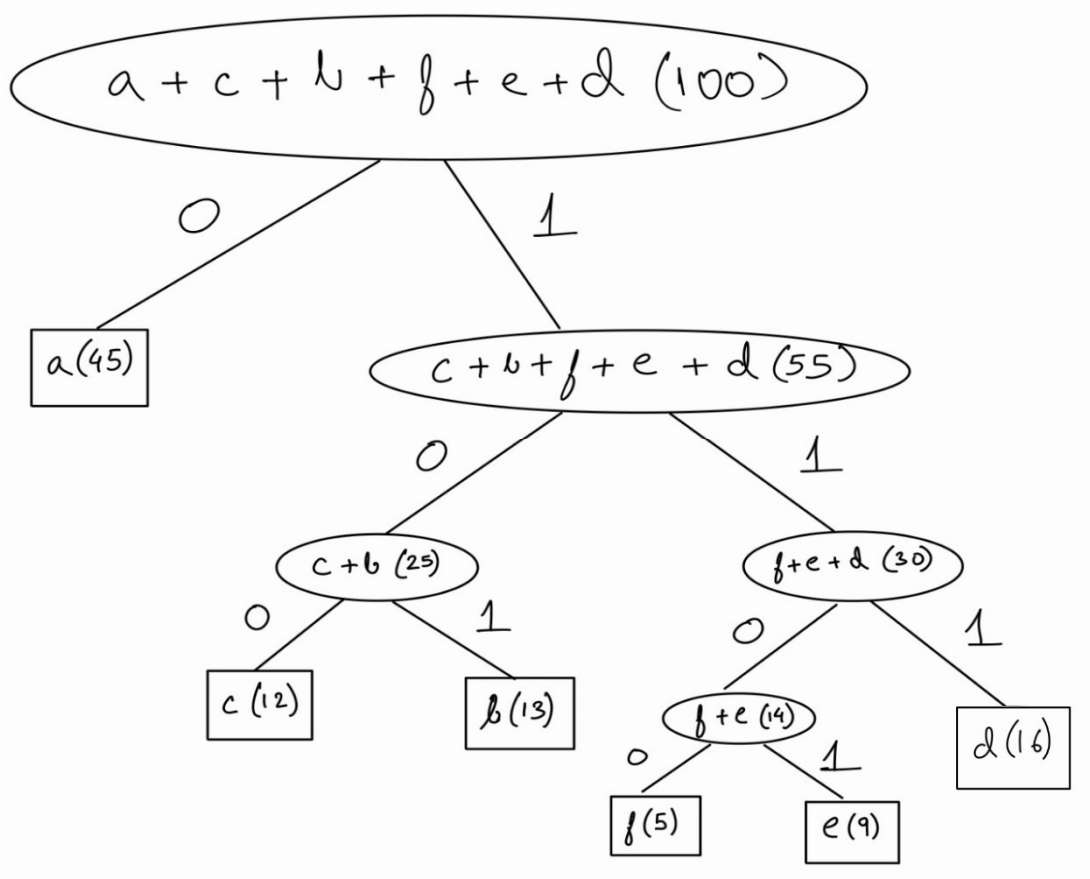
9. Add node c+b+f+e+d(55) to the list.



10. Extract a(45) and c+b+f+e+d(55) from the list and create a node a+c+b+f+e+d(100).



11. Assign 0 to the left edges and 1 to right edges of the created tree.



12. Obtain the codeword for each character.

***Conversion Table*:**

Character	*Codeword*	*No. of bits required*
a	0	1
b	101	3
c	100	3
d	111	3
e	1101	4
f	1100	4

Therefore,

Total no. of bits required to represent the message M

= (45 * 1 bit) + (13 * 3 bits) + (12 * 3 bits) + (16 * 3 bits) + (9 * 4 bits) + (5 * 4 bits)

= 224 bits.

Thus, the message can be represented using lesser no. of bits than before, indicating that the data is compressed.

Since, the original message can be easily obtained from the encoded data without any information loss, it is a lossless compression technique.

❖ Note that for decoding the encoded code either the Huffman tree or the conversion table is needed. This requires the table to also be included along with the encoded data. Thus, some extra bits will be used for the same.

For this example, message M, the conversion table will take:

(6 * 8 bits) + (6 * 4 bits) = 72 bits.

[considering character takes 8 bits and 4 bits is the maximum length of a codeword]

The actual no. of bits required to represent M = 224 + 72 bits = 296 bits

❖ Therefore,

The actual no. of bits required for any data = Encoded Data + Overhead for Conversion Table

❖ When large data is compressed, this overhead space becomes insignificant. Thus, Huffman code is an efficient ***lossless data compression algorithm*** as there is no loss information due to the encoding and decoding process.

PSEUDOCODE

- **C** is a set of **n** characters.
- Each character **c ∈ C** is an object with an attribute **c.freq** giving its frequency.
- **Q** is a min-priority queue Q implemented using binary min-heap, keyed on the **freq** attribute.
- **z** is a node of the tree.
- ***z.left*** is the left child of **z**.
- ***z.right*** is the right child of **z**.
- ***EXTRACT-MIN (Q):*** removes and returns the element of **Q** with the smallest key.
- ***INSERT (Q, z):*** Inserts element **z** in the set **Q** which is equivalent to the operation Q= Q ∪ {z}
- ***HUFFMAN (C):*** Returns the Huffman tree for the given set of characters **C**.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

COMPLEXITY ANALYSIS

For a set C of n characters, we can initialize Q in line 2 in O(n) time. [Building a min heap from an array]

The for loop in lines 3–8 executes exactly n-1 times,
line 4 takes constant time i.e. O(1)
line 5 takes O(lg n) time (extraction from heap)
line 6 also takes O(lg n) as it is the same heap operation as line 5
line 7 takes constant time i.e. O(1)
line 8 takes O(lg n) time (insertion in heap)

Total running time inside the loop = O(1) + O(lg n) + O(lg n) + O(1) + O(lg n) = O(lg n)

Thus the loop contributes O(n lg n) to the running time.

Thus, the total running time of HUFFMAN on a set of n characters is O(n) + O(n lg n) = O(n lg n).

Observations:

- Given a tree T corresponding to a prefix code, we can easily compute the number of bits required to encode a file. For each character c in the alphabet C, let the attribute c.*freq* denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree and is also the length of the codeword for character c.

  The no. of bits required to encode a file is thus

  $$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

  which we define as the cost of the tree T.

- Switching the left and right child of any node yields a different code of the same cost.

- Let C be an alphabet in which each character c in C has frequency c.freq. Let x and y be two characters in C having the lowest frequencies.
  Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.
  This implies that the process of building up an optimal tree by mergers can, without loss of generality, begin with the greedy choice of merging together those two characters of lowest frequency.
  Why is this a greedy choice?
  We can view the cost of a single merger as being the sum of the frequencies of the two items being merged. The total cost of the tree constructed equals the sum of the costs of its mergers. Of all possible mergers at each step, HUFFMAN chooses the one that incurs the least cost.