# DESIGN AND ANALYSIS OF ALGORITHMS

## B.Tech (CSE)

## NOTES

**Prepared By-** TOPPERWorld

# UNIT-1

## INTRODUCTION

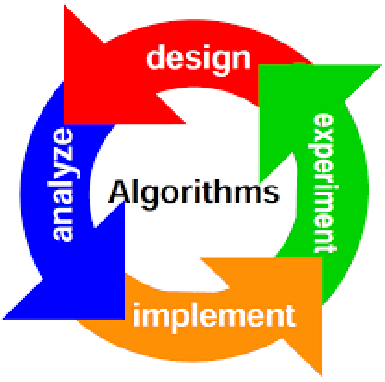
Review:- Elementary Data Structures, Algorithms and its complexity(Time and Space), Analysing Algorithms, Asymptotic Notations, Priority Queue, Quick Sort.

Recurrence relation:- Methods for solving recurrence(Substitution , Recursion tree, Master theorem), Strassen multiplication.

### Design and Analysis of Algorithms:

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.

An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the algorithm is independent from any programming languages.



## What is Algorithm

He word Algorithm means “a process or set of rules to be followed in calculations or other problem-solving operations”. Therefore Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.

## Why study Algorithm ?

The importance of algorithms is very high in today's world but in reality, what we focus on is the result, be it ios apps, android apps, or any other application. The reason we have these resultant applications is the Algorithm. If programming a building, then the algorithm is the pillar programming is standing on, and without pillars, there is no building. But why do we go for algorithms instead of going for the application directly? Let's get that from an example. Let's suppose we are building something, and we have the result in mind. We are not an expert, but still, we bring all the necessary items and design that thing. It also looks like what we had in mind. But it does not fulfill the purpose we built it for. Do we have any use of it? This is what's an algorithm for a program because it provides meaning to the program. There is much reason to study algorithms as it is used in almost every digital application we use today. To showcase the value algorithms have, here we have some of its applications.

## Properties of Algorithm

**All Algorithms must satisfy the following criteria -**

**1) Input**
There are more quantities that are extremely supplied.

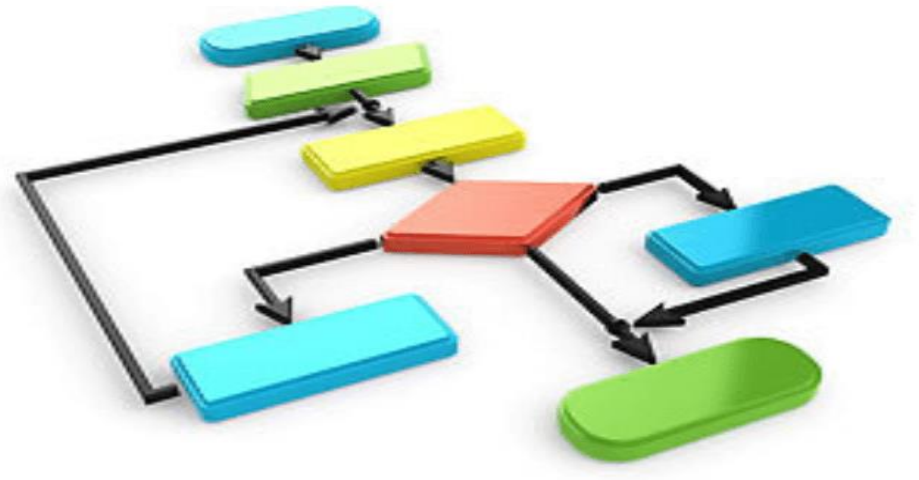
**2) Output**
At least one quantity is produced.

**3) Definiteness**
Each instruction of the algorithm should be clear and unambiguous.

**4) Finiteness**
The process should be terminated after a finite number of steps.

**5) Effectiveness**
Every instruction must be basic enough to be carried out theoretically or by using paper and pencil

*For example,suppose you are cooking a recipe and you chop vegetables which are not be used in the recipe then it is a waste of time.*



6)Independent

An algorithm should have step-by-step directions, which should be independent of any programming code. It should be such that it could be run on any of the programming languages.

## What are the Characteristics of an Algorithm

i) Clear and Unambiguous: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.

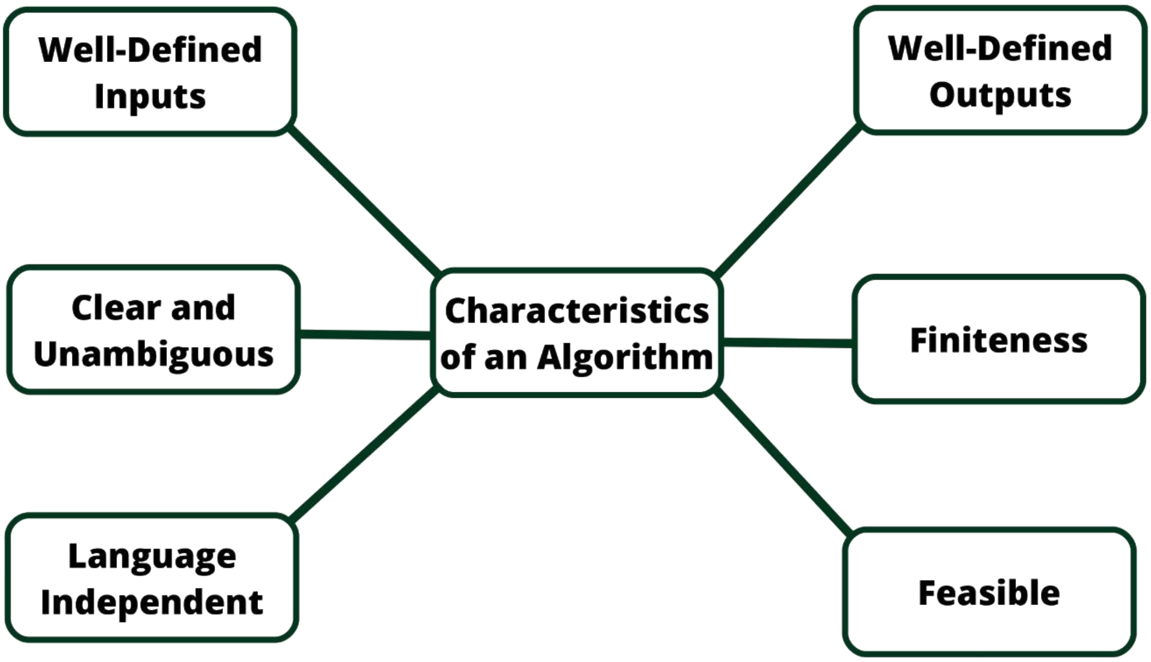
ii) Well-Defined Inputs: If an algorithm says to take inputs, it should be well-defined inputs.

iii) Well-Defined Outputs: The algorithm must clearly define what output will be yielded and it should be well-defined as well.

iv) Finite-ness: The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.

v) Feasible: The algorithm must be simple, generic and practical, such that it can be executed upon with the available resources. It must not contain some future technology, or anything.

vi) Language Independent: The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.



## Advantages of Algorithms:

i)It is easy to understand.

ii) Algorithm is a step-wise representation of a solution to a given problem.

iii)In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

## Disadvantages of Algorithms:

i)Writing an algorithm takes a long time so it is time-consuming.

ii)Branching and Looping statements are difficult to show in Algorithms.

## Performance Analysis of Algorithm

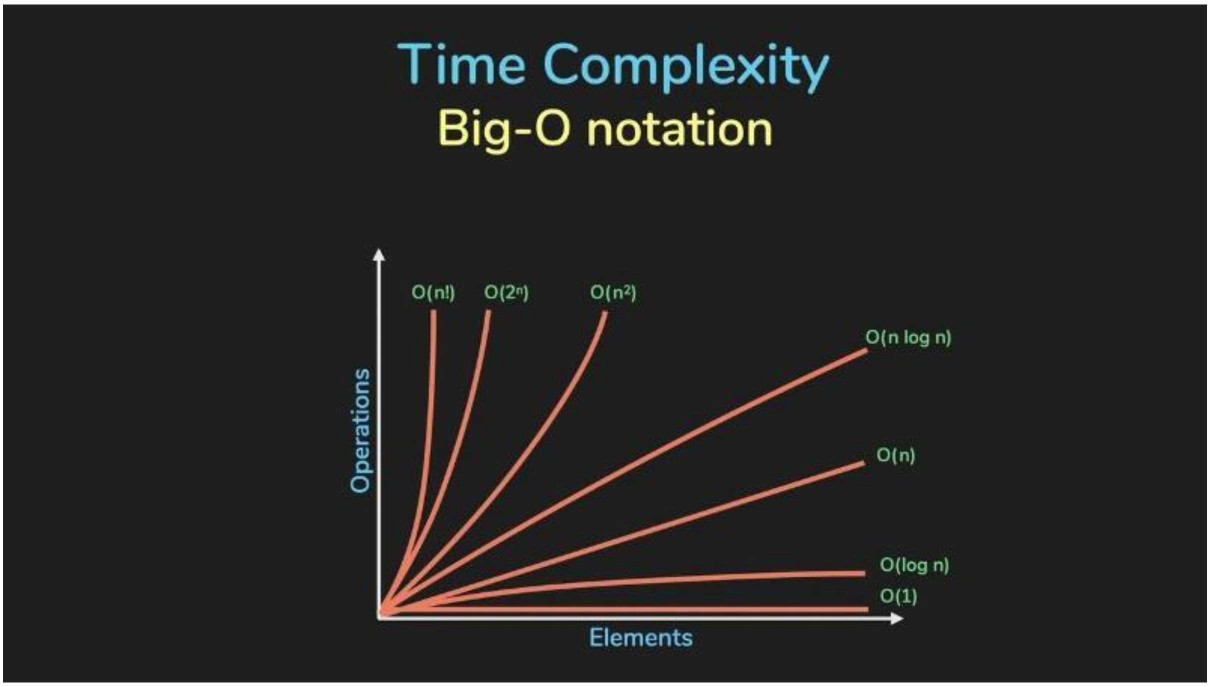
There are two types are:

i) Time Complexity
ii) Space Complexity

## Time Complexity

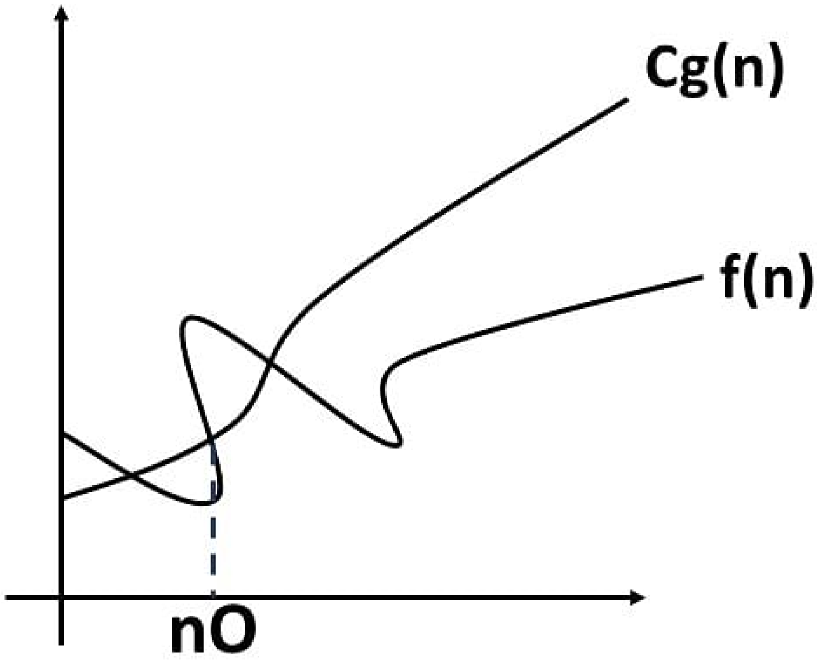
**Time complexity** is the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm. Upskilling with the help of an introduction to algorithms free course will help you understand time complexity clearly.

.



## What Is Space Complexity?

When an algorithm is run on a computer, it necessitates a certain amount of memory space. The amount of memory used by a program to execute it is represented by its space complexity. Because a program requires memory to store input data and temporal values while running, the space complexity is auxiliary and input space.



$$f(n) = O(g(n))$$

## Analyzing Algorithm

To analyze a programming code or algorithm, we must notice that each instruction affects the overall performance of the algorithm and therefore, each instruction must be analyzed separately to analyze overall performance. However, there are some algorithm control structures which are present in each programming code and have a specific asymptotic analysis.

Some Algorithm Control Structures are:

1. Sequencing
2. If-then-else
3. for loop
4. While loop

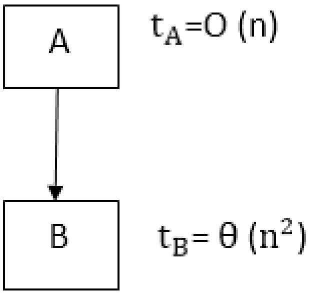
### 1. Sequencing:

Suppose our algorithm consists of two parts A and B. A takes time $t_A$ and B takes time $t_B$ for computation. The total computation "$t_A + t_B$" is according to the sequence rule. According to maximum rule, this computation time is (max ($t_A$,$t_B$)).

**Example:**

Suppose $t_A$ =O (n) and $t_B = \theta$ ($n^2$).

Then, the total computation time can be calculated as

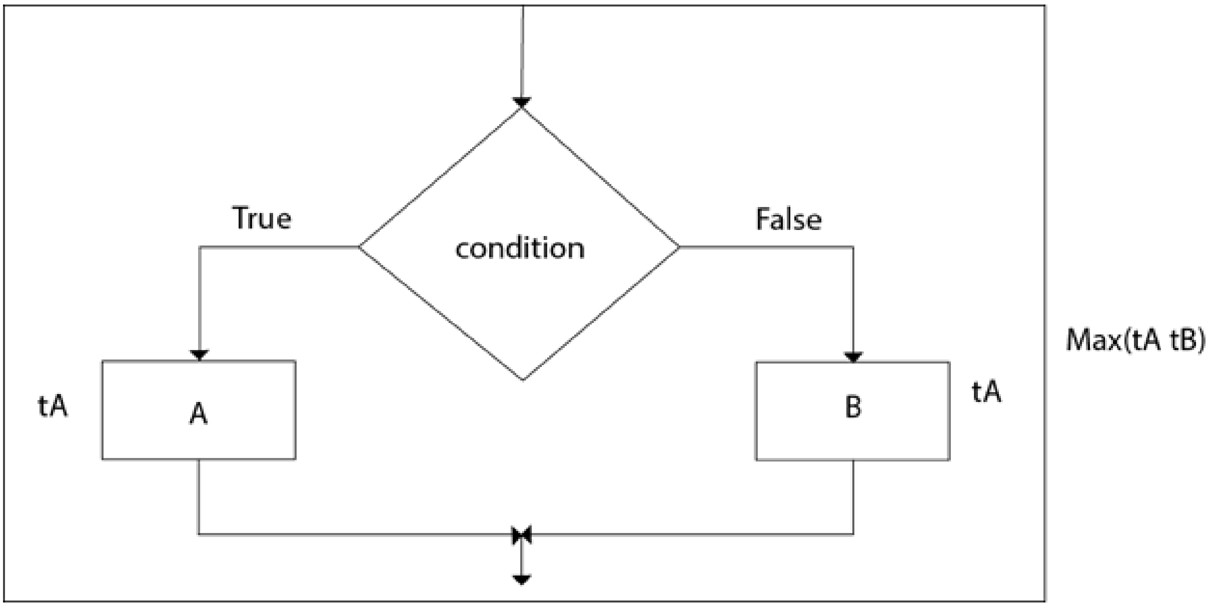


Computation Time = $t_A + t_B$

= (max ($t_A$,$t_B$)

= (max (O (n), θ ($n^2$)) = θ ($n^2$)

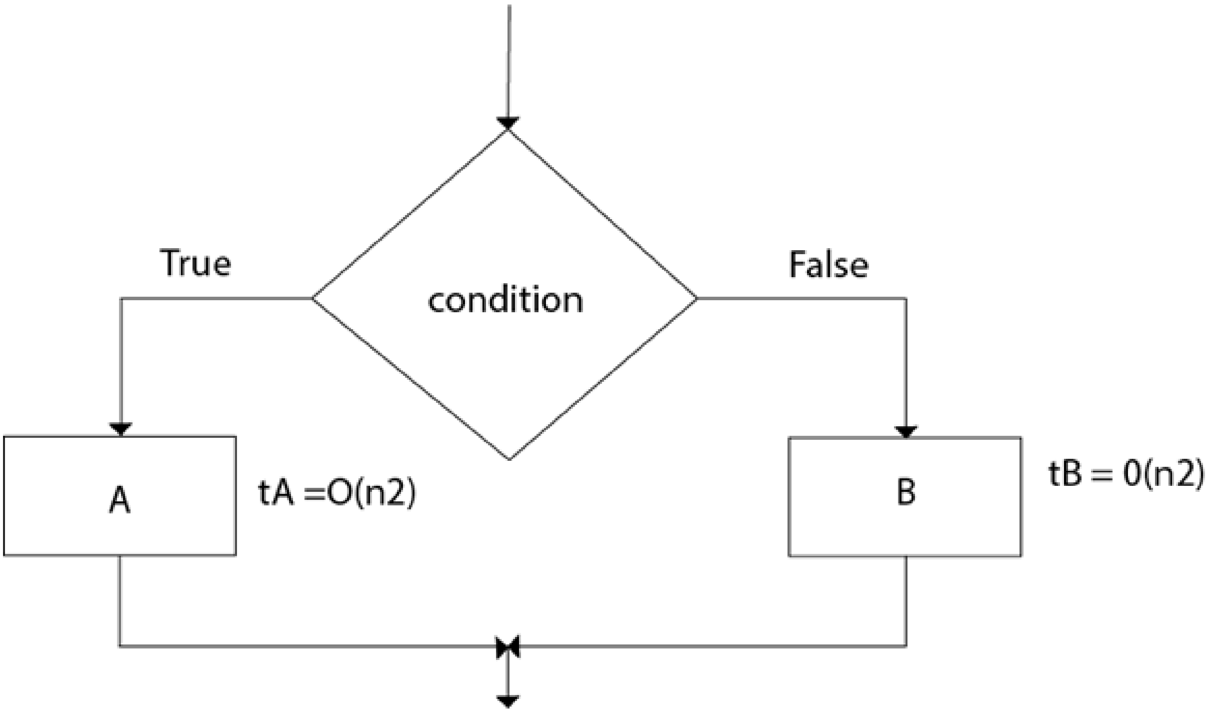
## 2. If-then-else:



The total time computation is according to the condition rule-"if-then-else." According to the maximum rule, this computation time is max ($t_A$,$t_B$).

**Example:**

Suppose $t_A = O(n^2)$ and $t_B = \theta(n^2)$
Calculate the total computation time for the following:



Total Computation = (max ($t_A$,$t_B$))

= max (O ($n^2$), θ ($n^2$) = θ ($n^2$)

### 3. For loop:

The general format of for loop is:

1. For (initialization; condition; updation)
2. 
3. Statement(s);

## Complexity of for loop:

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the **inner** loop execute a total of N * M times. Thus, the total complexity for the two loops is O (N2)

Consider the following loop:

```
for i ← 1 to n
{
        P (i)
}
```

If the computation time $t_i$ for ( $P_I$) various as a function of "i", then the total computation time for the loop is given not by a multiplication but by a sum i.e.

```
For i ← 1 to n
{
        P (i)
}
```

Takes $\sum_{i=1}^{n} t_i$ time, i.e.$\sum_{j=1}^{n} \theta(1) = \theta\sum_{i=1}^{n} \theta(n)$

If the algorithms consist of nested "for" loops, then the total computation time is

```
For i ← 1 to n
 {
     For j ← 1 to n
    {
     P (ij)
    }
 }
```

$$\sum_{i=1}^{n} \sum_{j=1}^{n} t_{ij}$$

**Example:**

Consider the following "for" loop, Calculate the total computation time for the following:

```
For i ← 2 to n-1
    {
        For j ← 3 to i
         {
                    Sum ← Sum+A [i] [j]
                }
        }
```

**Solution:**

The total Computation time is:

$\sum_{i=2}^{n-1} \sum_{j=3}^{i} t_{ij} = \sum_{i=2}^{n-1} \sum_{j=3}^{i} \theta\ (1)$

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

$= \theta(\sum_{i=2}^{n-1} i) = \theta\left(\frac{m^2}{2}\right) + \theta(m)$

$= \theta(m^2)$

## While loop:

The Simple technique for analyzing the loop is to determine the function of variable involved whose value decreases each time around. Secondly, for terminating the loop, it is necessary that value must be a positive integer. By keeping track of how many times the value of function decreases, one can obtain the number of repetition of the loop. The other approach for analyzing "while" loop is to treat them as recursive algorithms.

### Algorithm:

```
1. [Initialize] Set k: =1, LOC: =1 and MAX: = DA
   TA [1]
2. Repeat steps 3 and 4 while K≤N
3.  if MAX<DATA [k],then:
    Set LOC: = K and MAX: = DATA [k]
4. Set k: = k+1
   [End of step 2 loop]
5. Write: LOC, MAX
6. EXIT
```

## Example:

The running time of algorithm array Max of computing the maximum element in an array of n integer is O (n).

**Solution:**

```
array Max (A, n)
1. Current max ← A [0]
2. For i ←  1 to n-1
3. do if current max < A [i]
4. then  current max ← A [i]
5. return current max.
```

The number of primitive operation t (n) executed by this algorithm is at least.

1. $2 + 1 + n + 4 (n-1) + 1 = 5n$
2. $2 + 1 + n + 6 (n-1) + 1 = 7n-2$

The best case T(n) =5n occurs when A [0] is the maximum element. The worst case T(n) = 7n-2 occurs when element are sorted in increasing order.

We may, therefore, apply the big-Oh definition with c=7 and $n_0$=1 and conclude the running time of this is O (n).

## Asymptotic Notation

Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n.

Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm: There are three different notations: big O, big Theta (Θ), and big Omega (Ω).

## Why is Asymptotic Notation Important?

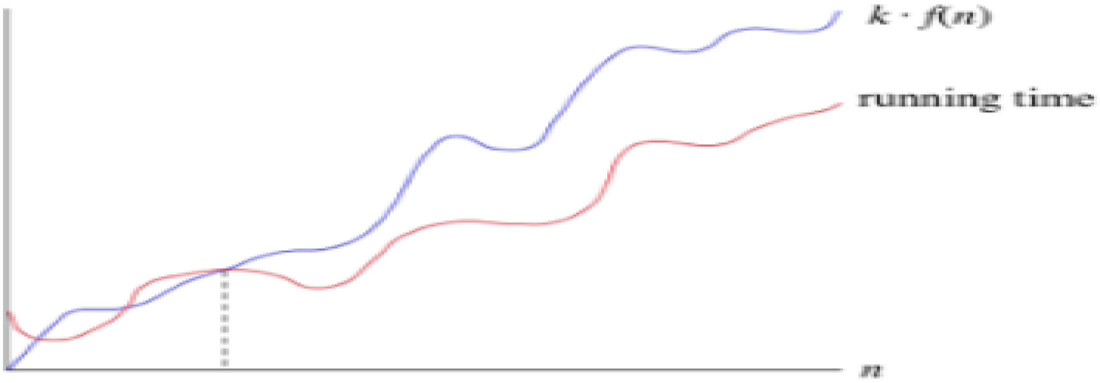
1. They give simple characteristics of an algorithm's efficiency.

2. They allow the comparisons of the performances of various algorithms.

## 1) Big-O Notation

The Big-O notation describes the worst-case running time of a program. We compute the Big-O of an algorithm by counting how many iterations an algorithm will take in the worst-case scenario with an input of N. We typically consult the Big-O because we must always plan for the worst case. For example, O(log n) describes the Big-O of a binary search algorithm.

1. f (n) ⩽ k.g (n)f(n)⩽k.g(n) **for** n>n0n>n0 in all **case**



ASYMPTOTIC UPPER BOUND

**For Example:**

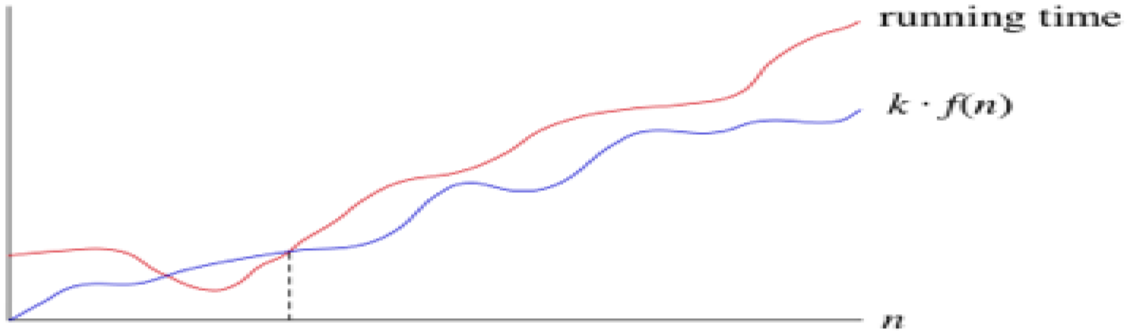
1. 1. 3n+2=O(n) as 3n+2≤4n **for** all n≥2
2. 2. 3n+3=O(n) as 3n+3≤4n **for** all n≥3

Hence, the complexity of **f(n)** can be represented as O (g (n))

2) Big-Ω Notation

Big-Ω (Omega) describes the best running time of a program. We compute the big-Ω by counting how many iterations an algorithm will take in the best-case scenario based on an input of N. For example, a Bubble Sort algorithm has a running time of Ω(N) because in the best case scenario the list is already sorted, and the bubble sort will terminate after the first iteration.

$F(n) \geq k^* g(n)$ for all n, $n \geq n_0$



ASYMPTOTIC LOWER BOUND

**For Example:**

$f(n) = 8n^2+2n-3 \geq 8n^2-3$

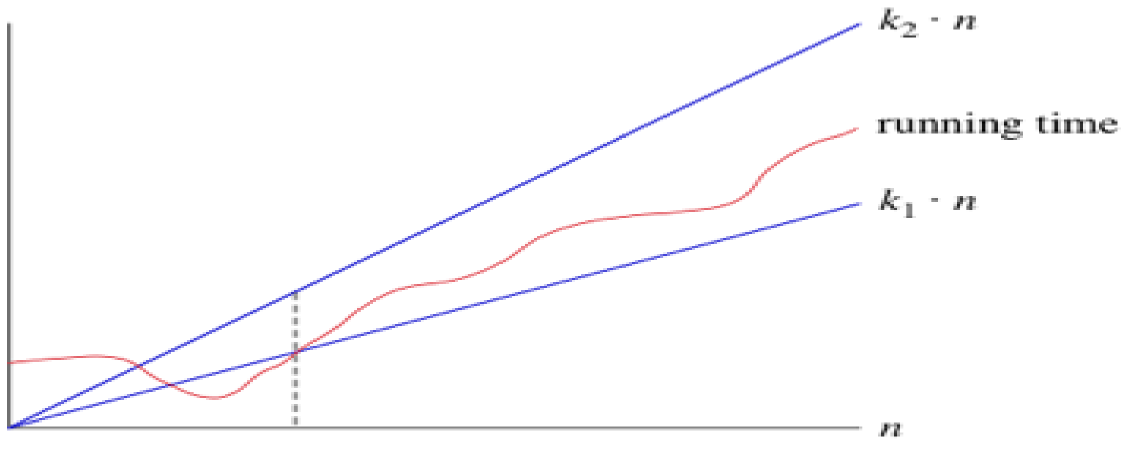
$= 7n^2+(n^2-3) \geq 7n^2$ (g(n))

Thus, $k_1=7$

Hence, the complexity of **f (n)** can be represented as **Ω** (g (n))

**3. Theta (θ) Notation:** The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant $k_1$, $k_2$ and $k_0$ such that

$k_1 * g(n) \leq f(n) \leq k_2\, g(n)$for all n, $n \geq n_0$



ASYMPTOTIC TIGHT BOUND

3n+2= θ (n) as 3n+2≥3n and 3n+2≤ 4n, for n
$k_1$=3,$k_2$=4, and $n_0$=2

Hence, the complexity of f (n) can be represented as θ (g(n)).

## Recurrence Relation

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

**For Example**, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

T (n) = θ (1) if n=1

$2T\left(\frac{n}{2}\right)$ + θ (n) if n>1

There are four methods for solving Recurrence:

1. Substitution Method

2. Iteration Method

3. Recursion Tree Method

4. Master Method

## 1. Substitution Method:

The Substitution Method Consists of two main steps:

1. Guess the Solution.
2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

**For Example1** Solve the equation by Substitution Method.

$$T(n) = T\left(\frac{n}{2}\right) + n$$

We have to show that it is asymptotically bound by O (log n).

**Solution:**

For T (n) = O (log n)

We have to show that for some constant c

1. T (n) ≤c logn.

Put this in given Recurrence Equation.

$$T(n) \leq c \log\left(\frac{n}{2}\right) + 1$$

$$\leq c \log\left(\frac{n}{2}\right) + 1 = c \log n - c\log_2 2 + 1$$

$$\leq c \log n \text{ for } c \geq 1$$

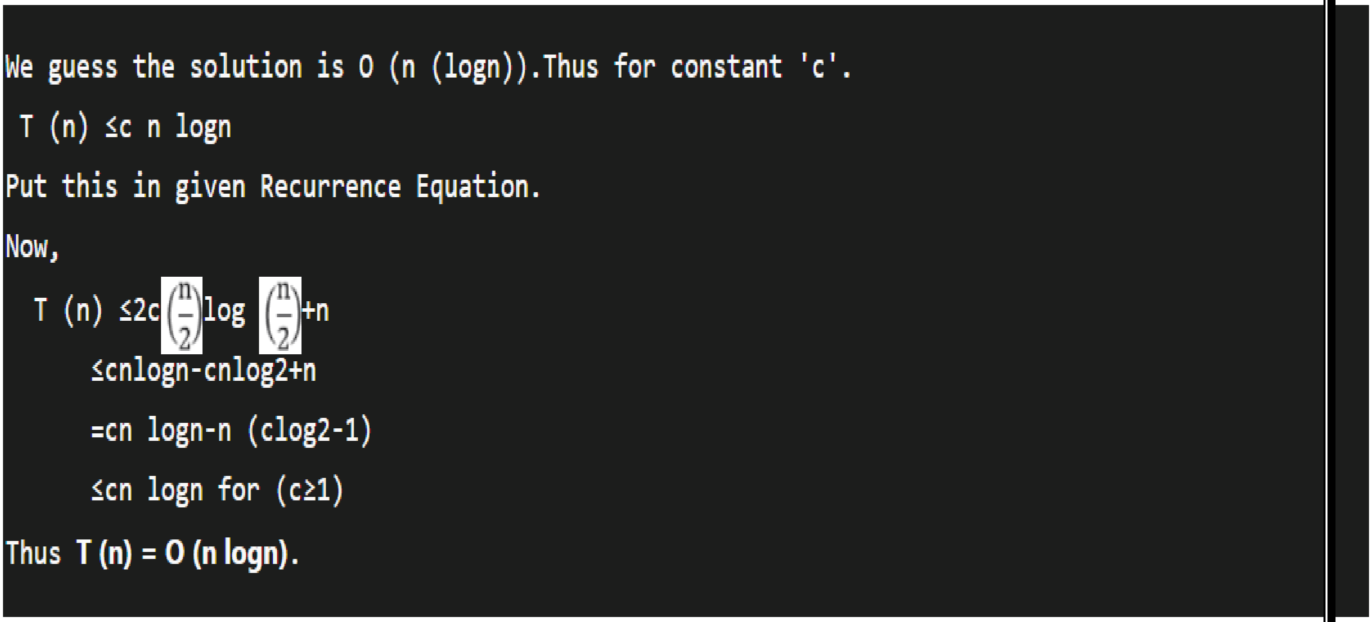
Thus **T (n) =O logn**.

**Example2.** Consider the Recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad n > 1$$

Find an Asymptotic bound on T.

Solution:



We guess the solution is O (n (logn)).Thus for constant 'c'.

T (n) ≤c n logn

Put this in given Recurrence Equation.

Now,

$$T\ (n)\ \leq 2c\left(\frac{n}{2}\right)\log\ \left(\frac{n}{2}\right)+n$$

$$\leq cnlogn-cnlog2+n$$

$$=cn\ logn-n\ (clog2-1)$$

$$\leq cn\ logn\ \text{for}\ (c\geq 1)$$

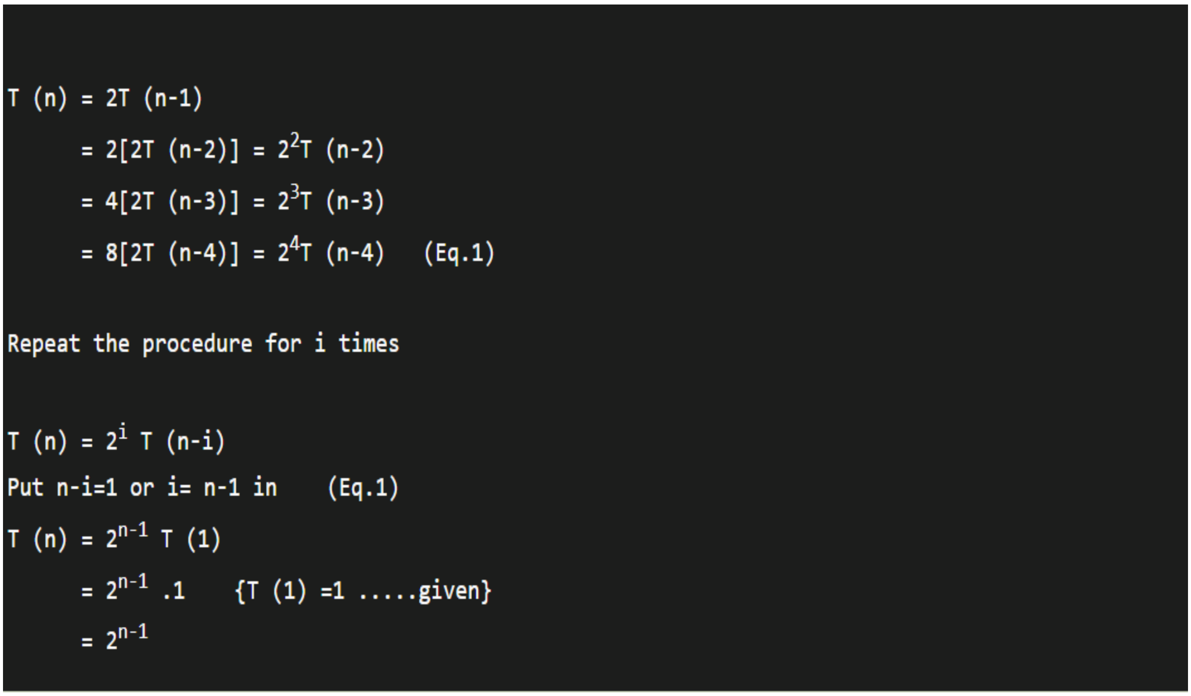
Thus **T (n) = O (n logn)**.

## 2. Iteration Methods

It means to expand the recurrence and express it as a summation of terms of n and initial condition.

**Example1:** Consider the Recurrence

1. T (n) = 1  **if** n=1
2.  = 2T (n-1) **if** n>1



T (n) = 2T (n-1)

$= 2[2T\ (n-2)] = 2^2T\ (n-2)$

$= 4[2T\ (n-3)] = 2^3T\ (n-3)$

$= 8[2T\ (n-4)] = 2^4T\ (n-4)$ (Eq.1)

Repeat the procedure for i times

$T\ (n) = 2^i\ T\ (n-i)$

Put n-i=1 or i= n-1 in (Eq.1)

$T\ (n) = 2^{n-1}\ T\ (1)$

$= 2^{n-1}\ .1$ {T (1) =1 .....given}

$= 2^{n-1}$

**3) Recurrence Tree Method:** In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

1. In general, we consider the second term in recurrence as root.

2. It is useful when the divide & Conquer algorithm is used.

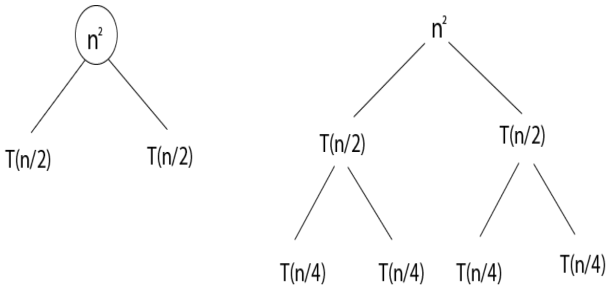
3. It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single subproblem.

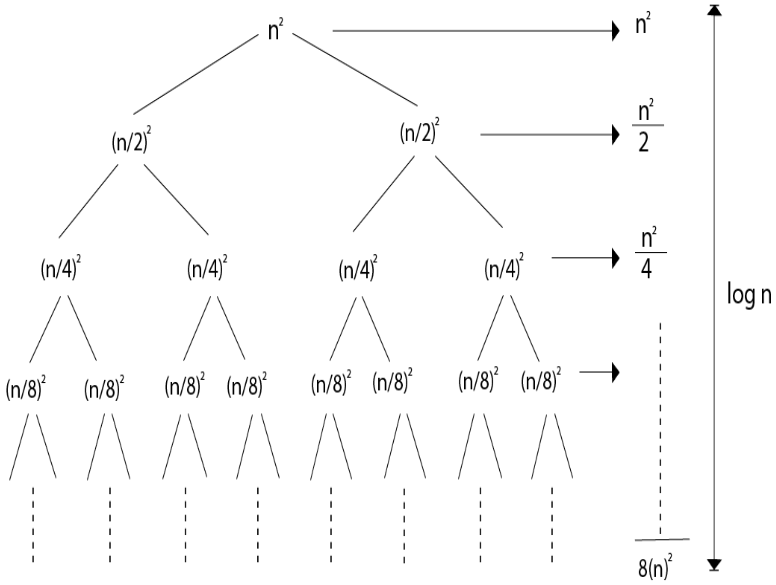
For example consider the recurrence relation

Consider $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

We have to obtain the asymptotic bound using recursion tree method.

**Solution:** The Recursion tree for the above recurrence is





$$T\,(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \text{log n times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

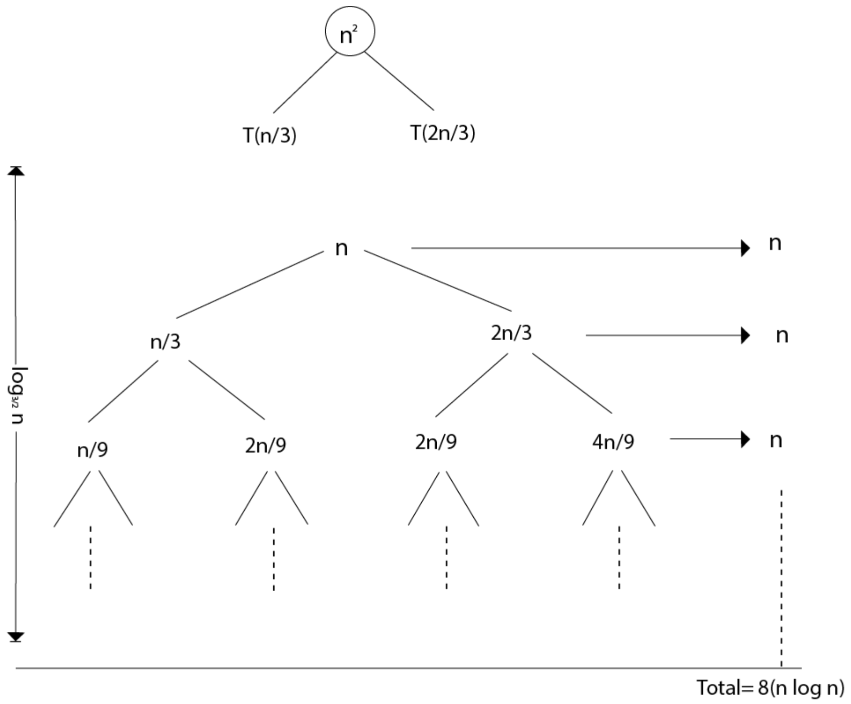
$$\mathbf{T\ (n) = \theta} n^2$$

**Example 3:** Consider the following recurrence

$$T\,(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

**Solution:** The given Recurrence has the following recursion tree



When we add the values across the levels of the recursion trees, we get a value of n for every level. The longest path from the root to leaf is

$$n \longrightarrow \frac{2}{3}n \longrightarrow \left(\frac{2}{3}\right) n \longrightarrow \ldots 1$$

Since $\left(\frac{2}{3}\right)$ n=1 when i=$\log_{\frac{3}{2}}$ n.

Thus the height of the tree is $\log_{\frac{3}{2}}$ n.

$$T\ (n) = n + n + n + \ldots\ldots + \log_{\frac{3}{2}} n \text{ times.} = \boldsymbol{\theta(n\ logn)}$$

## 4) Master Method

The Master Method is used for solving the following types of recurrence

T (n) = a $T\left(\frac{n}{b}\right)$ + f (n) with a≥1 and b≥1 be constant & f(n) be a function and $\frac{n}{b}$can be interpreted as

Let T (n) is defined on non-negative integers by the recurrence.

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

In the function to the analysis of a recursive algorithm, the constants and function take on the following significance:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.

- It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function

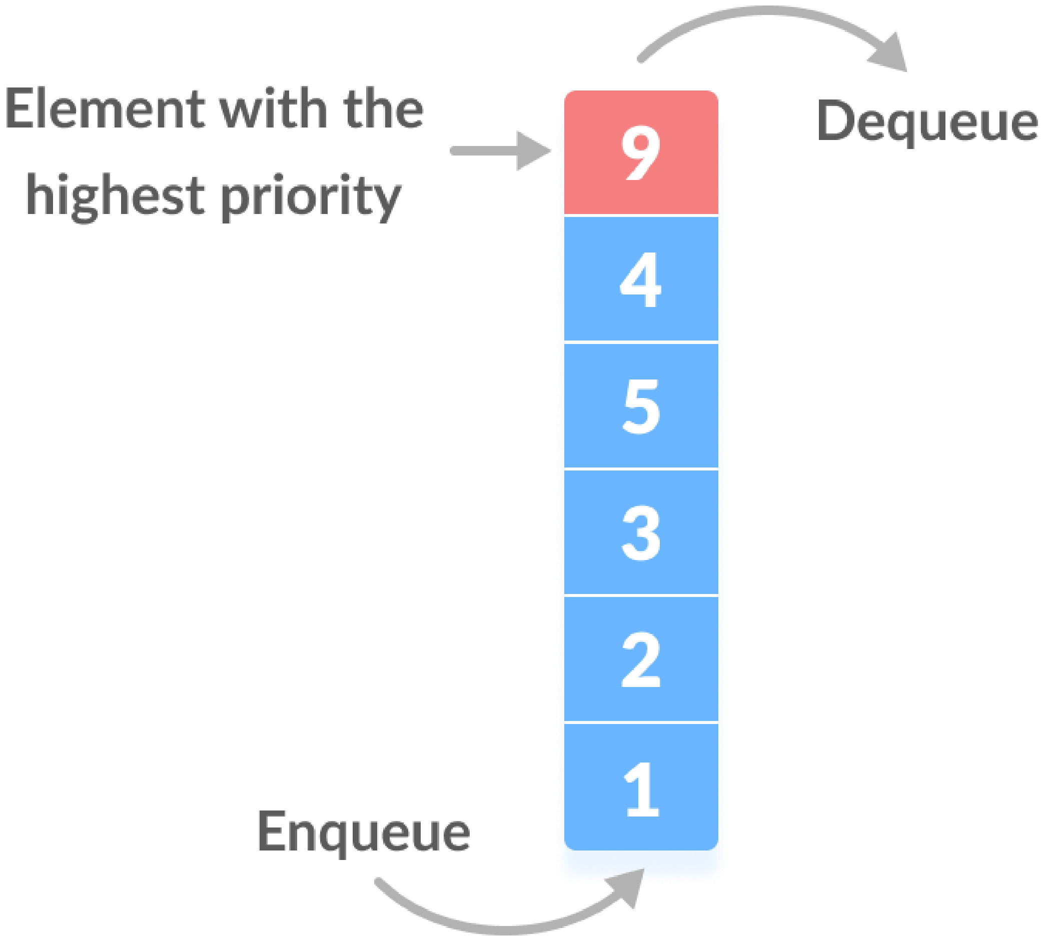
## Priority queue

A priority queue is a **special type of queue** in which each element is associated with a **priority value**. And, elements are served on the basis of their priority. That is, higher priority elements are served first.
However, if elements with the same priority occur, they are served according to their order in the queue.

**Assigning Priority Value**
Generally, the value of the element itself is considered for assigning the priority. For example,

The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element.

We can also set priorities according to our needs.



Removing highest Priority Element

## Implementation of Priority Queue

Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

Hence, we will be using the heap data structure to implement the priority queue in this tutorial. A max-heap is implement is in the following operations. If you want to learn more about it, please visit max-heap and mean-heap.
A comparative analysis of different implementations of priority queue is given below.

| Operations | peek | insert | delete |
|---|---|---|---|
| Linked List | O(1) | O(n) | O(1) |
| Binary Heap | O(1) | O(log n) | O(log n) |
| Binary Search Tree | O(1) | O(log n) | O(log n) |

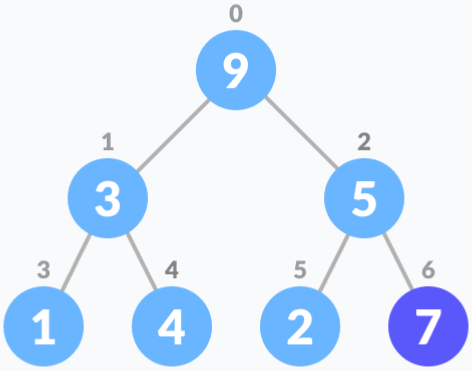
## Priority Queue Operations

Basic operations of a priority queue are inserting, removing, and peeking elements.

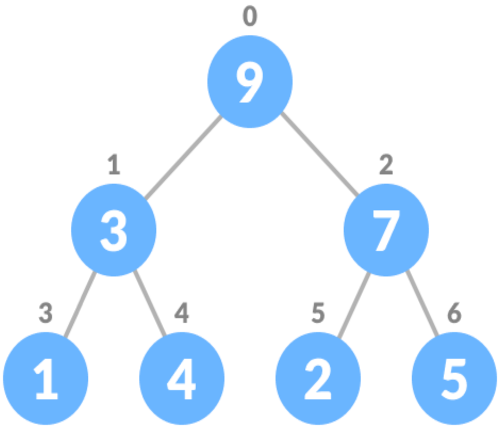
### . Inserting an Element into the Priority Queue

Inserting an element into a priority queue (max-heap) is done by the following steps.

- Insert the new element at the end of the tree.



Heapify the tree.



Heapify after insertion

Algorithm for insertion of an element into priority queue (max-heap)

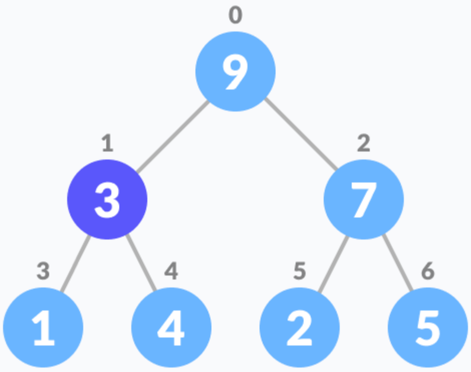
```
If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

heapify the array
```

For Min Heap, the above algorithm is modified so that `parentNode` is always smaller than `newNode`.

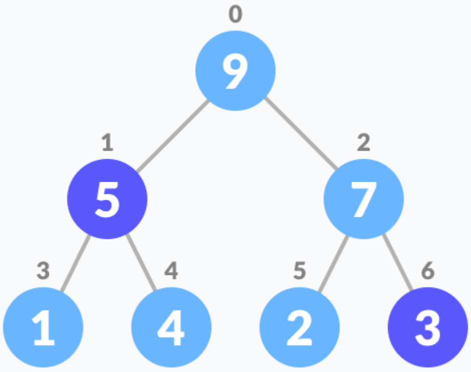
## 2. Deleting an Element from the Priority Queue

Deleting an element from a priority queue (max-heap) is done as follows:

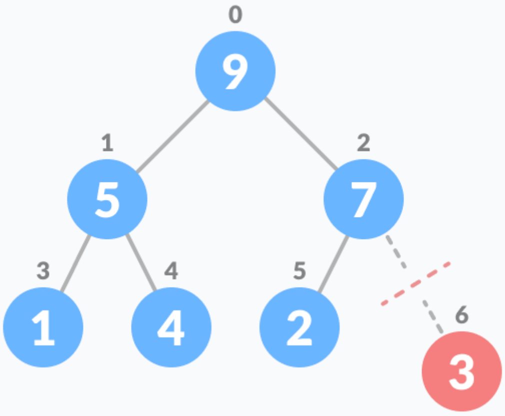


- Select the element to be deleted.

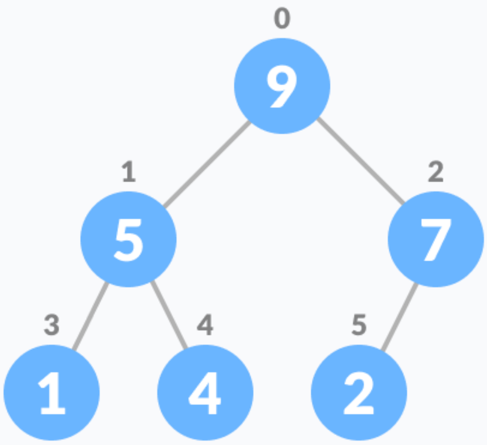
Select the element to be deleted



- Swap it with the last element. Swap with the last leaf node element



- Remove the last element. Remove the last element leaf



- Heapify the tree. Heapify the priority queue

Algorithm for deletion of an element in the priority queue (max-heap)

```
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted

heapify the array
```

For Min Heap, the above algorithm is modified so that the both `childNodes` are smaller than `currentNode`.

## 3. Peeking from the Priority Queue (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

```
return rootNode
```

## 4. Extract-Max/Min from the Priority Queue

Extract-Max returns the node with maximum value after removing it from a Max Heap whereas Extract-Min returns the node with minimum value after removing it from Min Heap.

