Syllabus:

Binary Search- Introduction, Applications: Median of two sorted arrays, Find the fixed point in a given array, Find Smallest Common Element in All Rows, Longest Common Prefix, Koko Eating Bananas.

-Page No: 01

Greedy Method: General method – Applications –Minimum product subset of an array, Best Time to Buy and Sell Stock, Knapsack problem, Minimum cost spanning trees, Single source shortest path problem.

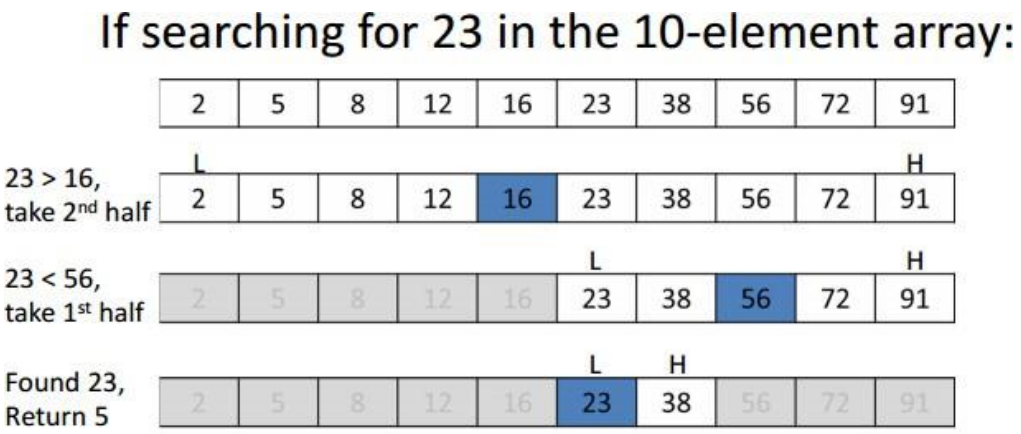
-Page No: 28

I. Binary Search

Introduction:

- Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].
- A simple approach is to do **Linear Search.**
- The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.
- A binary search or half-interval search algorithm finds the position of a specified value (the input "key") within a sorted array.
- In each step, the algorithm compares the input key value with the key value of the middle element of the array.
- If the keys match, then a matching element has been found so its index, or position, is returned.
- Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on the sub-array to the right.
- If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.
- Every iteration eliminates half of the remaining possibilities. This makes binary searches very efficient - even for large collections.
- Binary search requires a sorted collection. Also, binary searching can only be applied to a collection that allows random access (indexing).

Worst case performance: O(log n) **Best case performance: O(1)**



Working:

Binary Search Algorithm can be implemented in two ways which are discussed below.

1. **Recursive