# Priority Queue Implementations in Python, Java, C, and C++

Python

Java

C

C++

```
# Priority Queue implementation in Python


# Function to heapify the tree
def heapify(arr, n, i):
    # Find the largest among root, left child and right child
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r
    # Swap and continue heapifying if root is not largest
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


# Function to insert an element into the tree
def insert(array, newNum):
    size = len(array)
    if size == 0:
        array.append(newNum)
    else:
        array.append(newNum)
        for i in range((size // 2) - 1, -1, -1):
            heapify(array, size, i)


# Function to delete an element from the tree
def deleteNode(array, num):
    size = len(array)
    i = 0
    for i in range(0, size):
        if num == array[i]:
            break

    array[i], array[size - 1] = array[size - 1], array[i]

    array.remove(size - 1)

    for i in range((len(array) // 2) - 1, -1, -1):
        heapify(array, len(array), i)


arr = []

insert(arr, 3)
insert(arr, 4)
insert(arr, 9)
insert(arr, 5)
insert(arr, 2)

print ("Max-Heap array: " + str(arr))

deleteNode(arr, 4)
print("After deleting an element: " + str(arr))
```

## Heap Sort

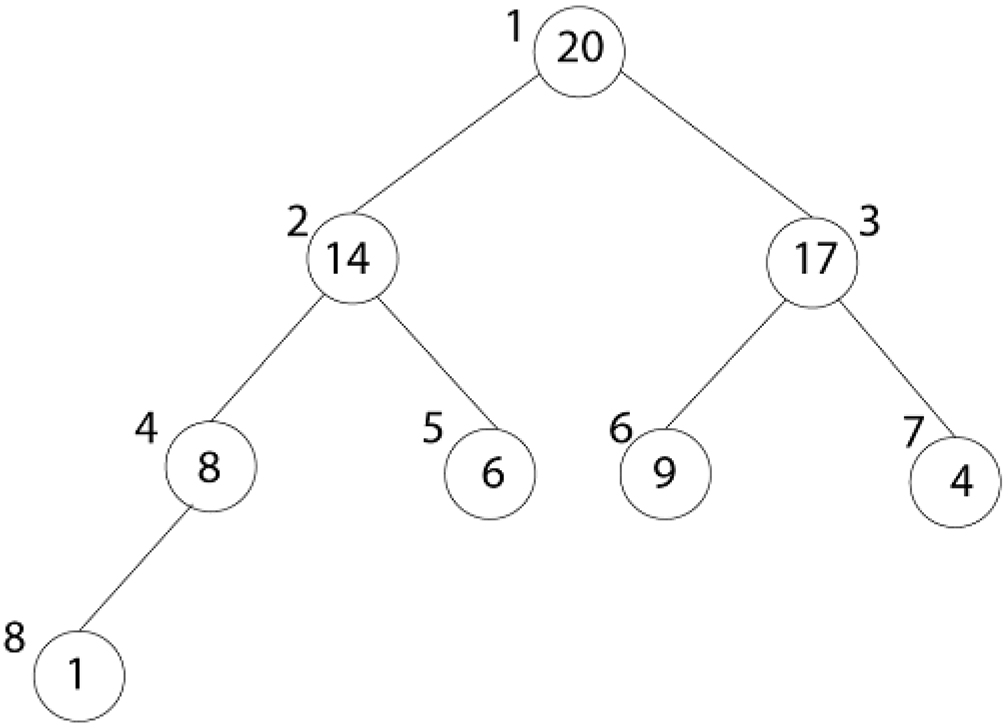
### Binary Heap:

Binary Heap is an array object can be viewed as Complete Binary Tree. Each node of the Binary Tree corresponds to an element in an array.

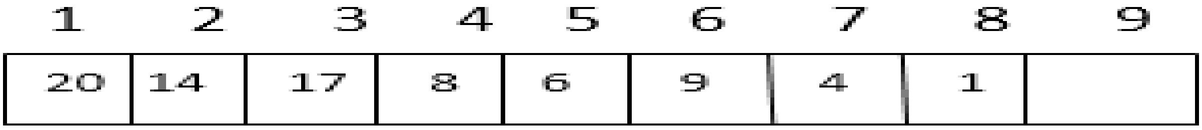
1. Length [A],number of elements in array
2. Heap-Size[A], number of elements in a heap stored within array A.

The root of tree A [1] and gives index 'i' of a node that indices of its parents, left child, and the right child can be computed.

```
PARENT (i)
    Return floor (i/2)
LEFT (i)
    Return 2i
RIGHT (i)
    Return 2i+1
```



**Representation of an array of the above figure is given below:**



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 20 | 14 | 17 | 8 | 6 | 9 | 4 | 1 |  |

The index of 20 is 1

**Stay**

To find the index of the left child, we calculate 1*2=2

This takes us (correctly) to the 14.

Now, we go right, so we calculate 2*2+1=5

This takes us (again, correctly) to the 6.

Now, 4's index is 7, we want to go to the parent, so we calculate 7/2 =3 which takes us to the 17.

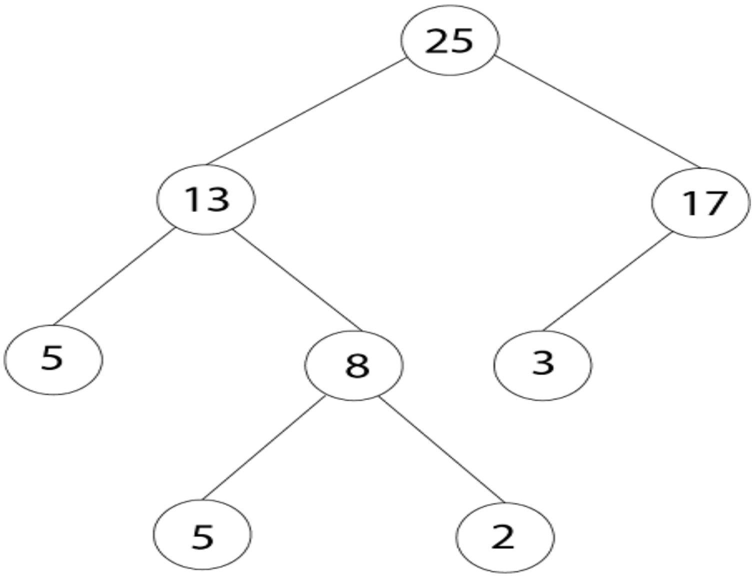
## Heap Property:

A binary heap can be classified as Max Heap or Min Heap

**1. Max Heap:** In a Binary Heap, for every node I other than the root, the value of the node is greater than or equal to the value of its highest child

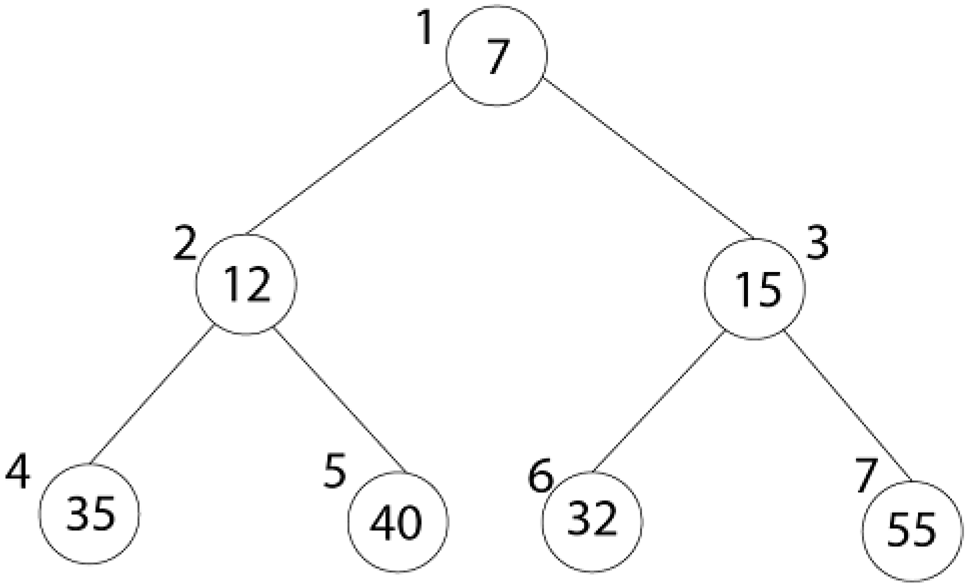
1. A [PARENT (i) ≥A[i]

Thus, the highest element in a heap is stored at the root. Following is an example of MAX-HEAP



**2. MIN-HEAP:** In MIN-HEAP, the value of the node is lesser than or equal to the value of its lowest child.

1. A [PARENT (i) ≤A[i]



## Heapify Method:

**1. Maintaining the Heap Property:** Heapify is a procedure for manipulating heap Data Structure. It is given an array A and index I into the array. The subtree rooted at the children of A [i] are heap but node A [i] itself may probably violate the heap property i.e. A [i] < A [2i] or A [2i+1]. The procedure 'Heapify' manipulates the tree rooted as A [i] so it becomes a heap.

**MAX-HEAPIFY (A, i)**

```
l ← left [i]
r ← right [i]
if l≤ heap-size [A] and A[l] > A [i]
then largest ← l
Else largest ← i
If r≤ heap-size [A] and A [r] > A[largest]
Then largest ← r
If largest ≠ i
Then exchange A [i]    A [largest]
MAX-HEAPIFY (A, largest)
```

## Analysis:

The maximum levels an element could move up are Θ (log n) levels. At each level, we do simple comparison which O (1). The total time for heapify is thus O (log n).

## Building a Heap:

BUILDHEAP (array A, int n)

```
for i ← n/2 down to 1
do
HEAPIFY (A, i, n)
```

## HEAP-SORT ALGORITHM:

**HEAP-SORT (A)**

```
BUILD-MAX-HEAP (A)
For I ← length[A] down to Z
Do exchange A [1] ←→ A [i]
Heap-size [A] ← heap-size [A]-1
MAX-HEAPIFY (A,1)
```

**Analysis:** Build max-heap takes O (n) running time. The Heap Sort algorithm makes a call to 'Build Max-Heap' which we take O (n) time & each of the (n-1) calls to Max-heap to fix up a new heap. We know 'Max-Heapify' takes time O (log n)

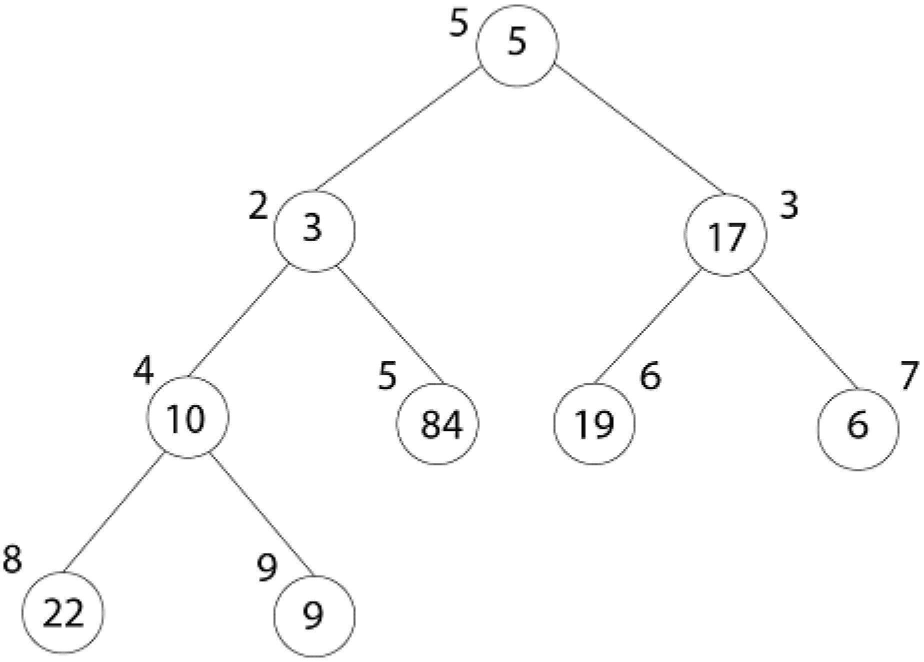
**The total running time of Heap-Sort is O (n log n).**

**Example:** Illustrate the Operation of BUILD-MAX-HEAP on the array.

1. A = (5, 3, 17, 10, 84, 19, 6, 22, 9)

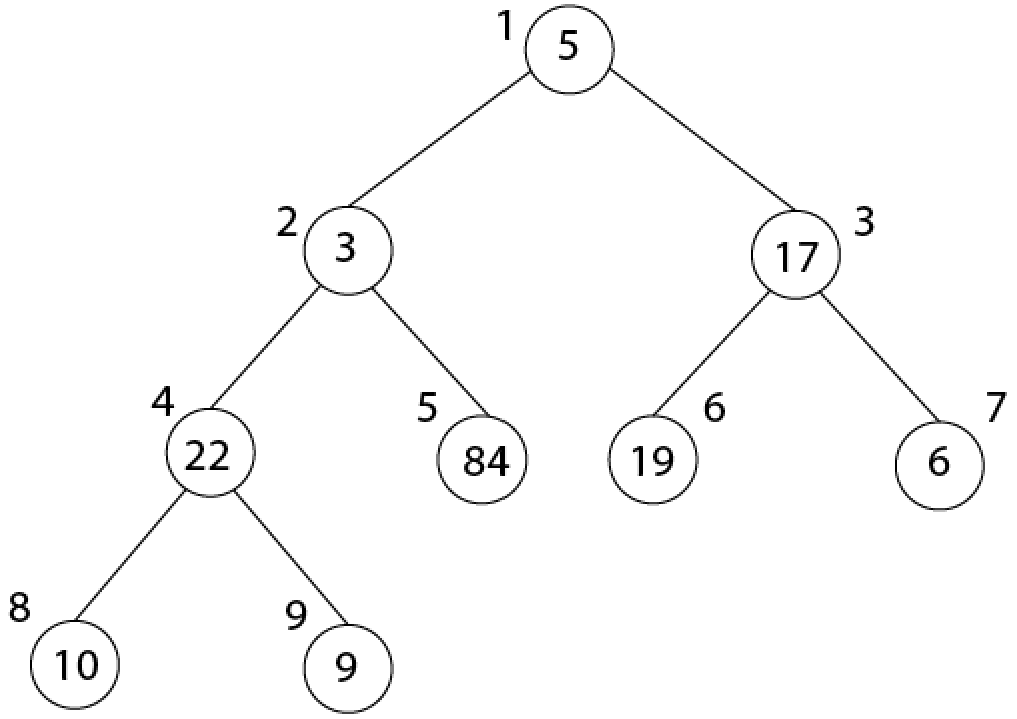
**Solution: Originally:**

1. Heap-Size (A) =9, so first we call MAX-HEAPIFY (A, 4)
2. And I = 4.5= 4 to 1



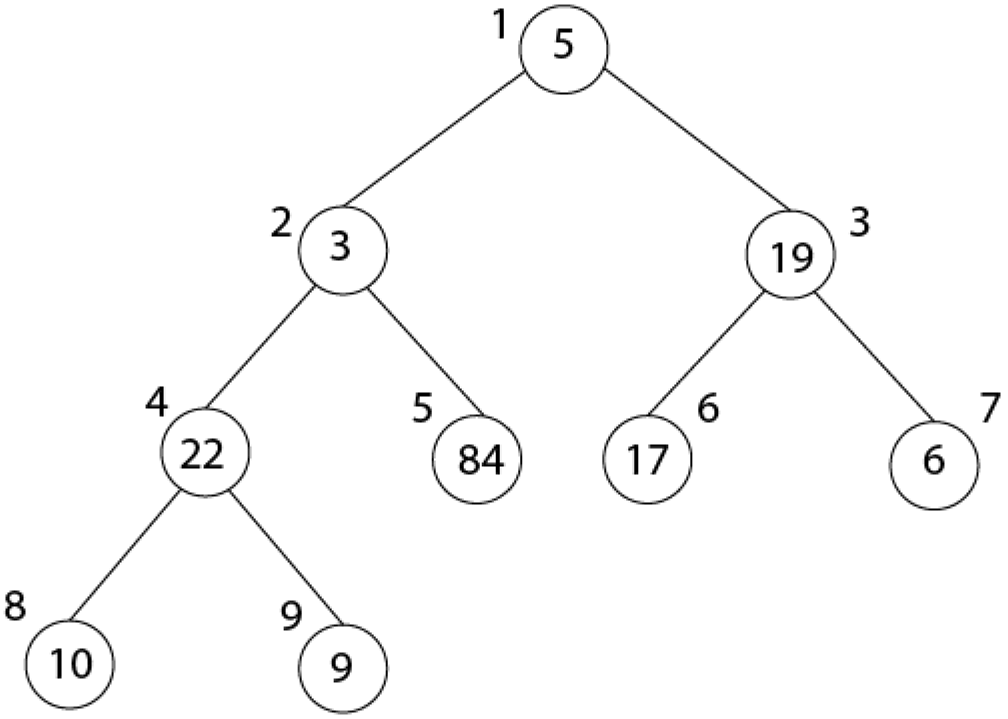
1. After MAX-HEAPIFY (A, 4) and i=4
2. L ← 8, r ← 9
3. l≤ heap-size[A] and A [l] >A [i]
4. 8 ≤9 and 22>10
5. Then Largest ← 8

6. If r≤ heap-size [A] and A [r] > A [largest]
7. 9≤9 and 9>22
8. If largest (8) ≠4
9. Then exchange A [4] ←→ A [8]
10. MAX-HEAPIFY (A, 8)



1. After MAX-HEAPIFY (A, 3) and i=3
2. l← 6, r ← 7
3. l≤ heap-size[A] and A [l] >A [i]
4. 6≤ 9 and 19>17
5. Largest ← 6
6. If r≤ heap-size [A] and A [r] > A [largest]
7. 7≤9 and 6>19
8. If largest (6) ≠3

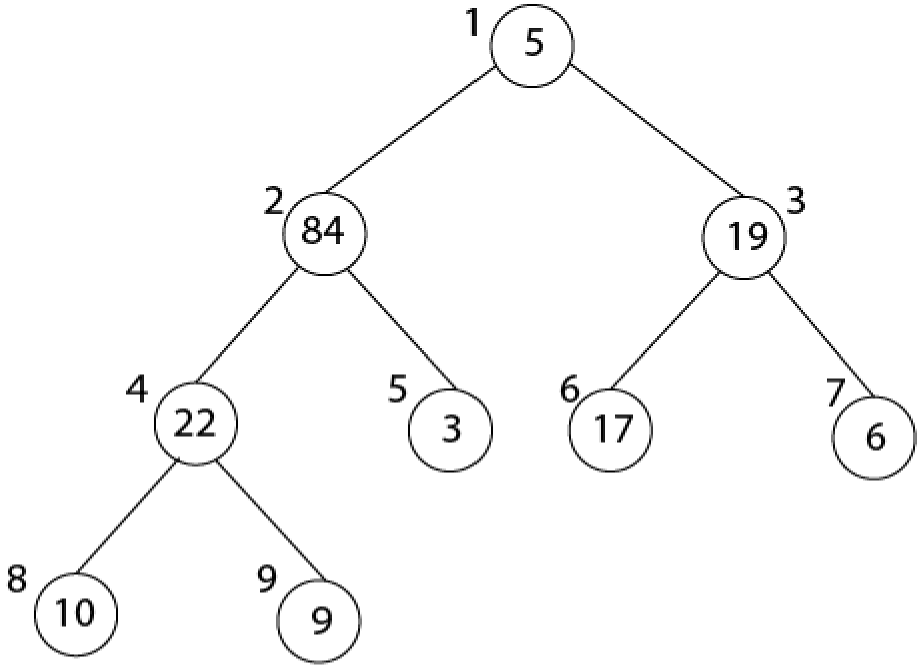
9. Then Exchange A [3] ←→ A [6]
10. MAX-HEAPIFY (A, 6)



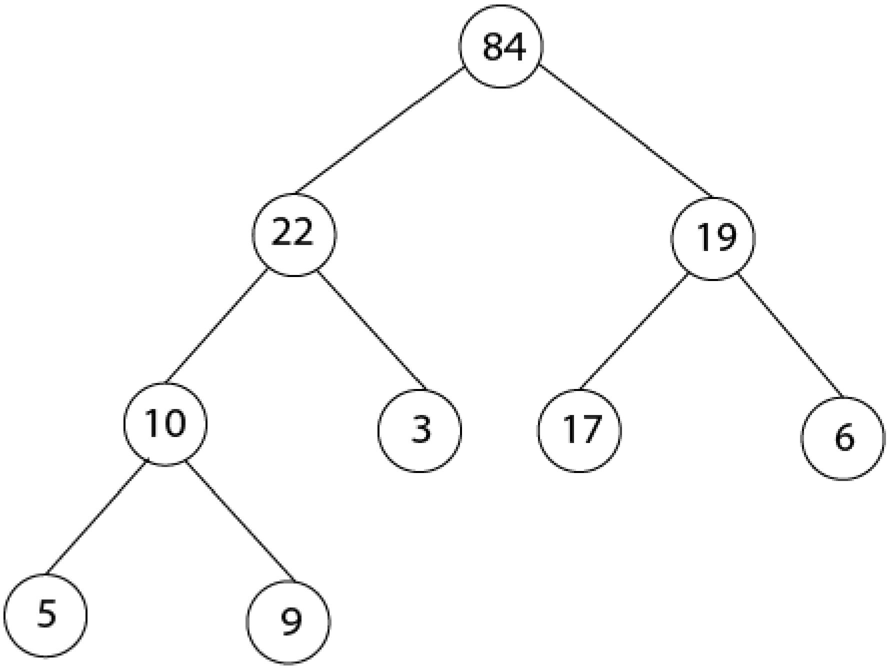
1. After MAX-HEAPIFY (A, 2) and i=2
2. l ← 4, r ← 5
3. l≤ heap-size[A] and A [l] >A [i]
4. 4≤9 and 22>3
5. Largest ← 4
6. If r≤ heap-size [A] and A [r] > A [largest]
7. 5≤9 and 84>22
8. Largest ← 5
9. If largest (4) ≠2

10. Then Exchange A [2] ←→ A [5]

11. MAX-HEAPIFY (A, 5)



1. After MAX-HEAPIFY (A, 1) and i=1
2. l ← 2, r ← 3
3. l≤ heap-size[A] and A [l] >A [i]
4. 2≤9 and 84>5
5. Largest ← 2
6. If r≤ heap-size [A] and A [r] > A [largest]
7. 3≤9 and 19<84
8. If largest (2) ≠1
9. Then Exchange A [1] ←→ A [2]
10. MAX-HEAPIFY (A, 2)



## Priority Queue:

As with heaps, priority queues appear in two forms: max-priority queue and min-priority queue.

A priority queue is a data structure for maintaining a set S of elements, each with a combined value called a key. A max-priority queue guides the following operations:

**INSERT(S, x):** inserts the element x into the set S, which is proportionate to the operation S=S∪[x].

**MAXIMUM (S)** returns the element of S with the highest key.

**EXTRACT-MAX (S)** removes and returns the element of S with the highest key.

**INCREASE-KEY(S, x, k)** increases the value of element x's key to the new value k, which is considered to be at least as large as x's current key value.

Let us discuss how to implement the operations of a max-priority queue. The procedure HEAP-MAXIMUM consider the MAXIMUM operation in θ (1) time.

### HEAP-MAXIMUM (A)

```
return A [1]
```

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation. It is similar to the for loop of Heap-Sort procedure.

### HEAP-EXTRACT-MAX (A)

```
if A. heap-size < 1
    error "heap underflow"
max ← A [1]
A [1] ← A [heap-size [A]]
heap-size [A] ← heap-size [A]-1
MAX-HEAPIFY (A, 1)
return max
```

The procedure HEAP-INCREASE-KEY implements the INCREASE-KEY operation. An index i into the array identify the priority-queue element whose key we wish to increase.

### HEAP-INCREASE-KEY.A, i, key)

```
1 if key < A[i]
2 errors "new key is smaller than current key"
3 A[i] = key
4 while i>1 and A [Parent (i)] < A[i]
5 exchange A [i] with A [Parent (i)]
6 i =Parent [i]
```

The running time of HEAP-INCREASE-KEY on an n-element heap is O (log n) since the path traced from the node updated in line 3 to the root has length O (log n).

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes as an input the key of the new item to be inserted into max-heap A. The procedure first expands the max-heap by calculating to the tree a new leaf whose key is - ∞. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its right value and maintain the max-heap property

**MAX-HEAP-INSERT (A, key)**

```
A. heap-size = A. heap-size + 1
A [A. heap-size] = - ∞
HEAP-INCREASE-KEY (A, A. heap-size, key)
```

**The running time of MAX-HEAP-INSERT on an n-element heap is O (log n).**

**Example:** Illustrate the operation of HEAP-EXTRACT-MAX on the heap

1. A= (15,13,9,5,12,8,7,4,0,6,2,1)

**Fig:** Operation of HEAP-INCREASE-KEY

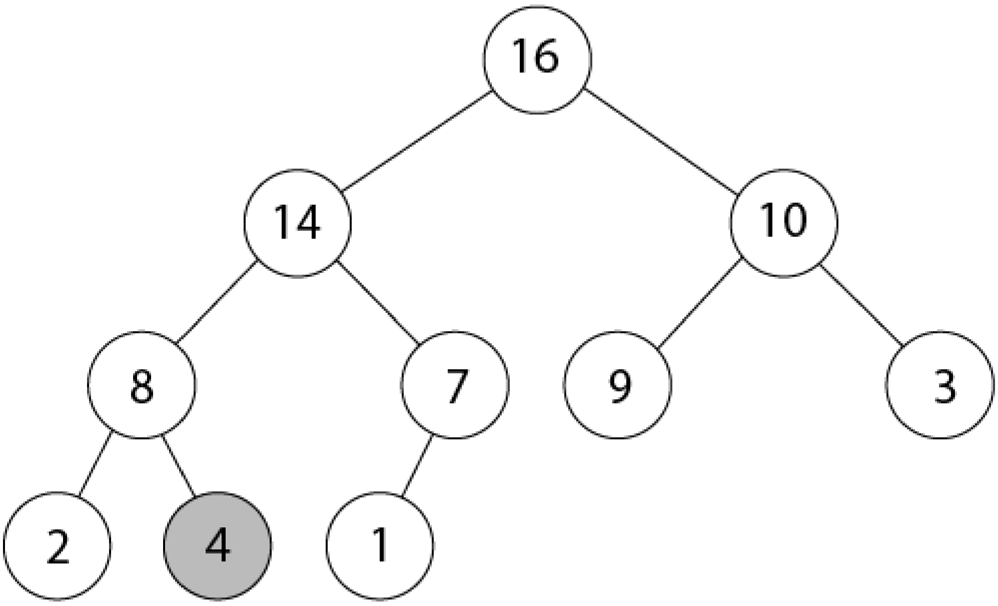


**Fig: (a)**

In this figure, that max-heap with a node whose index is 'i' heavily shaded

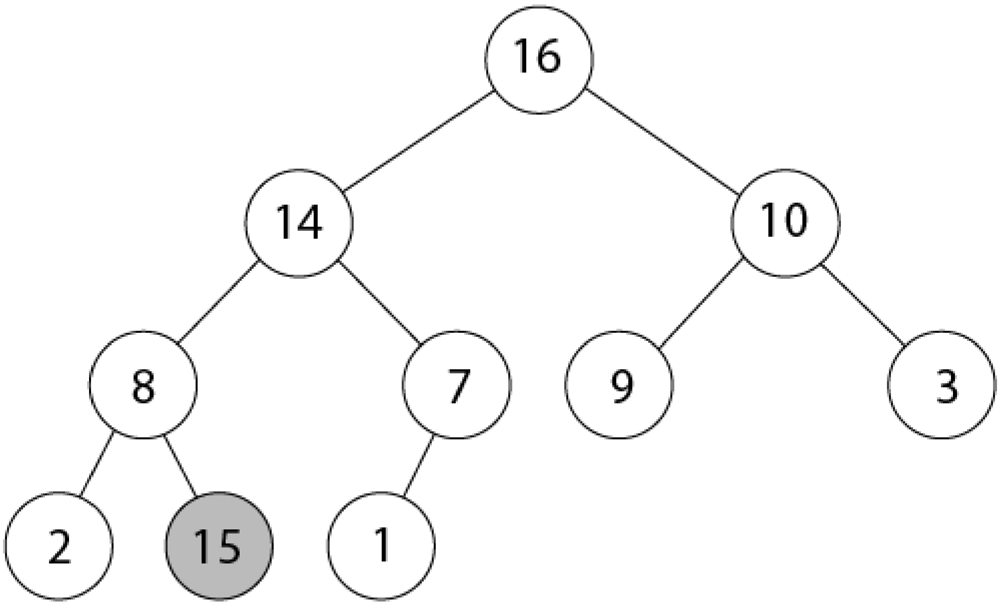


**Fig: (b)**

In this Figure, this node has its key increased to 15.

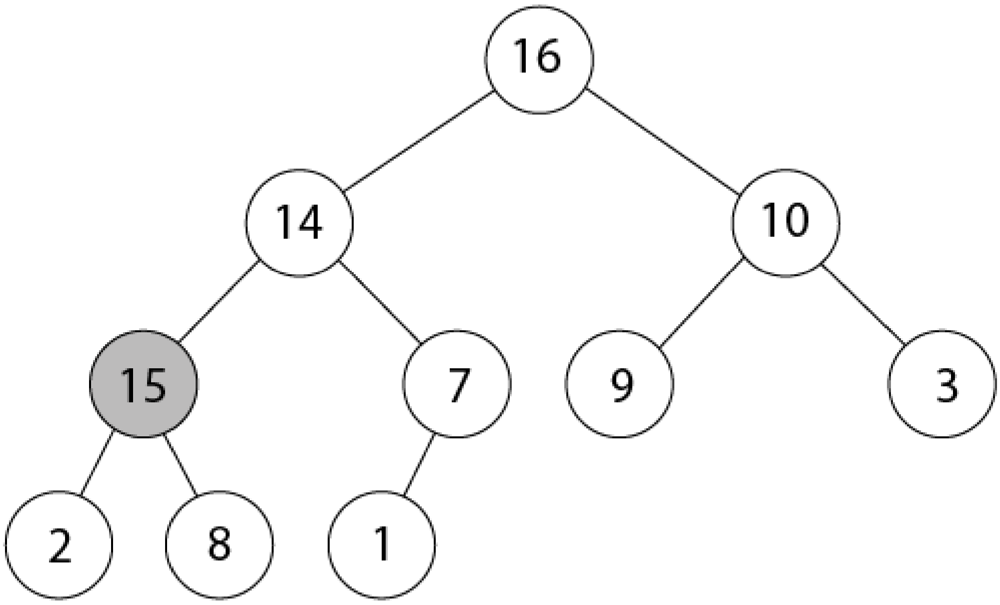


**Fig: (c)**

After one iteration of the while loop of lines 4-6, the node and its parent have exchanged keys, and the index i moves up to the parent.

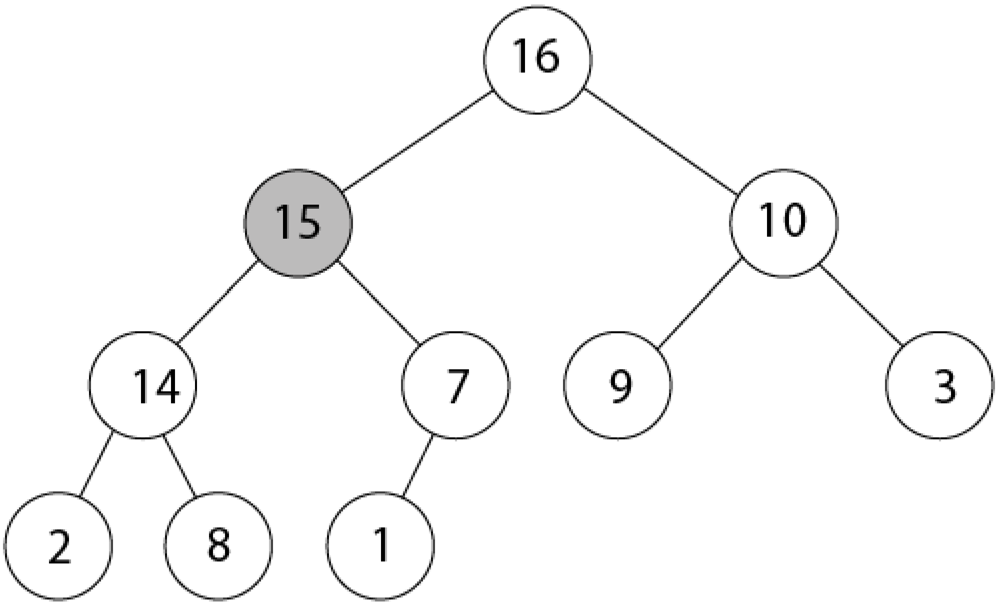


**Fig: (d)**

The max-heap after one more iteration of the while loops, the A [PARENT (i) ≥A (i)] the max-heap property now holds and the procedure terminates.

### Heap-Delete:

Heap-DELETE (A, i) is the procedure, which deletes the item in node 'i' from heap A, HEAP-DELETE runs in O (log n) time for n-element max heap.

```
HEAP-DELETE (A, i)
  1. A [i] ← A [heap-size [A]]
  2. Heap-size [A] ← heap-size [A]-1
  3. MAX-HEAPIFY (A, i)
```

## Quick sort

It is an algorithm of Divide & Conquer type.

**Divide:** Rearrange the elements and split arrays into two sub-arrays and an element in between search that each element in left sub array is less than or equal to the average element and each element in the right sub- array is larger than the middle element.

**Conquer:** Recursively, sort two sub arrays.

**Combine:** Combine the already sorted array.

- Quick Sort is a famous sorting algorithm.
- It sorts the given data items in ascending order.
- It uses the idea of divide and conquer approach.
- It follows a recursive algorithm

### Algorithm:

```
QUICKSORT (array A, int m, int n)
 1 if (n > m)
 2 then
 3 i ← a random index from [m,n]
```

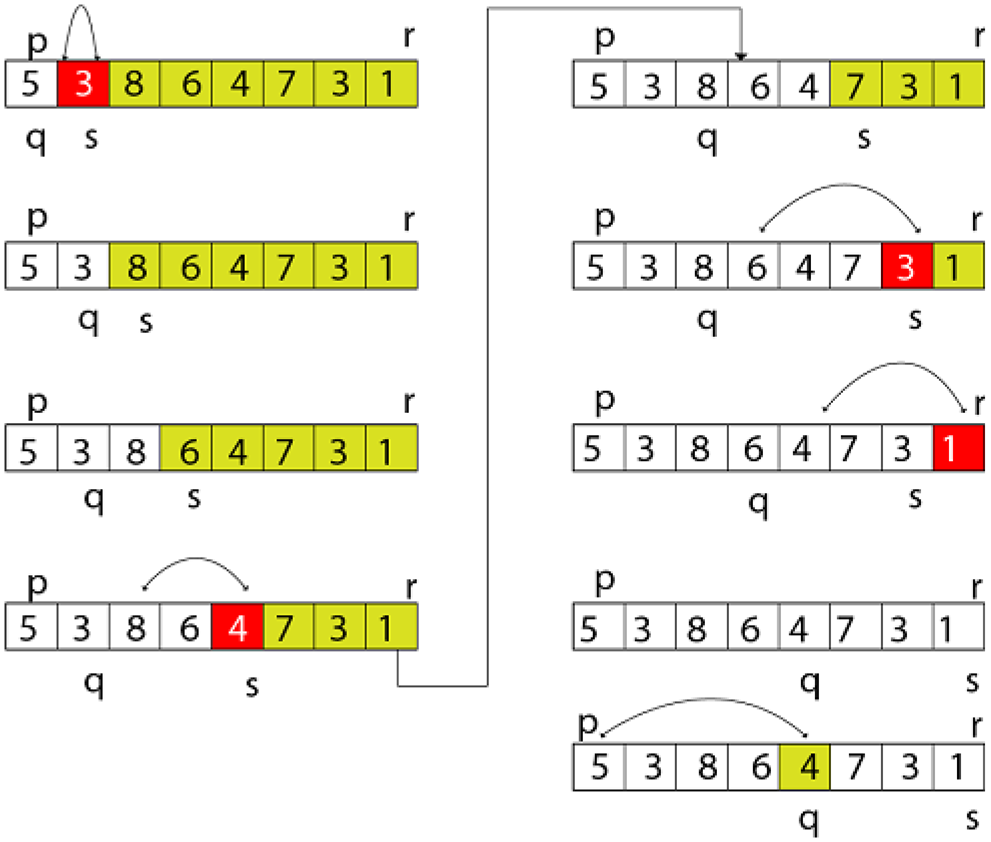
```
5.  4 swap A [i] with A[m]
6.  5 o ← PARTITION (A, m, n)
7.  6 QUICKSORT (A, m, o - 1)
8.  7 QUICKSORT (A, o + 1, n)
```

## Partition Algorithm:

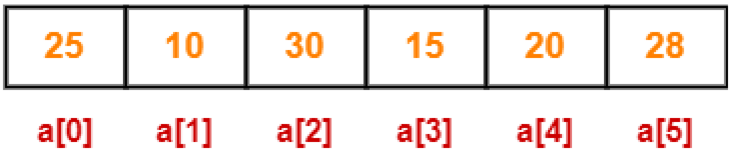
Partition algorithm rearranges the sub arrays in a place.

```
1. PARTITION (array A, int m, int n)
2.  1 x ← A[m]
3.  2 o ← m
4.  3 for p ← m + 1 to n
5.  4 do if (A[p] < x)
6.  5 then o ← o + 1
7.  6 swap A[o] with A[p]
8.  7 swap A[m] with A[o]
9.  8 return o
```

**Figure: shows the execution trace partition algorithm**



Consider the following array has to be sorted in ascending order using quick sort algorithm-



| 25 | 10 | 30 | 15 | 20 | 28 |
|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |

Quick Sort Example

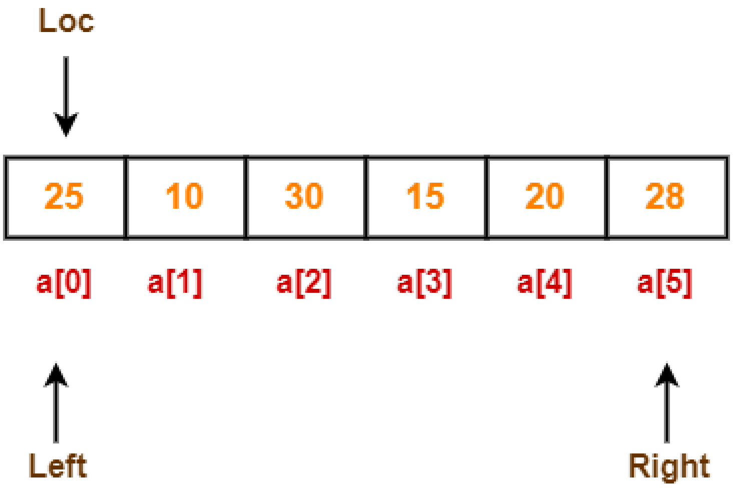
Quick Sort Algorithm works in the following steps-

## Step-01:

Initially-

- **Left** and **Loc** (pivot) points to the first element of the array.
- **Right** points to the last element of the array.

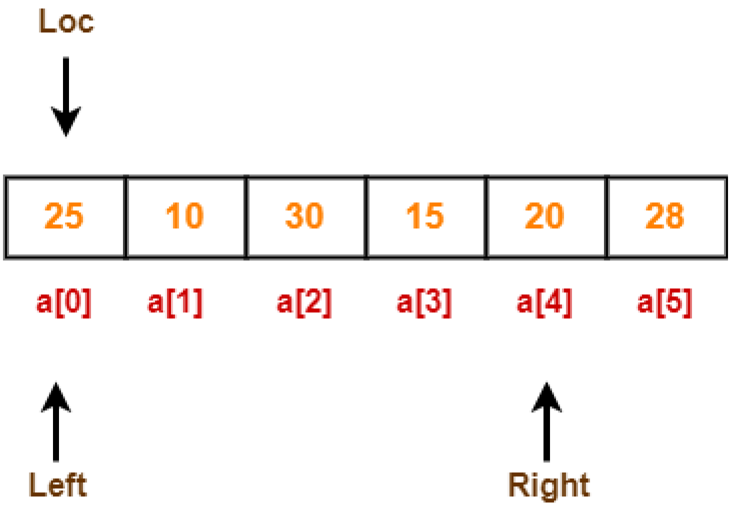
So to begin with, we set **loc** = 0, **left** = 0 and **right** = 5 as-



## Step-02:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] < a[right], so algorithm moves **right** one position towards left as-

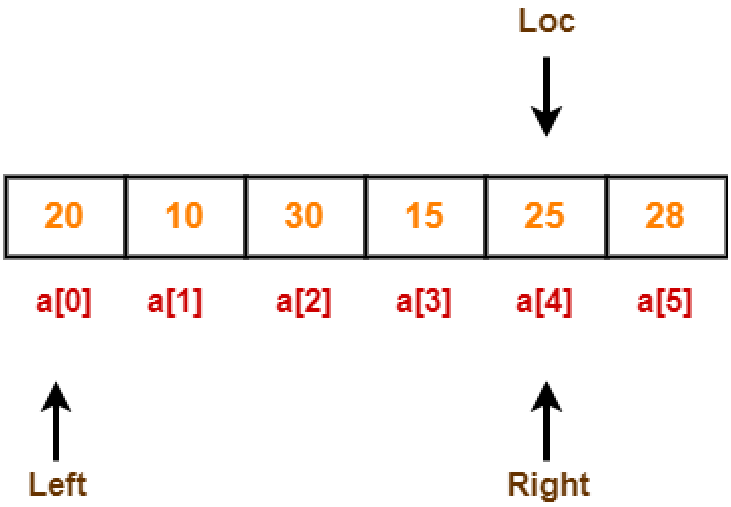


Now, **loc** = 0, **left** = 0 and **right** = 4.

Step-03:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as-

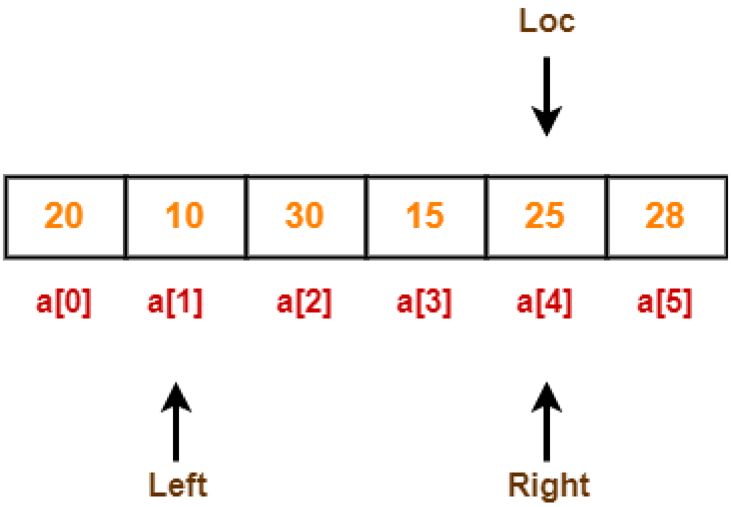


Now, **loc** = 4, **left** = 0 and **right** = 4.

Step-04:

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

As a[loc] > a[left], so algorithm moves **left** one position towards right as-

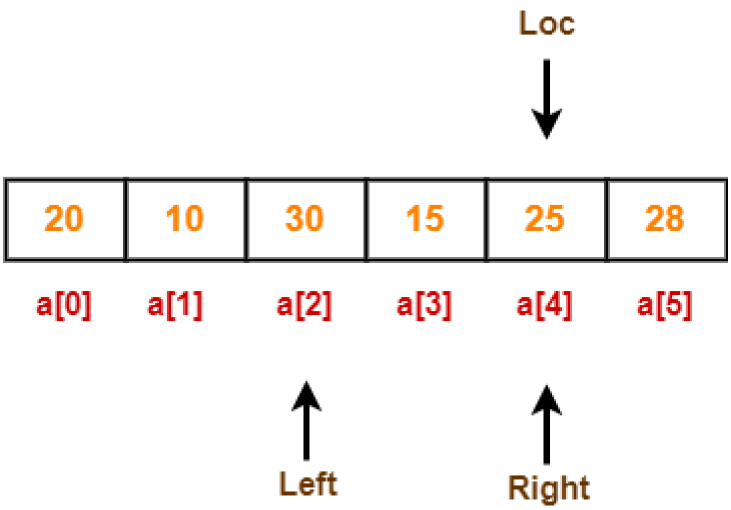


Now, **loc** = 4, **left** = 1 and **right** = 4.

Step-05:

Since **loc** points at right, so algorithm starts from **left** and move towards right.

As a[loc] > a[left], so algorithm moves **left** one position towards right as-

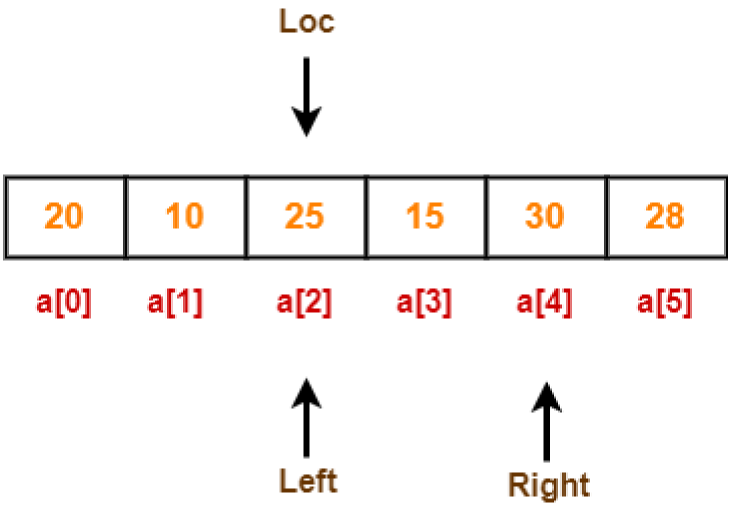


Now, **loc** = 4, **left** = 2 and **right** = 4.

Step-06:

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

As a[loc] < a[left], so we algorithm swaps a[loc] and a[left] and **loc** points at **left** as-

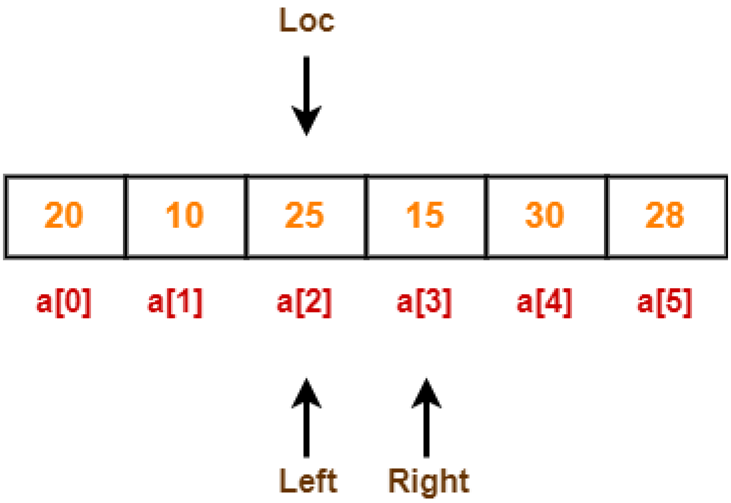


Now, **loc** = 2, **left** = 2 and **right** = 4.

## Step-07:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] < a[right], so algorithm moves **right** one position towards left as-

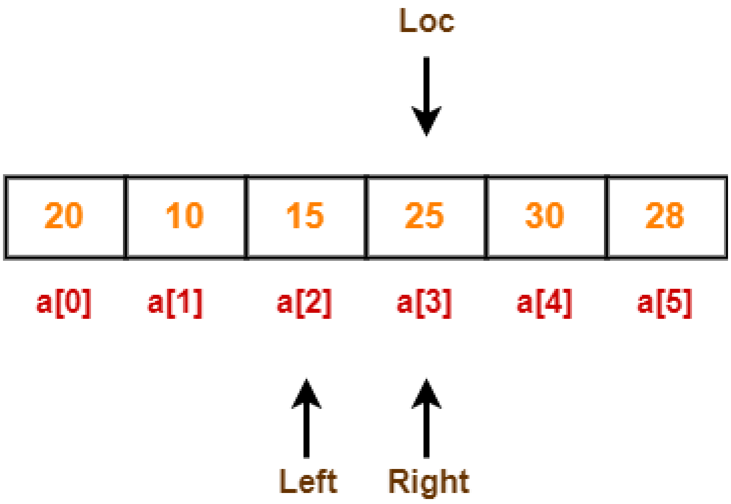


Now, **loc** = 2, **left** = 2 and **right** = 3.

Step-08:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as-

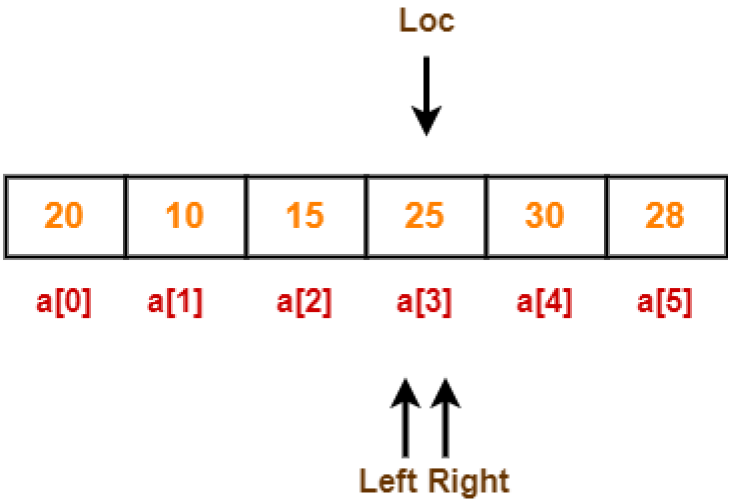


Now, **loc** = 3, **left** = 2 and **right** = 3.

Step-09:

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

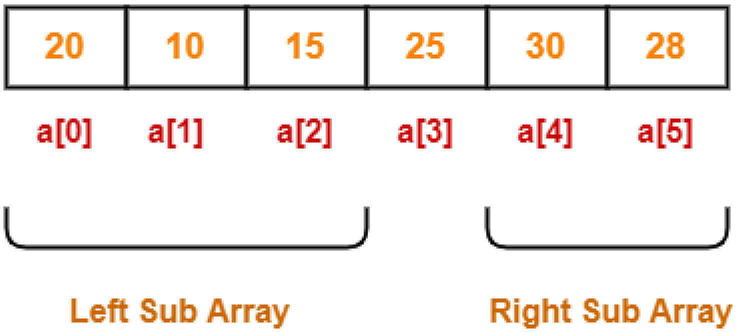
As a[loc] > a[left], so algorithm moves **left** one position towards right as-



Now, **loc** = 3, **left** = 3 and **right** = 3.

Now,

- **loc**, **left** and **right** points at the same element.
- This indicates the termination of procedure.
- The pivot element 25 is placed in its final position.
- All elements to the right side of element 25 are greater than it.
- All elements to the left side of element 25 are smaller than it.



Now, quick sort algorithm is applied on the left and right sub arrays separately in the similar manner.

## Quick Sort Analysis-

- To find the location of an element that splits the array into two parts, O(n) operations are required.
- This is because every element in the array is compared to the partitioning element.
- After the division, each section is examined separately.
- If the array is split approximately in half (which is not usually), then there will be $\log_2 n$ splits.
- Therefore, total comparisons required are $f(n) = n \times \log_2 n = O(n\log_2 n)$.

## Advantages of Quick Sort-

The advantages of quick sort algorithm are-

- Quick Sort is an in-place sort, so it requires no temporary memory.
- Quick Sort is typically faster than other algorithms.

(because its inner loop can be efficiently implemented on most architectures)

- Quick Sort tends to make excellent usage of the memory hierarchy like virtual memory or caches.
- Quick Sort can be easily parallelized due to its divide and conquer nature.

## **Disadvantages of Quick Sort-**

The disadvantages of quick sort algorithm are-

- The worst case complexity of quick sort is O($n^2$).
- This complexity is worse than O(nlogn) worst case complexity of algorithms like merge sort, heap sort etc.
- It is not a stable sort i.e. the order of equal elements may not be preserved.

## Time Complexity Analysis of Quick Sort

The average time complexity of quick sort is O(N log(N)).

At each step, the input of size N is broken into two parts say J and N-J.

```
T(N) = T(J) + T(N-J) + M(N)
```

The intuition is:

```
Time Complexity for N elements =
        Time Complexity for J elements +
        Time Complexity for N-J elements +
        Time Complexity for finding the pivot
```

where

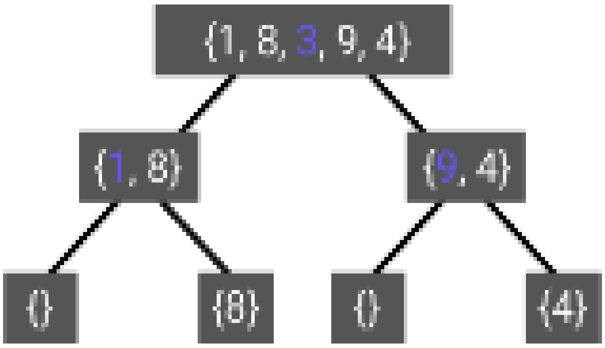
- T(N) = Time Complexity of Quick Sort for input of size N.
- T(J) = Time Complexity of Quick Sort for input of size J.
- T(N-J) = Time Complexity of Quick Sort for input of size N-J.
- M(N) = Time Complexity of finding the pivot element for N elements.

Quick Sort performs differently based on:

- How we choose the pivot? M(N) time
- How we divide the N elements -> J and N-J where J is from 0 to N-1

On solving for T(N), we will find the time complexity of Quick Sort.

# Best case Time Complexity of Quick Sort



- O(Nlog(N))
- the best case of quick sort is when we will select pivot as a mean element.
- In this case the recursion will look as shown in diagram, as we can see in diagram the height of tree is logN and in each level we will be traversing to all the elements with total operations will be logN * N
- as we have selected mean element as pivot then the array will be divided in branches of equal size so that the height of the tree will be mininum
- pivot for each recurssion is represented using blue color
- time complexity will be O(NlogN)

## Explanation

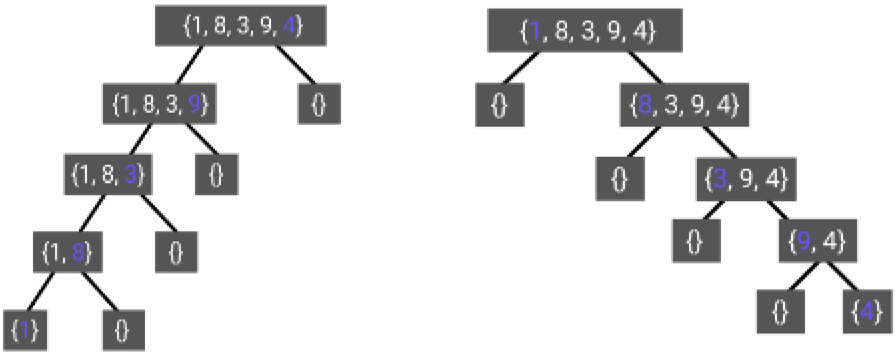
```
Lets T(n) be the time complexity for best cases
n = total number of elements
then
T(n) = 2*T(n/2) + constant*n
2*T(n/2) is because we are dividing array into two array of equal size
constant*n is because we will be traversing elements of array in each level of tree
```

```
therefore,
T(n) = 2*T(n/2) + constant*n
further we will devide arrai in to array of equalsize so
T(n) = 2*(2*T(n/4) + constant*n/2) + constant*n == 4*T(n/4) + 2*constant*n

for this we can say that
T(n) = 2^k * T(n/(2^k)) + k*constant*n
then n = 2^k
k = log2(n)

therefore,
T(n) = n * T(1) + n*logn = O(n*log2(n))
```

## Worst Case Time Complexity of Quick Sort



- O(N^2)
- This will happen when we will when our array will be sorted and we select smallest or largest indexed element as pivot
  as we can see in diagram we are always selecting pivot as corner index elements
  so height of the tree will be n and in top node we will be doing N operations
  then n-1 and so on till 1

## Explanation

```
lets T(n) ne total time complexity for worst case
```

```
n = total number of elements

T(n) = T(n-1) + constant*n
as we are dividing array into two parts one consist of single element and other of n-1
and we will traverse individual array

T(n) = T(n-2) + constant*(n-1) + constant*n =  T(n-2) + 2*constant*n - constant
T(n) = T(n-3) + 3*constant*n - 2*constant - constant
T(n) = T(n-k) + k*constant*n - (k-1)*constant ..... - 2*constant - constant

T(n) = T(n-k) + k*constant*n - constant*[(k-1) ....  + 3 + 2 + 1]
T(n) = T(n-k) + k*n*constant - constant*[k*(k-1)/2]
put n=k
T(n) = T(0) + constant*n*n - constant*[n*(n-1)/2]
removing constant terms
T(n) = n*n - n*(n-1)/2
T(n) = O(n^2)
```

- we can reduce complexity for worst case by randomly picking pivot instead of selecting start or end elements

## Average Case Time Complexity of Quick Sort

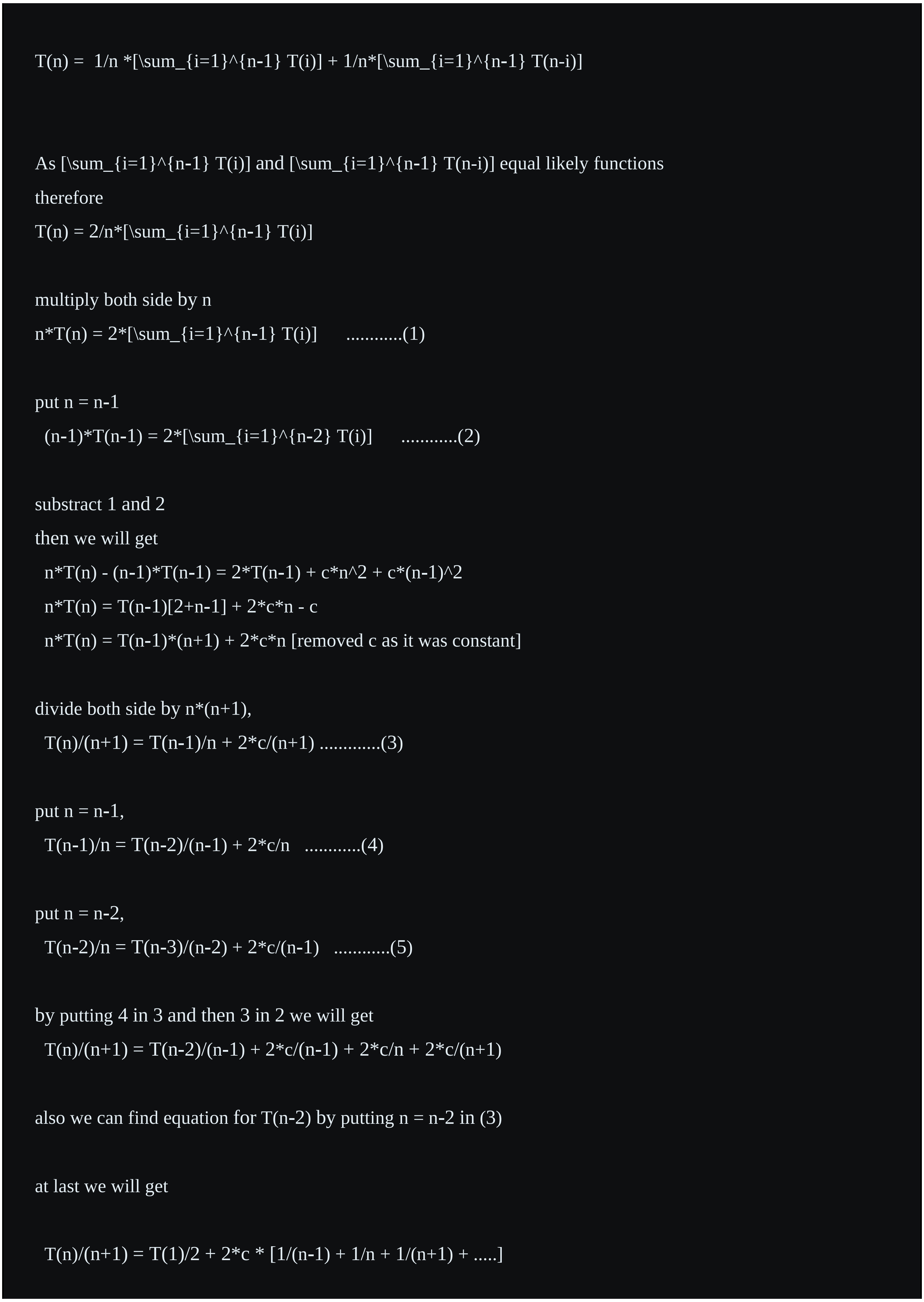
- O(Nlog(N))
- the overall average case for the quick sort is this which we will get by taking average of all complexities

## **Explanation**

```
lets T(n) be total time taken

then for average we will consider random element as pivot
lets index i be pivot

then time complexity will be
T(n) = T(i) + T(n-i)
```



T(n) =  1/n *[\sum_{i=1}^{n-1} T(i)] + 1/n*[\sum_{i=1}^{n-1} T(n-i)]

As [\sum_{i=1}^{n-1} T(i)] and [\sum_{i=1}^{n-1} T(n-i)] equal likely functions

therefore

T(n) = 2/n*[\sum_{i=1}^{n-1} T(i)]

multiply both side by n

n*T(n) = 2*[\sum_{i=1}^{n-1} T(i)]      ............(1)

put n = n-1

(n-1)*T(n-1) = 2*[\sum_{i=1}^{n-2} T(i)]      ............(2)

substract 1 and 2

then we will get

n*T(n) - (n-1)*T(n-1) = 2*T(n-1) + c*n^2 + c*(n-1)^2

n*T(n) = T(n-1)[2+n-1] + 2*c*n - c

n*T(n) = T(n-1)*(n+1) + 2*c*n [removed c as it was constant]

divide both side by n*(n+1),

T(n)/(n+1) = T(n-1)/n + 2*c/(n+1) .............(3)

put n = n-1,

T(n-1)/n = T(n-2)/(n-1) + 2*c/n   ............(4)

put n = n-2,

T(n-2)/n = T(n-3)/(n-2) + 2*c/(n-1)   ............(5)

by putting 4 in 3 and then 3 in 2 we will get

T(n)/(n+1) = T(n-2)/(n-1) + 2*c/(n-1) + 2*c/n + 2*c/(n+1)

also we can find equation for T(n-2) by putting n = n-2 in (3)

at last we will get

T(n)/(n+1) = T(1)/2 + 2*c * [1/(n-1) + 1/n + 1/(n+1) + .....]

```
  T(n)/(n+1) = T(1)/2 + 2*c*log(n) + C

  T(n) = 2*c*log(n) * (n+1)

now by removing constants,

  T(n) = log(n)*(n+1)

therefore,

  T(n) = O(n*log(n))
```

Space Complexity

- O(N)
- as we are not creating any container other then given array therefore Space complexity will be in order of N

The derivation is based on the following notation:
T(N) = Time Complexity of Quick Sort for input of size N.

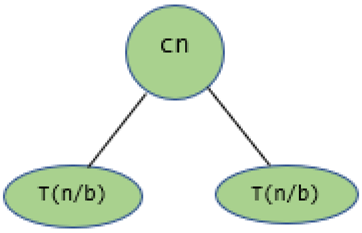
## Recursion Tree Method

The recursion tree method is commonly used in cases where the problem gets divided into smaller problems, typically of the same size. A recurrence tree is drawn, branching until the base case is reached. Then, we sum the total time taken at all levels in order to derive the overall time complexity.

For example, consider the following example:

T(n) = aT(n/b) + cn

Here, the problem is getting split into *a* subproblems, each of which has a size of *n/b*. Hence, the first level of the recurrence tree would look as follows:

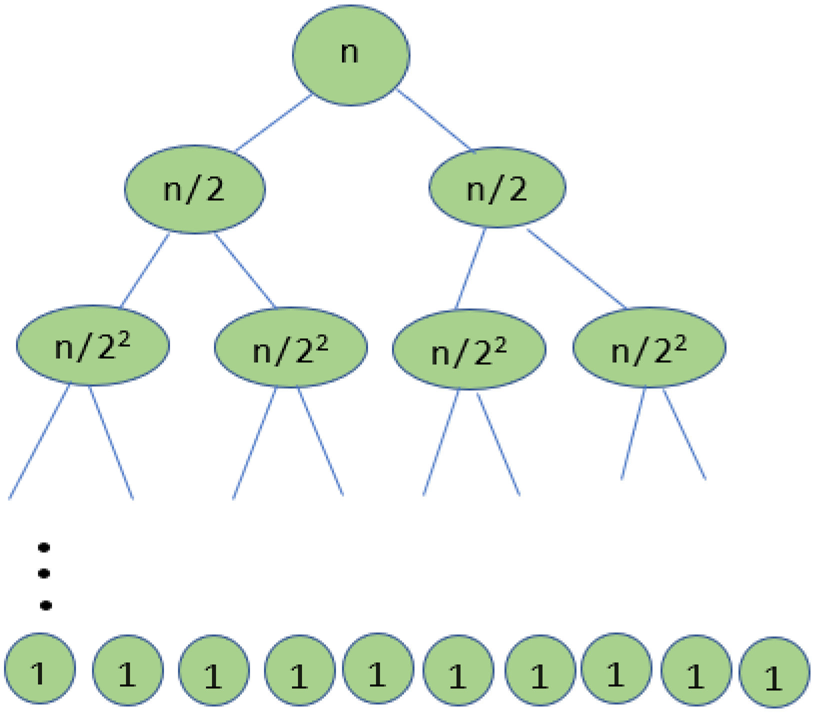


**Example**

## Example: T(n) = 2T(n/2) + n

In this problem, one can observe that the problem is getting split into two problems of half the initial size. Further the additional cost here equals the size. Hence, after the first division, the recursion tree will have two child nodes with input size *n/2*. We then proceed in this manner until the final input size becomes 1.

Therefore, the final recursion tree will look as follows (note that each node contains only the extra cost that is taken) :



As the recursion tree is complete, it remains to calculate the total sum of the entries. For that, we first need to determine the number of levels in the recursion tree. Since each level of the tree splits each of the nodes in that level to half the size of their parents, one can conclude that the total number of levels here is $log_2n$. The next thing we note here is that in each level, the sum of the nodes is *n*. Therefore, the overall time complexity is given by:

$T(n) = n + n + .... \ log_2n \ times$

$= n ( 1 + 1 + ....log_2n \ times)$

$= n \log_2 n$

$= \theta(n \log_2 n)$

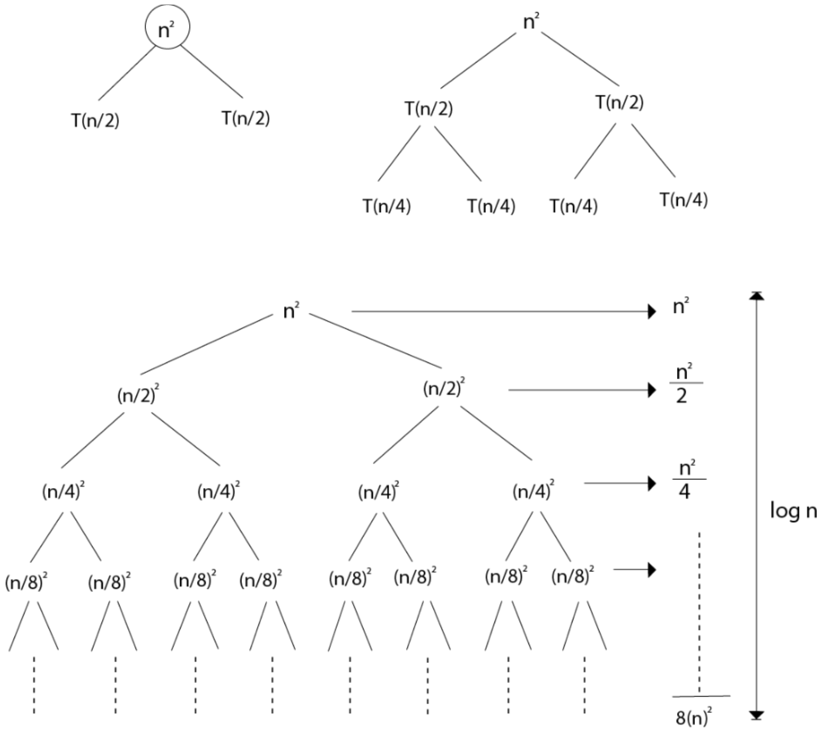
Therefore, the overall time complexity of the operation with the given recurrence equation is given by $\theta(n \log_2 n)$.

## Example 1

Consider $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

We have to obtain the asymptotic bound using recursion tree method.

**Solution:** The Recursion tree for the above recurrence is



$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \text{log n times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

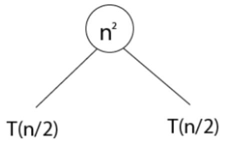
$$\mathbf{T\ (n) = \theta n^2}$$

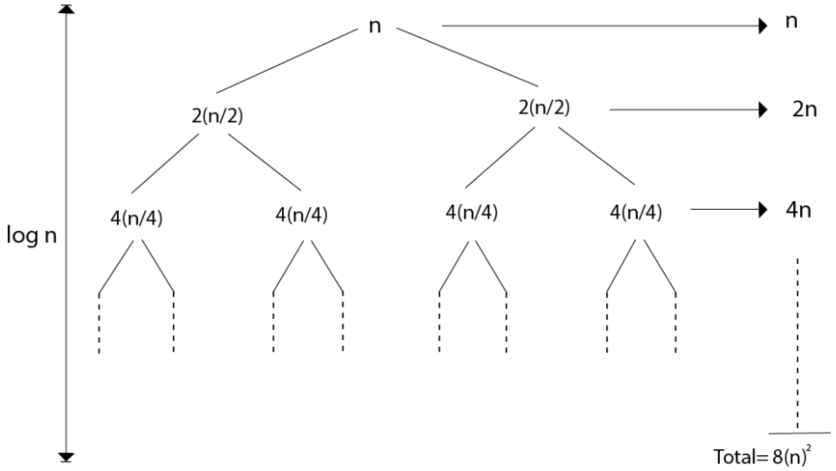
**Example 2:** Consider the following recurrence

$$T\ (n) = 4T\left(\frac{n}{2}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

**Solution:** The recursion trees for the above recurrence





We have n + 2n+ 4n +……$\log_2$ n times

= n (1+ 2+4+…..$\log_2$ n times)

$$= n\frac{(2\ \log_2 n-1)}{(2-1)} = \frac{n(n-1)}{1} = n^2 - n = \theta(n^2)$$

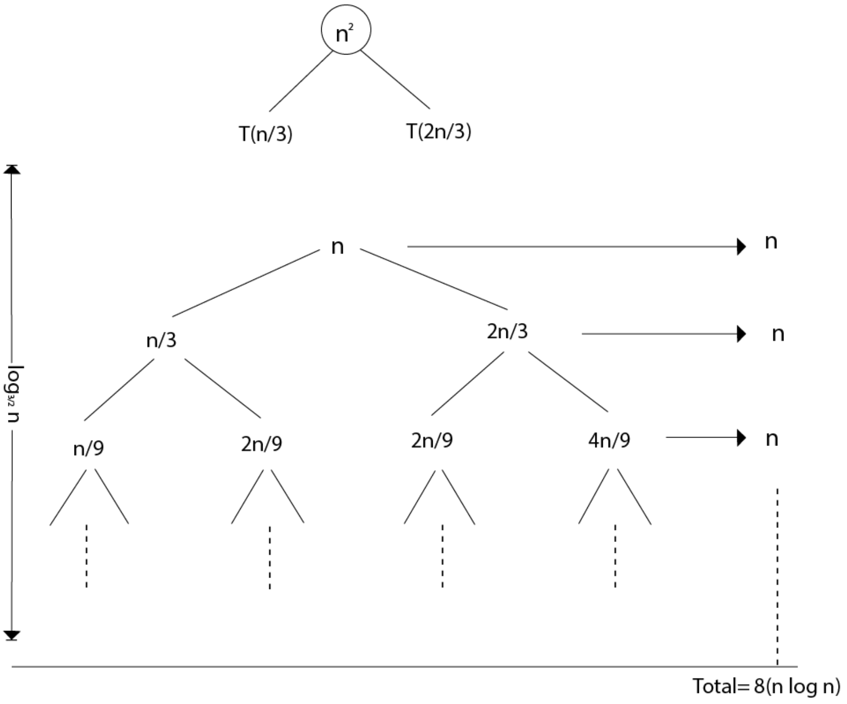
$$\mathbf{T\ (n) = \theta(n^2)}$$

**Example 3:** Consider the following recurrence

$$T\ (n) = T\left(\frac{n}{3}\right) + \ T\left(\frac{2n}{3}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

**Solution:** The given Recurrence has the following recursion tree



2

When we add the values across the levels of the recursion trees, we get a value of n for every level. The longest path from the root to leaf is

$$n \longrightarrow \frac{2}{3}n \longrightarrow \left(\frac{2}{3}\right) n \longrightarrow \ldots.1$$

Since $\left(\frac{2}{3}\right)$ n=1 when i=$\log_{\frac{3}{2}}$ n.

Thus the height of the tree is $\log_{\frac{3}{2}}$ n.

$$T\,(n) = n + n + n + \ldots\ldots + \log_{\frac{3}{2}} n \text{ times.} = \boldsymbol{\theta(n \log n)}$$

## Master Theorem-

**Master's theorem solves recurrence relations of the form**

$$\boxed{T\,(n) = a\,T\left(\frac{n}{b}\right) + \theta\,(\,n^{k} \log^{p} n)}$$

Master's Theorem

**Here, a >= 1, b > 1, k >= 0 and p is a real number.**

## Master Theorem Cases-

To solve recurrence relations using Master's theorem, we compare a with $b^k$.

hen, we follow the following cases-

## Case-01:

If $a > b^k$, then $T(n) = \theta\ (n^{\log_b a})$

## Case-02:

If $a = b^k$ and

- If $p < -1$, then $T(n) = \theta\ (n^{\log_b a})$
- If $p = -1$, then $T(n) = \theta\ (n^{\log_b a}.\log^2 n)$
- If $p > -1$, then $T(n) = \theta\ (n^{\log_b a}.\log^{p+1} n)$

## Case-03:

If $a < b^k$ and

- If $p < 0$, then $T(n) = O\ (n^k)$
- If $p >= 0$, then $T(n) = \theta\ (n^k \log^p n)$

## PRACTICE PROBLEMS BASED ON MASTER THEOREM-

### Problem-01:

Solve the following recurrence relation using Master's theorem-

$$T(n) = 3T(n/2) + n^2$$

**Solution-**

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta\ (n^k \log^p n)$.

Then, we have-

$$a = 3$$

$$b = 2$$

$$k = 2$$

$$p = 0$$

Now, $a = 3$ and $b^k = 2^2 = 4$.

Clearly, $a < b^k$.

So, we follow case-03.

Since $p = 0$, so we have-

$$T(n) = \theta\ (n^k \log^p n)$$

$$T(n) = \theta\ (n^2 \log^0 n)$$

Thus,

$$\boxed{\mathbf{T(n) = \theta\ (n^2)}}$$

## Problem-02:

Solve the following recurrence relation using Master's theorem-

$$T(n) = 2T(n/2) + n\log n$$

## Solution-

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta\ (n^k\log^p n)$.

Then, we have-

$$a = 2$$

$$b = 2$$

$$k = 1$$

$$p = 1$$

Now, $a = 2$ and $b^k = 2^1 = 2$.

Clearly, $a = b^k$.

So, we follow case-02.

Since $p = 1$, so we have-

$$T(n) = \theta\ (n^{\log_b a}.\log^{p+1}n)$$

$$T(n) = \theta\ (n^{\log_2 2}.\log^{1+1}n)$$

Thus,

$$\boxed{\mathbf{T(n) = \theta\ (nlog^2n)}}$$

## Problem-03:

Solve the following recurrence relation using Master's theorem-

$$T(n) = 2T(n/4) + n^{0.51}$$

## Solution-

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta\ (n^k log^p n)$.

Then, we have-

$$a = 2$$

$$b = 4$$

$$k = 0.51$$

$$p = 0$$

Now, $a = 2$ and $b^k = 4^{0.51} = 2.0279$.

Clearly, $a < b^k$.

So, we follow case-03.

Since $p = 0$, so we have-

$$T(n) = \theta\ (n^k \log^p n)$$

$$T(n) = \theta\ (n^{0.51} \log^0 n)$$

Thus,

$$\boxed{\mathbf{T(n) = \theta\ (n^{0.51})}}$$

## Problem-04:

Solve the following recurrence relation using Master's theorem-

$$T(n) = \sqrt{2}T(n/2) + \log n$$

## Solution-

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta\ (n^k \log^p n)$.

Then, we have-

$$a = \sqrt{2}$$

$$b = 2$$

$$k = 0$$

$$p = 1$$

Now, $a = \sqrt{2} = 1.414$ and $b^k = 2^0 = 1$.

Clearly, $a > b^k$.

So, we follow case-01.

So, we have-

$$T(n) = \theta\ (n^{\log_b a})$$

$$T(n) = \theta\ (n^{\log_2 \sqrt{2}})$$

$$T(n) = \theta\ (n^{1/2})$$

Thus,

$$\boxed{\mathbf{T(n) = \theta\ (\sqrt{n})}}$$

## Problem-05:

Solve the following recurrence relation using Master's theorem-

$$T(n) = 8T(n/4) – n^2 \log n$$

## Solution-

- The given recurrence relation does not correspond to the general form of Master's theorem.

- So, it can not be solved using Master’s theorem.

## Problem-06:

Solve the following recurrence relation using Master’s theorem-

$$T(n) = 3T(n/3) + n/2$$

## Solution-

- We write the given recurrence relation as T(n) = 3T(n/3) + n.
- This is because in the general form, we have θ for function f(n) which hides constants in it.
- Now, we can easily apply Master’s theorem.

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.

Then, we have-

$$a = 3$$

$$b = 3$$

$$k = 1$$

$$p = 0$$

Now, $a = 3$ and $b^k = 3^1 = 3$.

Clearly, $a = b^k$.

So, we follow case-02.

Since $p = 0$, so we have-

$$T(n) = \theta\ (n^{\log_b a}.\log^{p+1} n)$$

$$T(n) = \theta\ (n^{\log_3 3}.\log^{0+1} n)$$

$$T(n) = \theta\ (n^1.\log^1 n)$$

Thus,

$$\boxed{\mathbf{T(n) = \theta\ (nlogn)}}$$

## Problem-07:

Form a recurrence relation for the following code and solve it using Master’s theorem-

```
A(n)
{
if(n<=1)
return 1;
else
return A(√n);
}
```

**Solution-**

- We write a recurrence relation for the given code as $T(n) = T(\sqrt{n}) + 1$.
- Here 1 = Constant time taken for comparing and returning the value.
- We can not directly apply Master's Theorem on this recurrence relation.
- This is because it does not correspond to the general form of Master's theorem.
- However, we can modify and bring it in the general form to apply Master's theorem.

Let-

$$n = 2^m \ldots\ldots(1)$$

Then-

$$T(2^m) = T(2^{m/2}) + 1$$

Now, let $T(2^m) = S(m)$, then $T(2^{m/2}) = S(m/2)$

So, we have-

$$S(m) = S(m/2) + 1$$

Now, we can easily apply Master's Theorem.

We compare the given recurrence relation with $S(m) = aS(m/b) + \theta\ (m^k \log^p m)$.

Then, we have-

$$a = 1$$

$$b = 2$$

$$k = 0$$

$$p = 0$$

Now, $a = 1$ and $b^k = 2^0 = 1$.

Clearly, $a = b^k$.

So, we follow case-02.

Since $p = 0$, so we have-

$$S(m) = \theta\ (m^{\log_b a} . \log^{p+1} m)$$

$$S(m) = \theta\ (m^{\log_2 1} . \log^{0+1} m)$$

$$S(m) = \theta\ (m^0 . \log^1 m)$$

Thus,

$$S(m) = \theta(\log m) \text{ ……(2)}$$

Now,

- From (1), we have $n = 2^m$.
- So, $\log n = m\log 2$ which implies $m = \log_2 n$.

Substituting in (2), we get-

$$S(m) = \theta(\log\log_2 n)$$

# UNIT-2

## Advanced Design and analysis Techniques

Dynamic programming:- Elements, Matrix-chain multiplication, longest common subsequence,

Greedy algorithms:- Elements , Activity- Selection problem, Huffman codes, Task scheduling problem, Travelling Salesman Problem.

Advanced data Structures:- Binomial heaps, Fibonacci heaps, Splay Trees, Red-Black Trees.

## Dynamic Programming:-

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.

### Elements of Dynamic programming

**Dynamic programming** posses two important elements which are as given below:

1. **Overlapping sub problem**
   One of the main characteristics is to split the problem into subproblem, as similar as divide and conquer approach. The overlapping subproblem is found in that problem where bigger problems share the same smaller problem. However unlike divide and conquer there are many subproblems in which overlap cannot be treated distinctly or independently. Basically, there are two ways for handling the overlapping subproblems:

   **a .Top down approach**
   It is also termed as memoization technique. In this, the problem is broken into subproblem and these subproblems are solved and the solutions are remembered, in case if they need to be solved in future. Which means that the values are stored in a data structure, which will help us to reach them efficiently when the same problem will occur during the program execution.

   **b.Bottom up approach**
   It is also termed as tabulation technique. In this, all subproblems are needed to be solved in advance and then used to build up a solution to the larger problem.

2. **Optimal sub structure**
   It implies that the optimal solution can be obtained from the optimal solution of its subproblem. So optimal substructure is simply an optimal selection

among all the possible substructures that can help to select the best structure of the same kind to exist.

Example :

1. LCS(Longest Chain Subsequence)

2. MCM(Matrix Chain Multiplication)

1.LCS (Longest Chain Subsequence )

Subsequence: A subsequence of a given sequence is just the given sequence with some elements left out.

Given two sequences X and Y, we say that the sequence is:

X={x1,x2,x3……xn}

Y={ y1,y2,y3……yn}

The aim this problem is to find a maximum Length Common Sequence of X & Y . This problem is applicable in DNA matching in which we need to find out how similar are two stands of DNA or how closely related .

Algorithm of LCS:

**Algorithm: LCS-Length-Table-Formulation (X, Y)**

```
m := length(X)
n := length(Y)
for i = 1 to m do
```

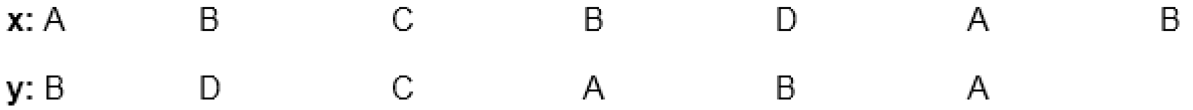
```
  C[i, 0] := 0
for j = 1 to n do
  C[0, j] := 0
for i = 1 to m do
  for j = 1 to n do
    if $x_i = y_j$
      C[i, j] := C[i - 1, j - 1] + 1
      B[i, j] := ‘D’
    else
      if C[i -1, j] ≥ C[i, j -1]
        C[i, j] := C[i - 1, j] + 1
        B[i, j] := ‘U’
      else
      C[i, j] := C[i, j - 1]
      B[i, j] := ‘L’
return C and B
```

```
Algorithm: Print-LCS (B, X, i, j)
if i = 0 and j = 0
  return
if B[i, j] = ‘D’
  Print-LCS(B, X, i-1, j-1)
  Print($x_i$)
else if B[i, j] = ‘U’
  Print-LCS(B, X, i-1, j)
else
  Print-LCS(B, X, i, j-1)
```

This algorithm will print the longest common subsequence of **X** and **Y**.

## Example of Longest Common Sequence

**Example:** Given two sequences X [1...m] and Y [1.....n]. Find the longest common subsequences to both.



| | | | | | | |
|---|---|---|---|---|---|---|
| **x:** A | B | C | B | D | A | B |
| **y:** B | D | C | A | B | A | |

here X = (A,B,C,B,D,A,B) and Y = (B,D,C,A,B,A)

m = length [X] and n = length [Y]

m = 7 and n = 6

Here $x_1$= x [1] = A   $y_1$= y [1] = B

$x_2$= B  $y_2$= D

$x_3$= C  $y_3$= C

$x_4$= B  $y_4$= A

$x_5$= D  $y_5$= B

$x_6$= A  $y_6$= A

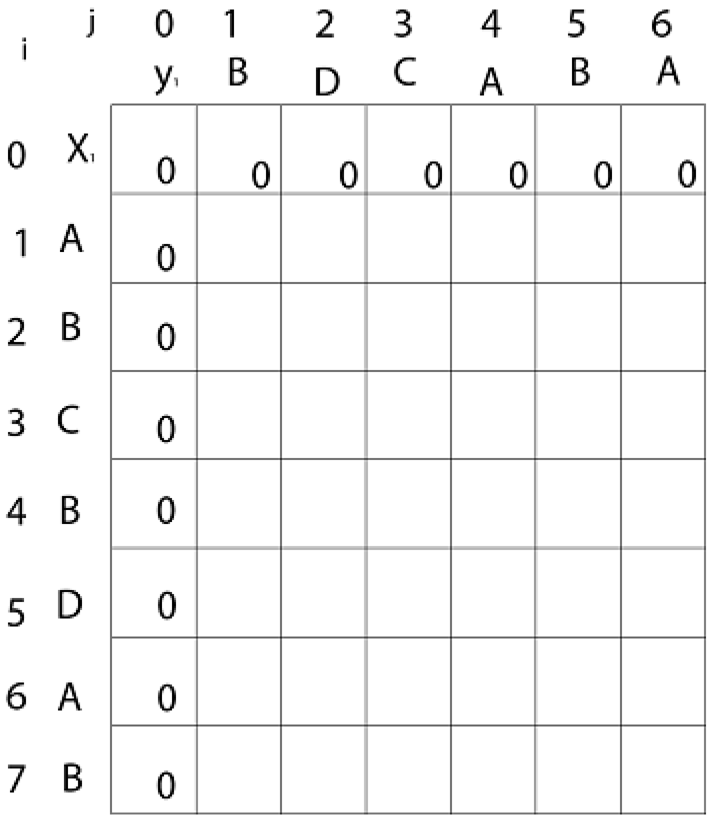
$x_7$= B

Now fill the values of c [i, j] in m x n table

Initially, for i=1 to 7 c [i, 0] = 0

For j = 0 to 6 c [0, j] = 0

That is:



| i | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
|  |  | $y_i$ | B | D | C | A | B | A |
| 0 | $X_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 |  |  |  |  |  |  |
| 2 | B | 0 |  |  |  |  |  |  |
| 3 | C | 0 |  |  |  |  |  |  |
| 4 | B | 0 |  |  |  |  |  |  |
| 5 | D | 0 |  |  |  |  |  |  |
| 6 | A | 0 |  |  |  |  |  |  |
| 7 | B | 0 |  |  |  |  |  |  |

**Now for i=1 and j = 1**

$x_1$ and $y_1$ we get $x_1 \neq y_1$ i.e. $A \neq B$

And c [i-1,j] = c [0, 1] = 0

c [i, j-1] = c [1,0 ] = 0

That is, c [i-1,j]= c [i, j-1] so c [1, 1] = 0 and b [1, 1] = ' ↑ '

**Now for i=1 and j = 2**

$x_1$ and $y_2$ we get $x_1 \neq y_2$ i.e. $A \neq D$

c [i-1,j] = c [0, 2] = 0

c [i, j-1] = c [1,1 ] = 0

That is, c [i-1,j]= c [i, j-1] and c [1, 2] = 0 b [1, 2] = ' ↑ '

**Now for i=1 and j = 3**

$x_1$ and $y_3$ we get $x_1 \neq y_3$ i.e. $A \neq C$

c [i-1,j] = c [0, 3] = 0

c [i, j-1] = c [1,2 ] = 0

so c [1,3] = 0 b [1,3] = ' ↑ '

**Now for i=1 and j = 4**

$x_1$ and $y_4$ we get. $x_1=y_4$ i.e A = A

c [1,4] = c [1-1,4-1] + 1

= c [0, 3] + 1

= 0 + 1 = 1

c [1,4] = 1

b [1,4] = ' ↖ '

**Now for i=1 and j = 5**

$x_1$ and $y_5$ we get $x_1 \neq y_5$

c [i-1,j] = c [0, 5] = 0

c [i, j-1] = c [1,4 ] = 1

Thus c [i, j-1] > c [i-1,j] i.e. c [1, 5] = c [i, j-1] = 1. So b [1, 5] = '←'

**Now for i=1 and j = 6**

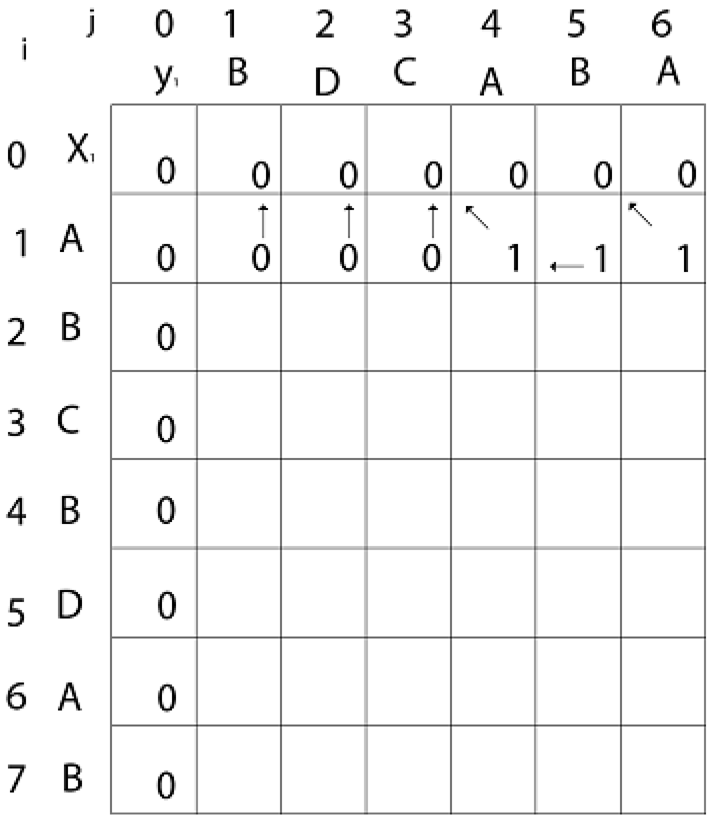
$x_1$ and $y_6$ we get $x_1=y_6$

c [1, 6] = c [1-1,6-1] + 1

= c [0, 5] + 1 = 0 + 1 = 1

c [1,6] = 1

b [1,6] = ' ↖ '



| i | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
|  |  | $y_i$ | B | D | C | A | B | A |
| 0 | $X_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ↖ 1 |
| 2 | B | 0 |  |  |  |  |  |  |
| 3 | C | 0 |  |  |  |  |  |  |
| 4 | B | 0 |  |  |  |  |  |  |
| 5 | D | 0 |  |  |  |  |  |  |
| 6 | A | 0 |  |  |  |  |  |  |
| 7 | B | 0 |  |  |  |  |  |  |

**Now for i=2 and j = 1**

We get $x_2$ and $y_1$ B = B i.e. $x_2 = y_1$

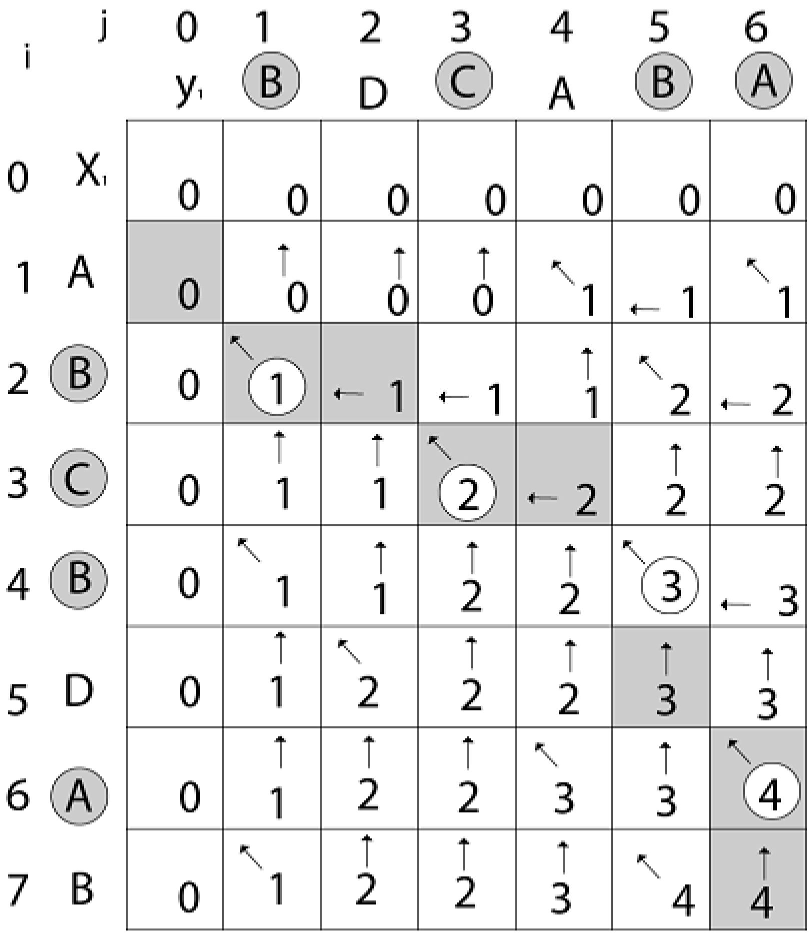
c [2,1] = c [2-1,1-1] + 1

= c [1, 0] + 1

= 0 + 1 = 1

c [2, 1] = 1 and b [2, 1] = ' ↖ '

Similarly, we fill the all values of c [i, j] and we get



| i | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
|  |  | $y_i$ | B | D | C | A | B | A |
| 0 | $X_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ↖ 1 |
| 2 | B | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 1 | ↖ 2 | ← 2 |
| 3 | C | 0 | ↑ 1 | ↑ 1 | ↖ 2 | ← 2 | ↑ 2 | ↑ 2 |
| 4 | B | 0 | ↖ 1 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3 |
| 5 | D | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| 6 | A | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ↑ 3 | ↖ 4 |
| 7 | B | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 | ↑ 4 |

**Step 4: Constructing an LCS:** The initial call is PRINT-LCS (b, X, X.length, Y.length)

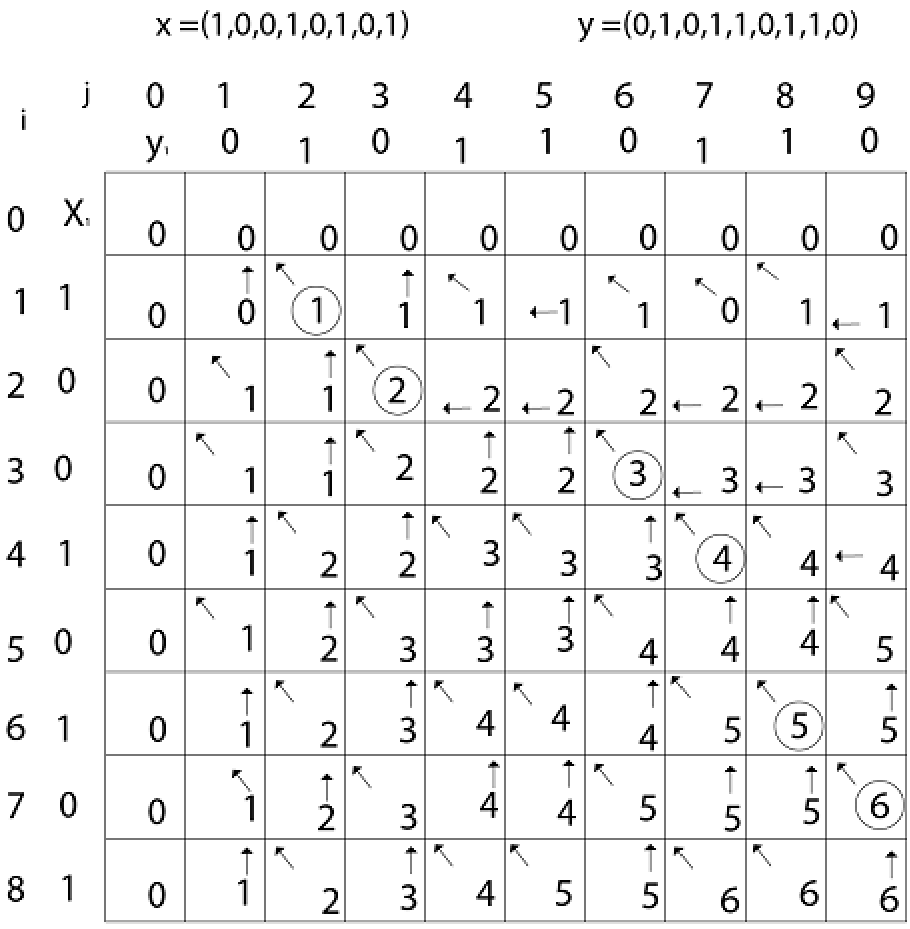
```
PRINT-LCS (b, x, i, j)
if i=0 or j=0
then return
if b [i,j] = ' ↖ '
then PRINT-LCS (b,x,i-1,j-1)
print x_i
else if b [i,j] = ' ↑ '
then PRINT-LCS (b,X,i-1,j)
else PRINT-LCS (b,X,i,j-1)
```

**Example:** Determine the LCS of (1,0,0,1,0,1,0,1) and (0,1,0,1,1,0,1,1,0).

**Solution:** let X = (1,0,0,1,0,1,0,1) and Y = (0,1,0,1,1,0,1,1,0).

$$c\,[i,j] = \left\{ \begin{array}{lll} 0 & \text{if } i = 0 & \text{or} \quad j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 & \text{and} \quad x_i = y_i \\ \max(c[i, j-1], c[i-1, j]) \text{if } i, j > 0 \text{ and } & & x_i \neq y_i \end{array} \right\}$$

We are looking for c [8, 9]. The following table is built.



x =(1,0,0,1,0,1,0,1) y =(0,1,0,1,1,0,1,1,0)

| i | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  | $y_i$ | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | $X_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | ↑ 0 | ↖ ① | ↑ 1 | ↖ 1 | ← 1 | ↖ 1 | ↖ 0 | ↖ 1 | ← 1 |
| 2 | 0 | 0 | ↖ 1 | ↑ 1 | ↖ ② | ← 2 | ← 2 | ↖ 2 | ← 2 | ← 2 | ↖ 2 |
| 3 | 0 | 0 | ↖ 1 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↖ ③ | ← 3 | ← 3 | ↖ 3 |
| 4 | 1 | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↖ 3 | ↖ 3 | ↑ 3 | ↖ ④ | ↖ 4 | ← 4 |
| 5 | 0 | 0 | ↖ 1 | ↑ 2 | ↖ 3 | ↑ 3 | ↑ 3 | ↖ 4 | ↑ 4 | ↑ 4 | ↖ 5 |
| 6 | 1 | 0 | ↑ 1 | ↖ 2 | ↑ 3 | ↖ 4 | ↖ 4 | ↑ 4 | ↖ 5 | ↖ ⑤ | ↑ 5 |
| 7 | 0 | 0 | ↖ 1 | ↑ 2 | ↖ 3 | ↑ 4 | ↑ 4 | ↖ 5 | ↑ 5 | ↑ 5 | ↖ ⑥ |
| 8 | 1 | 0 | ↑ 1 | ↖ 2 | ↑ 3 | ↖ 4 | ↖ 5 | ↑ 5 | ↖ 6 | ↖ 6 | ↑ 6 |

From the table we can deduct that LCS = 6. There are several such sequences, for instance (1,0,0,1,1,0) (0,1,0,1,0,1) and (0,0,1,1,0,1)

## 2) Matrix Chain Multiplication

It is a Method under Dynamic Programming in which previous output is taken as input for next.

Here, Chain means one matrix's column is equal to the second matrix's row

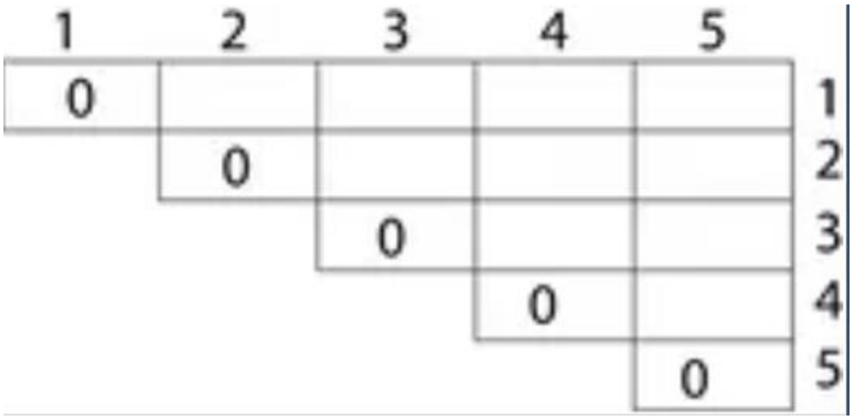
## Algorithm of Matrix Chain Multiplication

**MATRIX-CHAIN-ORDER (p)**

```
n   length[p]-1
for i ← 1 to n
    do m [i, i] ← 0
for l ← 2 to n    // l is the chain length
    do for i ← 1 to n-l + 1
        do j ← i+ l -1
            m[i,j] ← ∞
            for k ← i to j-1
                do q ← m [i, k] + m [k + 1, j] + $p_{i-1} p_k p_j$
                If q < m [i,j]
                    then m [i,j] ← q
                    s [i,j] ← k
return m and s.
```

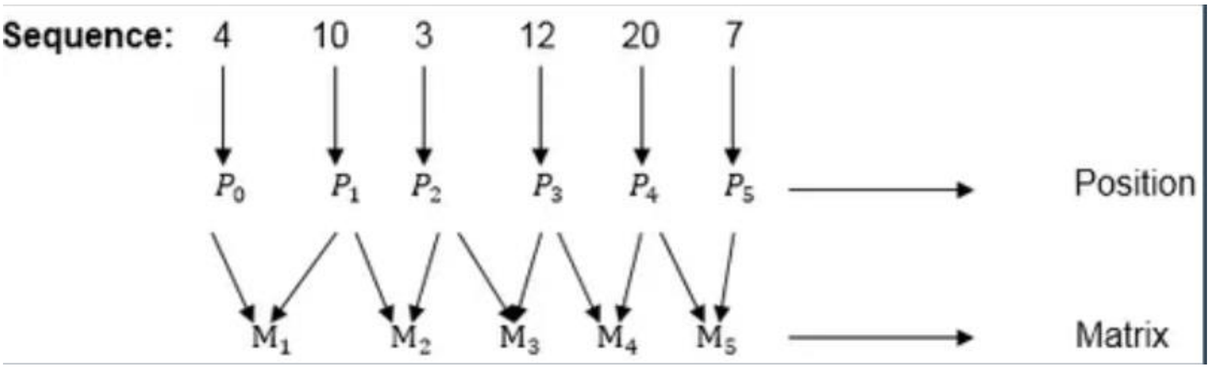
## Example Problem of Matrix Chain Multiplication

**Example-1 :** We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], 0 ≤ i, j≤ 5. We know M [i, i] = 0 for all i.



| 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|
| 0 | | | | | 1 |
| | 0 | | | | 2 |
| | | 0 | | | 3 |
| | | | 0 | | 4 |
| | | | | 0 | 5 |

Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.


Sequence: 4 10 3 12 20 7
$P_0$ $P_1$ $P_2$ $P_3$ $P_4$ $P_5$
Position
$M_1$ $M_2$ $M_3$ $M_4$ $M_5$
Matrix

Here P0 to P5 are Position and M1 to M5 are matrix of size (pi to pi-1)

On the basis of sequence, we make a formula , for $M_i$ ------> p[i] as column and p[i-1] as row .

In Dynamic Programming, initialization of every method done by '0'.So we initialize it by '0'.It will sort out diagonally.

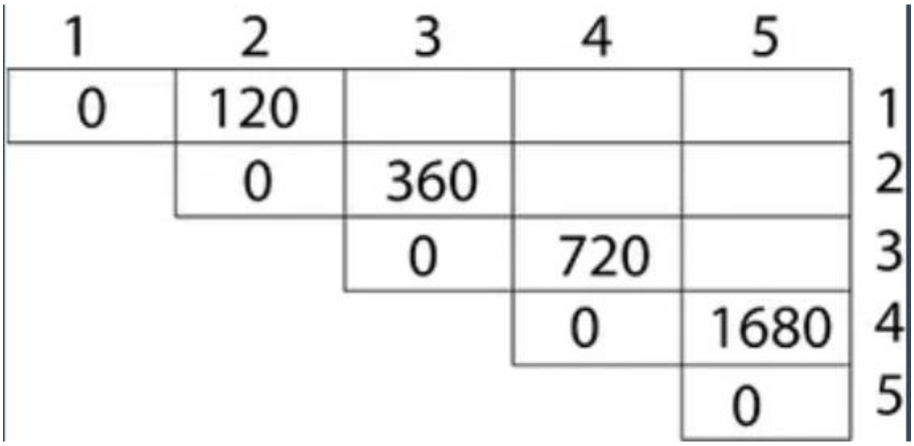
We have to sort out all the combination but the minimum output combination is taken into consideration.

**Calculation of Product of 2 matrices:**
**1. m (1,2) = m1 x m2**
**= 4 x 10 x 10 x 3**
**= 4 x 10 x 3 = 120**

**2. m (2, 3) = m2 x m3**
**= 10 x 3 x 3 x 12**
**= 10 x 3 x 12 = 360**

**3. m (3, 4) = m3 x m4**
**= 3 x 12 x 12 x 20**
**= 3 x 12 x 20 = 720**

**4. m (4,5) = m4 x m5**
**= 12 x 20 x 20 x 7**
**= 12 x 20 x 7 = 1680**



| 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|
| 0 | 120 | | | | 1 |
| | 0 | 360 | | | 2 |
| | | 0 | 720 | | 3 |
| | | | 0 | 1680 | 4 |
| | | | | 0 | 5 |

- We initialize the diagonal element with equal i,j value with '0'.
- After that second diagonal is sorted out and we get all the values corresponded to it

Now the third diagonal will be solved out in the same way.

**Now product of 3 matrices:**

M [1, 3] = $M_1$ $M_2$ $M_3$

1. There are two cases by which we can solve this multiplication: ( $M_1$ x $M_2$) + $M_3$, $M_1$+ ($M_2$x $M_3$)

2. After solving both cases we choose the case in which minimum output is there.

$$M\,[1,3] = \min \begin{cases} M\,[1,2] + M\,[3,3] + p_0\,p_2 p_3 = 120 + 0 + 4.3.12 \;=\; 264 \\ M\,[1,1] + M\,[2,3] + p_0\,p_1 p_3 = 0 + 360 + 4.10.12 \;=\; 840 \end{cases}$$

**M [1, 3] =264**

As Comparing both output **264** is minimum in both cases so we insert **264** in table and ( $M_1$ x $M_2$) + $M_3$ this combination is chosen for the output making.

M [2, 4] = $M_2$ $M_3$ $M_4$

1. There are two cases by which we can solve this multiplication: ($M_2$x $M_3$)+$M_4$, $M_2$+($M_3$ x $M_4$)
2. After solving both cases we choose the case in which minimum output is there.

$$M\,[2,4] = \min \begin{cases} M[2,3] + M[4,4] + \; p_1 p_3 p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + \; M[3,4] + \; p_1 p_2 p_4 = 0 + 720 + 10.3.20 = \; 1320 \end{cases}$$

**M [2, 4] = 1320**

As Comparing both output **1320** is minimum in both cases so we insert **1320** in table and $M_2$+($M_3$ x $M_4$) this combination is chosen for the output making.

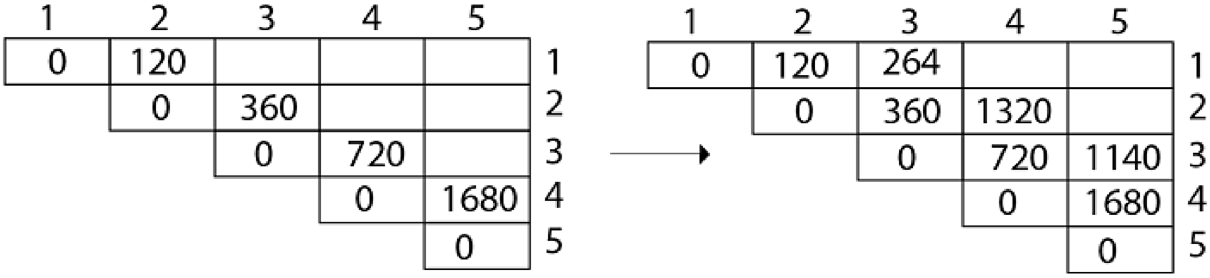
M [3, 5] = $M_3$  $M_4$  $M_5$

1. There are two cases by which we can solve this multiplication: ( $M_3$ x $M_4$) + $M_5$, $M_3$+ ( $M_4$x$M_5$)
2. After solving both cases we choose the case in which minimum output is there.

$$M\,[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3\,.20.7 \;=\; 1140 \\ M[3,3] + \; M\,[4,5] + \; p_2 p_3 p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases}$$

$M[3, 5] = 1140$

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and ( $M_3$ x $M_4$) + $M_5$this combination is chosen for the output making.



| 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|
| 0 | 120 | | | | 1 |
| | 0 | 360 | | | 2 |
| | | 0 | 720 | | 3 |
| | | | 0 | 1680 | 4 |
| | | | | 0 | 5 |

→

| 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|
| 0 | 120 | 264 | | | 1 |
| | 0 | 360 | 1320 | | 2 |
| | | 0 | 720 | 1140 | 3 |
| | | | 0 | 1680 | 4 |
| | | | | 0 | 5 |

Now Product of 4 matrices:

$M[1, 4] = M_1 \; M_2 \, M_3 \, M_4$

There are three cases by which we can solve this multiplication:

1. ( $M_1$ x $M_2$ x $M_3$) $M_4$
2. $M_1$ x($M_2$ x $M_3$ x $M_4$)
3. ($M_1$ x$M_2$) x ( $M_3$ x $M_4$)

After solving these cases we choose the case in which minimum output is there

$$M[1, 4] = \min \begin{cases} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = & 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = & 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = & 2120 \end{cases}$$

**M [1, 4] =1080**

As comparing the output of different cases then '**1080**' is minimum output, so we insert 1080 in the table and

($M_1$ x$M_2$) x ($M_3$ x $M_4$) combination is taken out in output making,

M [2, 5] = $M_2$ $M_3$ $M_4$ $M_5$

There are three cases by which we can solve this multiplication:

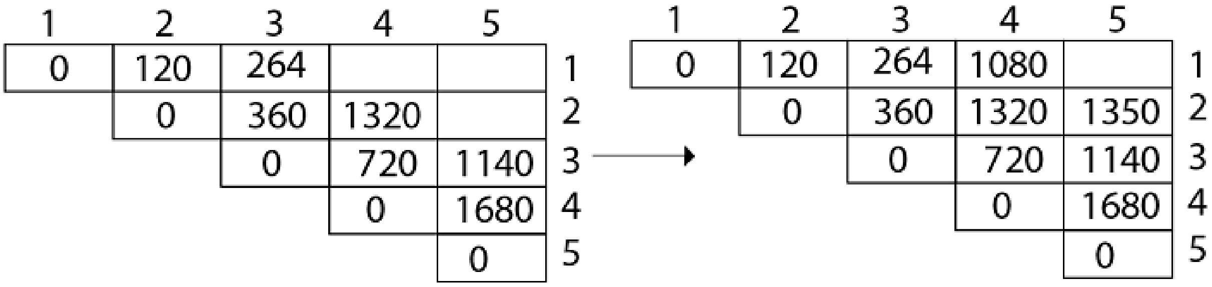
1. ($M_2$ x $M_3$ x $M_4$)x $M_5$
2. $M_2$ x( $M_3$ x $M_4$ x $M_5$)
3. ($M_2$ x $M_3$)x ( $M_4$ x $M_5$)

After solving these cases we choose the case in which minimum output is there

$$M\ [2,\ 5] = \min \begin{Bmatrix} M[2,4] + M[5,5] + p_1p_4p_5 = 1320 + 0 + 10.20.7 = & 2720 \\ M[2,3] + M[4,5] + p_1p_3p_5 = 360 + 1680 + 10.12.7 = & 2880 \\ M[2,2] + M[3,5] + p_1p_2p_5 = 0 + 1140 + 10.3.7 = & 1350 \end{Bmatrix}$$

M [2, 5] = 1350

As comparing the output of different cases then '**1350**' is minimum output, so we insert 1350 in the table and $M_2$ x( $M_3$ x $M_4$ x$M_5$)combination is taken out in output making.



| 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|
| 0 | 120 | 264 | | | 1 |
| | 0 | 360 | 1320 | | 2 |
| | | 0 | 720 | 1140 | 3 |
| | | | 0 | 1680 | 4 |
| | | | | 0 | 5 |

| 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|
| 0 | 120 | 264 | 1080 | | 1 |
| | 0 | 360 | 1320 | 1350 | 2 |
| | | 0 | 720 | 1140 | 3 |
| | | | 0 | 1680 | 4 |
| | | | | 0 | 5 |

**Now Product of 5 matrices:**

M [1, 5] = $M_1$ $M_2$ $M_3$ $M_4$ $M_5$

There are five cases by which we can solve this multiplication:

1. $(M_1 \text{ x } M_2 \text{ x} M_3 \text{ x } M_4 )\text{x } M_5$
2. $M_1 \text{ x}( M_2 \text{ x} M_3 \text{ x } M_4 \text{ x} M_5)$
3. $(M_1 \text{ x } M_2 \text{ x} M_3)\text{x } M_4 \text{ x} M_5$
4. $M_1 \text{ x } M_2\text{x}(M_3 \text{ x } M_4 \text{ x} M_5)$

After solving these cases we choose the case in which minimum output is there

$$M\ [1,\ 5] = \min \begin{cases} M[1,4] + M[5,5] + p_0p_4p_5 = 1080 + 0 + 4.20.7 = \quad 1544 \\ M[1,3] + M[4,5] + p_0p_3p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0p_2p_5 = 120 + 1140 + 4.3.7 = \ 1344 \\ M[1,1] + M[2,5] + \ p_0p_1p_5 = 0 + 1350 + 4.10.7 = \quad 1630 \end{cases}$$

M [1, 5] = 1344

As comparing the output of different cases then '**1344**' is minimum output, so we insert 1344 in the table and $M_1$ x $M_2$ x($M_3$ x $M_4$ x $M_5$)combination is taken out in output making.

**Final Output is:**

| 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|
| 0 | 120 | 264 | 1080 | | 1 |
| | 0 | 360 | 1320 | 1350 | 2 |
| | | 0 | 720 | 1140 | 3 |
| | | | 0 | 1680 | 4 |
| | | | | 0 | 5 |

→

| 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|
| 0 | 120 | 264 | 1080 | 1344 | 1 |
| | 0 | 360 | 1320 | 1350 | 2 |
| | | 0 | 720 | 1140 | 3 |
| | | | 0 | 1680 | 4 |
| | | | | 0 | 5 |

**Step 3: Computing Optimal Costs:** let us assume that matrix $A_i$ has dimension $p_{i-1}$x $p_i$ for i=1, 2, 3....n. The input is a sequence ($p_0$,$p_1$,......$p_n$) where length [p] = n+1. The procedure uses an auxiliary table m [1....n, 1.....n]

for storing m [i, j] costs an auxiliary table s [1.....n, 1.....n] that record which index of k achieved the optimal costs in computing m [i, j].

The algorithm first computes m [i, j] ← 0 for i=1, 2, 3.....n, the minimum costs for the chain of length 1.

## Greedy Algorithm

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

## Elements of Greedy Algorithm

**1. Greedy Choice Property**

If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

**2. Optimal Substructure**

If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

Method of Greedy Algorithm:

i) Huffman Coding
ii) Knapsack problem
iii) Activity Selection Problem (ASP)
iv) Travelling Salesman Problem (TSP)
v) Task Scheduling

Huffman Coding : Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.

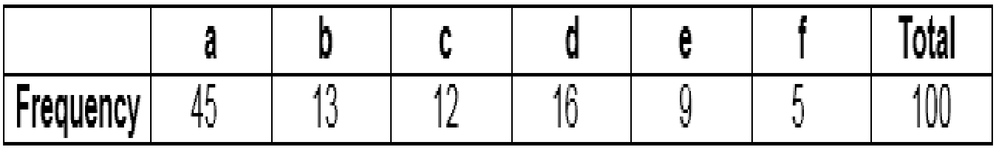
Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

- i) Data can be encoded efficiently using Huffman Codes.

- (ii) It is a widely used and beneficial technique for compressing data.

- (iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.

- Suppose we have $10^5$ characters in a data file. Normal Storage: 8 bits per character (ASCII) - 8 x $10^5$ bits in a file. But we want to compress the file and save it compactly. Suppose only six characters appear in the file:

- 

|  | a | b | c | d | e | f | Total |
|---|---|---|---|---|---|---|---|
| Frequency | 45 | 13 | 12 | 16 | 9 | 5 | 100 |

- How can we represent the data in a Compact way?

**(i) Fixed length Code:** Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character:

**For example:**

| | |
|---|---|
| a | 0 |
| b | 101 |
| c | 100 |

| | |
|---|---|
| d | 111 |
| e | 1101 |
| f | 1100 |

Number of bits = (45 x 1 + 13 x 3 + 12 x 3 + 16 x 3 + 9 x 4 + 5 x 4) x 1000

**= 2.24 x $10^5$bits**

Thus, 224,000 bits to represent the file, a saving of approximately 25%.This is an optimal character code for this file.

## Algorithm of Huffman Code

```
Huffman (C)
n=|C|
Q ← C
for i=1 to n-1
do
    z= allocate-Node ()
    x=  left[z]=Extract-Min(Q)
    y= right[z] =Extract-Min(Q)
    f [z]=f[x]+f[y]
    Insert (Q, z)
return Extract-Min (Q)
```

**Example:** Find an optimal Huffman Code for the following set of frequencies:

1. a: 50 b: 25 c: 15 d: 40 e: 75

**Solution:**

Given that: C = {a, b, c, d, e}

f (C) = {50, 25, 15, 40, 75}

n = 5

Q $\leftarrow$ c

i.e.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| c | 15 | b | 25 | d | 40 | a | 50 | e | 75 |

for i $\leftarrow$ 1 to 4

i = 1 Z $\leftarrow$ Allocate node

x $\leftarrow$ Extract-Min (Q)

y $\leftarrow$ Extract-Min (Q)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| c | 15 | b | 25 | d | 40 | a | 50 | e | 75 |

Left [z] $\leftarrow$ x

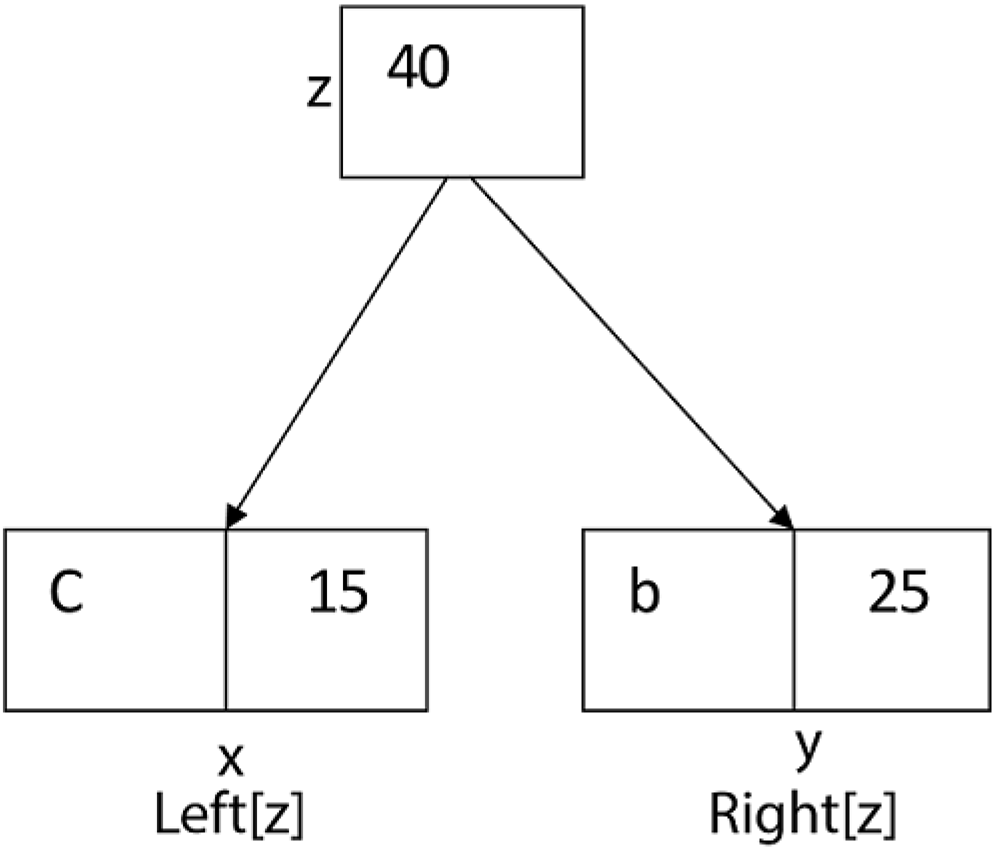
Right [z] $\leftarrow$ y

f (z) $\leftarrow$ f (x) + f (y) = 15 + 25

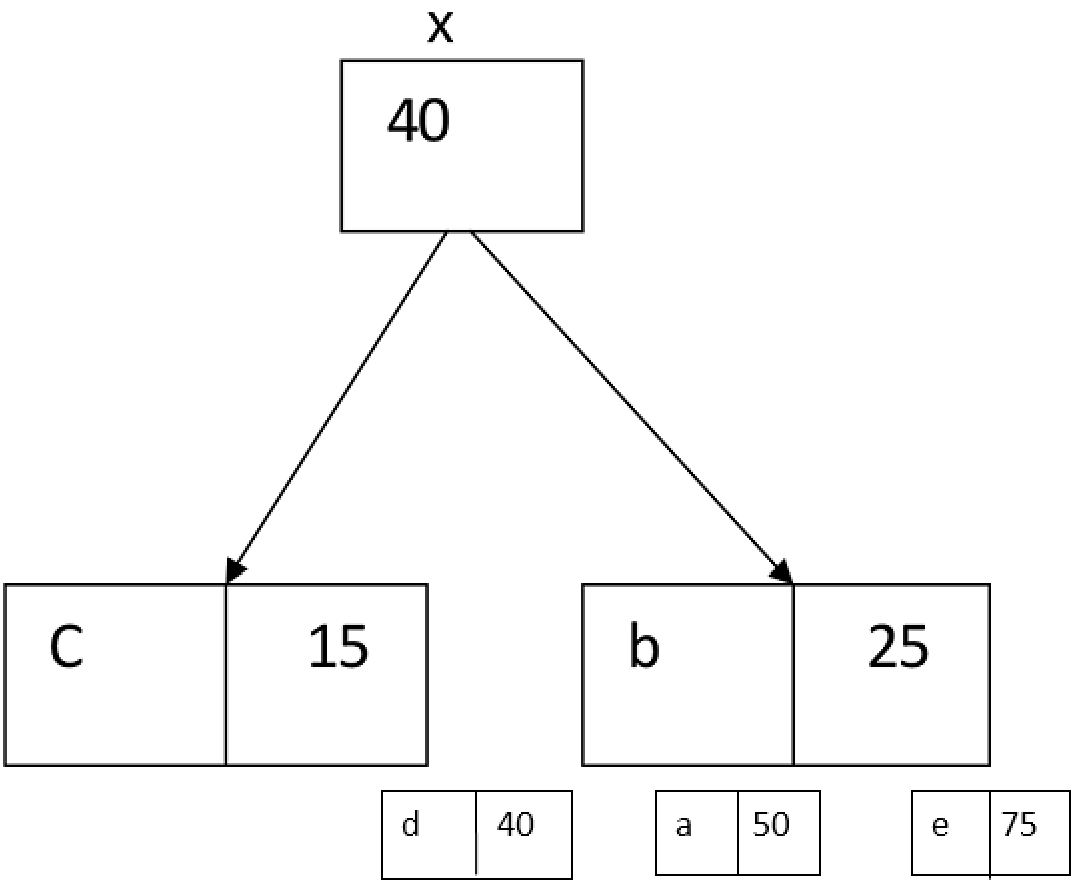
f (z) = 40

| | | | | | |
|---|---|---|---|---|---|
| d | 40 | a | 50 | e | 75 |

i.e.



Again for i=2



z ← Allocate node

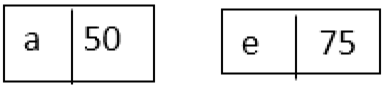
x ← 40

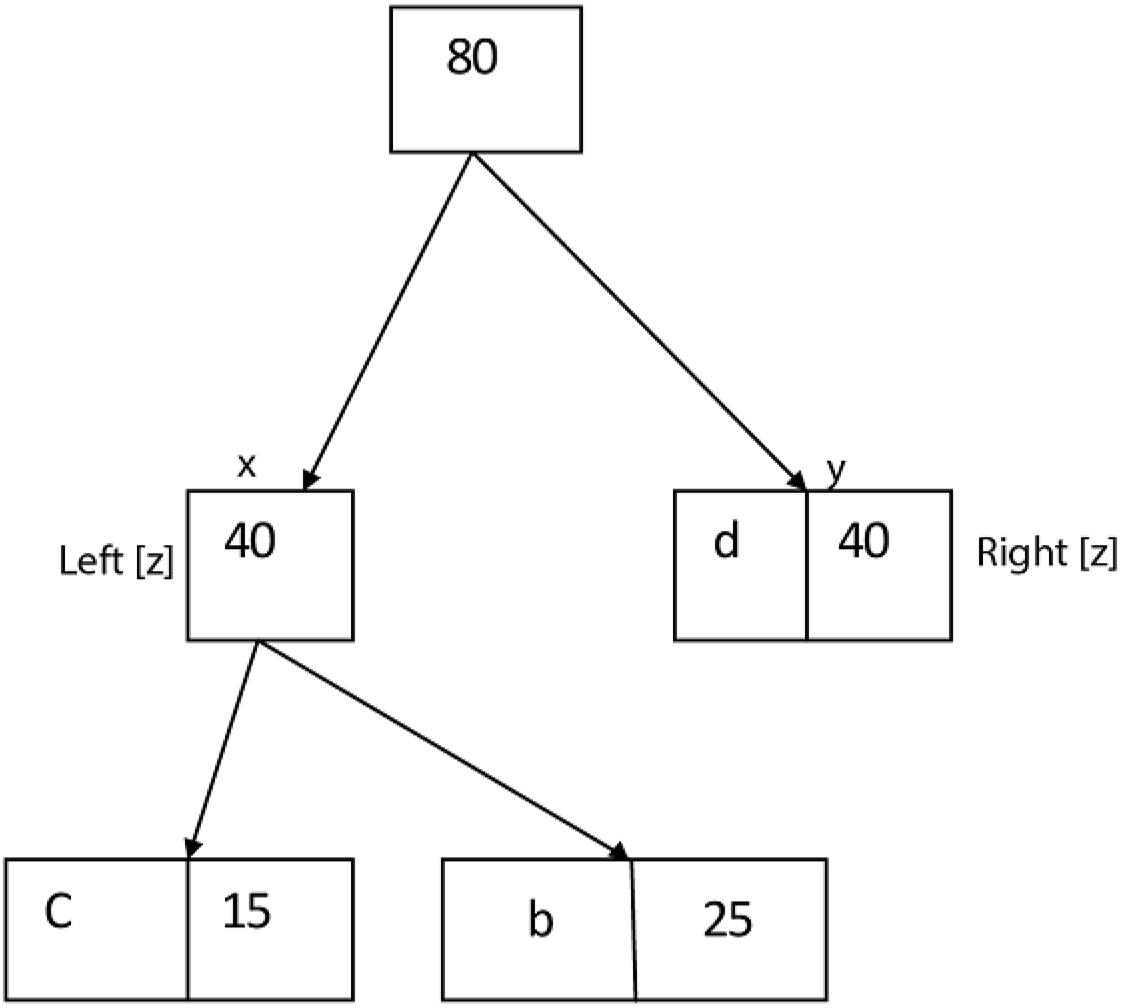
y ← 40

left [z] ← x

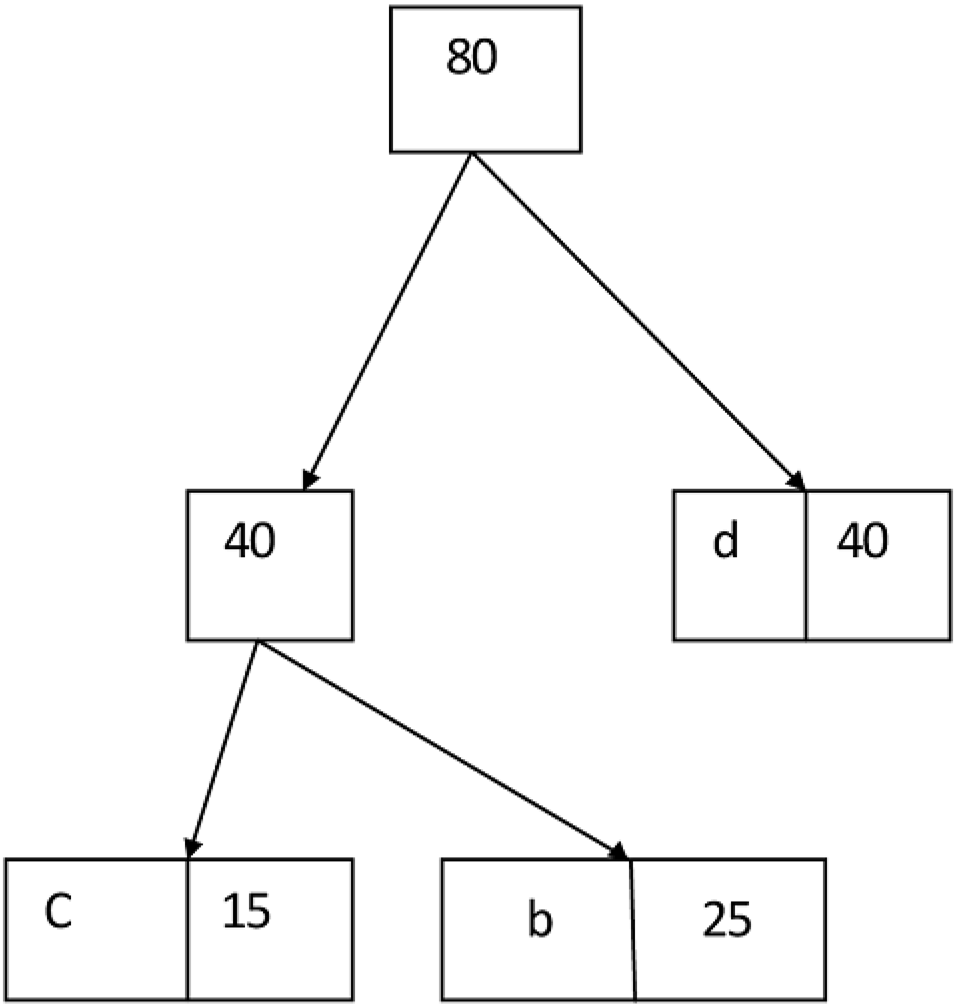
right [z] ← y

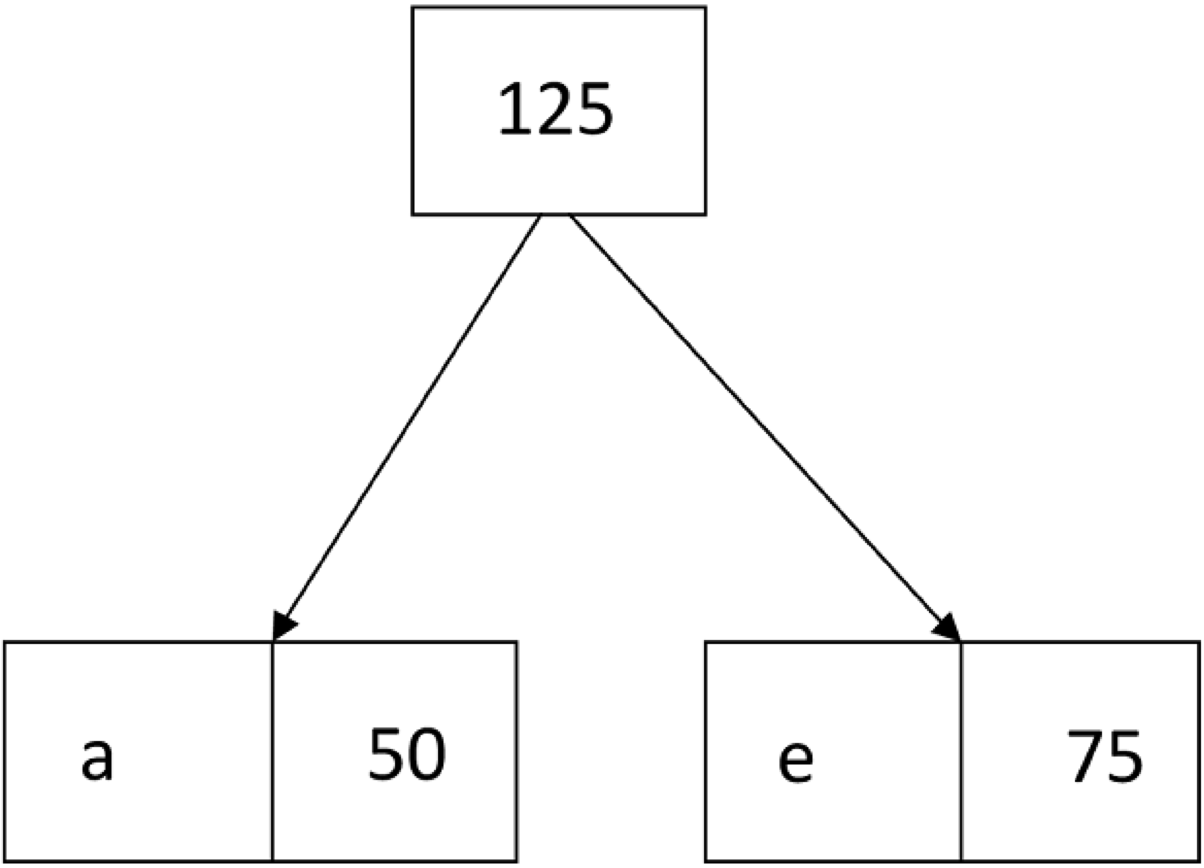
f (z) = 40 + 40 = 80



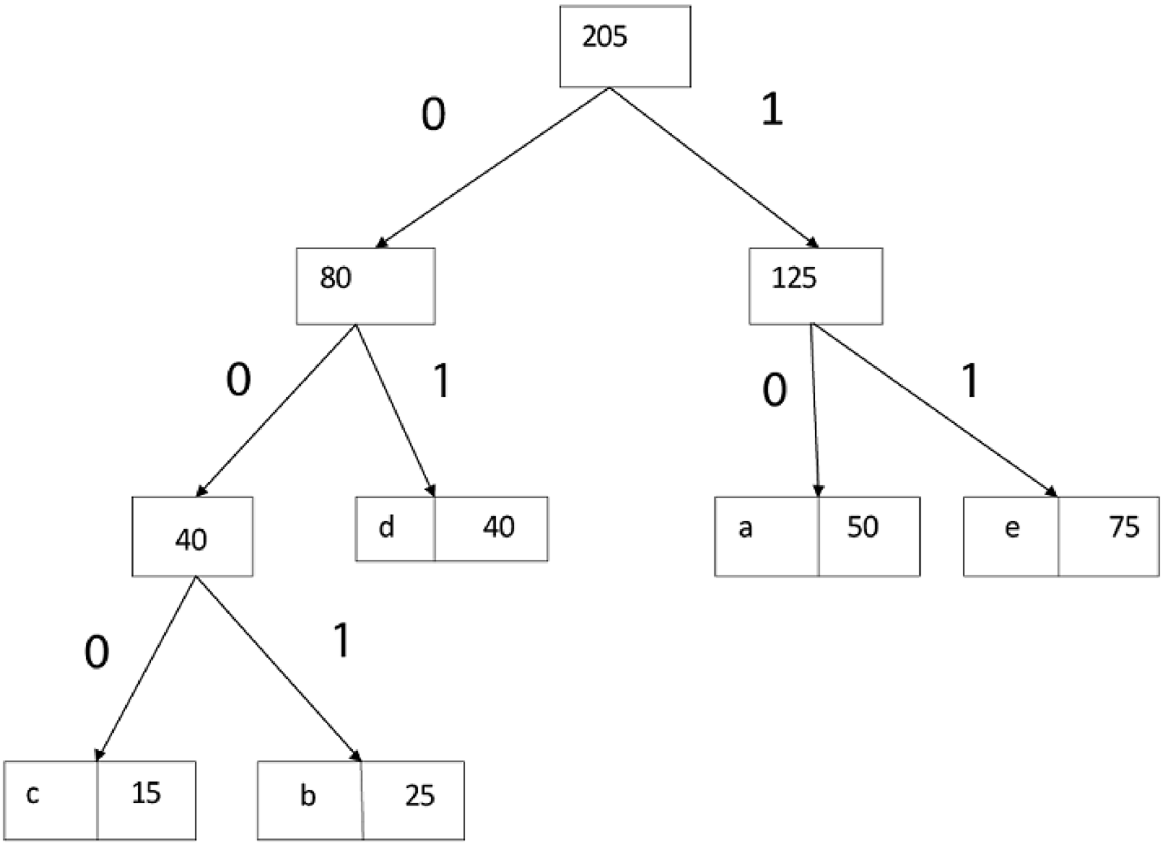


Similarly, we apply the same process we get


80
40
d
40
C
15
b
25



Thus, the final output is:



## 2) Knapsack Problem:

The **knapsack problem** is a problem in combinational optimization : Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

For example, the weight of the container is 20 kg. We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

$$\text{Maximize } \sum n=1n\ (\ xi\ .\ pi)$$

subject to constraint,

$$\sum n=1n\ (\ xi\ .\ wi\ ) \leqslant W$$

There are two types of knapsack problems:

- 0/1 knapsack problem
- Fractional knapsack problem

### 1)Fractional Knapsack Problem

**Algorithm: Greedy-Fractional-Knapsack (w [ 1..n], p[1..n], W)**

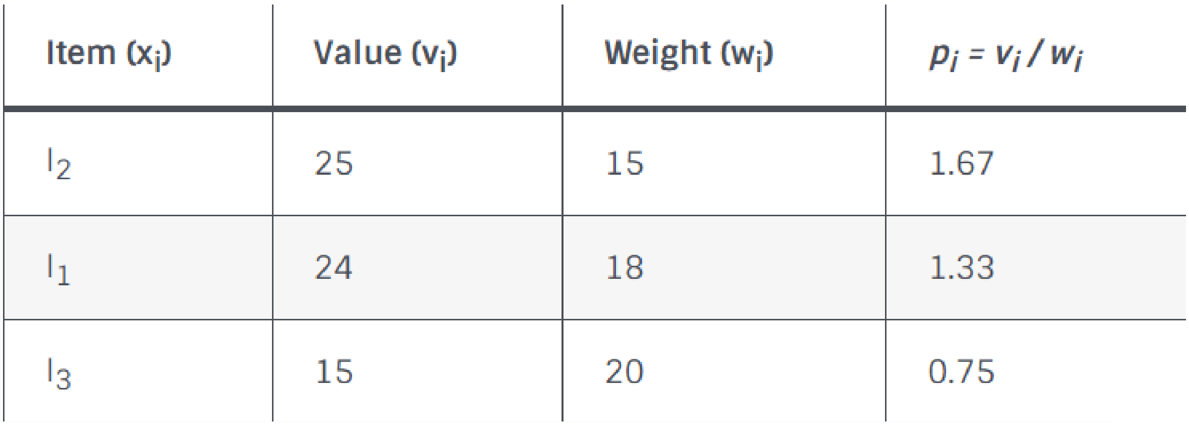
```
for i = 1 to n
   do x[i] = 0
    weight = 0
    for i = 1 to n
       if weight + w[i] ≤ W then
          x[i] = 1
          weight = weight + w[i]
       else
          x[i] = (W - weight) / w[i]
          weight = W
          break
    return x
```

## Examples of Fractional Knapsack

**Problem: Consider the following instances of the fractional knapsack problem: n = 3, M = 20, V = (24, 25, 15) and W = (18, 15, 20) find the feasible solutions.**

**Solution:**

Let us arrange items by decreasing order of profit density. Assume that items are labeled as X = ($I_1$, $I_2$, $I_3$), have profit V = {24, 25, 15} and weight W = {18, 15, 20}.



| Item ($x_i$) | Value ($v_i$) | Weight ($w_i$) | $p_i = v_i / w_i$ |
|---|---|---|---|
| $I_2$ | 25 | 15 | 1.67 |
| $I_1$ | 24 | 18 | 1.33 |
| $I_3$ | 15 | 20 | 0.75 |

We shall select one by one item from Table. If the inclusion of an item does not cross the knapsack capacity, then add it. Otherwise, break the current item and select only the portion of item equivalent to remaining knapsack capacity. Select the profit accordingly. We should stop when knapsack is full or all items are scanned.

Initialize, Weight of selected items, SW = 0,

Profit of selected items, SP = 0,

Set of selected items, S = { },

Here, Knapsack capacity M = 20.

**Iteration 1 :** SW= (SW + $w_2$) = 0 + 15 = 15

SW $\leq$ M, so select $I_2$

S = { $I_2$ }, SW = 15, SP = 0 + 25 = 25

**Iteration 2 :** SW + $w_1$ > M, so break down item $I_{1.}$

The remaining capacity of the knapsack is 5 unit, so select only 5 units of item $I_1$.

*frac* = (M – SW) / W[i] = (20 – 15) / 18 = 5 / 18

S = { $I_{2,}$ $I_1$ * 5/18 }

SP = SP + $v_1$ * *frac* = 25 + (24 * (5/18)) = 25 + 6.67 = 31.67

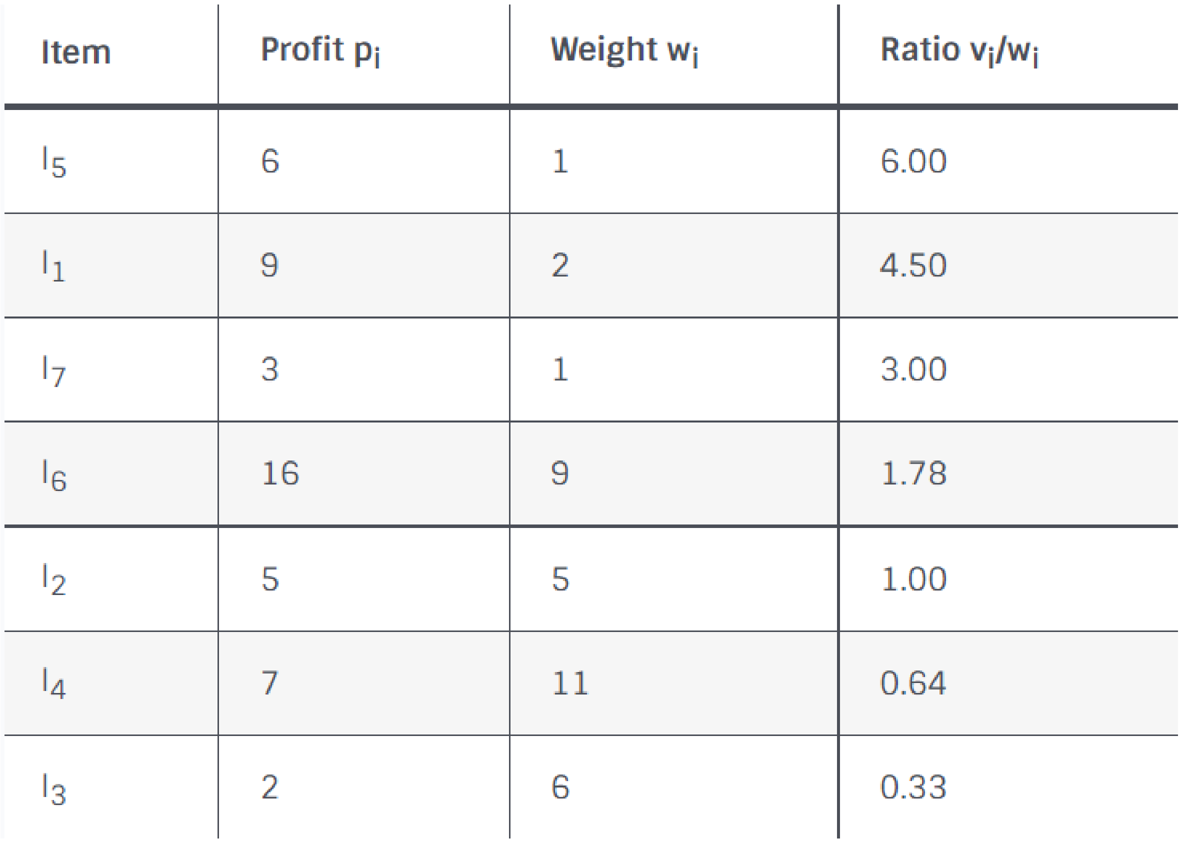
SW = SW + $w_1$ * *frac* = 15 + (18 * (5/18)) = 15 + 5 = 20

The knapsack is full. Fractional Greedy algorithm selects items { $I_{2,}$ $I_1$ * 5/18 }, and it gives a profit of 31.67 units.

**Problem: Find the optimal solution for knapsack problem (fraction) where knapsack capacity = 28, P = {9, 5, 2, 7, 6, 16, 3} and w = {2, 5, 6, 11, 1, 9, 1}.**

**Solution:**

Arrange items in decreasing order of profit to weight ratio



| Item | Profit $p_i$ | Weight $w_i$ | Ratio $v_i/w_i$ |
|---|---|---|---|
| $I_5$ | 6 | 1 | 6.00 |
| $I_1$ | 9 | 2 | 4.50 |
| $I_7$ | 3 | 1 | 3.00 |
| $I_6$ | 16 | 9 | 1.78 |
| $I_2$ | 5 | 5 | 1.00 |
| $I_4$ | 7 | 11 | 0.64 |
| $I_3$ | 2 | 6 | 0.33 |

Initialize, Weight = 0, P = 0, M = 28, S = { }

Where S is the solution set, P and W is profit and weight of included items, respectively. M is the capacity of the knapsack.

**Iteration 1**

(Weight + $w_5$) ≤ M, so select $I_5$

So, S = { $I_5$ }, Weight = 0 + 1 = 1, P = 0 + 6= 6

**Iteration 2**

(Weight + $w_1$) ≤ M, so select $I_1$

So, S = {$I_5$ ,$I_1$ }, Weight = 1 + 2 = 3, P = 6 + 9= 15

**Iteration 3**

(Weight + $w_7$) ≤ M, so select $I_7$

o, S = {$I_5$, $I_1$, $I_7$ }, Weight = 3 + 1 = 4, P = 15 + 3= 18

**Iteration 4**

(Weight + $w_6$) ≤ M, so select $I_6$

So, S = {$I_5$, $I_1$, $I_7$, $I_6$ }, Weight = 4 + 9 = 13, P = 18 + 16= 34

**Iteration 5**

(Weight + $w_2$) ≤ M, so select $I_2$

So, S = {$I_5$, $I_1$, $I_7$, $I_6$, $I_2$ }, Weight = 13 + 5 = 18, P = 34 + 5= 39

**Iteration 6**

(Weight + $w_4$) > M, So $I_4$ must be broken down into two parts x and y such that x = capacity left in knapsack and y = $I_4$ – x.

Available knapsack capacity is 10 units. So we can select only (28 – 18) / 11 = 0.91 unit of $I_4$

So S = {$I_5$, $I_1$, $I_7$, $I_6$, $I_2$, 0.91 * $I_4$ }, Weight = 18 + 0.91*11 = 28, P = 39 + 0.91 * 7= 45.37

## Activity Selection Problem

The activity selection problem is a mathematical optimization problem. Our first illustration is the problem of scheduling a resource among several challenge activities. We find a greedy algorithm provides a well designed and simple method for selecting a maximum- size set of manually compatible activities.

- Span of activity is defined by its start time and finishing time. Suppose we have such n activities.
- Aim of algorithm is to find optimal schedule with maximum number of activities to be carried out with limited resources. Suppose S = $\{a_1, a_2, a_3, ..\ a_n\}$ is the set of activities that we want to schedule.
- Scheduled activities must be compatible with each other. Start time of activities is let's say $s_i$ and finishing time is $f_i$, then activities i and j are called compatible if and only if $f_i < s_j$ or $f_j < s_i$. In other words, two activities are compatible if their time durations do not overlap.
- Consider the below time line. Activities $\{A_1, A_3\}$ and $\{A_2, A_3\}$ are compatible set of activities.
- For given n activities, there may exist multiple such schedules. Aim of activity selection algorithm is to find out the longest schedule without overlap.

Greedy Approach sort activities by their finishing time in increasing order, so that $f_1 \leq f_2 \leq f_3 \leq . . . \leq f_n$. By default it schedules the first activity in sorted list. Subsequent next activities are scheduled whose start time is larger than finish time of previous activity. Run through all possible activities and do the same.

## Algorithm for Activity Selection Problem

### GREEDY- ACTIVITY SELECTOR (s, f)

```
// A is Set of n activities sorted by finishing time.
// S = { A[1] }, solution set, initially which contains first activity.

1. n ← length [s]
2. A ← {1}
3. j ← 1.
4. for i ← 2 to n
5. do if $s_i \geq f_i$
6. then A ← A ∪ {i}
7. j ← i
8. return A
```

**Example:** Given 10 activities along with their start and end time as

S = ($A_1$ $A_2$ $A_3$ $A_4$ $A_5$ $A_6$ $A_7$ $A_8$ A9 A10)

Si = (1,2,3,4,7,8,9,9,11,12)

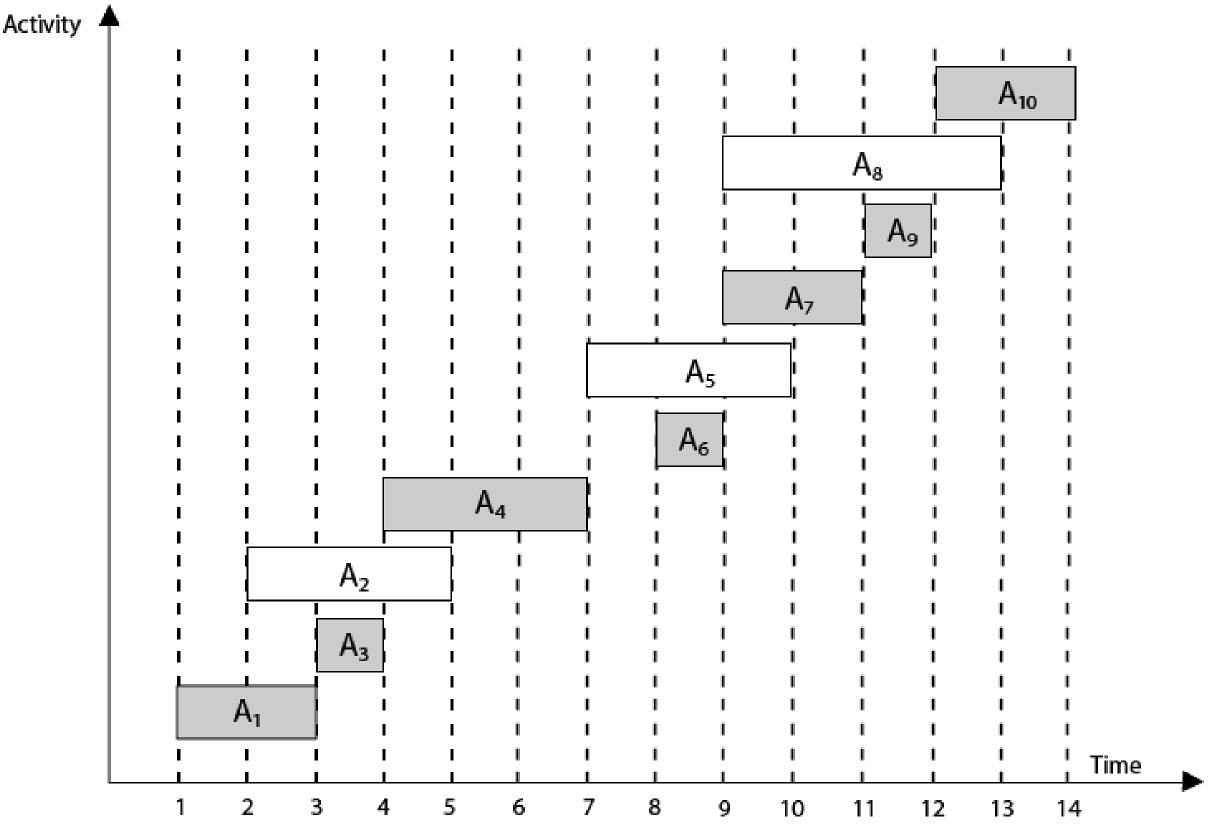
fi = (3,5,4,7,10,9,11,13,12,14)

Compute a schedule where the greatest number of activities takes place.

**Solution:** The solution to the above Activity scheduling problem using a greedy strategy is illustrated below:

Arranging the activities in increasing order of end time

| Activity | $A_1$ | $A_3$ | $A_2$ | $A_4$ | $A_6$ | $A_5$ | $A_7$ | $A_9$ | $A_8$ | $A_{10}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | 1 | 3 | 2 | 4 | 8 | 7 | 9 | 11 | 9 | 12 |
| Finish | 3 | 4 | 5 | 7 | 9 | 10 | 11 | 12 | 13 | 14 |



Now, schedule $A_1$

Next schedule $A_3$ as $A_1$ and $A_3$ are non-interfering.

Next **skip** $A_2$ as it is interfering.

Next, schedule $A_4$ as $A_1$ $A_3$ and $A_4$ are non-interfering, then next, schedule $A_6$ as $A_1$ $A_3$ $A_4$ and $A_6$ are non-interfering.

Skip $A_5$ as it is interfering.

Next, schedule $A_7$ as $A_1$ $A_3$ $A_4$ $A_6$ and $A_7$ are non-interfering.

Next, schedule $A_9$ as $A_1$ $A_3$ $A_4$ $A_6$ $A_7$ and $A_9$ are non-interfering.

Skip $A_8$ as it is interfering.

Next, schedule $A_{10}$ as $A_1$ $A_3$ $A_4$ $A_6$ $A_7$ $A_9$ and $A_{10}$ are non-interfering.

Thus the final Activity schedule is:

$(A_1\ A_3\ \ A_4\ A_6\ \ A_7\ \ A_9\ A_{10})$

Now we can understand another example :

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| START | 1 | 3 | 2 | 0 | 5 | 8 | 11 |
| END | 3 | 4 | 5 | 7 | 9 | 10 | 12 |

SELECTED

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| START | 1 | 3 | 2 | 0 | 5 | 8 | 11 |
| END | 3 | 4 | 5 | 7 | 9 | 10 | 12 |

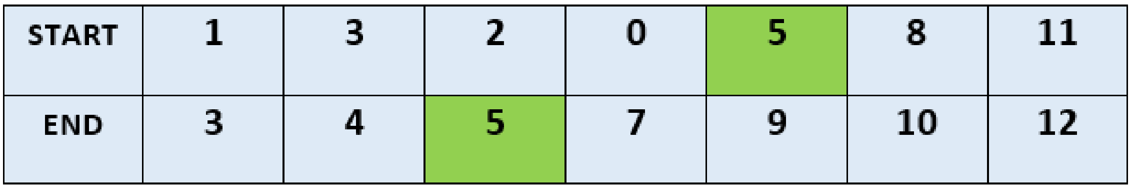
START[1]>=END[0], SELECTED

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| START | 1 | 3 | 2 | 0 | 5 | 8 | 11 |
| END | 3 | 4 | 5 | 7 | 9 | 10 | 12 |

START[2]<END[1], REJECTED

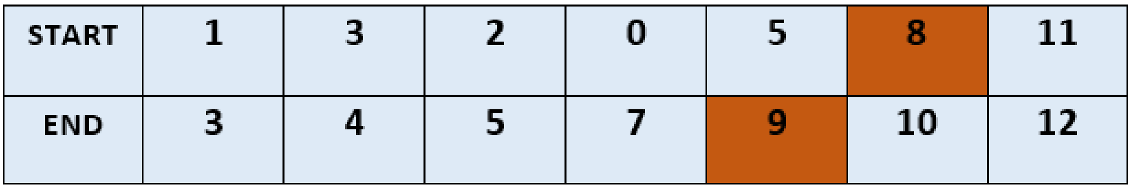
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| START | 1 | 3 | 2 | 0 | 5 | 8 | 11 |
| END | 3 | 4 | 5 | 7 | 9 | 10 | 12 |

START[3]<END[1], REJECTED



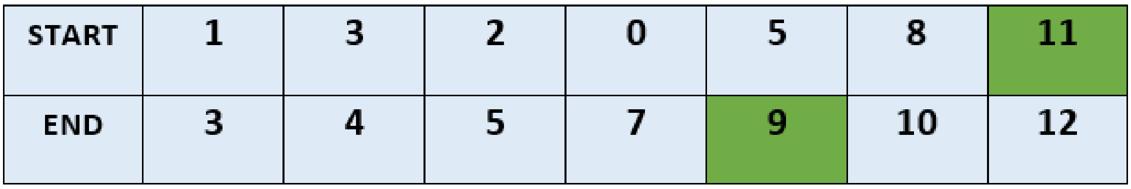
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| START | 1 | 3 | 2 | 0 | 5 | 8 | 11 |
| END | 3 | 4 | 5 | 7 | 9 | 10 | 12 |

START[4] >=END[2], SELECTED



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| START | 1 | 3 | 2 | 0 | 5 | 8 | 11 |
| END | 3 | 4 | 5 | 7 | 9 | 10 | 12 |

START[5]<END[4], REJECTED



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| START | 1 | 3 | 2 | 0 | 5 | 8 | 11 |
| END | 3 | 4 | 5 | 7 | 9 | 10 | 12 |

START[6]>=END[4], SELECTED

In this example, we take the start and finish time of activities as follows:

start = [1, 3, 2, 0, 5, 8, 11]
finish = [3, 4, 5, 7, 9, 10, 12]

Sorted by their finish time, the activity 0 gets selected. As the activity 1 has starting time which is equal to the finish time of activity 0, it gets selected.

Activities 2 and 3 have smaller starting time than finish time of activity 1, so they get rejected. Based on similar comparisons, activities 4 and 6 also get selected, whereas activity 5 gets rejected.

In this example, in all the activities 0, 1, 4 and 6 get selected, while others get rejected.

## Task Scheduling Algorithm

This is the dispute of optimally scheduling unit-time tasks on a single processor, where each job has a deadline and a penalty that necessary be paid if the deadline is missed.

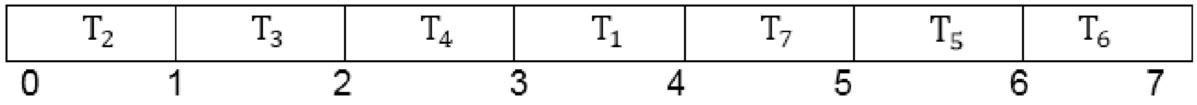
**Example:** Find the optimal schedule for the following task with given weight (penalties) and deadlines.

| | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| $d_i$ | 4 | 2 | 4 | 3 | 1 | 4 |
| $w_i$ | 70 | 60 | 50 | 40 | 30 | 20 |

**Solution:** According to the Greedy algorithm we sort the jobs in decreasing order of their penalties so that minimum of penalties will be charged.

In this problem, we can see that the maximum time for which uniprocessor machine will run in 6 units because it is the maximum deadline.

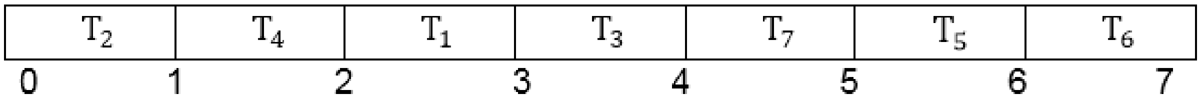
Let $T_i$ represents the tasks where i = 1 to 7



$T_5$ and $T_6$ cannot be accepted after $T_7$ so penalty is

$w_5 + w_6 = 30 + 20 = 50$ (2 3 4 1 7 5 6)

Other schedule is

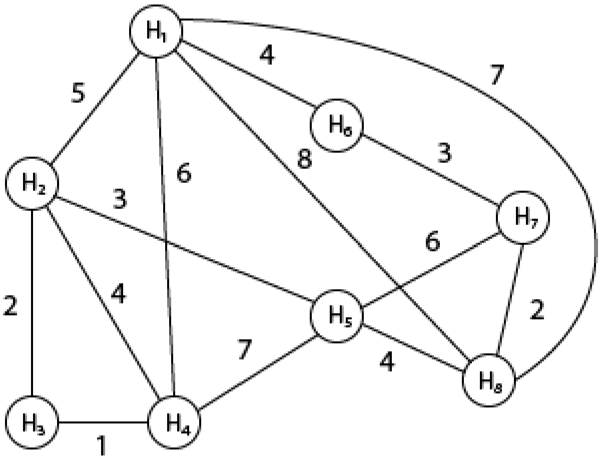


(2 4 1 3 7 5 6)

There can be many other schedules but (2 4 1 3 7 5 6) is optimal.

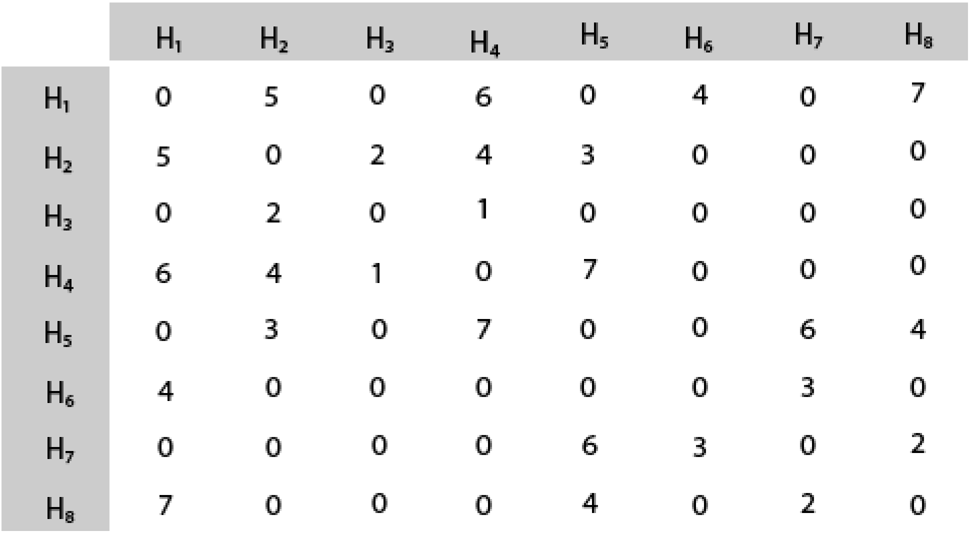
## Travelling Salesman Problem ( TSP)

The traveling salesman problem (TSP) is an algorithmic problem tasked with finding the shortest route between a set of points and locations that must be visited. In the problem statement, the points are the cities a salesperson might visit. The salesman‘s goal is to keep both the travel costs and the distance traveled as low as possible.



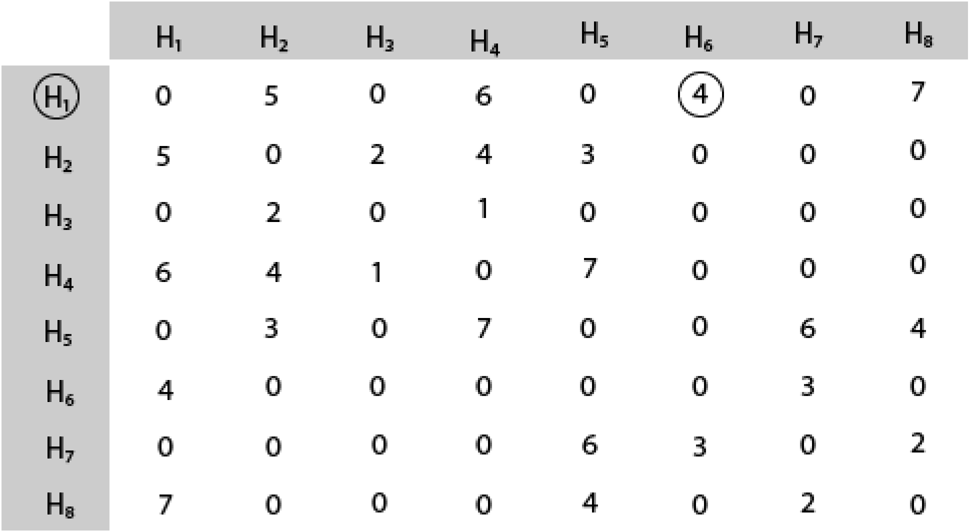
Solution: The cost- adjacency matrix of graph G is as follows:

$\text{cost}_{ij}$ =



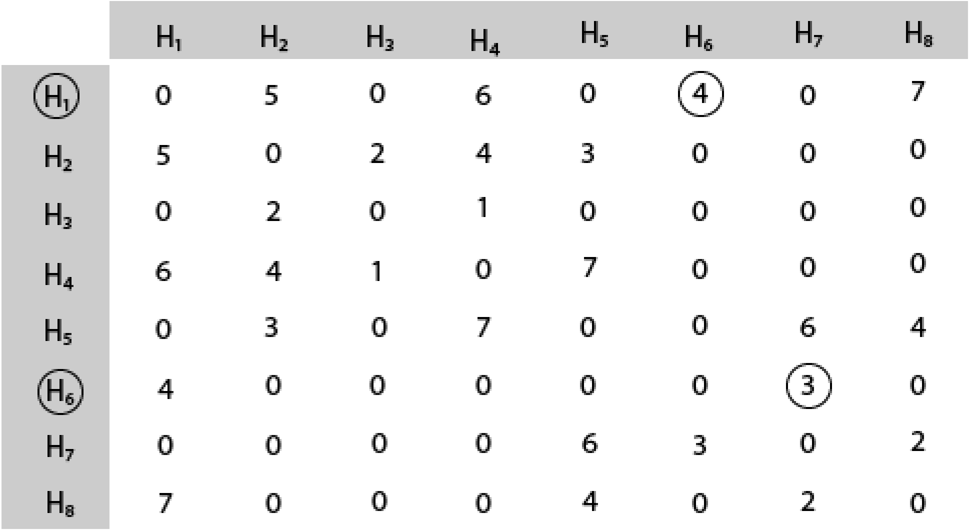
| | $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ | $H_6$ | $H_7$ | $H_8$ |
|---|---|---|---|---|---|---|---|---|
| $H_1$ | 0 | 5 | 0 | 6 | 0 | 4 | 0 | 7 |
| $H_2$ | 5 | 0 | 2 | 4 | 3 | 0 | 0 | 0 |
| $H_3$ | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 |
| $H_4$ | 6 | 4 | 1 | 0 | 7 | 0 | 0 | 0 |
| $H_5$ | 0 | 3 | 0 | 7 | 0 | 0 | 6 | 4 |
| $H_6$ | 4 | 0 | 0 | 0 | 0 | 0 | 3 | 0 |
| $H_7$ | 0 | 0 | 0 | 0 | 6 | 3 | 0 | 2 |
| $H_8$ | 7 | 0 | 0 | 0 | 4 | 0 | 2 | 0 |

The tour starts from area $H_1$ and then select the minimum cost area reachable from $H_1$.



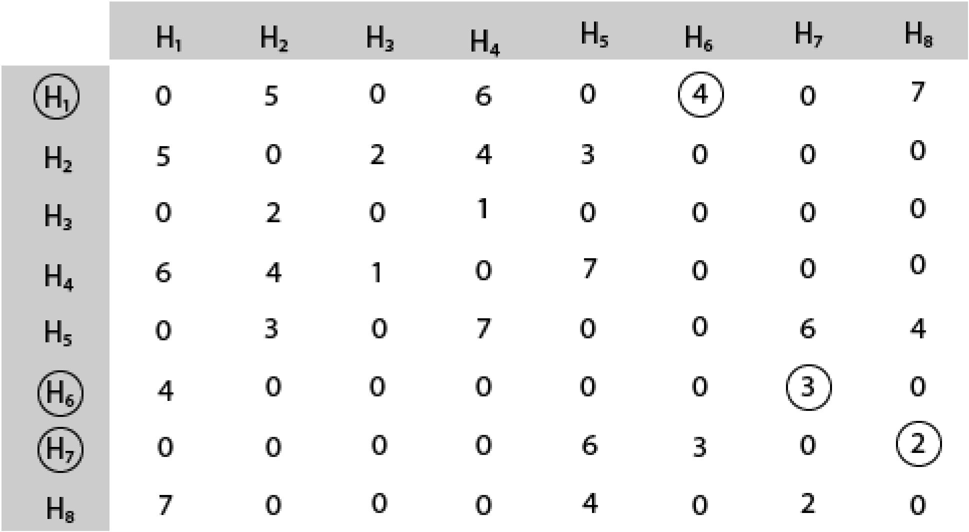
| | $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ | $H_6$ | $H_7$ | $H_8$ |
|---|---|---|---|---|---|---|---|---|
| $H_1$ | 0 | 5 | 0 | 6 | 0 | 4 | 0 | 7 |
| $H_2$ | 5 | 0 | 2 | 4 | 3 | 0 | 0 | 0 |
| $H_3$ | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 |
| $H_4$ | 6 | 4 | 1 | 0 | 7 | 0 | 0 | 0 |
| $H_5$ | 0 | 3 | 0 | 7 | 0 | 0 | 6 | 4 |
| $H_6$ | 4 | 0 | 0 | 0 | 0 | 0 | 3 | 0 |
| $H_7$ | 0 | 0 | 0 | 0 | 6 | 3 | 0 | 2 |
| $H_8$ | 7 | 0 | 0 | 0 | 4 | 0 | 2 | 0 |

Mark area $H_6$ because it is the minimum cost area reachable from $H_1$ and then select minimum cost area reachable from $H_6$.



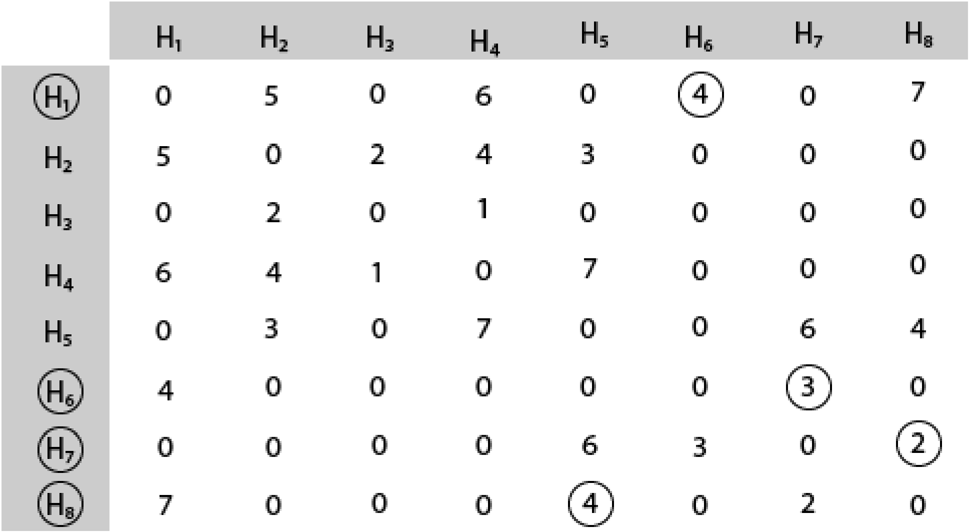
|  | $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ | $H_6$ | $H_7$ | $H_8$ |
|---|---|---|---|---|---|---|---|---|
| $\textcircled{H_1}$ | 0 | 5 | 0 | 6 | 0 | ④ | 0 | 7 |
| $H_2$ | 5 | 0 | 2 | 4 | 3 | 0 | 0 | 0 |
| $H_3$ | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 |
| $H_4$ | 6 | 4 | 1 | 0 | 7 | 0 | 0 | 0 |
| $H_5$ | 0 | 3 | 0 | 7 | 0 | 0 | 6 | 4 |
| $\textcircled{H_6}$ | 4 | 0 | 0 | 0 | 0 | 0 | ③ | 0 |
| $H_7$ | 0 | 0 | 0 | 0 | 6 | 3 | 0 | 2 |
| $H_8$ | 7 | 0 | 0 | 0 | 4 | 0 | 2 | 0 |

Mark area $H_7$ because it is the minimum cost area reachable from $H_6$ and then select minimum cost area reachable from $H_7$.



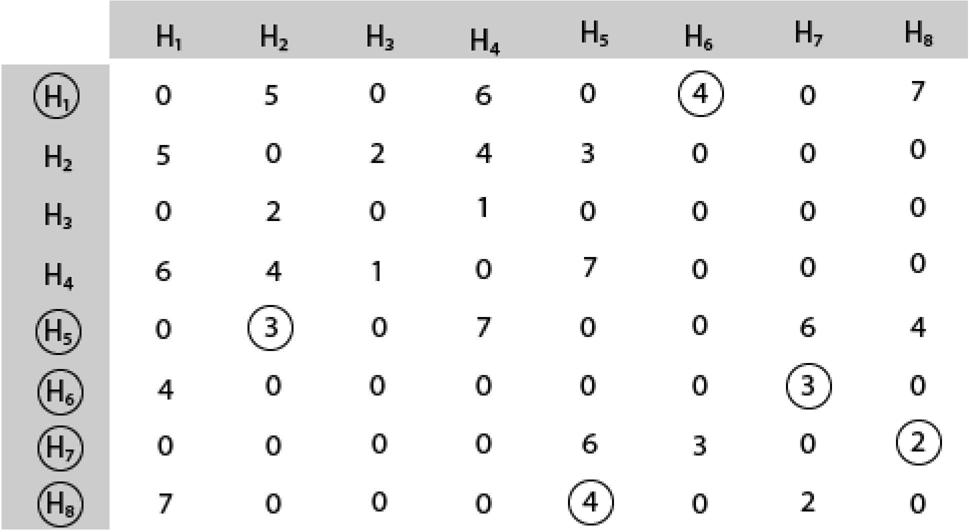
|  | $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ | $H_6$ | $H_7$ | $H_8$ |
|---|---|---|---|---|---|---|---|---|
| $\textcircled{H_1}$ | 0 | 5 | 0 | 6 | 0 | ④ | 0 | 7 |
| $H_2$ | 5 | 0 | 2 | 4 | 3 | 0 | 0 | 0 |
| $H_3$ | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 |
| $H_4$ | 6 | 4 | 1 | 0 | 7 | 0 | 0 | 0 |
| $H_5$ | 0 | 3 | 0 | 7 | 0 | 0 | 6 | 4 |
| $\textcircled{H_6}$ | 4 | 0 | 0 | 0 | 0 | 0 | ③ | 0 |
| $\textcircled{H_7}$ | 0 | 0 | 0 | 0 | 6 | 3 | 0 | ② |
| $H_8$ | 7 | 0 | 0 | 0 | 4 | 0 | 2 | 0 |

Mark area $H_8$ because it is the minimum cost area reachable from $H_8$.



| | $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ | $H_6$ | $H_7$ | $H_8$ |
|---|---|---|---|---|---|---|---|---|
| $\textcircled{H_1}$ | 0 | 5 | 0 | 6 | 0 | ④ | 0 | 7 |
| $H_2$ | 5 | 0 | 2 | 4 | 3 | 0 | 0 | 0 |
| $H_3$ | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 |
| $H_4$ | 6 | 4 | 1 | 0 | 7 | 0 | 0 | 0 |
| $H_5$ | 0 | 3 | 0 | 7 | 0 | 0 | 6 | 4 |
| $\textcircled{H_6}$ | 4 | 0 | 0 | 0 | 0 | 0 | ③ | 0 |
| $\textcircled{H_7}$ | 0 | 0 | 0 | 0 | 6 | 3 | 0 | ② |
| $\textcircled{H_8}$ | 7 | 0 | 0 | 0 | ④ | 0 | 2 | 0 |

Mark area $H_5$ because it is the minimum cost area reachable from $H_5$.



| | $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ | $H_6$ | $H_7$ | $H_8$ |
|---|---|---|---|---|---|---|---|---|
| $\textcircled{H_1}$ | 0 | 5 | 0 | 6 | 0 | ④ | 0 | 7 |
| $H_2$ | 5 | 0 | 2 | 4 | 3 | 0 | 0 | 0 |
| $H_3$ | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 |
| $H_4$ | 6 | 4 | 1 | 0 | 7 | 0 | 0 | 0 |
| $\textcircled{H_5}$ | 0 | ③ | 0 | 7 | 0 | 0 | 6 | 4 |
| $\textcircled{H_6}$ | 4 | 0 | 0 | 0 | 0 | 0 | ③ | 0 |
| $\textcircled{H_7}$ | 0 | 0 | 0 | 0 | 6 | 3 | 0 | ② |
| $\textcircled{H_8}$ | 7 | 0 | 0 | 0 | ④ | 0 | 2 | 0 |

Mark area $H_2$ because it is the minimum cost area reachable from $H_2$.

| | $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ | $H_6$ | $H_7$ | $H_8$ |
|---|---|---|---|---|---|---|---|---|
| $H_1$ | 0 | 5 | 0 | 6 | 0 | ④ | 0 | 7 |
| $H_2$ | 5 | 0 | ② | 4 | 3 | 0 | 0 | 0 |
| $H_3$ | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 |
| $H_4$ | 6 | 4 | 1 | 0 | 7 | 0 | 0 | 0 |
| $H_5$ | 0 | ③ | 0 | 7 | 0 | 0 | 6 | 4 |
| $H_6$ | 4 | 0 | 0 | 0 | 0 | 0 | ③ | 0 |
| $H_7$ | 0 | 0 | 0 | 0 | 6 | 3 | 0 | ② |
| $H_8$ | 7 | 0 | 0 | 0 | ④ | 0 | 2 | 0 |

Mark area $H_3$ because it is the minimum cost area reachable from $H_3$.

| | $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ | $H_6$ | $H_7$ | $H_8$ |
|---|---|---|---|---|---|---|---|---|
| $H_1$ | 0 | 5 | 0 | 6 | 0 | ④ | 0 | 7 |
| $H_2$ | 5 | 0 | ② | 4 | 3 | 0 | 0 | 0 |
| $H_3$ | 0 | 2 | 0 | ① | 0 | 0 | 0 | 0 |
| $H_4$ | 6 | 4 | 1 | 0 | 7 | 0 | 0 | 0 |
| $H_5$ | 0 | ③ | 0 | 7 | 0 | 0 | 6 | 4 |
| $H_6$ | 4 | 0 | 0 | 0 | 0 | 0 | ③ | 0 |
| $H_7$ | 0 | 0 | 0 | 0 | 6 | 3 | 0 | ② |
| $H_8$ | 7 | 0 | 0 | 0 | ④ | 0 | 2 | 0 |

Mark area $H_4$ and then select the minimum cost area reachable from $H_4$ it is $H_1$.So, using the greedy strategy, we get the following.

$$H_1 \xrightarrow{4} H_6 \xrightarrow{3} H_7 \xrightarrow{2} H_8 \xrightarrow{4} H_5 \xrightarrow{3} H_2 \xrightarrow{2} H_3 \xrightarrow{1} H_4 \xrightarrow{6} H_1.$$

Thus, the minimum travel cost = 4 + 3 + 2 + 4 + 3 + 2 + 1 + 6 = 25

.

## Binomial heaps

**A binomial heap** is a heap similar to a binary heap but also supports quickly merging two heaps. This is achieved by using a special tree structure. It is important as an implementation of the **mergeable heap** abstract data type (also called meldable heap), which is a priority queue supporting merge operation.

## Binomial tree

A binomial heap is implemented as a collection of binomial trees (compare with a binary heap, which has a shape of a single binary tree). A **binomial tree** is defined recursively:

- A binomial tree of order 0 is a single node
- A binomial tree of order *k* has a root node whose children are roots of binomial trees of orders *k*−1, *k*−2, ..., 2, 1, 0 (in this order).

### Types of Binary Heap

A binary heap can be classified further as either a ***max-heap*** or a ***min-heap*** based on the ordering property.

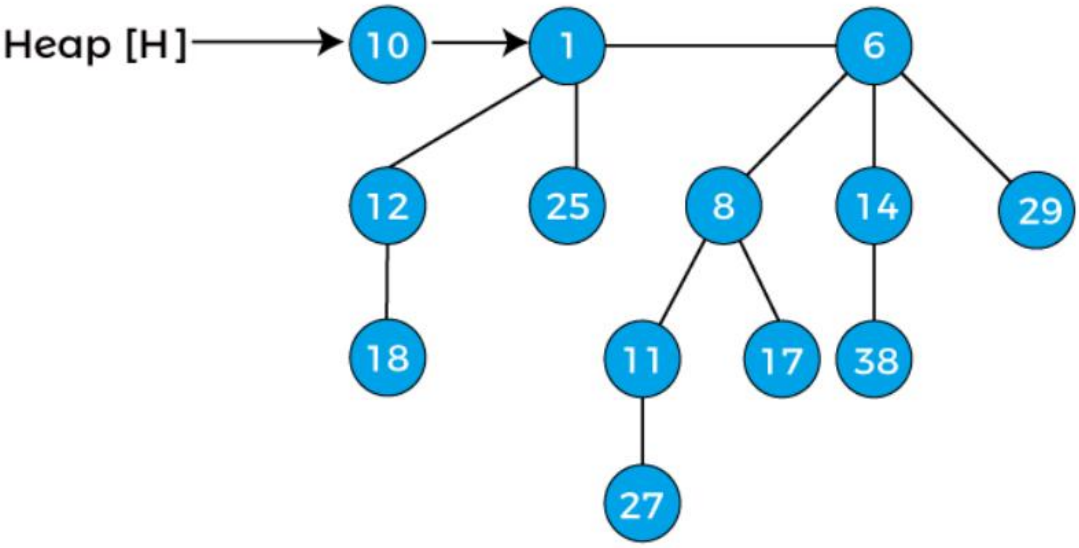
➢ Max-Heap :

In this heap, the key value of a node is greater than or equal to the key value of the highest child.

Hence, ***H[Parent(i)] ≥ H[i]***

- Max Heap conforms to the above properties of heap.

- In max heap, every node contains greater or equal value element than its child nodes.
- Thus, root node contains the largest value element.



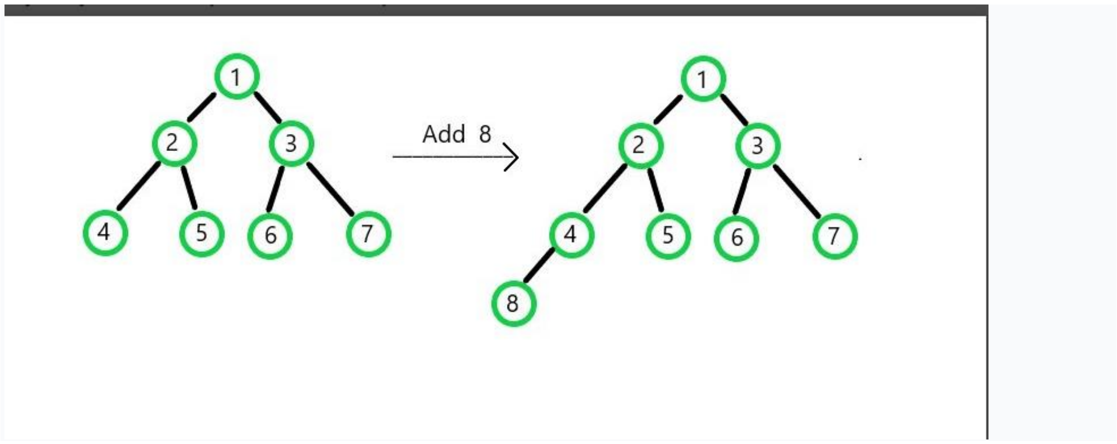
Min- heap:

In mean-heap, the key value of a node is lesser than or equal to the key value of the lowest child.

Hence, ***H[Parent(i)] ≤ H[i]***

In this context, basic operations are shown below with respect to Max-Heap. Insertion and deletion of elements in and from heaps need rearrangement of elements. Hence, **Heapify** function needs to be called

- Min Heap conforms to the above properties of heap.
- In min heap, every node contains lesser value element than its child nodes.
- Thus, root node contains the smallest value element.



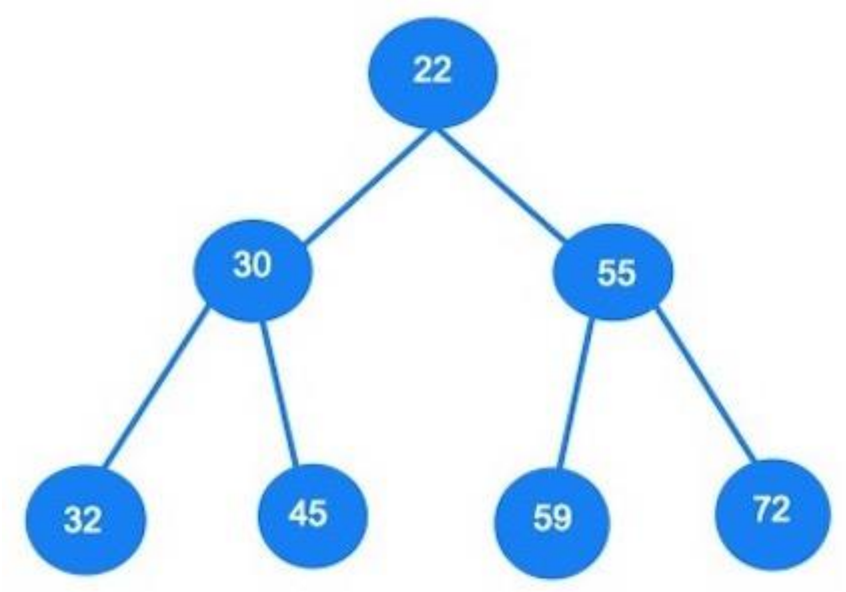
## Properties of Binary Heap

All right, now with the basics out of the way, let's take a closer look at the specific properties of the heap data structure.

### 1. Ordering

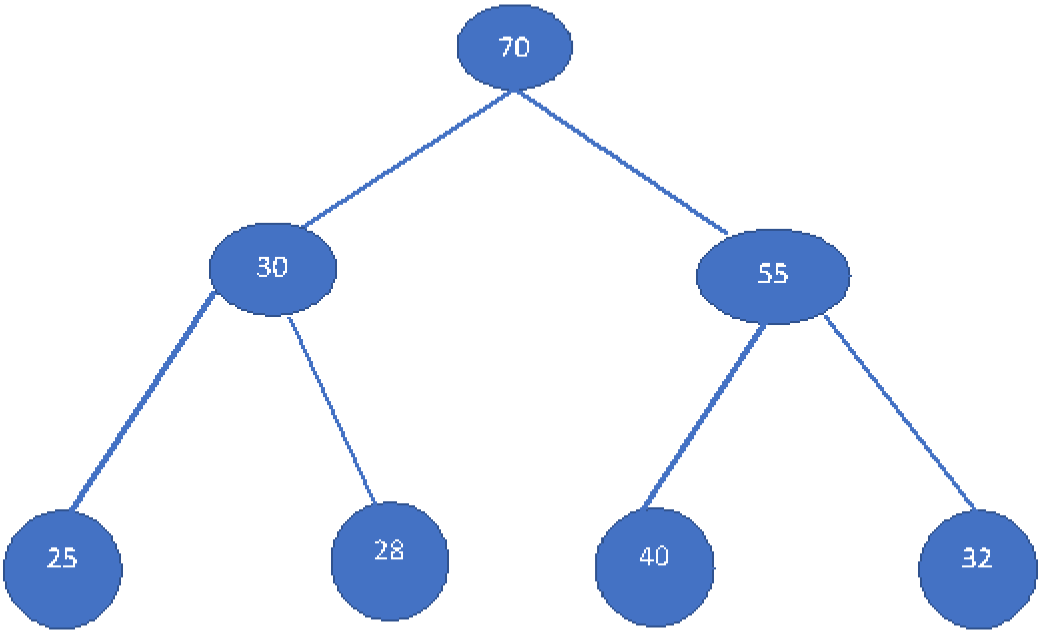
Nodes must be arranged in an order according to values. The values should follow min-heap or max-heap property.

In **min-heap** property, the value of each node, or child, is greater than or equal to the value of its parent, with the minimum value at the root node.



*Min-heap*

In **max-heap** property, the value of each node, or child, is less than or equal to the value of its parent, with the maximum value at the root node.

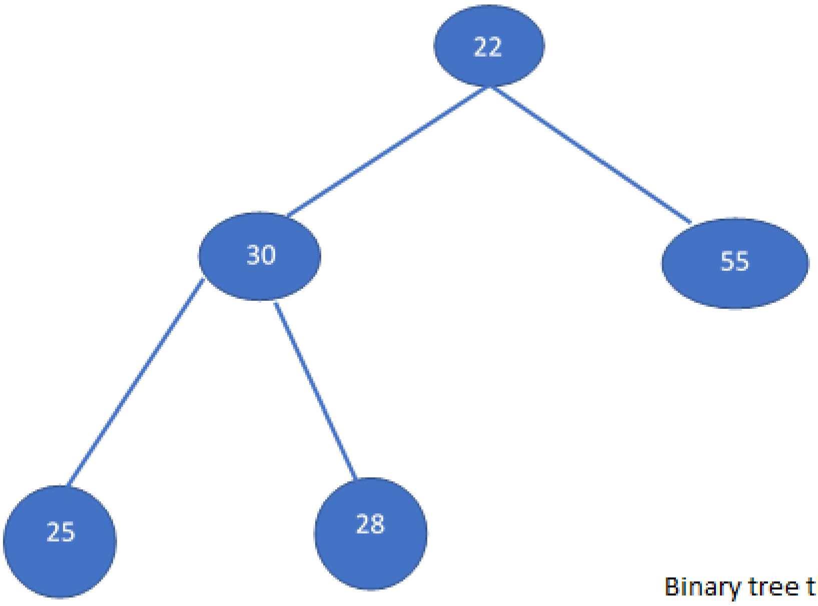


***Max-heap***

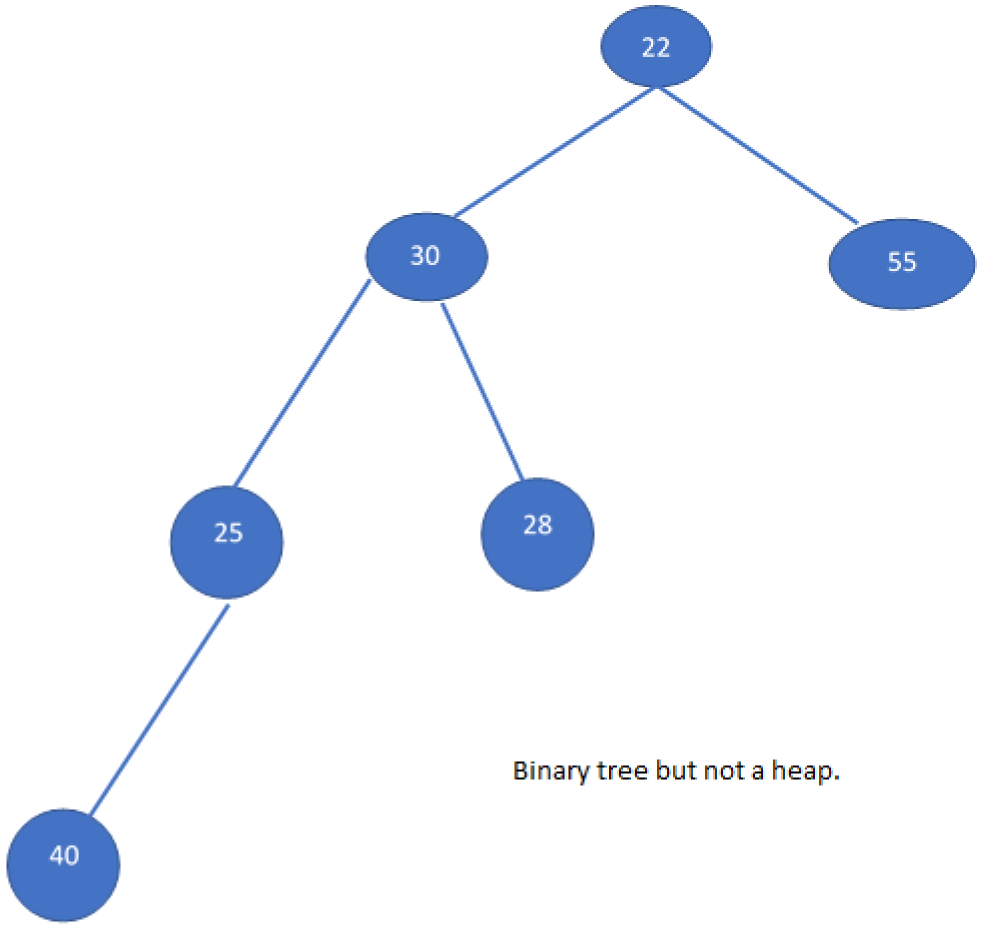
## 2. Structural

All levels in a heap should be full. In other words, it should be a complete binary tree:

- All levels of heap should be full, except the last one.
- Nodes or child must be filled from left to right strictly.
- Heap doesn't follow binary search tree principle. The values in right and left child or nodes don't matter.



Binary tree that is a Heap

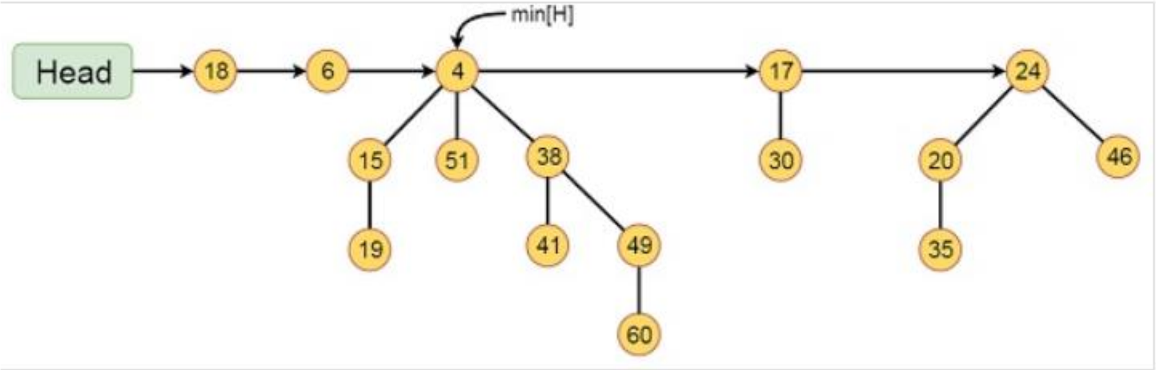


Binary tree but not a heap.

Fibonacci Heap :

A fibonacci heap is a data structure that consists of a collection of trees which follow min heap or max heap property. We have already discussed **min heap** and **max heap property** in the Heap Data Structure article. These two properties are the characteristics of the trees present on a fibonacci heap.

Example of fabonacci Series :



his Fibonacci Heap H consists of five Fibonacci Heaps and 16 nodes. The line with arrow head indicates the root list. Minimum node in the list is denoted by min[H] which is holding 4.

### What is a Splay Tree?

A splay tree is a self-balancing tree, but AVL and Red-Black trees are also self-balancing trees then. What makes the splay tree unique two trees. It has one extra property that makes it unique is splaying.

### Splaying

After an element is accessed, the splay operation is performed, which brings the element to the root of the tree. If the element is not in a root position, splaying can take one of three patterns:

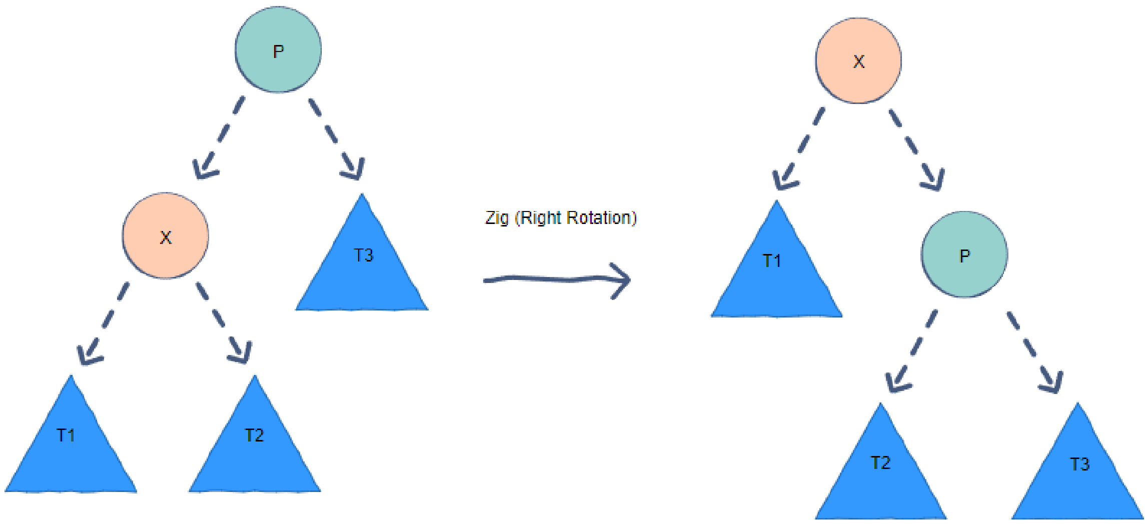
1. Zig (or zag) step
2. Zig-zig (or zag-zag) step

3. Zig-zag (or zag-zig) step

The step you take is dependent on the position of the node. *If the node is at the root, it is immediately returned.*

1. Zig (or zag)

When no grandparent node exists, the splay function will move the node up to the parent with a single rotation. A *left rotation* is a *zag* and a *right rotation* is a *zig*.



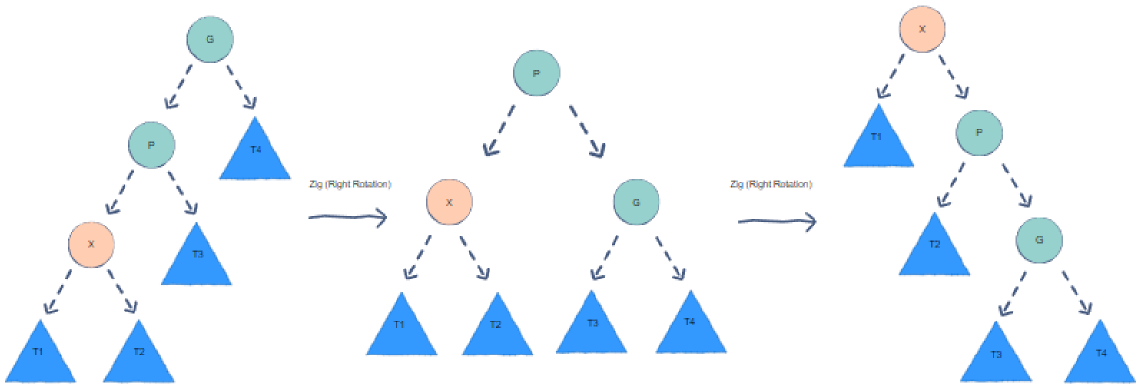
X is splayed, P is parent, T1, T2 and T3 are subtrees

Note: A left rotation corresponds to a right placement (the node is right of the parent) and vice versa.

X is splayed, P is parent, T1, T2 and T3 are subtrees

2. Zig-zig (or zag-zag)

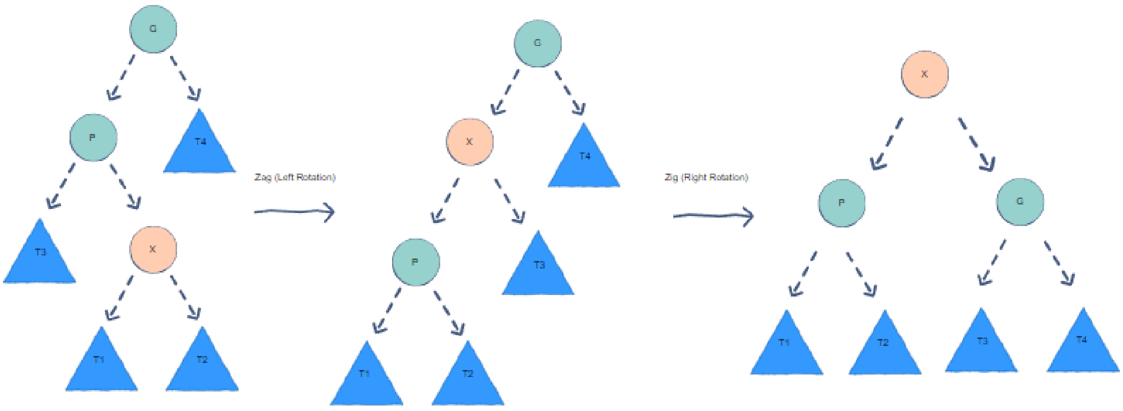
When a grandparent and parent node exist and are placed in a similar orientation (e.g., the parent is left of the grandparent and the node is left of the parent), the operation is either zig-zig (left) or zag-zag (right).



X is splayed, P is parent, and G is grandparent. T1, T2 , T3 and T4 are subtrees.

### 3. Zig-zag (or zag-zig)

A zig-zag corresponds to the parent being left of the grandparent and right of the parent. A zag-zig is the opposite.



X is splayed, P is parent, and G is grandparent. T1, T2 , T3, and T4 are subtrees

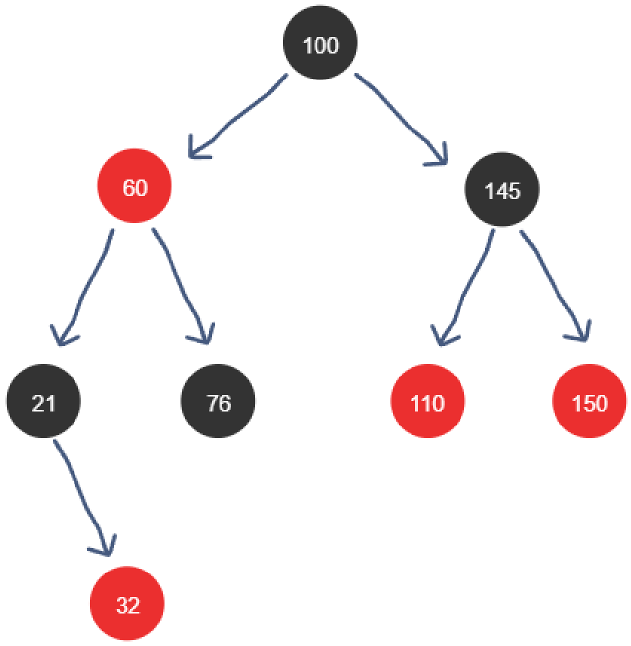
## Red -Black Tree

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

A red-black tree satisfies the following properties:

1. **Red/Black Property:** Every node is colored, either red or black.
2. **Root Property:** The root is black.
3. **Leaf Property:** Every leaf (NIL) is black.
4. **Red Property:** If a red node has children then, the children are always black.
5. **Depth Property:** For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).

Example



A correct red-black tree.

The tree above ensures that every path from the root to a leaf node has the same amount of black nodes. In this case, there is *one* (excluding the root node).

## Properties of a red-black tree

- Each tree node is colored either *red* or *black*.
- The *root* node of the tree is always *black*.
- Every path from the root to any of the leaf nodes must have the same number of black nodes.
- No two red nodes can be *adjacent*, i.e., a red node cannot be the parent or the child of another red node.

# UNIT-3

## GRAPH ALGORITHMS

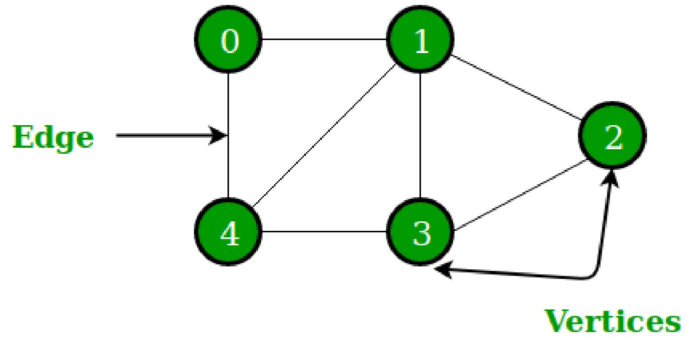
Review of graph algorithms:-Traversal Methods(Depth first and Breadth first search),Topological sort, Strongly connected components, Minimum spanning trees- Kruskal and Prims, Single source shortest paths, Relaxation, Dijkstras Algorithm, Bellman- Ford algorithm, Single source shortest paths for directed acyclic graphs, All pairs shortest paths- shortest paths and matrix multiplication, Floyd-Warshall algorithm.

Computational Complexity:-Basic Concepts, Polynomial Vs Non-Polynomial Complexity, NP- hard and NP-complete classes.

Topperworld.in

### Graph:-

It is a non -linear , non – primitive data structure i.e. represented with a set of verticle that are connected by edge i.e. G (V , e)



## Classification of Graph

The following are some of the commonly used terms in graph :

| **Term** | **Description** |
|---|---|
| Vertex | Every individual data element is called a vertex or a node. In the above image,<br>A, B, C, D & E are the vertices. |
| Edge (Arc) | It is a connecting link between two nodes or vertices. Each edge has two ends<br>and is represented as (startingVertex , endingVertex). |
| Undirected Edge | It is a bidirectional edge. |
| Directed Edge | It is a unidirectional edge. |
| Weighted Edge. | An edge with value (cost) on it. |
| Degree | The total number of edges connected to a vertex in a graph. |
| Indegree | The total number of incoming edges connected to a vertex. |
| Outdegree | The total number of outgoing edges connected to a vertex. |
| Self-loop | An edge is called a self-loop if its two endpoints coincide with each other. |
| Adjacency | Vertices are said to be adjacent to one another if there is an edge connecting them. |

## Graph Traversal Algorithm

i)DFS (Depth First Search )

ii)BFS ( Breadth First Search )

## DFS (Depth First Search )

Breadth-first search (BFS) is an algorithm that is used to graph data or searching tree or traversing structures. The full form of BFS is the Breadth-first search.

BFS is a graph traversal approach in which you start at a source node and layer by layer through the graph, analyzing the nodes directly related to the source node. Then, in BFS traversal, you must move on to the next-level neighbor nodes.

According to the BFS, you must traverse the graph in a breadthwise direction:

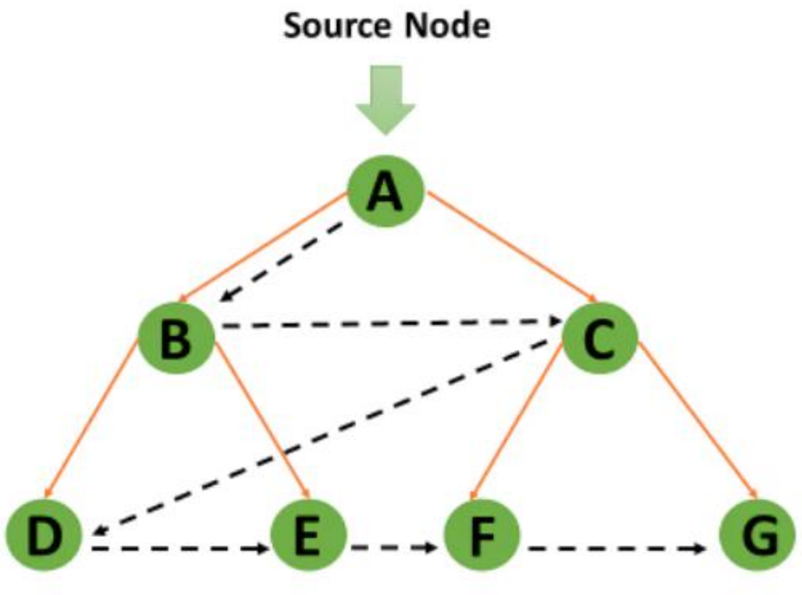
- To begin, move horizontally and visit all the current layer's nodes.
- Continue to the next layer.

Algorithm of BFS :

Algorithm of BFS ( G , s)

```
1.for each vertex  $v \in V[G]-\{s\}$
2. color [u]= White
3.d[u] := $\infty$
4. $\pi$[u]=NIL
5 color [s]:= GRAY
6.d[s]= NIL
7. d[s]:= NIL
8.Q= $\phi$
9. ENQUEUE =( Q,s)
```

```
while ($Q \neq \phi$)
{ u=DEQUEUE(Q)
    for each v € Adj (u) in 'G'
    { If colour [v] = WHITE
        color [v]:= GRAY
        d[v]=d[u]+1
        $\pi$[v]=u
        ENQUEUE (Q, v) }
    Color [u]= BLACK }
```



Breadth-First Search uses a Queue data Structure to store the node and mark it as "visited" until it marks all the neighboring vertices directly related to it. The queue operates on the First

In First Out (FIFO) principle, so the node's neighbors will be viewed in the order in which it inserts them in the node, starting with the node that was inserted first.

## Application of DFS

i)Tropological Sort

ii)Strongly Connected Components

1)Tropological Sort :

Tropological sort is an algorithm which sorts a directed graph by returning an array or a vector, or a list, that consists of nodes where each node appears **before** all the nodes it points to.

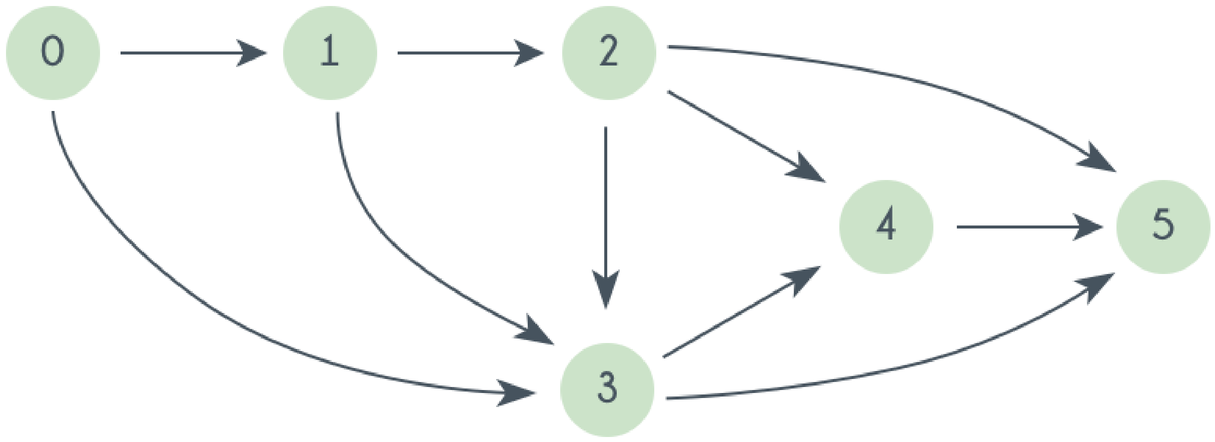
Here, we'll simply refer to it as an array, you can use a vector or a list too.

Say we had a graph,

```
a --> b --> c
```

then the topological sort algorithm would return **- [a, b, c]**. Why? Because, a points to b, which means that a must come before b in the sort. b points to c, which means that b must come before c in the sort.

Let's take a graph and see the algorithm in action. Consider the graph given below:



Initially in_degree[0]=0 and T is empty

QUEUE : 0

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| *in_degree* | 0 | 1 | 2 | 2 | 2 | 3 |
|  | *0* | *1* | *2* | *3* | *4* | *5* |

T : --

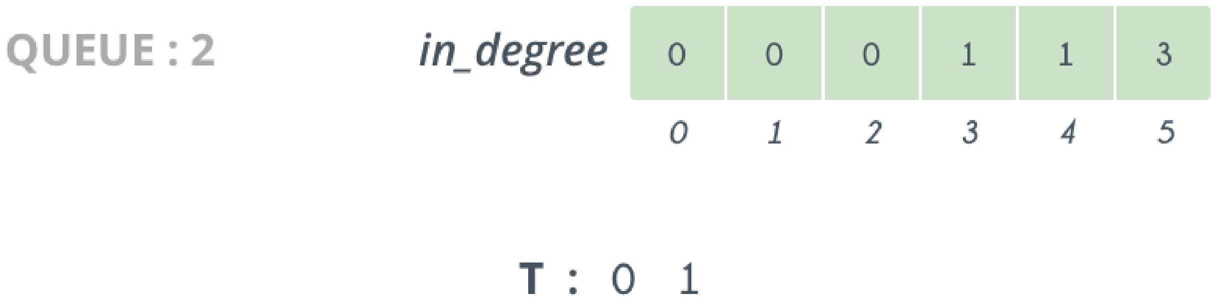
So, we delete 0 from Queue and append it to T. The vertices directly connected to 0 are 1 and 2 so we decrease their in_degree[] by 1. So, now in_degree[1]=0 and so 1 is pushed in Queue.

QUEUE : 1

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| *in_degree* | 0 | 0 | 1 | 1 | 2 | 3 |
|  | *0* | *1* | *2* | *3* | *4* | *5* |

T : 0

Next we delete 1 from Queue and append it to T. Doing this we decrease in_degree[2] by 1, and now it becomes 0 and 2 is pushed into Queue.

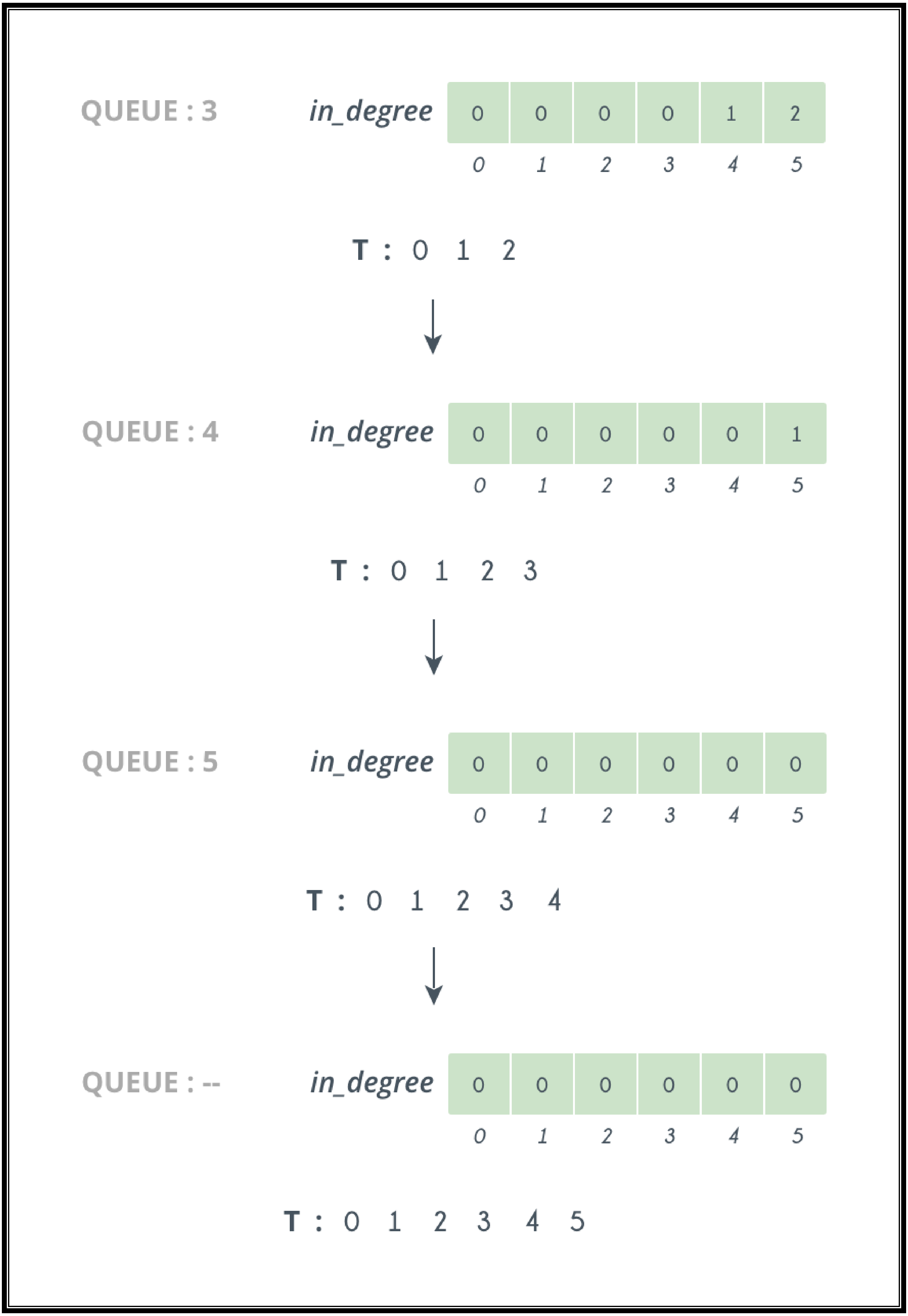


QUEUE : 2

| in_degree | 0 | 0 | 0 | 1 | 1 | 3 |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |

T : 0 1

So, we continue doing like this, and further iterations looks like as follows:



QUEUE : 3

| | | | | | | |
|---|---|---|---|---|---|---|
| *in_degree* | 0 | 0 | 0 | 0 | 1 | 2 |
| | *0* | *1* | *2* | *3* | *4* | *5* |

T : 0 1 2

QUEUE : 4

| | | | | | | |
|---|---|---|---|---|---|---|
| *in_degree* | 0 | 0 | 0 | 0 | 0 | 1 |
| | *0* | *1* | *2* | *3* | *4* | *5* |

T : 0 1 2 3

QUEUE : 5

| | | | | | | |
|---|---|---|---|---|---|---|
| *in_degree* | 0 | 0 | 0 | 0 | 0 | 0 |
| | *0* | *1* | *2* | *3* | *4* | *5* |

T : 0 1 2 3 4

QUEUE : --

| | | | | | | |
|---|---|---|---|---|---|---|
| *in_degree* | 0 | 0 | 0 | 0 | 0 | 0 |
| | *0* | *1* | *2* | *3* | *4* | *5* |

T : 0 1 2 3 4 5

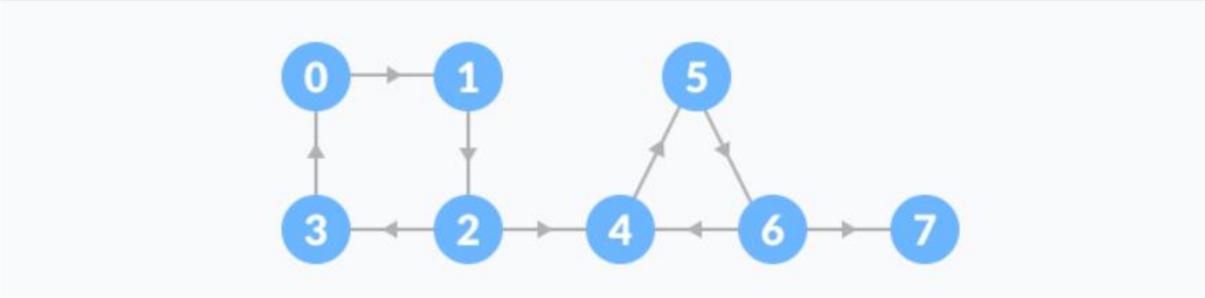
So at last we get our Topological sorting in T i.e. : 0, 1, 2, 3, 4, 5

## ii) Strongly Connected Component

A strongly connected component is the portion of a directed graph in which there is a path from each vertex to another vertex. **It is applicable only on a directed graph**.

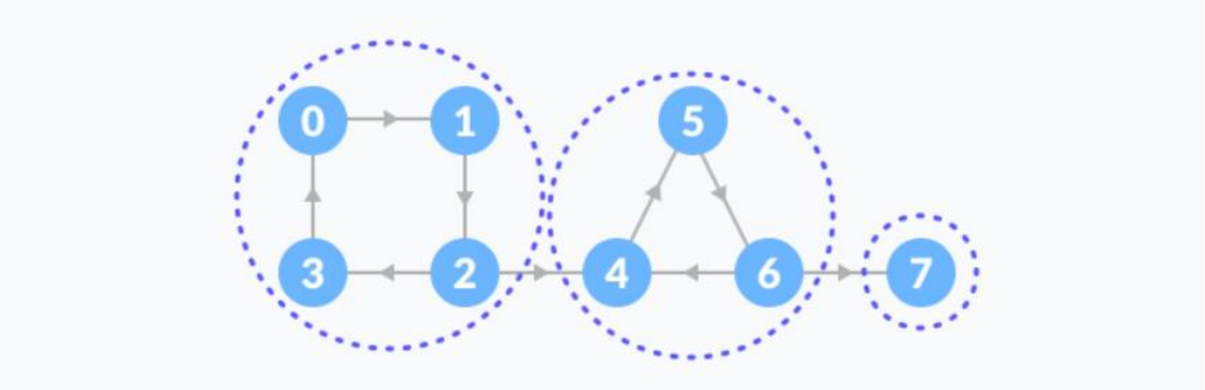
For example:

Let us take the graph below.



Initial Graph

The strongly connected components of the above graph are:



You can observe that in the first strongly connected component, every vertex can reach the other vertex through the directed path.

## 2 . DFS (Depth First Search)

DFS is a recursive traversal algorithm for searching all the vertices of a graph or tree data structure. It starts from the first node of graph G and then goes to further vertices until the goal vertex is reached.

- DFS uses stack as its backend data structure
- edges that lead to an unvisited node are called discovery edges while the edges that lead to an already visited node are called block edges.

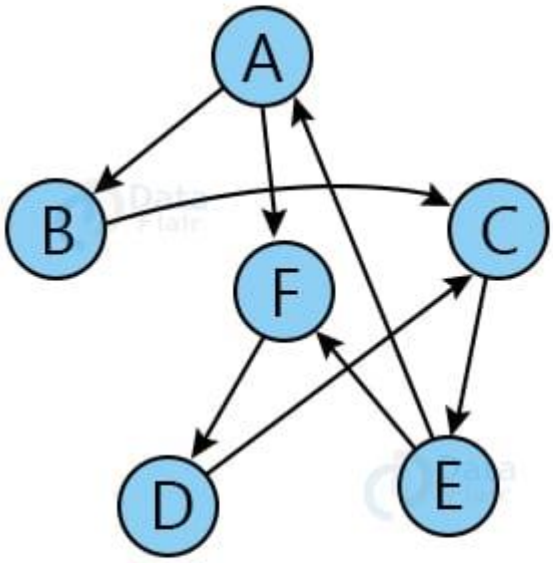
```
1.Algorithm of DFS ( G )
2.for each vertex u ∈ V[G]
3.color u =WHITE
4.π[u]=NIL
5.Time =0;
6.for each value u ∈ V[G]
7.if ( color [u]= WHITE
8.DFS -VISIT (G ,u )
Algo DFS VISIT ( G , u )
Time = time +1;  // white vertex ‘ u’ has just been discover
d[u] = time
color[u] =GRAY
for each v $\in$ Adj[u] in ‘G’ // explore edge ( V , v )
if  ( color [v] =WHITE )
$\pi$[v]=u
DFS -VISIT ( G , V)
Color [u] =BLACK
Time =time +1
f[u] = time
d[v]=d[u]+1
$\pi$[v]=u
ENQUEUE (Q, v) }
Color [u]= BLACK }
```

G = is a graph consisting of ‘v’ vertex where  :-

$\pi$ denotes the parents model from which a particular nodes will be explore

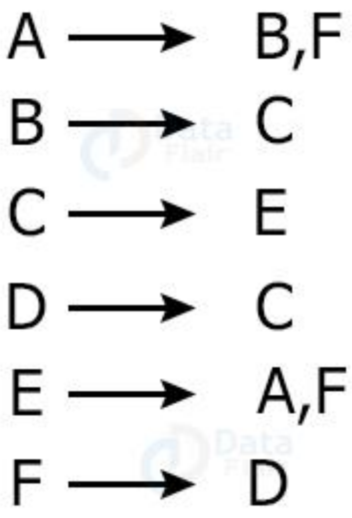
d[u] & f[u] = denotes the discovery & finishing time of the nodes u.

Let us consider the graph below:



Starting node: A

**Step 1:** Create an adjacency list for the above graph.



A → B,F

B → C

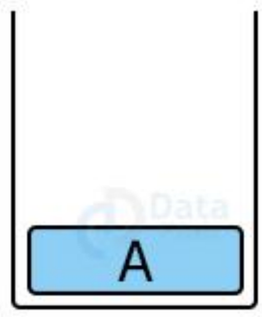
C → E

D → C

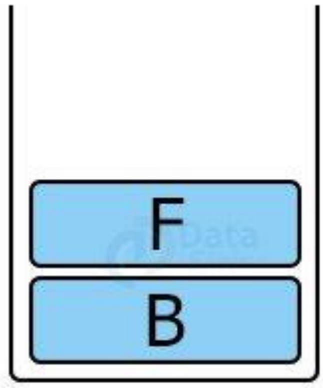
E → A,F

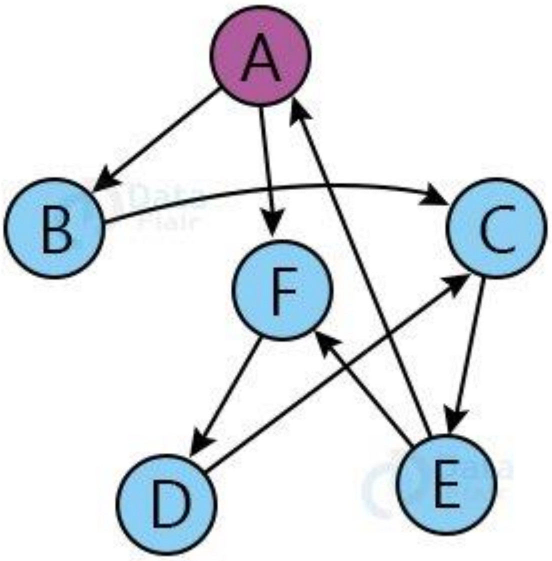
F → D

**tep 2:** Create an empty stack.
**Step 3:** Push 'A' into the stack

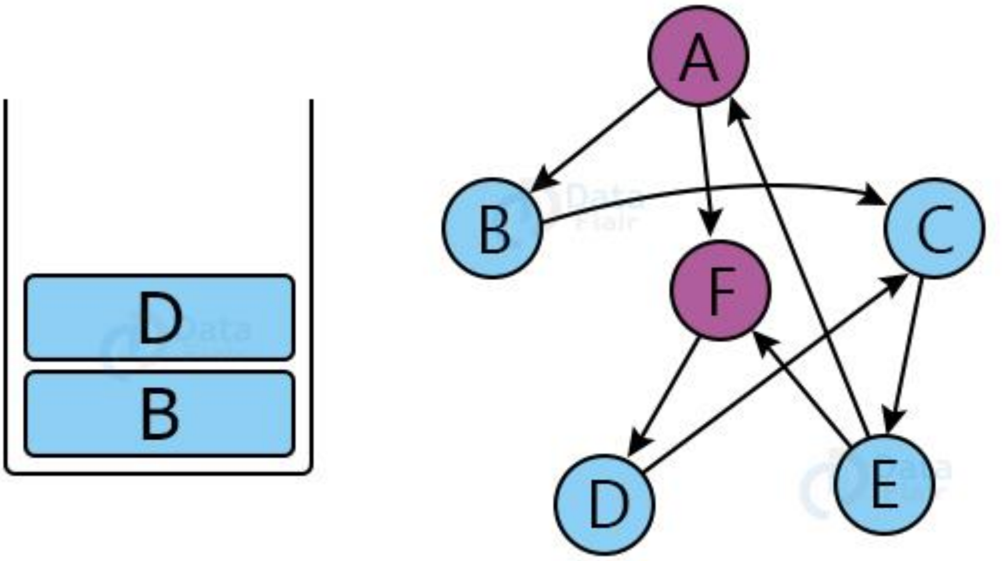


**Step 4:** Pop 'A' and push 'B' and 'F'. Mark node 'A' as the visited node

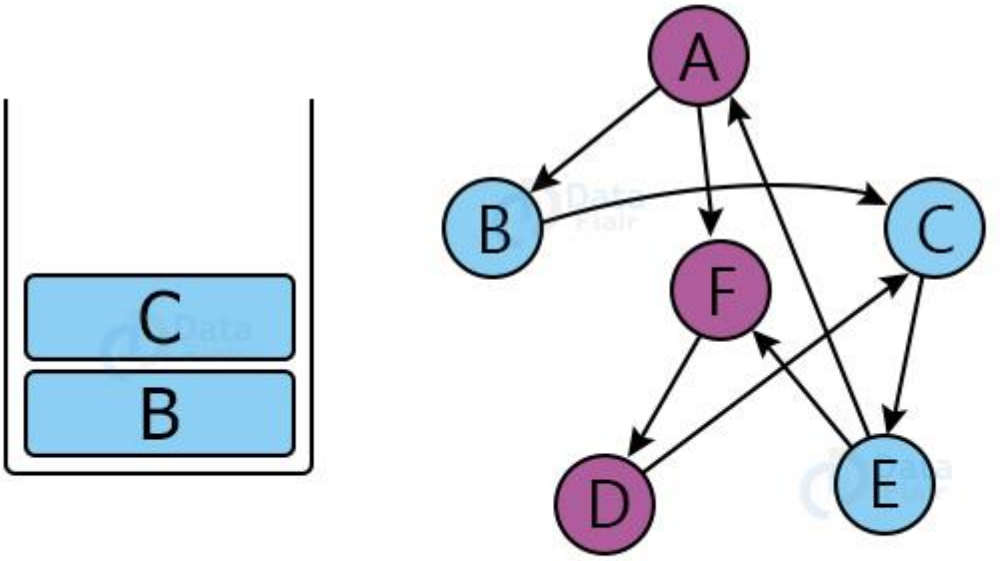




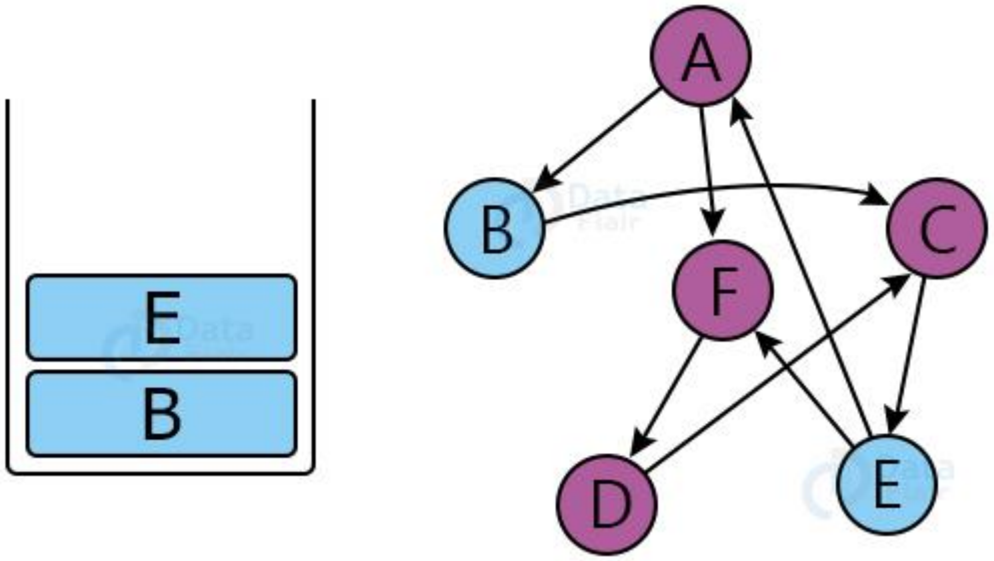
**Step 5:** pop 'F' and push 'D'. Mark 'F' as a visited node.



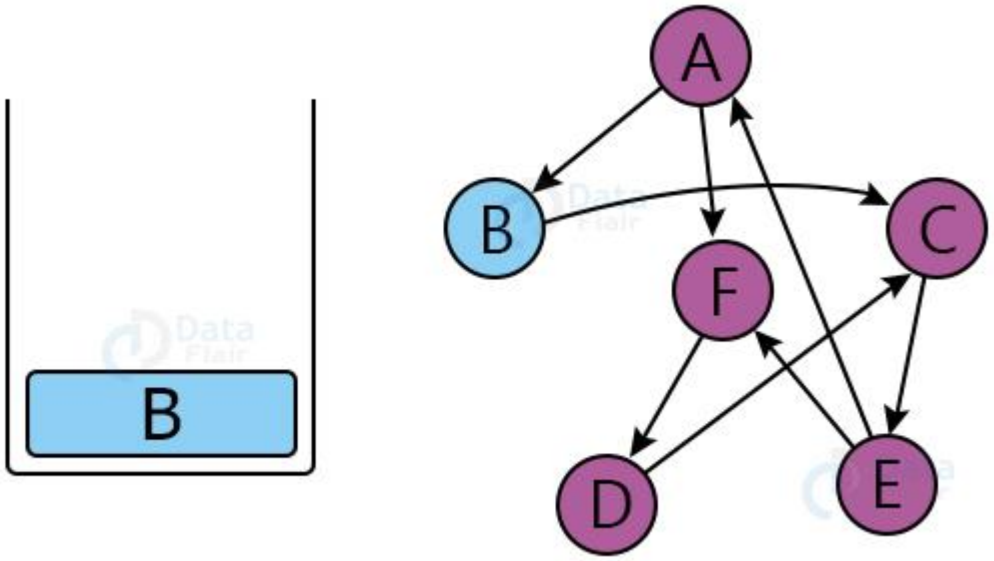
**Step 6:** pop ‘D’ and push ‘C’. Mark ‘D’ as a visited node.



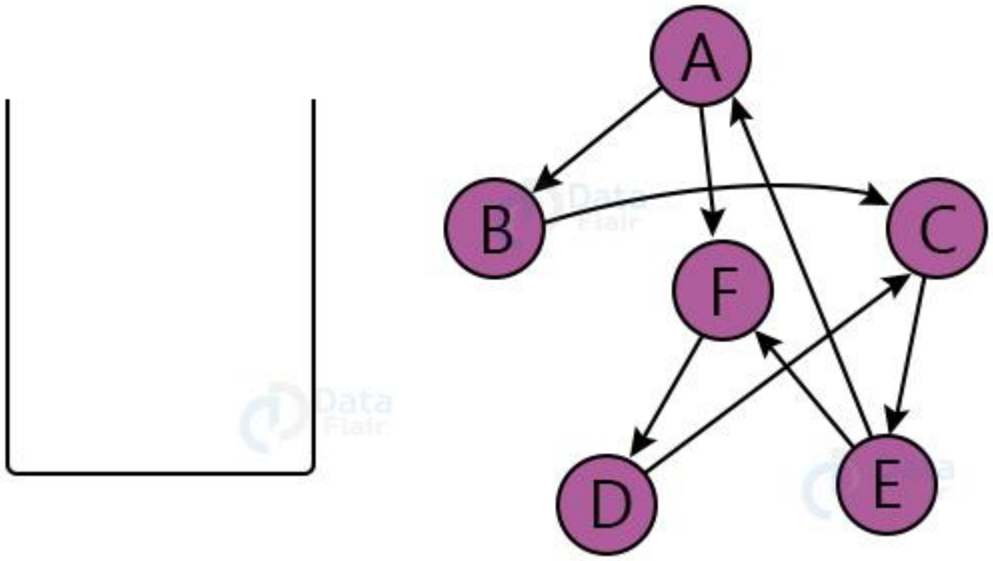
**Step 7:** pop ‘C’ and push ‘E’. Mark ‘C’ as a visited node.



**Step 8:** pop ‘E’. Mark ‘E’ as a visited node. No new node is left.



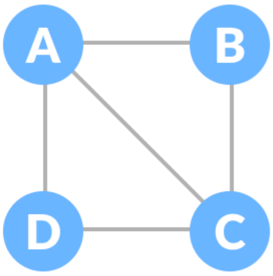
**Step 9:** pop ‘B’. Mark ‘B’ as visited. All the nodes in the graph are visited now.



# Minimum Spanning Tree

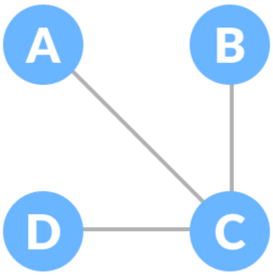
Before we learn about spanning trees, we need to understand two graphs: undirected graphs and connected graphs.

An **undirected graph** is a graph in which the edges do not point in any direction (ie. the edges are bidirectional).



Undirected Graph

A **connected graph** is a graph in which there is always a path from a vertex to any other vertex.



Connected Graph

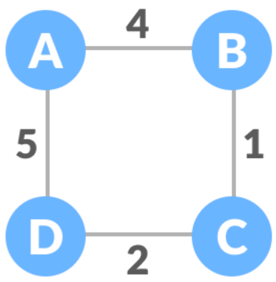
## Spanning tree

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

## Example of a Spanning Tree

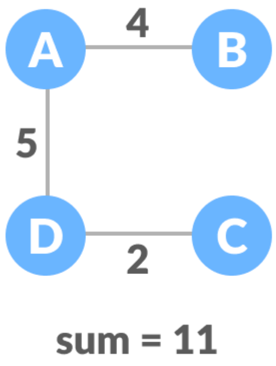
Let's understand the above definition with the help of the example below.

The initial graph is:

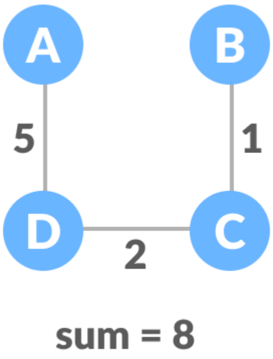


Weighted graph

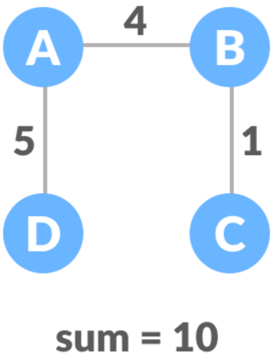
The possible spanning trees from the above graph are:



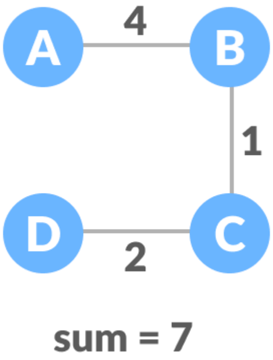
Minimum spanning tree - 1



Minimum spanning tree - 2

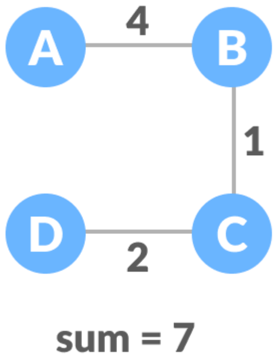


Minimum spanning tree - 3



Minimum spanning tree - 4

The minimum spanning tree from the above spanning trees is:



Minimum spanning tree

The minimum spanning tree from a graph is found using the two method  in MST

i) Kruskal algorithm

ii) Prims algorithm

## Kruskal's Algorithm :

**Kruskal's Algorithm** is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

### How does Kruskal's algorithm work?

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- First, sort all the edges from low weight to high.
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

The applications of Kruskal's algorithm are -

- Kruskal's algorithm can be used to layout electrical wiring among cities.
- It can be used to lay down LAN connections.

## Algorithm of Kruskal :

1.{ constant a min -heap out of thr edge cost using HEAPIFY

2. fir I =1 to n ;

3. do parent [i] =1 // each vertex is in a different set

4. i=0; min cost =0;

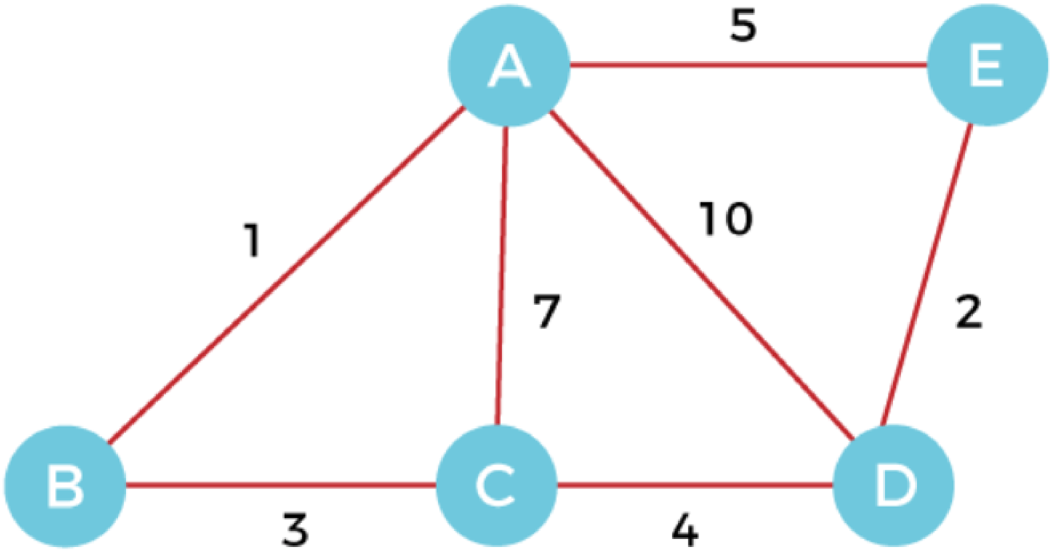
5. while ( ( i<n-1) && ( heap not empty)) do

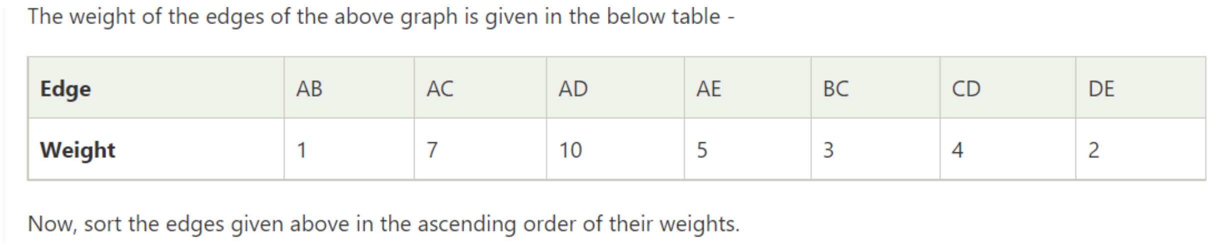
```
6.{ Delete 0 min cost edge (u,v) from the heap & re-heapify using ADJUST
7. j= FIND (u) ,k = FIND (v)
8.if ( j ≠ k)
9. { i= i+1
10. t [ i ,1] =u , t[i,z]=v;
11. Min cost = min cost + cost ( u, v)
12. UNION (j , k);
13.}}
14. if (i≠ n-1) then write ( No spanning Tree is possible else return Min cost
15.}
```

## Example of Kruskal's algorithm

Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.

Suppose a weighted graph is -

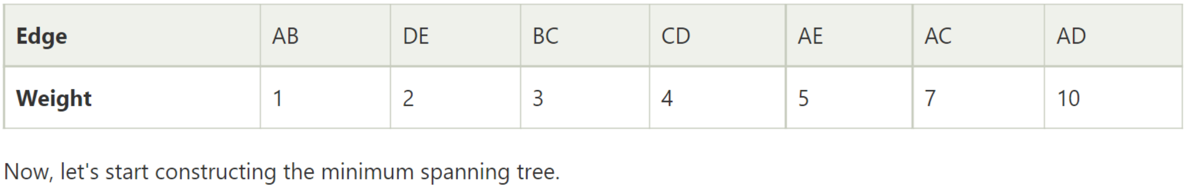




The weight of the edges of the above graph is given in the below table -

| Edge | AB | AC | AD | AE | BC | CD | DE |
|---|---|---|---|---|---|---|---|
| Weight | 1 | 7 | 10 | 5 | 3 | 4 | 2 |

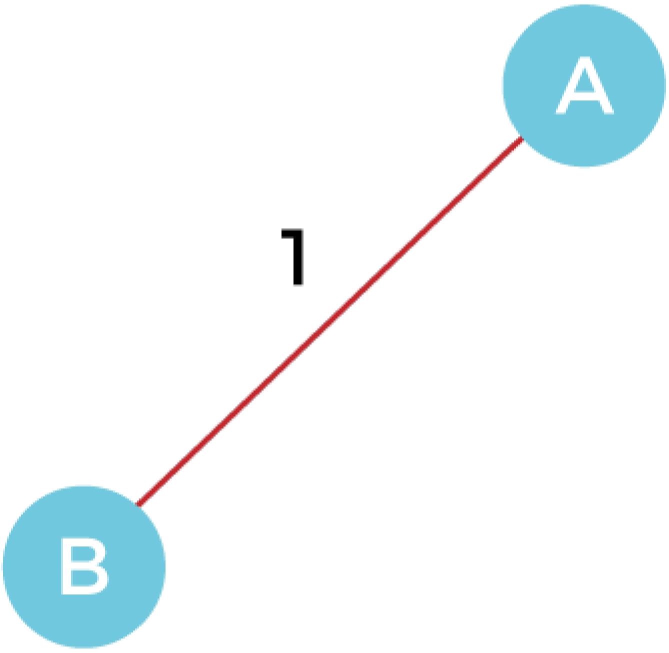
Now, sort the edges given above in the ascending order of their weights.



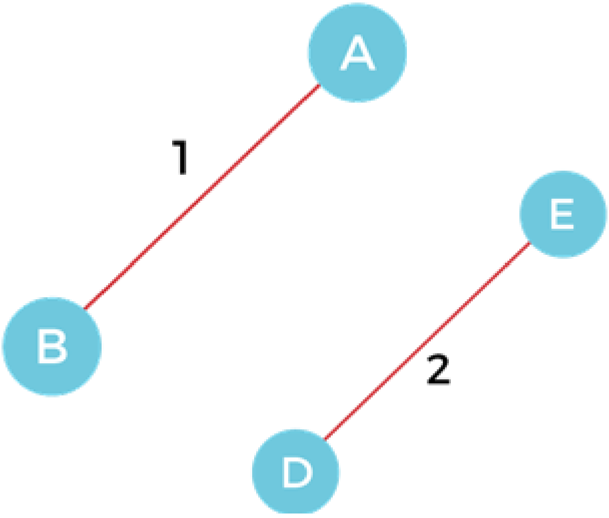
| Edge | AB | DE | BC | CD | AE | AC | AD |
|---|---|---|---|---|---|---|---|
| Weight | 1 | 2 | 3 | 4 | 5 | 7 | 10 |

Now, let's start constructing the minimum spanning tree.

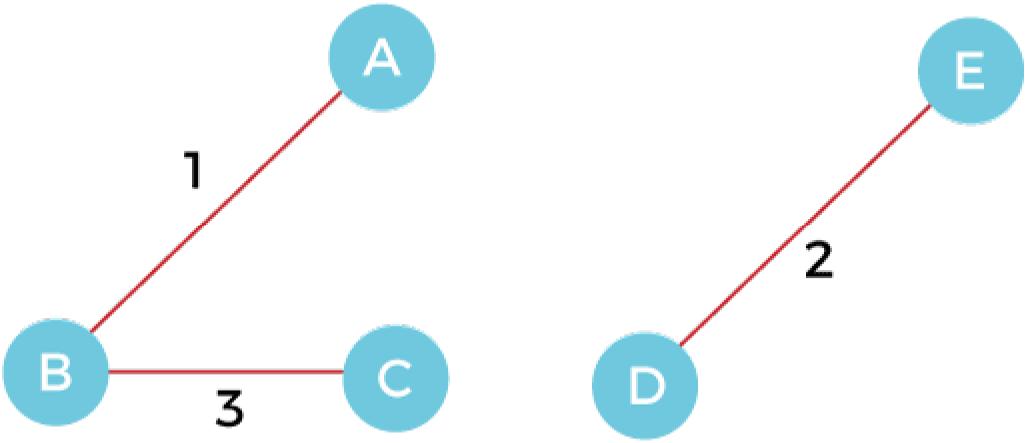
**Step 1** - First, add the edge **AB** with weight **1** to the MST.



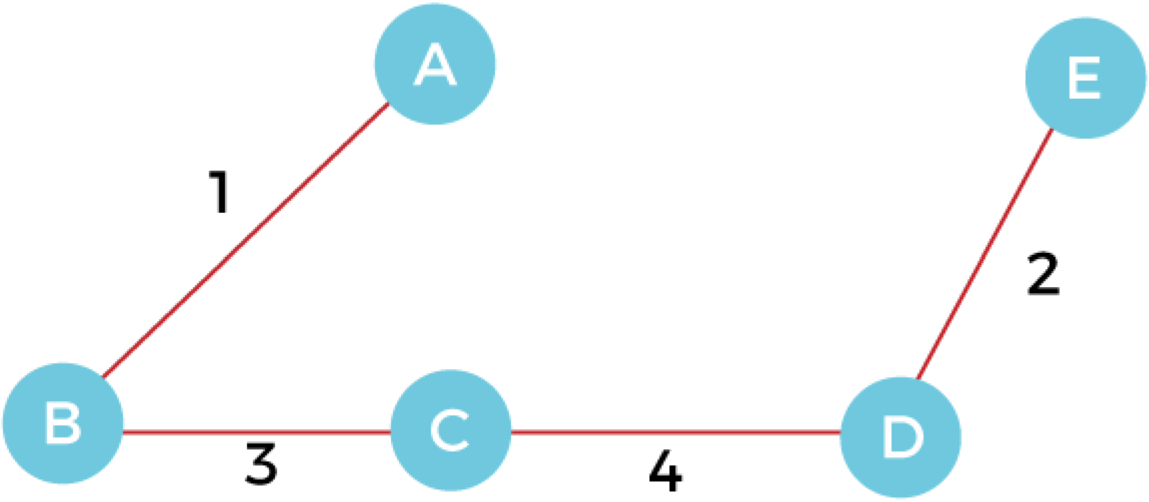
**Step 2** - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



**Step 3** - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.



**Step 4 -** Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.

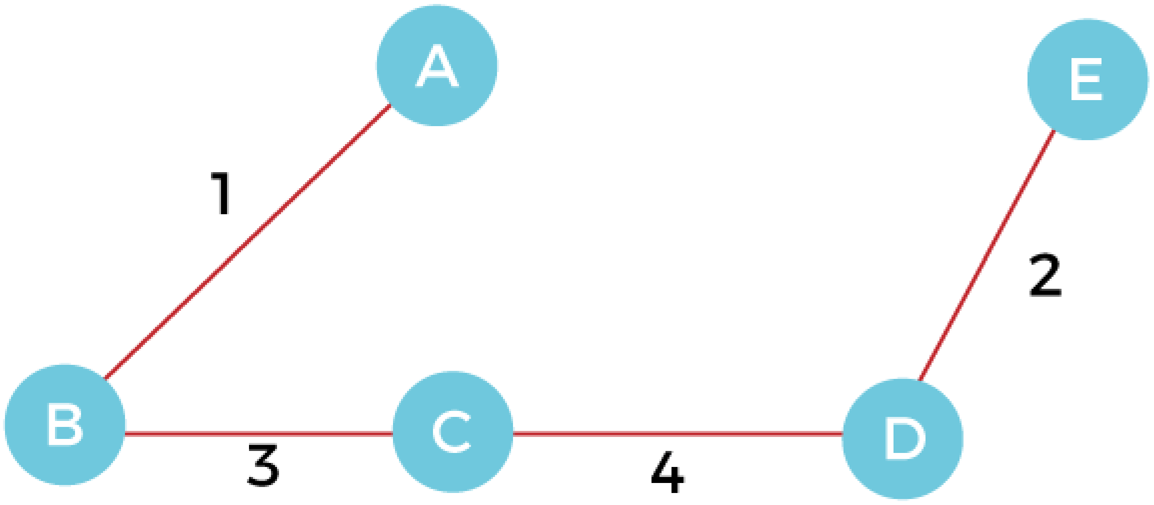


**Step 5 -** After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

**Step 6 -** Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

**Step 7 -** Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

## Prims Algorithm

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

- Prim’s Algorithm is a famous greedy algorithm.

- It is used for finding the Minimum Spanning Tree (MST) of a given graph.

- To apply Prim’s algorithm, the given graph must be weighted, connected and undirected

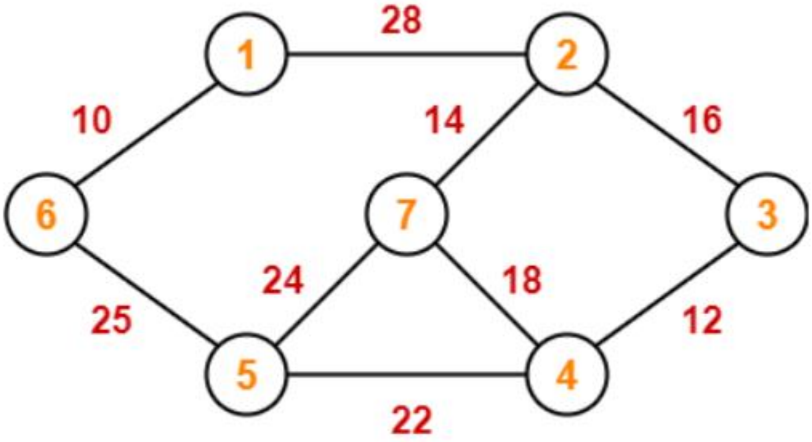
```
Algorithm of Prims ( e , cost , n, t)

1.{ let ( k,l) be an edge of minimum cost in e
2. Min cost = cost ( k, l ) ,I =1
3. { [ I ,1 ] = k  , t [ I , z ] =l
4. for I =1 to n  // initialize near
5. if ( cost [ I ,k] < cost [ i ,l ] )
6. then near [i] =k;
7.else near [i]= l }
8.near [k] = near [l]=0;
9. for I =3 to n-1  do ;
10. {  Let  j is an index such that near [;] ≠ 0  & cost [ j,
near [j] ] is minimum

11.  t [i , j] = j ; t[i , 2 ] = near [j]
12. Min cost = Min  cost  + cost [ j , near [ i ]
13. near [i] =o;
14. for k = 1 to n do  // update near
15. if ((near [k] ≠ 0) && cost [k ,near [k] > cost [ k, j])
16.then near [k] =I }
17.return Min cost }
```

Let’s take it one example :

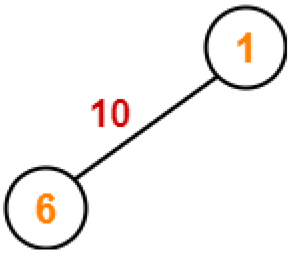
Construct the minimum spanning tree (MST) for the given graph using Prim’s Algorithm-



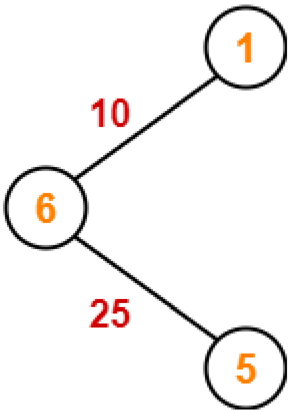
**Solution-**

The above discussed steps are followed to find the minimum cost spanning tree using Prim’s Algorithm-

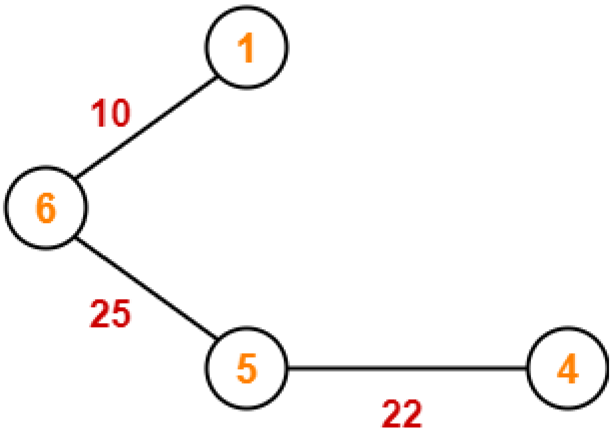
Step-01:



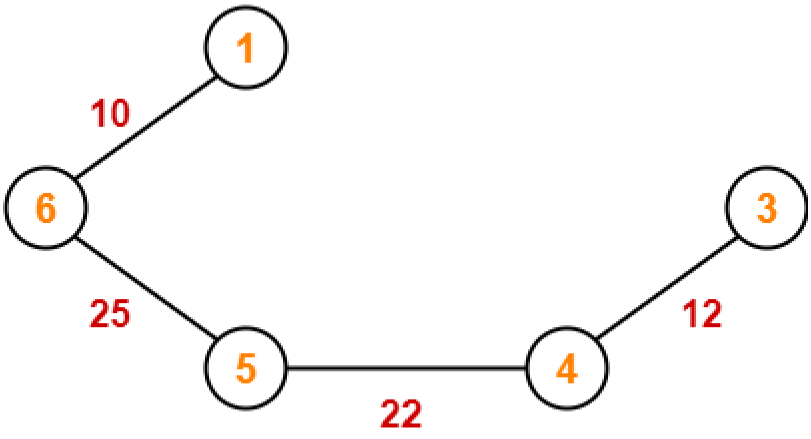
## Step-02:



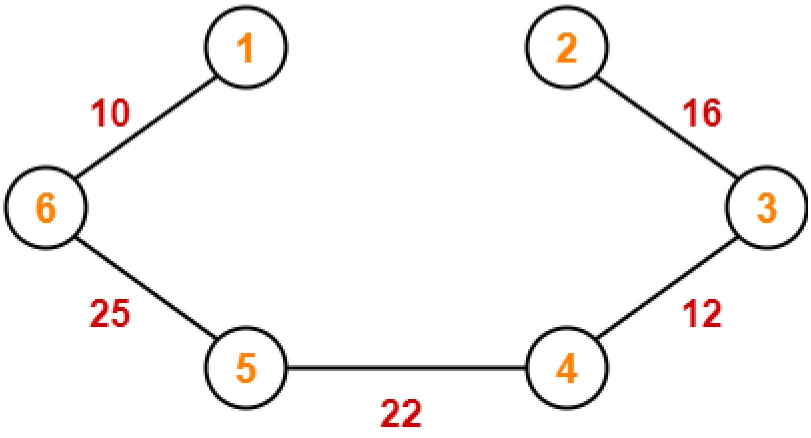
## Step-03:



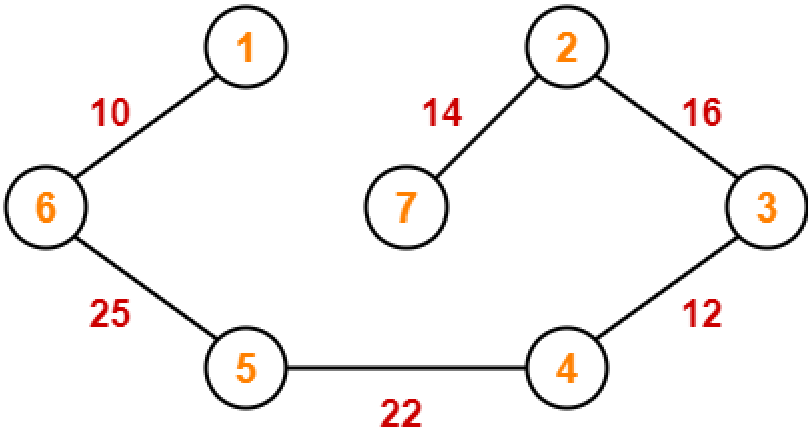
## Step-04:



## Step-05:



## Step-06:



Since all the vertices have been included in the MST, so we stop.

Now, Cost of Minimum Spanning Tree

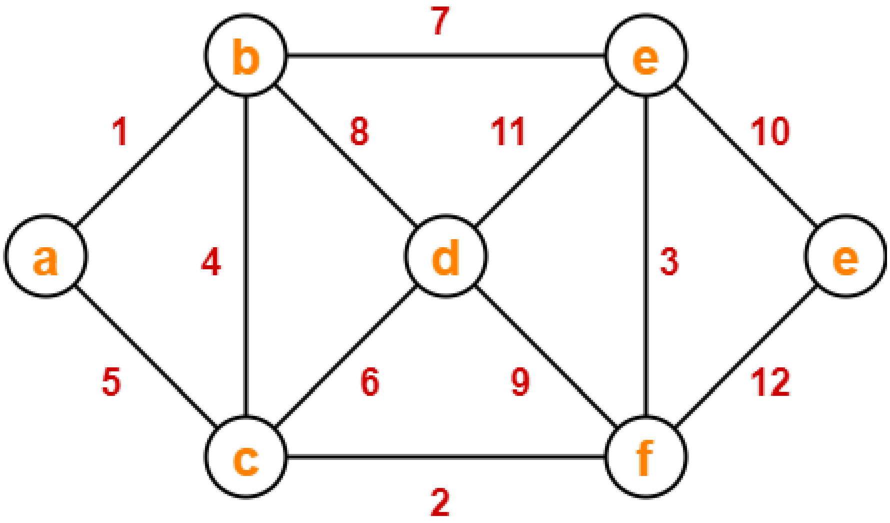
= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

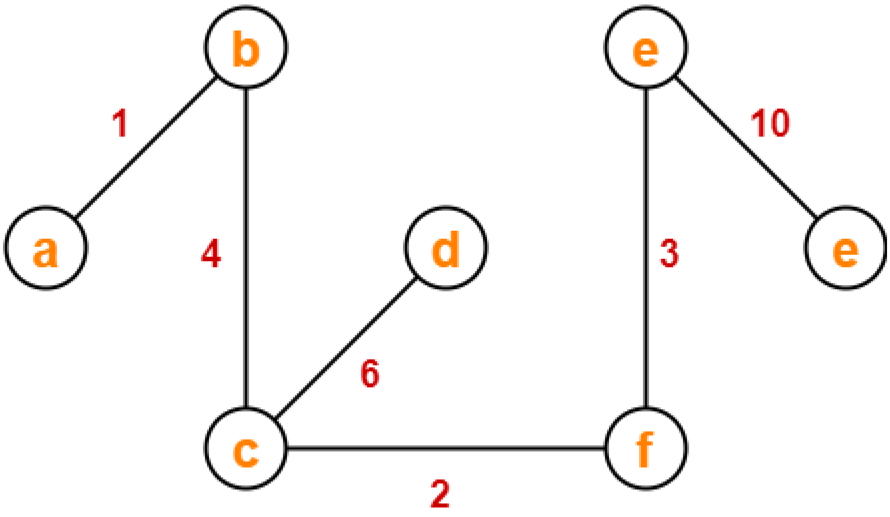
**Problem-02:**

Using Prim's Algorithm, find the cost of minimum spanning tree (MST) of the given graph-



**Solution-**

The minimum spanning tree obtained by the application of Prim's Algorithm on the given graph is as shown below-



Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

= 1 + 4 + 2 + 6 + 3 + 10

= 26 units

To gain better understanding about Prim's Algorithm,

Single source shortest paths ( SSSP)

The Single-Source Shortest Path (SSSP) problem consists of finding the shortest paths between a given vertex *v* and all other vertices in the graph. Algorithms such as Breadth-First-Search (BFS) for unweighted graphs or Dijkstra [1] solve this problem.

In a **shortest- paths problem**, we are given a weighted, directed graphs G = (V, E), with weight function **w: E** → **R** mapping edges to real-valued weights. The weight of path p = ($v_0$,$v_1$,..... $v_k$) is the total of the weights of its constituent edges:

$$w(P) = \sum_{i=1}^{k} w(v_{i-1}v_i)$$

1) Dijkstra's Algorithm,
2) Bellman Ford Algorithm

a) INITIALIZE -SINGLE SOURCE ( r)

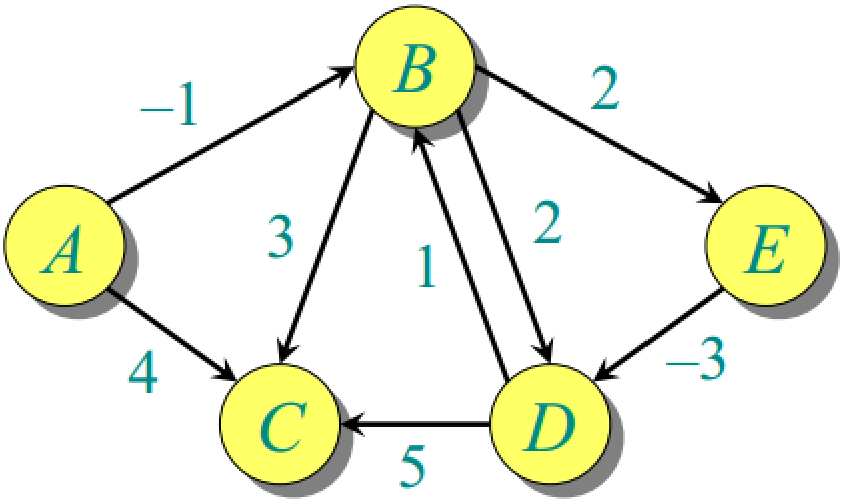
```
1.d[r] = 0;
2.d[v]=0;
```

b) ALGORITHM RELAXATION ( u , v , w)

```
1.if d[v]> d[u] + w[ u , v]
2.then d[v]=d[u]+w[u ,v]
```

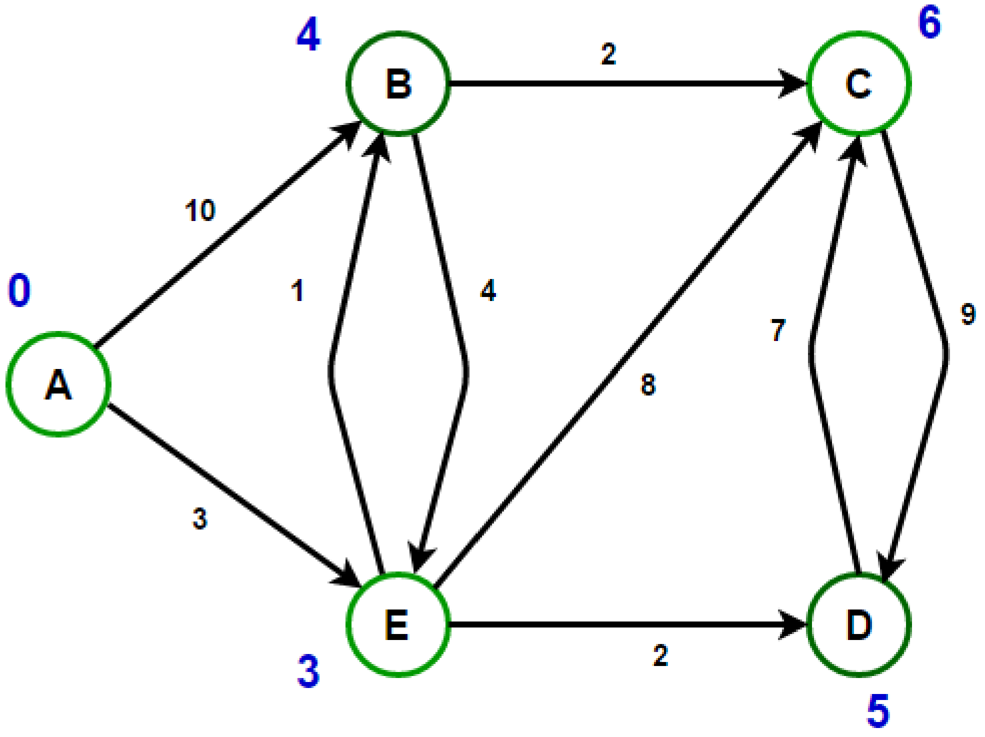
Here d[v] = distance of node v from source

Here,
W= denotes the weight of the edges
r = Source code
t= destination node
u = any intermediate node



Dijkstra's Algorithm( v ,cost , dist , n)

```
1.for I =1to n do
2 { r [ i ] = false , dist [ i ] = cost [ v , i] } // initialize s
3. r[v]=true , dist[v]=0.0 // put v in s
4.for sum = 2 to n-1 do // determine " n-1" path from
5.{ choose 'u' among those vertex not in 's'
such that d[ u] is minimum
6. r[u] = true // put ‘u’ in s
7. for ( each w adjacent to ‘u’ with r [ w] = fix )
8.if ( d[w]>d[u] + cost[u,w]) then // relaxctive of edge
9. d[w] = d[u] + cost[ u,w]
10.}
```



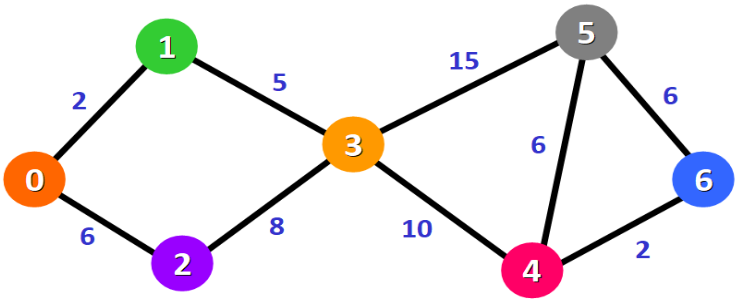
In this algorithm ,
v = source node
u = intermediate node
cost = the weight of the edge
d = distance from source node
n= total no. of vertice

‘S’ define the set which is intically empty & the vertices are choosen in set as according to the Shortest distance .

## Example of Dijkstra's Algorithm

Now that you know more about this algorithm, let's see how it works behind the scenes with a a step-by-step example.

We have this graph:



The algorithm will generate the shortest path from node 0 to all the other nodes in the graph.

**Note :** For this graph, we will assume that the weight of the edges represents the distance between two nodes.
We will have the shortest path from node 0 to node 1, from node 0 to node 2, from node 0 to node 3, and so on for every node in the graph.
Initially, we have this list of distances (please see the list below):

- The distance from the source node to itself is 0. For this example, the source node will be node 0 but it can be any node that you choose.
- The distance from the source node to all other nodes has not been determined yet, so we use the infinity symbol to represent this initially.

**Distance:**

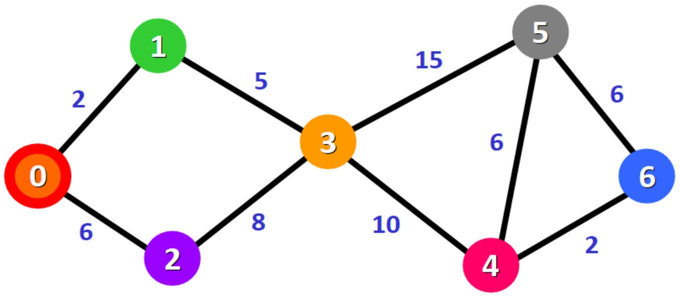
0: 0
1: ∞
2: ∞
3: ∞
4: ∞
5: ∞
6: ∞

We also have this list (see below) to keep track of the nodes that have not been visited yet (nodes that have not been included in the path):

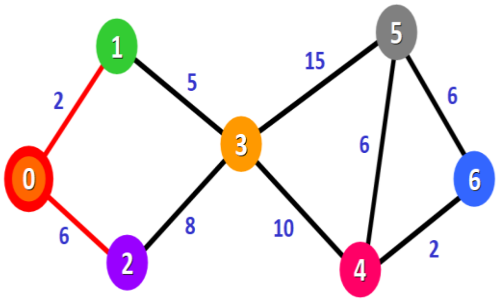
**Unvisited Nodes:** {0, 1, 2, 3, 4, 5, 6}

**Note:** Since we are choosing to start at node 0, we can mark this node as visited. Equivalently, we cross it off from the list of unvisited nodes and add a red border to the corresponding node in diagram:

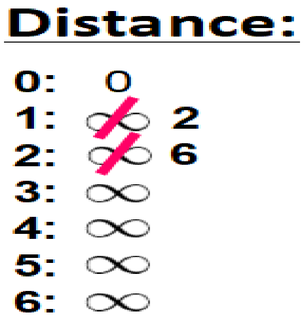
**Unvisited Nodes:** {~~0~~, 1, 2, 3, 4, 5, 6}



Now we need to start checking the distance from node 0 to its adjacent nodes. As you can see, these are nodes 1 and 2 (see the red edges):



We need to update the distances from node 0 to node 1 and node 2 with the weights of the edges that connect them to node 0 (the source node). These weights are 2 and 6, respectively:



**Distance:**

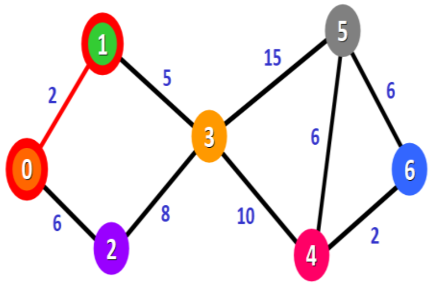
0: 0<br>
1: ~~∞~~ 2<br>
2: ~~∞~~ 6<br>
3: ∞<br>
4: ∞<br>
5: ∞<br>
6: ∞

After updating the distances of the adjacent nodes, we need to:

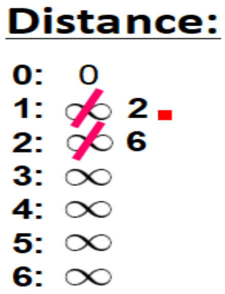
- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

If we check the list of distances, we can see that node 1 has the shortest distance to the source node (a distance of 2), so we add it to the path.

In the diagram, we can represent this with a red edge:



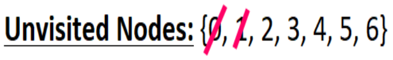
We mark it with a red square in the list to represent that it has been "visited" and that we have found the shortest path to this node:



**Distance:**

0: 0<br>
1: ~~∞~~ 2 🟥<br>
2: ~~∞~~ 6<br>
3: ∞<br>
4: ∞<br>
5: ∞<br>
6: ∞

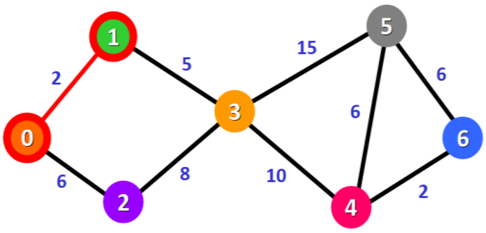
We cross it off from the list of unvisited nodes:



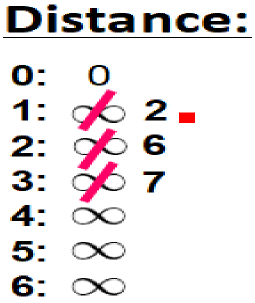
Unvisited Nodes: {~~0~~, ~~1~~, 2, 3, 4, 5, 6}

Now we need to analyze the new adjacent nodes to find the shortest path to reach them. We will only analyze the nodes that are adjacent to the nodes that are already part of the shortest path (the path marked with red edges).

Node 3 and node 2 are both adjacent to nodes that are already in the path because they are directly connected to node 1 and node 0, respectively, as you can see below. These are the nodes that we will analyze in the next step.



Since we already have the distance from the source node to node 2 written down in our list, we don't need to update the distance this time. We only need to update the distance from the source node to the new adjacent node (node 3):

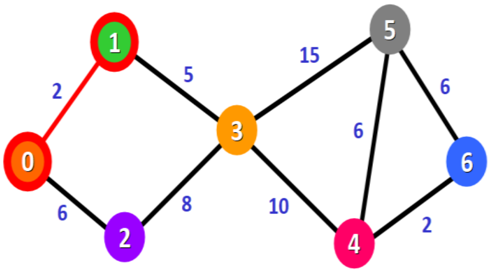


**Distance:**

0: 0
1: ~~∞~~ 2
2: ~~∞~~ 6
3: ~~∞~~ 7
4: ∞
5: ∞
6: ∞

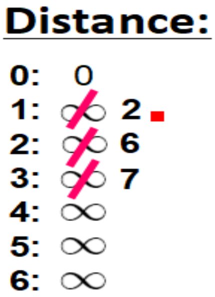
This distance is **7**. Let's see why.
To find the distance from the source node to another node (in this case, node 3), we add the weights of all the edges that form the shortest path to reach that node:

- **For node** 3**:** the total distance is **7** because we add the weights of the edges that form the path 0 -> 1 -> 3 (2  for the edge 0 -> 1 and 5 for the edge 1 -> 3).



Now that we have the distance to the adjacent nodes, we have to choose which node will be added to the path. We must select the **unvisited** node with the shortest (currently known) distance to the source node.

From the list of distances, we can immediately detect that this is node 2 with distance **6**:



**Distance:**

0: 0

1: ~~∞~~ 2 ■

2: ~~∞~~ 6

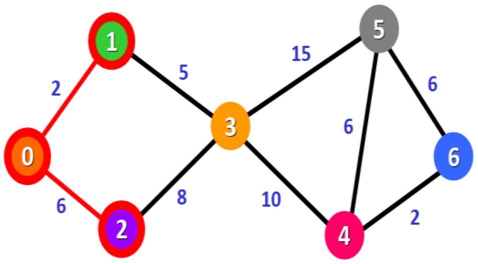
3: ~~∞~~ 7

4: ∞

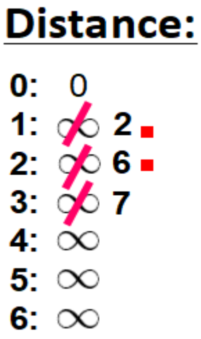
5: ∞

6: ∞

We add it to the path graphically with a red border around the node and a red edge:



We also mark it as visited by adding a small red square in the list of distances and crossing it off from the list of unvisited nodes:



**Distance:**

0: 0

1: ~~∞~~ 2 ■

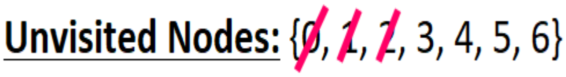
2: ~~∞~~ 6 ■

3: ~~∞~~ 7

4: ∞

5: ∞

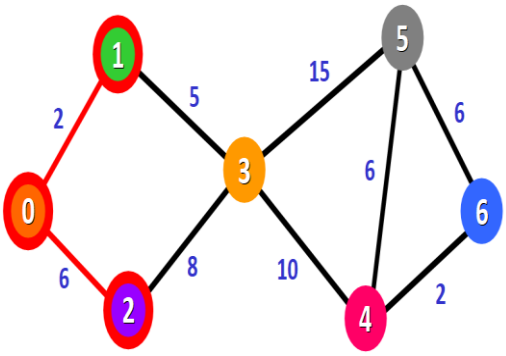
6: ∞



**Unvisited Nodes:** {~~0~~, ~~1~~, ~~2~~, 3, 4, 5, 6}

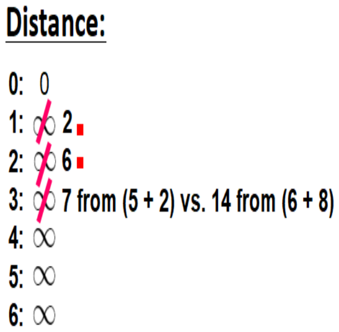
Now we need to repeat the process to find the shortest path from the source node to the new adjacent node, which is node 3.

You can see that we have two possible paths 0 -> 1 -> 3 or 0 -> 2 -> 3. Let's see how we can decide which one is the shortest path.



Node 3 already has a distance in the list that was recorded previously (**7,** see the list below). This distance was the result of a previous step, where we added the weights 5 and 2 of the two edges that we needed to cross to follow the path 0 -> 1 -> 3.

But now we have another alternative. If we choose to follow the path 0 -> 2 -> 3, we would need to follow two edges 0 -> 2 and 2 -> 3 with weights **6** and **8**, respectively, which represents a total distance of **14**.

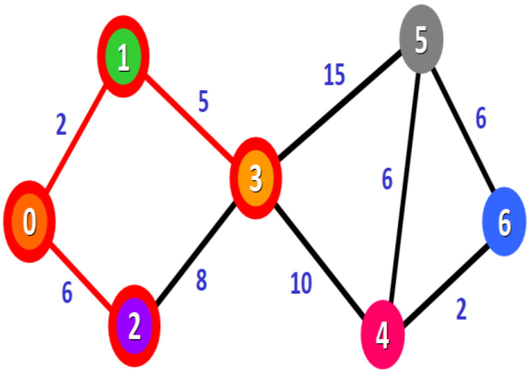


Distance:

0: 0
1: ~~∞~~ 2 ■
2: ~~∞~~ 6 ■
3: ~~∞~~ 7 from (5 + 2) vs. 14 from (6 + 8)
4: ∞
5: ∞
6: ∞

Clearly, the first (existing) distance is shorter (7 vs. 14), so we will choose to keep the original path 0 -> 1 -> 3. **We only update the distance if the new path is shorter.**

Therefore, we add this node to the path using the first alternative: 0 -> 1 -> 3.



We mark this node as visited and cross it off from the list of unvisited nodes:

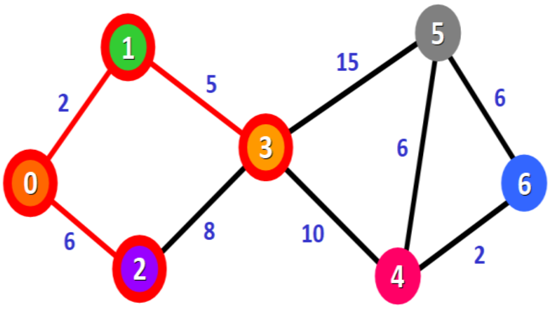
**Distance:**

0: 0<br>
1: ~~∞~~ 2 ■<br>
2: ~~∞~~ 6 ■<br>
3: ~~∞~~ 7 ■<br>
4: ∞<br>
5: ∞<br>
6: ∞

**Unvisited Nodes:** {~~0~~, ~~1~~, ~~2~~, ~~3~~, 4, 5, 6}

Now we repeat the process again.

We need to check the new adjacent nodes that we have not visited so far. This time, these nodes are node 4 and node 5 since they are adjacent to node 3.



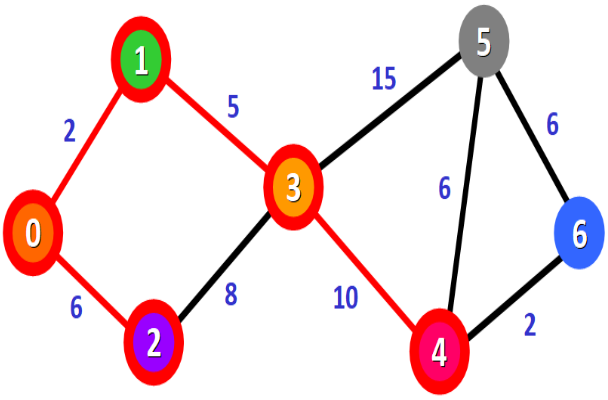
We update the distances of these nodes to the source node, always trying to find a shorter path, if possible:

- **For node** 4**:** the distance is **17** from the path 0 -> 1 -> 3 -> 4.
- **For node** 5**:** the distance is **22** from the path 0 -> 1 -> 3 -> 5.

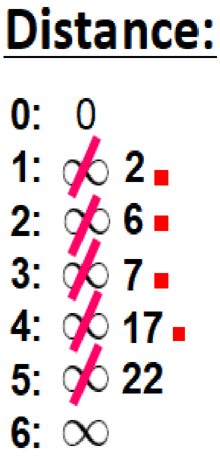
**Distance:**

0: 0<br>
1: ~~∞~~ 2 ■<br>
2: ~~∞~~ 6 ■<br>
3: ~~∞~~ 7 ■<br>
4: ~~∞~~ 17 from (2 + 5 + 10)<br>
5: ~~∞~~ 22 from (2 + 5 + 15)<br>
6: ∞

We need to choose which unvisited node will be marked as visited now. In this case, it's node 4 because it has the shortest distance in the list of distances. We add it graphically in the diagram:



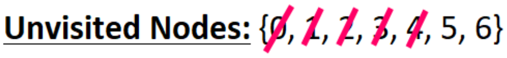
We also mark it as "visited" by adding a small red square in the list:



Distance:

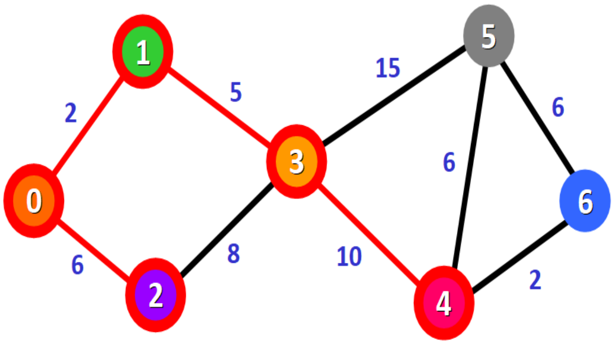
0: 0
1: ~~∞~~ 2 ■
2: ~~∞~~ 6 ■
3: ~~∞~~ 7 ■
4: ~~∞~~ 17 ■
5: ~~∞~~ 22
6: ∞

And we cross it off from the list of unvisited nodes:



Unvisited Nodes: {~~0~~, ~~1~~, ~~2~~, ~~3~~, ~~4~~, 5, 6}

And we repeat the process again. We check the adjacent nodes: node 5 and node 6. We need to analyze each possible path that we can follow to reach them from nodes that have already been marked as visited and added to the path.



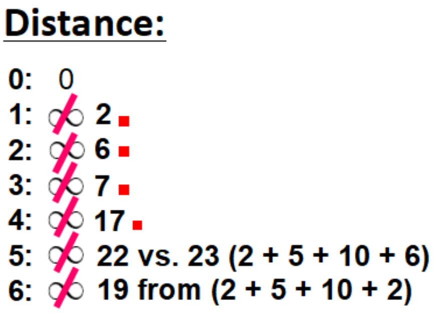
**For node 5:**

- The first option is to follow the path 0 -> 1 -> 3 -> 5, which has a distance of **22** from the source node (2 + 5 + 15). This distance was already recorded in the list of distances in a previous step.
- The second option would be to follow the path 0 -> 1 -> 3 -> 4 -> 5, which has a distance of **23** from the source node (2 + 5 + 10 + 6).

Clearly, the first path is shorter, so we choose it for node 5.

**For node** 6**:**

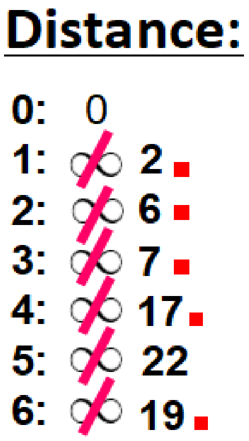
- The path available is 0 -> 1 -> 3 -> 4 -> 6, which has a distance of **19** from the source node (2 + 5 + 10 + 2).



Distance:

0: 0
1: ~~∞~~ 2 ■
2: ~~∞~~ 6 ■
3: ~~∞~~ 7 ■
4: ~~∞~~ 17 ■
5: ~~∞~~ 22 vs. 23 (2 + 5 + 10 + 6)
6: ~~∞~~ 19 from (2 + 5 + 10 + 2)

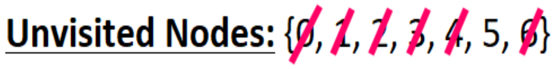
We mark the node with the shortest (currently known) distance as visited. In this case, node 6.



Distance:

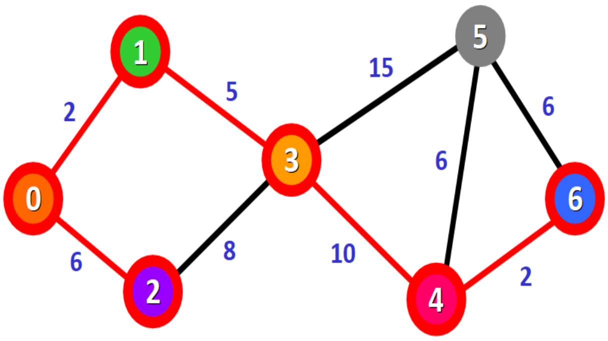
0: 0
1: ~~∞~~ 2 ■
2: ~~∞~~ 6 ■
3: ~~∞~~ 7 ■
4: ~~∞~~ 17 ■
5: ~~∞~~ 22
6: ~~∞~~ 19 ■

And we cross it off from the list of unvisited nodes:



Unvisited Nodes: {~~0~~, ~~1~~, ~~2~~, ~~3~~, ~~4~~, 5, ~~6~~}

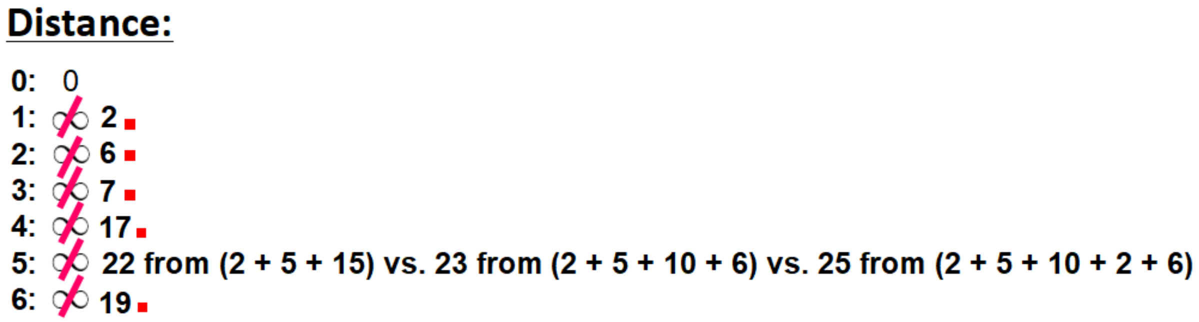
Now we have this path (marked in red):



Only one node has not been visited yet, node 5. Let's see how we can include it in the path.

There are three different paths that we can take to reach node 5 from the nodes that have been added to the path:

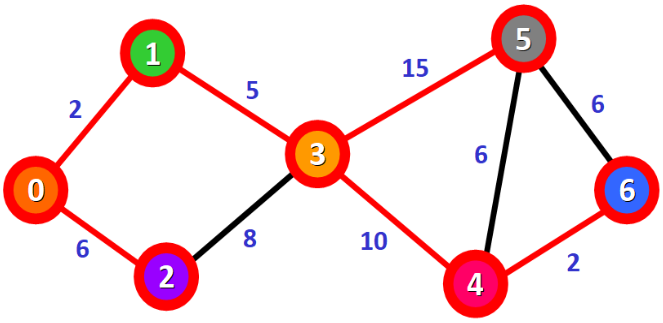
- **Option 1:** 0 -> 1 -> 3 -> 5 with a distance of **22** (2 + 5 + 15).
- **Option 2:** 0 -> 1 -> 3 -> 4 -> 5 with a distance of **23** (2 + 5 + 10 + 6).
- **Option 3:** 0 -> 1 -> 3 -> 4 -> 6 -> 5 with a distance of **25** (2 + 5 + 10 + 2 + 6).



**Distance:**

0: 0
1: ~~∞~~ 2 ■
2: ~~∞~~ 6 ■
3: ~~∞~~ 7 ■
4: ~~∞~~ 17 ■
5: ~~∞~~ 22 from (2 + 5 + 15) vs. 23 from (2 + 5 + 10 + 6) vs. 25 from (2 + 5 + 10 + 2 + 6)
6: ~~∞~~ 19 ■

We select the shortest path: 0 -> 1 -> 3 -> 5 with a distance of **22**.



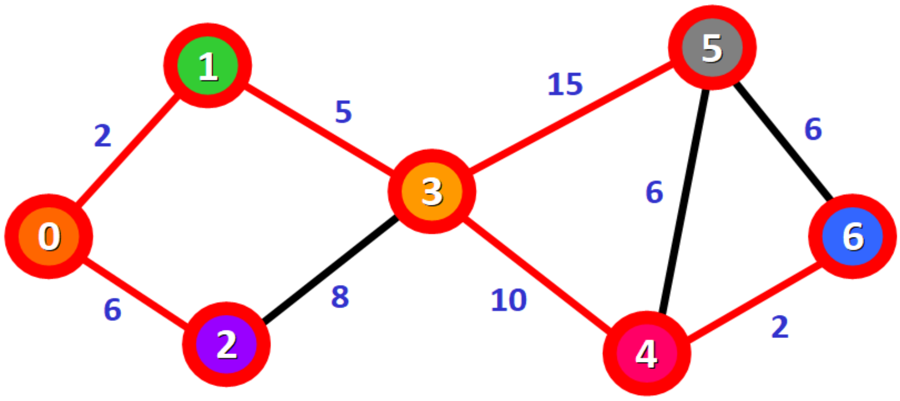
We mark the node as visited and cross it off from the list of unvisited nodes:

**Distance:**

0: 0
1: ~~∞~~ 2 ■
2: ~~∞~~ 6 ■
3: ~~∞~~ 7 ■
4: ~~∞~~ 17 ■
5: ~~∞~~ 22 ■
6: ~~∞~~ 19 ■

**Unvisited Nodes:** {~~0~~, ~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~}

**And voilà!** We have the final result with the shortest path from node 0 to each node in the graph.



In the diagram, the red lines mark the edges that belong to the shortest path. You need to follow these edges to follow the shortest path to reach a given node in the graph starting from node 0.

## 2. Bellman-Ford algorithm

```
Bellman -Ford Algorithm( v ,cost, dist, n)
1.for i=1 to n do
2.{ d[i] = cost[v , i] }
3.for k=2 to n-1 do
4.{ for each 'u' such that u ≠ v & has atleast one in long edge
```

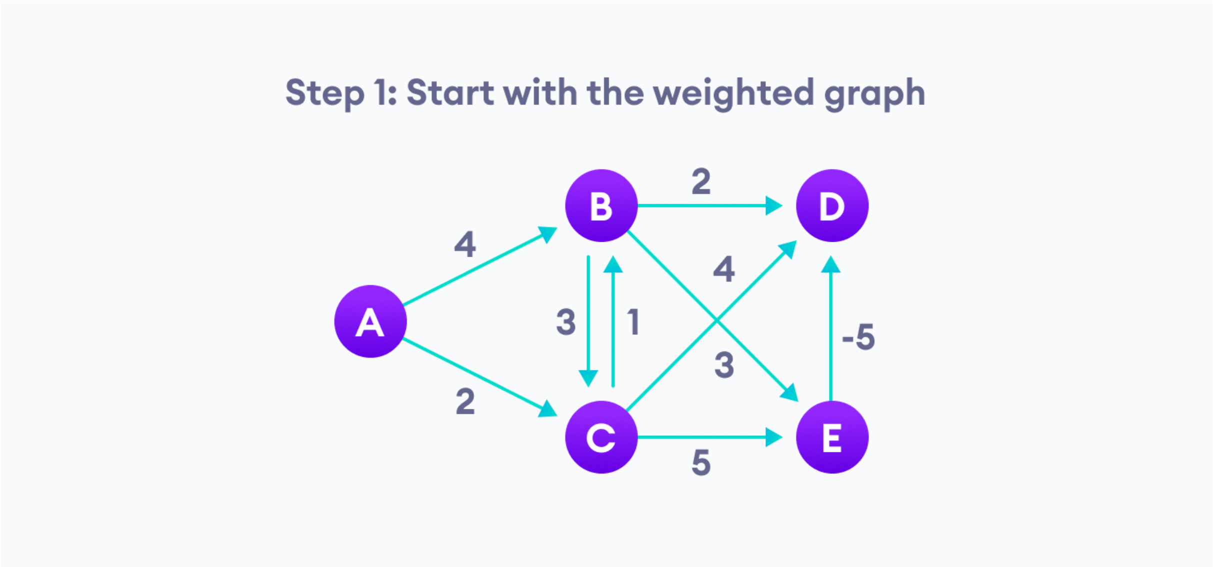
```
do
5. { for each ( i, u) in the graph do
6.{ if ( do[u] > d[i] + cost [i, u]
   Then d[u]=d[i] + cost [i, u]
7.}}}
```

In this Algorithm

n= total no. of vertices<br>
d= distance from source<br>
v= destination node<br>
cost= weight of edge<br>
a = denotes that total no. of intermediate edge i.e. use for SD computation

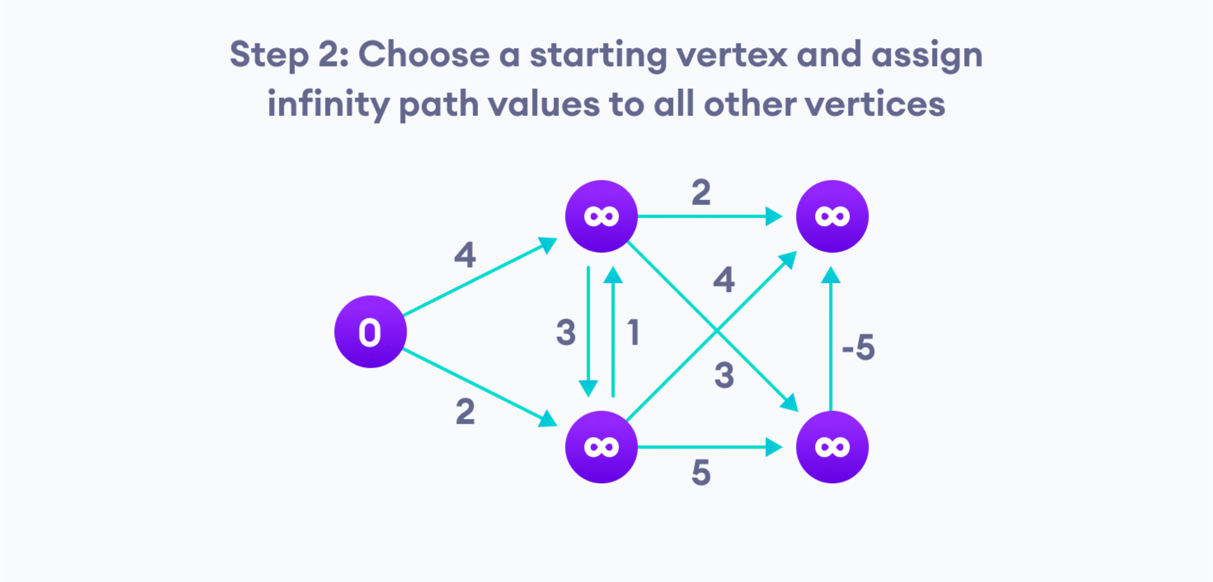
- Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.
  - t is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.
  - if the weighted graph contains the negative weight values, then the Dijkstra algorithm does not confirm whether it produces the correct answer or not.
  - It begins with a starting vertex and calculates the distances between other vertices that a single edge can reach.

By doing this repeatedly for all vertices, we can guarantee that the result is optimized.



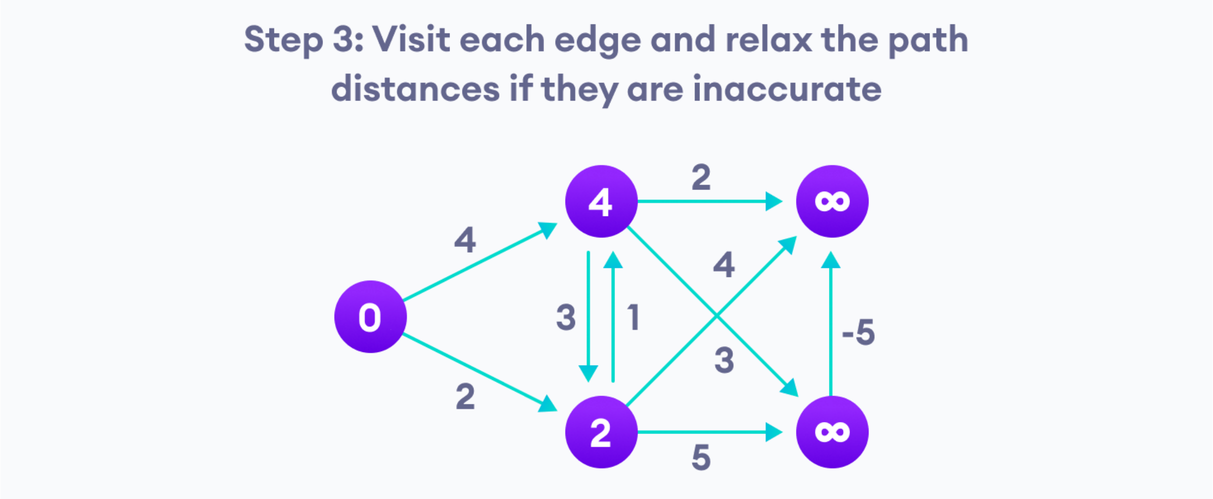
Step 1: Start with the weighted graph

Step-1 for Bellman Ford's algorithm

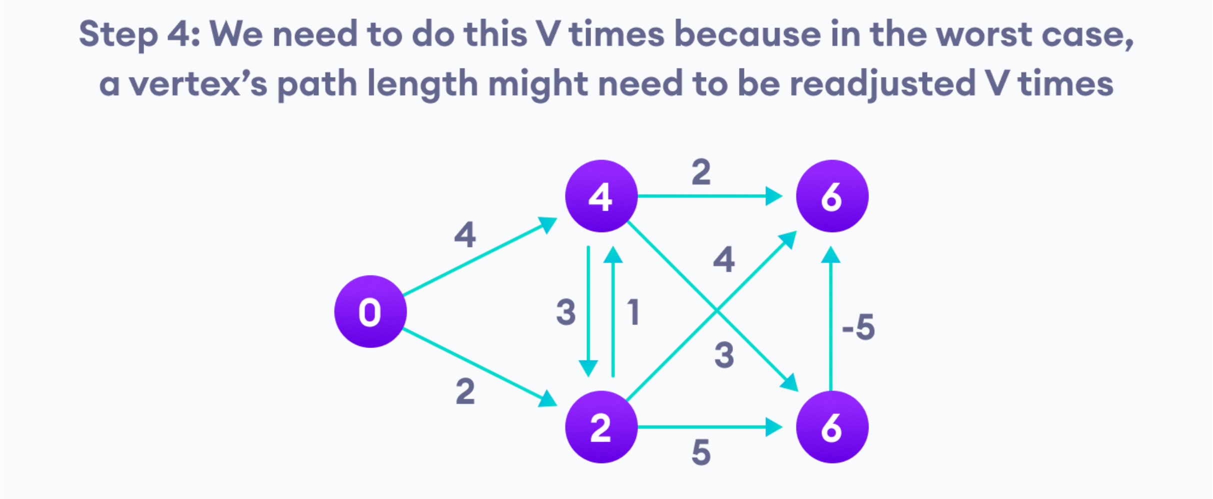


Step 2: Choose a starting vertex and assign infinity path values to all other vertices

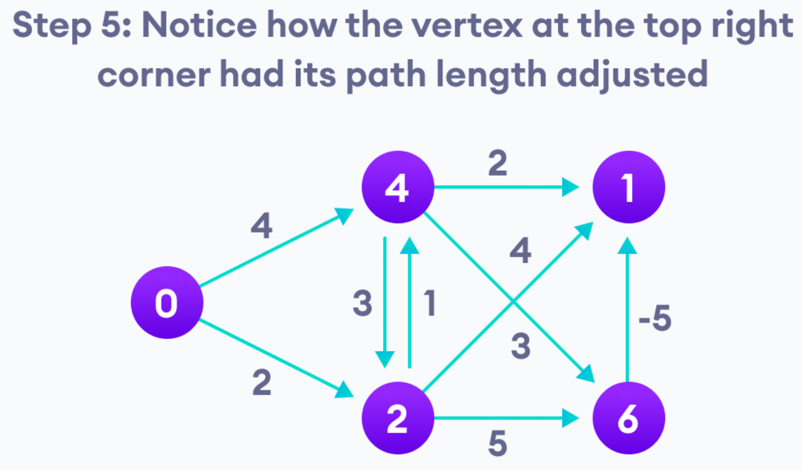
Step-2 for Bellman Ford's algorithm


Step 3: Visit each edge and relax the path distances if they are inaccurate

Step-3 for Bellman Ford's algorithm

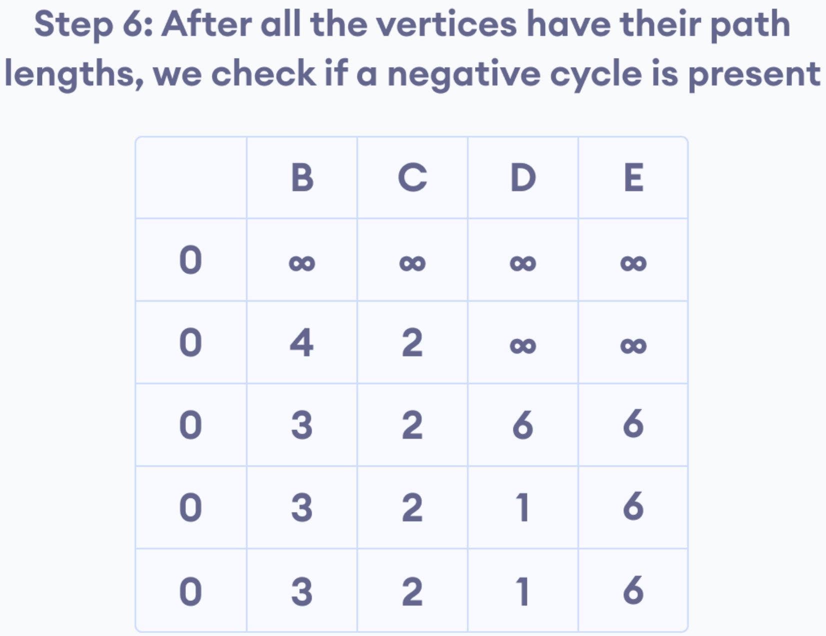

Step 4: We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times

Step-4 for Bellman Ford's algorithm



Step 5: Notice how the vertex at the top right corner had its path length adjusted

Step-5 for Bellman Ford's algorithm



Step 6: After all the vertices have their path lengths, we check if a negative cycle is present

|   | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | 4 | 2 | ∞ | ∞ |
| 0 | 3 | 2 | 6 | 6 |
| 0 | 3 | 2 | 1 | 6 |
| 0 | 3 | 2 | 1 | 6 |

Step-6 for Bellman Ford's algorithm

## Single source shortest path in DAG

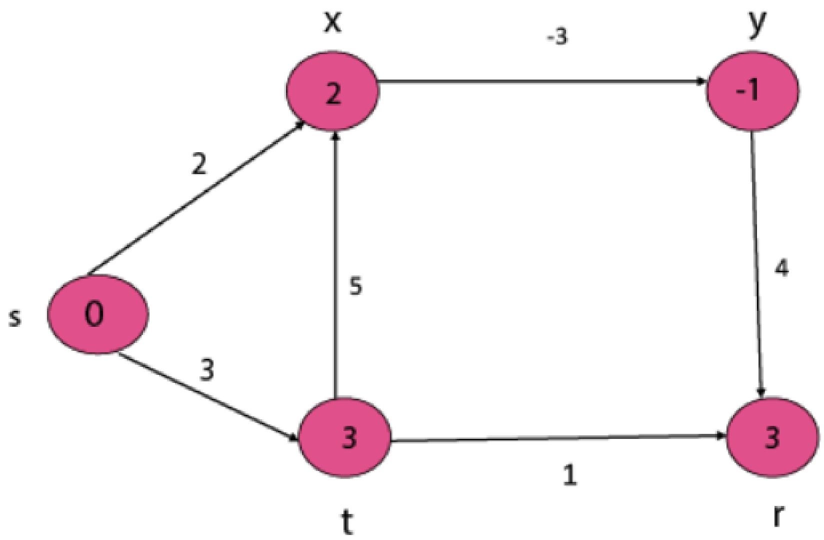
- Single Source shortest path is basically the shortest distance between the source and other vertices in the graph.

- We can find the shortest path from the source to every other vertex by relaxing the edges of the weighted directed acyclic graph **G=(V,E)** according to the topological sort of its vertices.

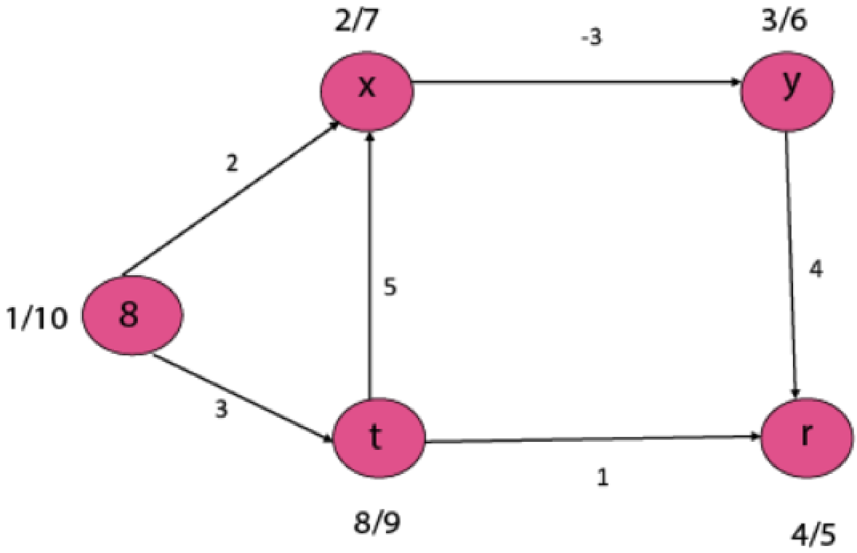
**DAG - SHORTEST - PATHS (G, w, s)**

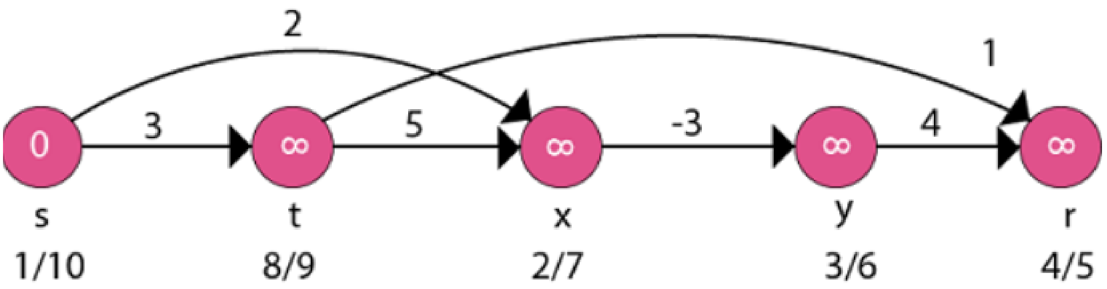
```
1. Topologically sort the vertices of G.
2. INITIALIZE - SINGLE- SOURCE (G, s)
3. for each vertex u taken in topologically sorted order
4. do for each vertex v ∈ Adj [u]
5. do RELAX (u, v, w)
```

**Example:**



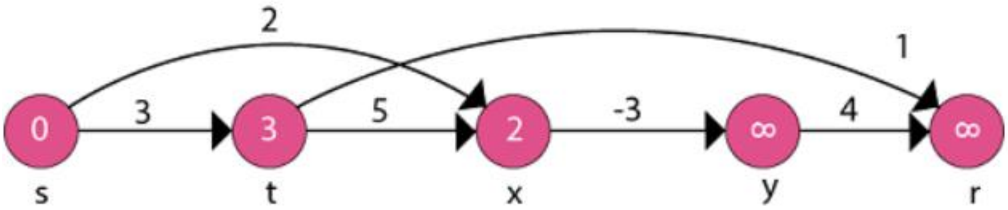
**Step1:** To topologically sort vertices apply **DFS (Depth First Search)** and then arrange vertices in linear order by decreasing order of finish time.





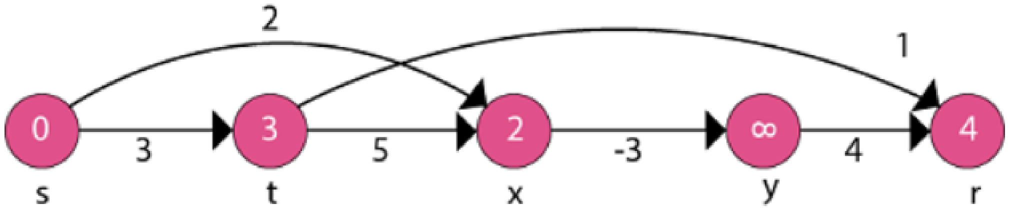
Initialize Single Source

Now, take each vertex in topologically sorted order and relax each edge

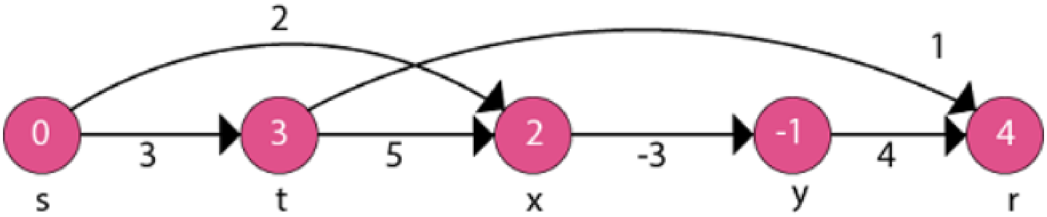


1. adj [s] →t, x
2. 0 + 3 < ∞
3. d [t] ← 3
4. 0 + 2 < ∞

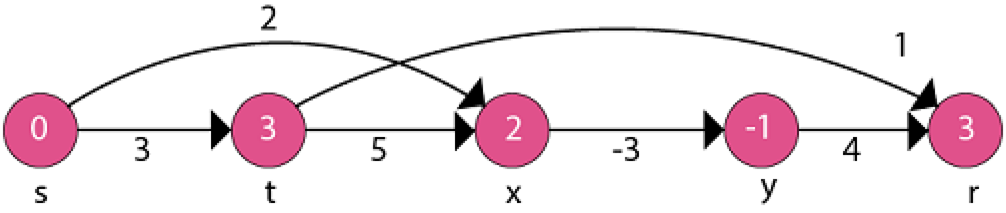
5. d [x] ← 2



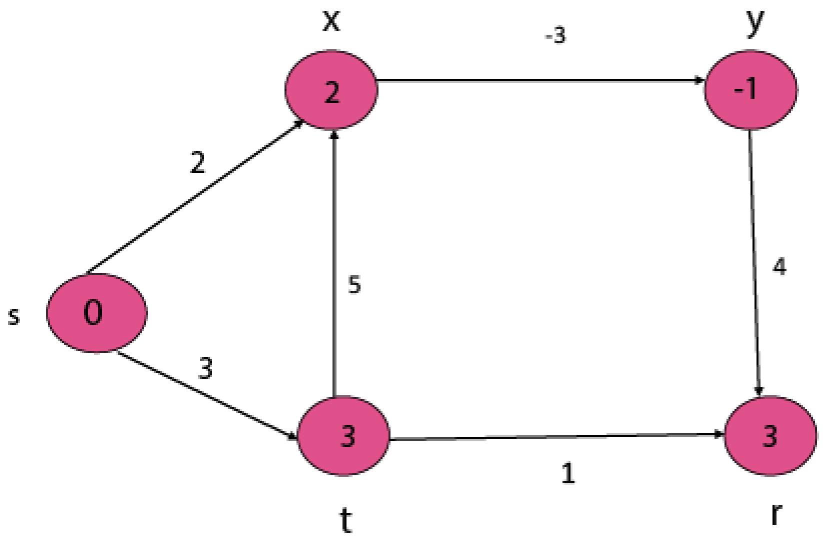
1. adj [t] ⟶ r, x
2. 3 + 1 < ∞
3. d [r] ← 4
4. 3 + 5 ≤ 2



1. adj [x] ⟶ y
2. 2 - 3 < ∞
3. d [y] ← -1



1. adj [y] ⟶ r
2. -1 + 4 < 4
3. 3 <4
4. d [r] ← 3



**Thus the Shortest Path is:**

1. s to x is 2
2. s to y is -1
3. s to t is 3
4. s to r is 3

## Computational complexity

In computer science, the **computational complexity** or simply **complexity** of an algorithm is the amount of resources required to run it.

Particular focus is given to time and memory requirements.

The complexity of a problem is the complexity of the best algorithms that allow solving the problem

There are lots of variants of this bit that we are generally looking at when we are doing any computer programming or in general or in most practical purposes are just two main complexities, one is Time Complexity, and the other is Space (memory) Complexity.

Time complexity is simple as how fast your code runs, how much time it will take, depends on the number of steps

**Example of Complexity in Time (execution) and Space (memory) factors :**

**Example-1 : More Complex**

```
i = 1;                    1s
while( i <= 10 )              11s
{
    a = 5;                  10s
    result = i * a;           10s
    printf(“\n” /d”, result);    10s
    i++;                    10s
}
```

Here, we assume each variable is equal to **2 Bytes**. In the above example we use three variables (i, a, result) which is **6 Bytes.**

**Execution Time :** 52s

**Memory (Space) :** 6 Bytes

**Example-2 : Less Complex**

```
a = 5;                    1s
```

```
i = 1;                        1s
while( i<=10)                   11s
{
    result = i * a;             10s
    printf("\n" /d", result);    10s
    i++;                        10s
}
```

**Execution Time :** 43s
**Memory (Space) :** 6 Bytes

**Definition of Polynomial time:** - If we produce an output according to the given input within a specific amount of time such as within a minute, hours. This is known as Polynomial time.

**Definition of Non-Polynomial time:** - If we produce an output according to the given input but there are no time constraints is known as Non-Polynomial time. But yes output will produce but time is not fixed yet.

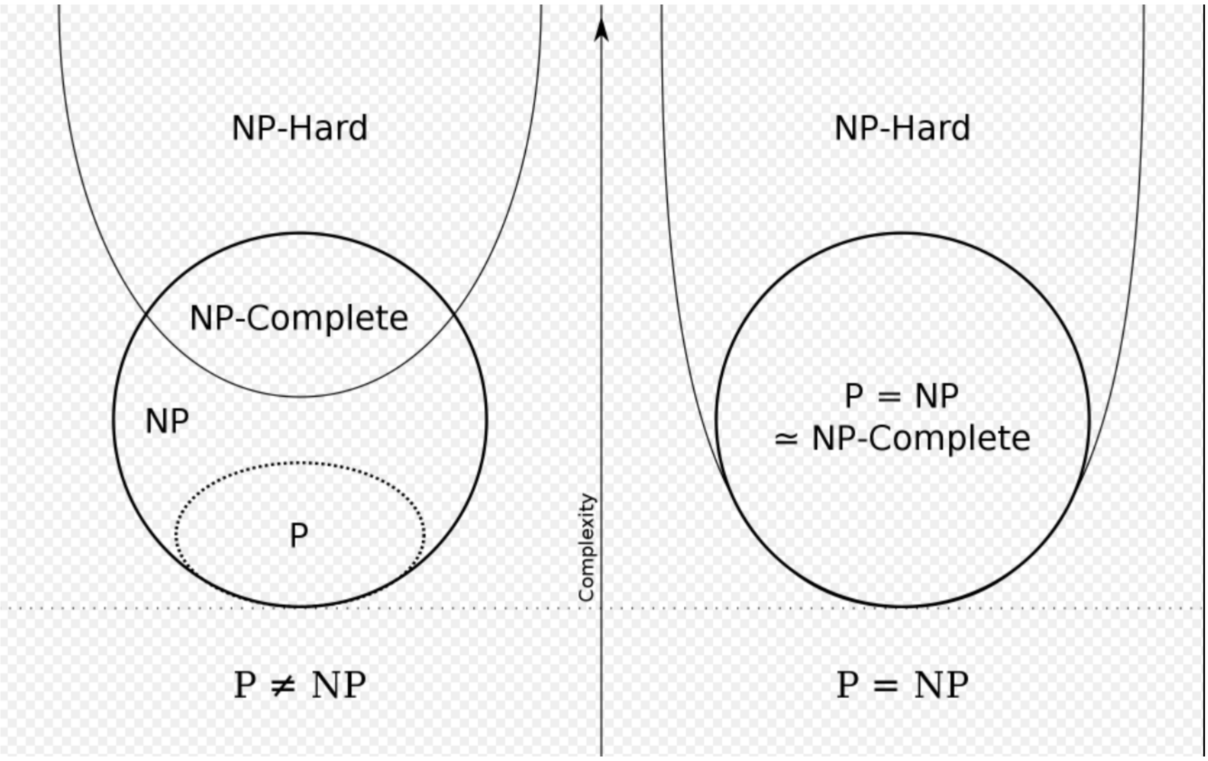
## NP -hard:

An NP-hard problem is at least as hard as the hardest problem in NP and it is the class of the problems such that every problem in NP reduces to NP-hard.

## NP- Complete

**NP-complete problem**, any of a class of computational problems for which no efficient solution algorithm has been found.

Many significant computer-science problems belong to this class—e.g., the traveling salesman problem, satisfiability problems, and graph-covering problems.



# UNIT-4

## Network and Sorting Algorithms

Flow and Sorting Networks Flow networks, Ford- Fulkerson method, Maximum Bipartite matching, Sorting Networks, Comparison network, The zero- One principle, Bitonic sorting network, Merging networks

## Flow Network:

Flow Network is a directed graph that is used for modeling material Flow.

There are two different vertices; one is a **source** which produces material at some steady rate, and another one is sink which consumes the content at the same constant speed.

**Definition:** A Flow Network is a directed graph G = (V, E) such that

1. For each edge (u, v) ∈ E, we associate a nonnegative weight capacity c (u, v) ≥ 0.If (u, v) ∉ E, we assume that c (u, v) = 0.

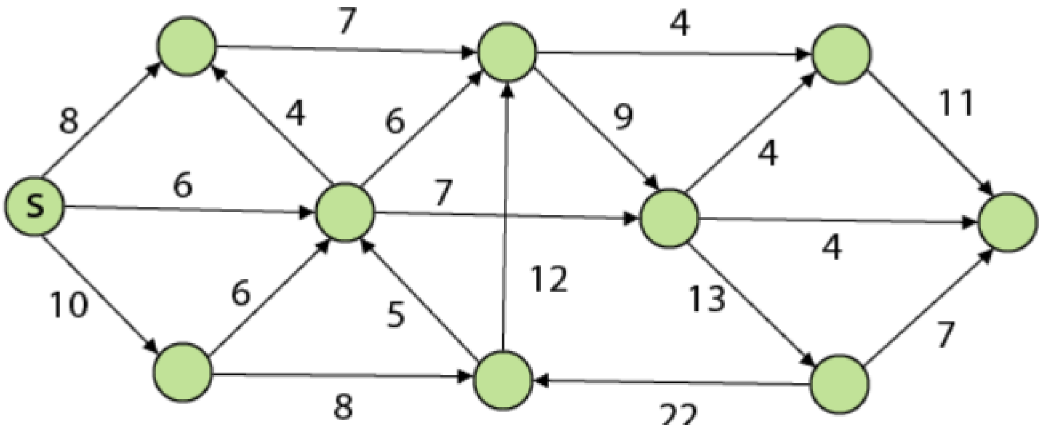
2. There are two distinguishing points, the source s, and the sink t;
3. For every vertex v ∈ V, there is a path from s to t containing v.

Let G = (V, E) be a flow network. Let s be the source of the network, and let t be the sink. A flow in G is a real-valued function f: V x V→R such that the following properties hold:

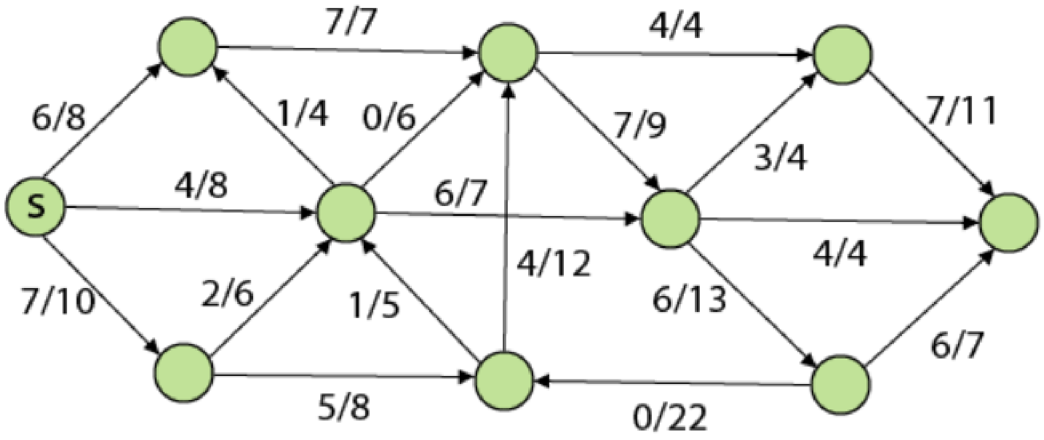
- **Capacity Constraint:** For all u, v ∈ V, we need f (u, v) ≤ c (u, v).
- **Skew Symmetry:** For all u, v ∈ V, we need f (u, v) = - f (u, v).
- **Flow Conservation:** For all u ∈ V- {s, t}, we need

$$\boxed{\sum_{v \in V} f(u,v) = \sum_{u \in V} f(u,v) = 0}$$

The quantity f (u, v), which can be positive or negative, is known as the net flow from vertex u to vertex v. In the **maximum-flow problem**, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value from s to t.



(a)



(b)

The value of the flow is the net flow from the source,

$$|f| = \sum_{v \in V} f(s, v)$$

The **positive net flow entering** a vertex v is described by

$$\sum_{\{u \in V : f(u,v) > 0\}} f(u, v)$$

The **positive net flow** leaving a vertex is described symmetrically. One interpretation of the Flow-Conservation Property is that the positive net flow entering a vertex other than the source or sink must equal the positive net flow leaving the vertex.

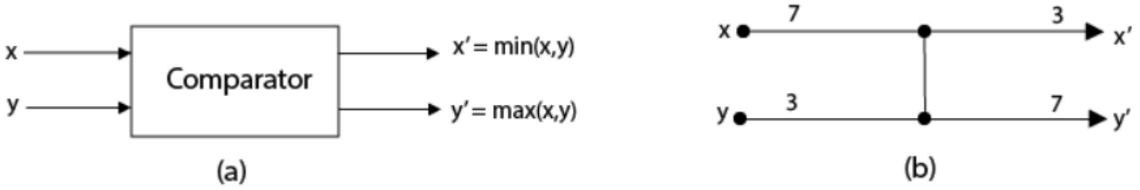
A flow f is said to be **integer-valued** if f (u, v) is an integer for all (u, v) ∈ E. Clearly, the value of the flow is an integer is an integer-valued flow.

## Comparison Networks

**Comparison Networks** are a type of sorting networks which always sort their inputs. Wires and comparator comprise comparison network. A *Comparator* is a device which has two inputs (x, y) and outputs (x', y'). It performs the following function:

x' = min (x, y),

y' = max (x, y)

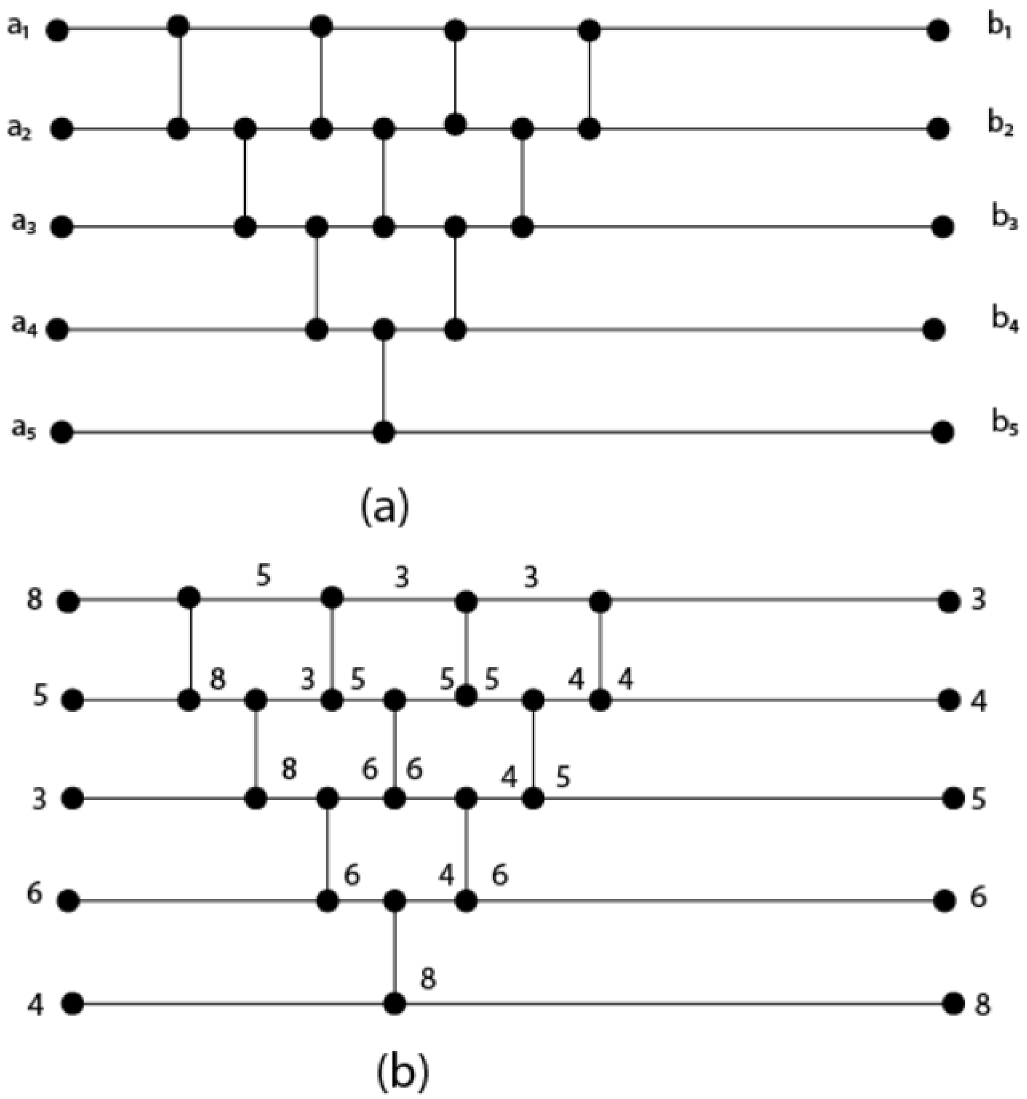


Comparison Network is a set of comparators interconnected by wires. Running time of comparator can define regarding **depth**.

**Depth of a Wire:** An input wire of a comparison network has depth 0. Now, if a comparator has two input wires with depths $d_x$ and $d_y$' then its output wires have depth max $(d_x,d_y) + 1$.

A sorting network is a comparison network for which the output sequence is monotonically increasing (that is $b_1 \leq b_2 \leq$ ....$b_n$) for every input sequence.

**Fig: A Sorting network based on Insertion Sort**



## Network Flow Problems

The most obvious flow network problem is the following:

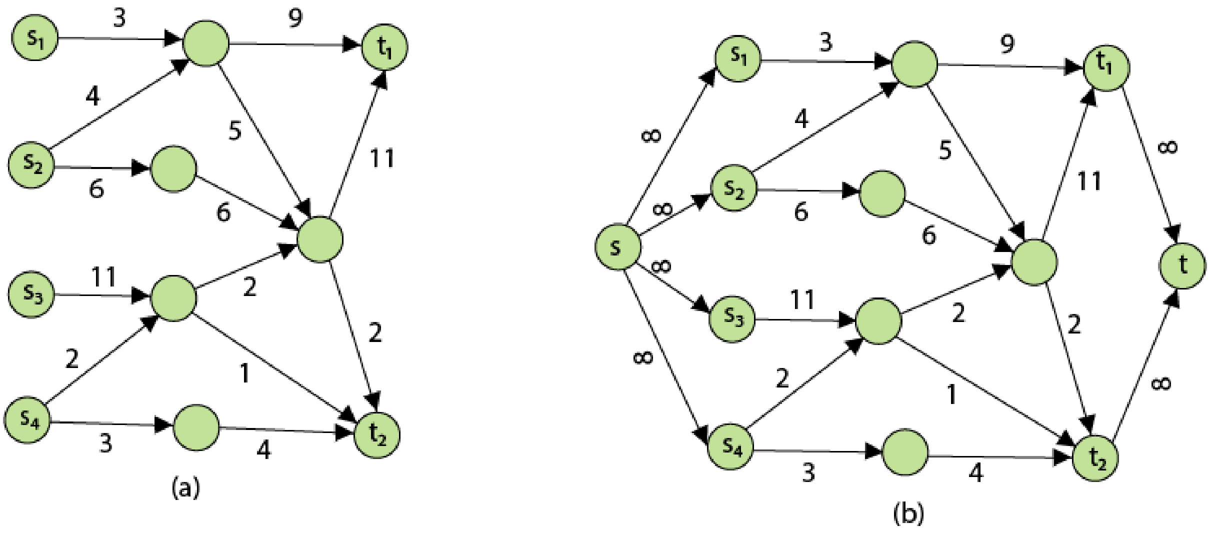
**Problem1:** Given a flow network G = (V, E), the maximum flow problem is to find a flow with maximum value.

**Problem 2:** The multiple source and sink maximum flow problem is similar to the maximum flow problem, except there is a set $\{s_1,s_2,s_3.......s_n\}$ of sources and a set $\{t_1,t_2,t_3..........t_n\}$ of sinks.

Fortunately, this problem is no solid than regular maximum flow. Given multiple sources and sink flow network G, we define a new flow network G' by adding

- A super source s,
- A super sink t,
- For each $s_i$, add edge (s, $s_i$) with capacity ∞, and
- For each $t_i$,add edge ($t_i$,t) with capacity ∞

Figure shows a multiple sources and sinks flow network and an equivalent single source and sink flow network



(a)

(b)

**Residual Networks:** The Residual Network consists of an edge that can admit more net flow. Suppose we have a flow network G = (V, E) with source s and sink t. Let f be a flow in G, and examine a pair of vertices u, v ∈ V. The sum of additional net flow we can push from u to v before exceeding the capacity c (u, v) is the residual capacity of (u, v) given by

$$c_f(u, v) = c\ (u, v) - \ f\ (u, v).$$

When the net flow f (u, v) is negative, the residual capacity $c_f$ (u,v) is greater than the capacity c (u, v).

**For Example:** if c (u, v) = 16 and f (u, v) =16 and f (u, v) = -4, then the residual capacity $c_f$ (u,v) is 20.

Given a flow network G = (V, E) and a flow f, the residual network of G induced by f is $G_f$ = (V, $E_f$), where

$$E_f = \{(u, v) \in V \times V: C_f(u, v) \geq 0\}$$

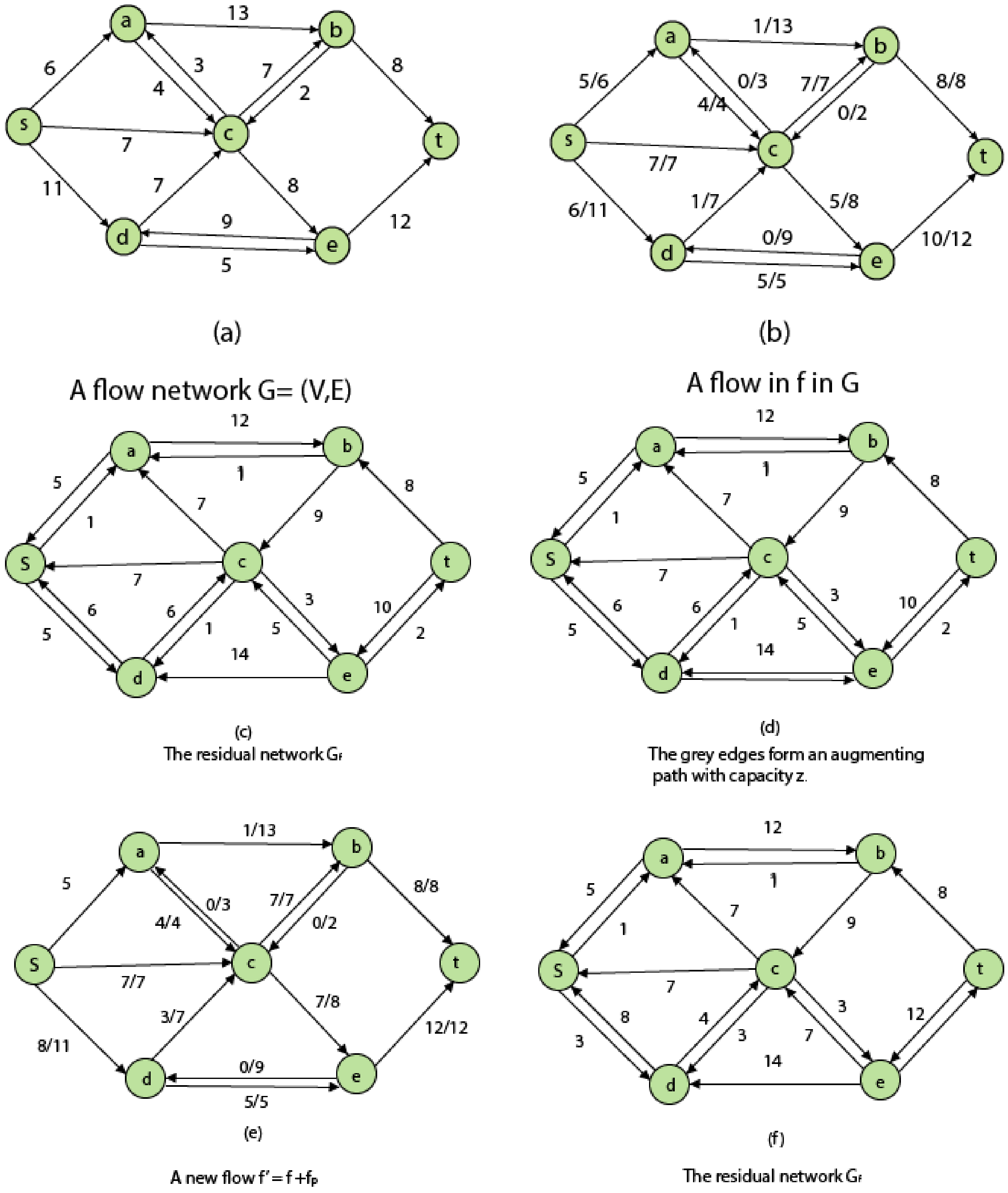
That is, each edge of the residual network, or residual edge, can admit a strictly positive net flow.

**Augmenting Path:** Given a flow network G = (V, E) and a flow f, an **augmenting path** p is a simple path from s to t in the residual network$G_f$. By the solution of the residual network, each edge (u, v) on an augmenting path admits some additional positive net flow from u to v without violating the capacity constraint on the edge.

Let G = (V, E) be a flow network with flow f. The **residual capacity** of an augmenting path p is

$$C_f(p) = \min\ \{C_f(u, v): (u, v) \text{is on p}\}$$

The residual capacity is the maximal amount of flow that can be pushed through the augmenting path. If there is an augmenting path, then each edge on it has a positive capacity. We will use this fact to compute a maximum flow in a flow network.



(a)

A flow network G= (V,E)

(b)

A flow in f in G

(c)
The residual network $G_f$

(d)
The grey edges form an augmenting path with capacity z.

(e)

A new flow $f' = f + f_p$

(f)

The residual network $G_f$

## Ford – Fulkerson Algorithm

The **Ford–Fulkerson method** or **Ford–Fulkerson algorithm** (**FFA**) is a greedy algorithm that computes the maximum flow in a flow network.

A term, **flow network**, is used to describe a network of vertices and edges with a source (S) and a sink (T). Each vertex, except **S** and **T**, can receive and send an equal amount of stuff through it.

Algorithm of Ford Fulkerson

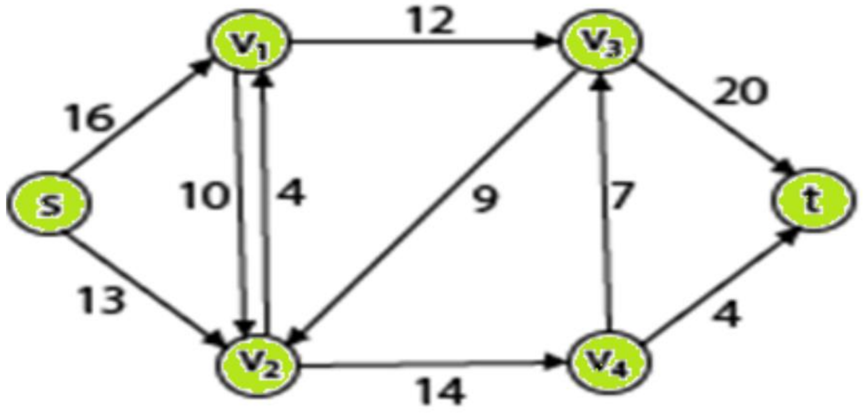
### FORD-FULKERSON METHOD (G, s, t)

```
Initialize flow f to 0
while there exists an augmenting path p
do argument flow f along p
Return f
```

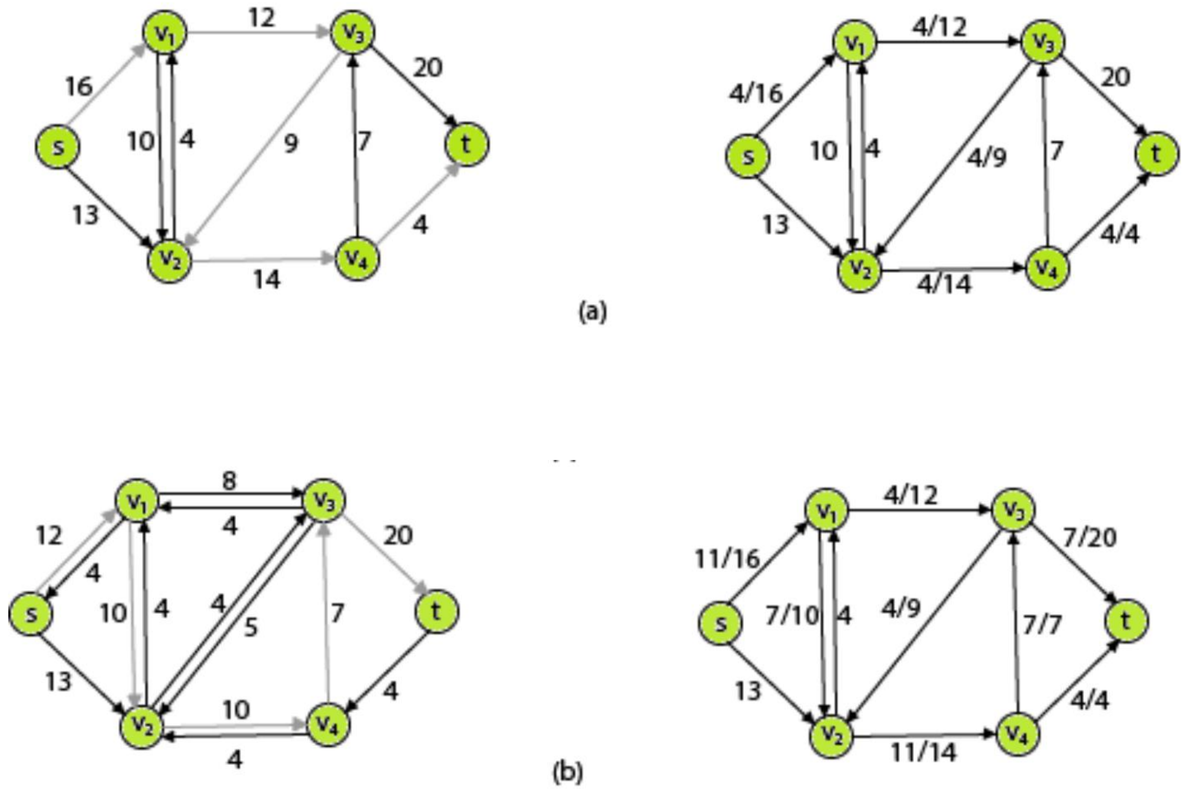
### FORD-FULKERSON (G, s, t)

```
for each edge (u, v) ∈ E [G]
do f [u, v] ← 0
f [u, v] ← 0
while there exists a path p from s to t in the residual network $G_f$.
do $c_f$ (p)←min? { $C_f$ (u ,v):(u ,v)is on p}
for each edge (u, v) in p
do f [u, v] ← f [u, v] + $c_f$ (p)
f [u, v] ←f[ u ,v]
```

**Example:** Each Directed Edge is labeled with capacity. Use the Ford-Fulkerson algorithm to find the maximum flow.

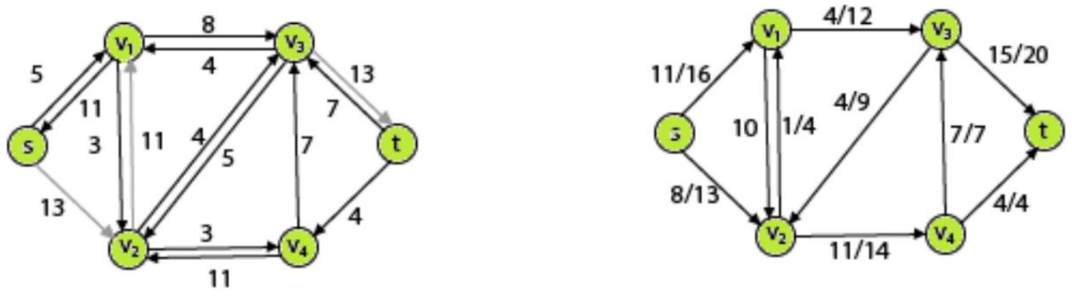


**Solution:** The left side of each part shows the residual network G$_f$ with a shaded augmenting path p,and the right side of each part shows the net flow f.



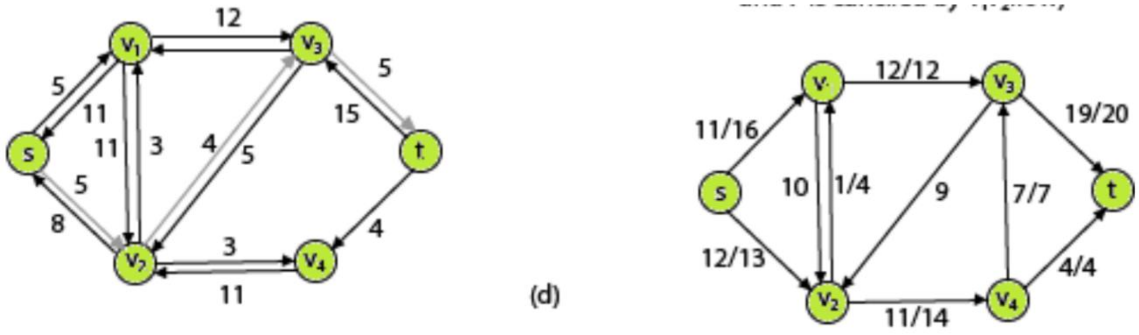
(a)

(b)

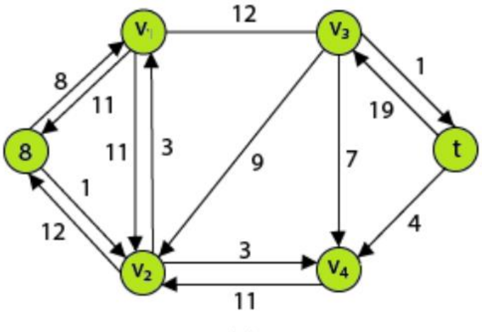


(c)

(In this , 8 is break into 7 and 1 and 7 is canclled by $v_1v_2$flow)



(d)

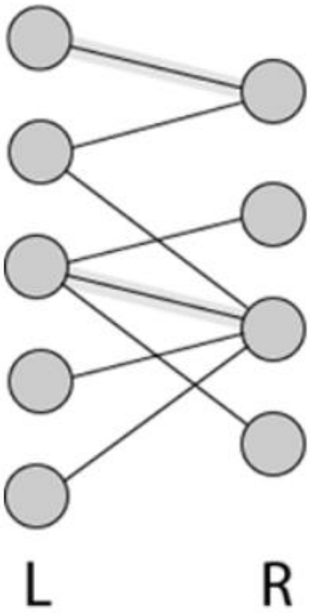


(e)

Now , it has no augmenting paths .So, the maximum flow shown in (d) is 23 is a maximum flow .

# Maximum Bipartite matching

A Bipartite Graph is a graph whose vertices can be divided into two independent sets L and R such that every edge (u, v) either connect a vertex from L to R or a vertex from R to L. In other words, for every edge (u, v) either u ∈ L and v ∈ L. We can also say that no edge exists that connect vertices of the same set.



(a)

Matching is a Bipartite Graph is a set of edges chosen in such a way that no two edges share an endpoint. Given an undirected Graph G = (V, E), a Matching is a subset of edge M ⊆ E such that for all vertices v ∈ V, at most one edge of M is incident on v.

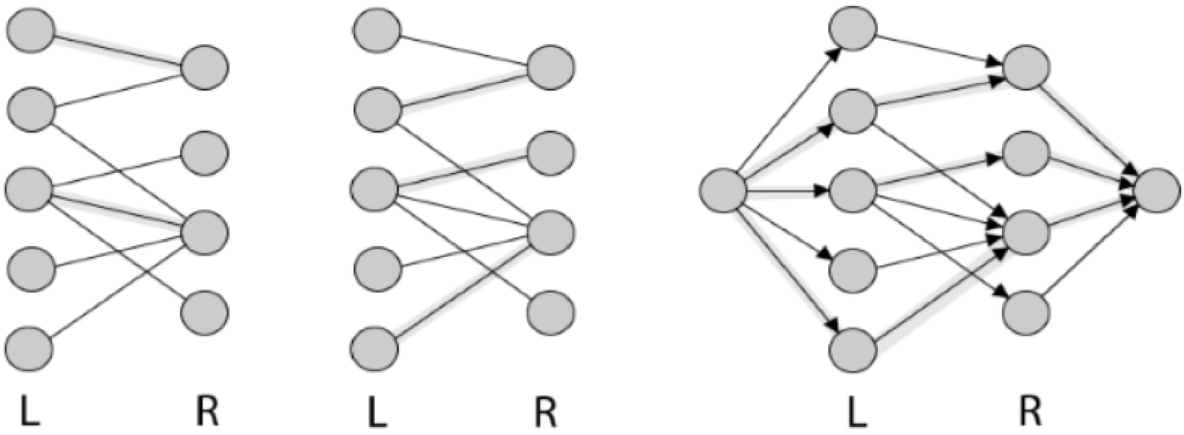
A Maximum matching is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have|M|>|M' |.

## Finding a maximum bipartite matching

We can use the Ford-Fulkerson method to find a maximum matching in an undirected bipartite graph G= (V, E) in time polynomial in |V| and |E|. The trick is to construct a flow network G= (V',E') for the bipartite graph G as follows. We let the source s and sink t be new vertices not in V, and we let V'=V ∪{s,t}.If the vertex partition of G is V = L∪R, the directed edges of G' are the edges of E, directed from L to R, along with |V| new directed edges:

$$\mathbf{E}' = \{(\mathbf{s},\mathbf{u}) : \mathbf{u} \in \mathbf{L}\} \cup \{(\mathbf{u},\mathbf{v}) : (\mathbf{u},\mathbf{v}) \in \mathbf{E}\} \cup \{(\mathbf{v},\mathbf{t}) : \mathbf{v} \in \mathbf{R}\}$$

**Fig: A Bipartite Graph G = (V, E) with vertex partition V = L ∪ R.**



(a) Matching with Cardinality 2.

(b) Matching with Cardinality 3.

(c) The corresponding flow network G′ with a maximum flow shown. Each edge has unit Capacity. The shaded edges from L to R correspond to those in the maximum matching from (b).

# Algorithm of Maximum Bipartite matching:

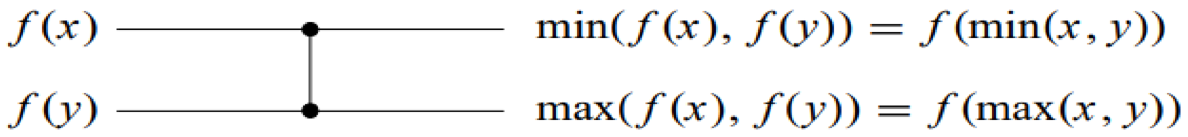
```
bool kuhn(vertex v) {
  if (used[v]) return false;
  used[v] = true;
  for (vertex q adjacent to v) {
    if ((q has no pair) or kuhn(pairs[q])) {
      pairs[q] = v;
      return true;
    }
  }
}
find_max_matching {
  for (vertex v = {
      1,
      ..,
      n
    }) {
    used = {
      0
    };
    kuhn(v);
  }
}
```

## The zero -one principal

The zero-one principle says that if a sorting network works correctly when each input is drawn from the set {0, 1}, then it works correctly on arbitrary input numbers. (The numbers can be integers, reals, or, in general, any set of values from any linearly ordered set.) As we construct sorting networks and other comparison networks, the zero-one principle will allow us to focus on their operation for input sequences consisting solely of 0's and 1's. Once we have constructed a sorting network and proved that it can sort all zero-one sequences, we shall appeal to the zero-one principle to show that it properly sorts sequences of arbitrary values. The proof of the zero-one principle relies on the notion of a monotonically increasing function.

If a comparison network transforms the input sequence a = ha1, a2, . . . , an into the output sequence b = hb1, b2, . . . , bn, then for any monotonically increasing function f , the network transforms the input sequence f (a) = h f (a1), f (a2), . . . , f (an) into the output sequence f (b) = h f (b1), f (b2), . . . , f (bn).

Proof We shall first prove the claim that if f is a monotonically increasing function, then a single comparator with inputs f (x) and f (y) produces outputs f (min(x, y)) and f (max(x, y)). We then use induction to prove the lemma.


$f(x)$
$f(y)$
$\min(f(x), f(y)) = f(\min(x, y))$
$\max(f(x), f(y)) = f(\max(x, y))$

Theorem (Zero-one principle) If a comparison network with n inputs sorts all 2n possible sequences of 0's and 1's correctly, then it sorts all sequences of arbitrary numbers correctly.

## Proof

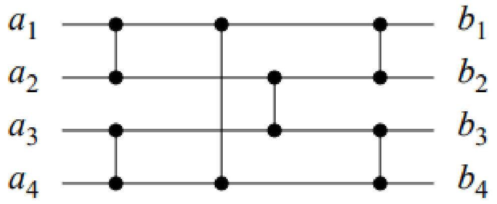
Suppose for the purpose of contradiction that the network sorts all zero-one sequences, but there exists a sequence of arbitrary numbers that the network does not correctly sort. That is, there exists an input sequence ha1,a2,. . . ,ani containing elements ai and aj such that ai < aj , but the network places aj before ai in the output sequence. We define a monotonically increasing function f as

$$f(x) = \begin{cases} 0 & \text{if } x \leq a_i \ , \\ 1 & \text{if } x > a_i \ . \end{cases}$$

Since the network places aj before ai in the output sequence when ha1, a2, . . . , ani is input, that it places f (aj) before f (ai) in the output sequence when h f (a1), f (a2), . . . , f (an) is input. But since f (aj) = 1 and f (ai) = 0, we obtain the contradiction that the network fails to sort the zero-one sequence h f (a1), f (a2), . . . , f (an) correctly

Exercises: Prove that applying a monotonically increasing function to a sorted sequence produces a sorted sequence.

Prove that a comparison network with n inputs correctly sorts the input sequence (n, n − 1, . . . ,1) if and only if it correctly sorts the n − 1 zero-one sequences <1, 0, 0, . . . ,0, 0>, <1, 1, 0, . . . ,0, 0>, . . ., <1, 1, 1, . . . ,1, 0>.



A sorting network for sorting 4 numbers

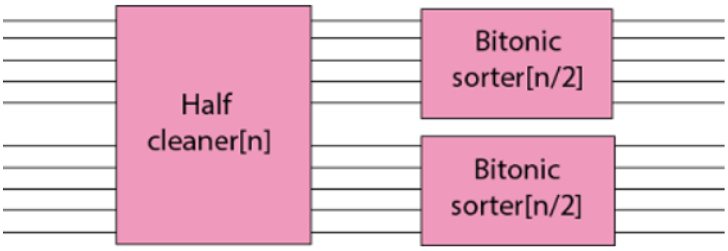
### Bitonic Sorting Network

A sequence that monotonically increases and then monotonically decreases, or else monotonically decreases and then monotonically increases is called a bitonic sequence. For example: the sequence (2, 5, 6, 9, 3, 1) and (8, 7, 5, 2, 4, 6) are both bitonic. The bitonic sorter is a comparison network that sorts bitonic sequence of 0's and 1's.

**Half-Cleaner:** A bitonic sorter is containing several stages, each of which is called a half-cleaner. Each half-cleaner is a comparison network of depth 1 in which input line i is compared with line 1+ $\frac{n}{2}$ for i = 1, 2..... $\frac{n}{2}$.

**Bitonic Sorter:** By recursively connecting half-cleaners, we can build a bitonic sorter, which is a network that sorts bitonic sequences. The first stage of BITONIC-SORTER [n] consists of HALF-CLEANER [n], which produces two bitonic sequences of half the size such that every element in the top half is at least as small as each element in the bottom half. Thus, we can complete the sort by utilizing two copies of BITONIC-SORTER [n/2] to sort the two halves recursively.

Fig: The depth D (n) of BITONIC-SORTER [n] is given by recurrence whose solution is D (n) = log n.



## Bitonic Sort Algorithm

Bitonic sort is a parallel sorting algorithm that performs O($n^2$log n) comparisons.

Although the number of comparisons is more than that in any other popular sorting algorithm, it performs better for the parallel implementation because elements are compared in a predefined sequence that must not depend upon the data being sorted. The predefined sequence is called the Bitonic sequence.

To understand the bitonic sort, we first have to understand the **Bitonic sequence**.

In **Bitonic sequence,** elements are first arranged in increasing order, and then after some particular index, they start decreasing.

An array with A[0…i…n-1] is said to be bitonic, if there is an index i, such that

1. A[0] < A[1] < A[2] .... A[i1] < A[i] > A[i+1] > A[i+2] > A[i+3] > ... >A[ n-1]

Where, 0 ≤ i ≤ n-1.

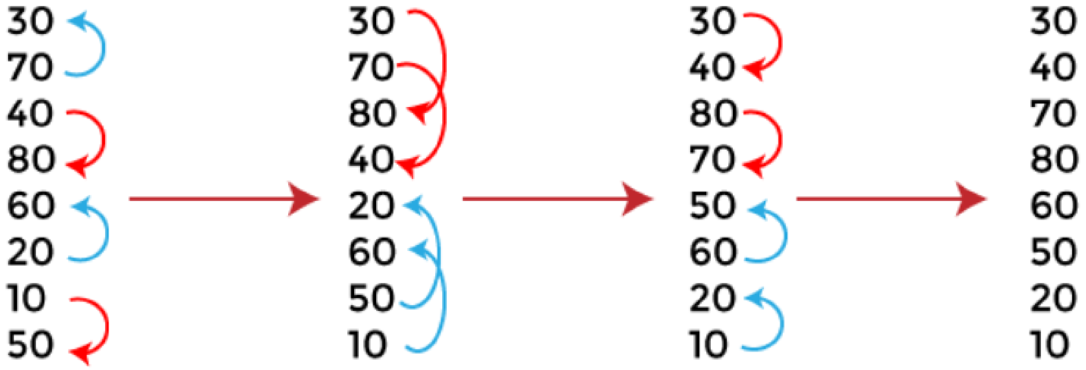
Before moving directly towards the algorithm of bitonic sort, first, understand the conversion of any random sequence into a bitonic sequence.

## How to convert the random sequence into a bitonic sequence?

Consider a sequence A[ 0 ... n-1] of n elements. First, start constructing a Bitonic sequence by using 4 elements of the sequence. Sort the first 2 elements in ascending order and the last 2 elements in descending order, concatenate this pair to form a Bitonic sequence of 4 elements. Repeat this process for the remaining pairs of the element until we find a Bitonic sequence.

Let's understand the process to convert the random sequence into a bitonic sequence using an example.

Suppose the elements of array are - **{30, 70, 40, 80, 60, 20, 10, 50}**



After conversion, the bitonic sequence that we will get is -

**30, 40, 70, 80, 60, 50, 20, 10**

Now, move towards the steps of performing the bitonic sort.

## Merging Network

Merging Network is the network that can join two sorted input sequences into one sorted output sequence. We adapt BITONIC-SORTER [n] to create the merging network MERGER [n].

The merging network is based on the following assumption:

Given two sorted sequences, if we reverse the order of the second sequence and then connect the two sequences, the resulting sequence is bitonic.

For Example: Given two sorted zero-one sequences X = 00000111 and Y =00001111, we reverse Y to get $Y^R$ = 11110000.

he sorting network SORTER [n] need the merging network to implement a parallel version of merge sort. The first stage of SORTER [n] consists of n/2 copies of MERGER [2] that work in parallel to merge pairs of a 1-element sequence to produce a sorted sequence of length 2. The second stage subsists of n/4 copies of MERGER [4] that merge pairs of these 2-element sorted sequences to generate sorted sequences of length 4. In general, for k = 1, 2..... log n, stage k consists of $n/2^k$ copies of MERGER [$2^k$] that merge pairs of the $2^k$-1 element sorted sequence to produce a sorted sequence of length$2^k$. At the last stage, one sorted sequence consisting of all the input values is produced. This sorting network can be shown by induction to sort zero-one sequences, and therefore by the zero-one principle, it can sort arbitrary values.

The recurrence given the depth of SORTER [n]

$$D\ (n) = \begin{Bmatrix} 0 & \text{if} & n = 1 \\ D\left(\frac{n}{2}\right) + \log n & \text{if}\, n = 2^k & \text{and}\, k \geq 1 \end{Bmatrix}$$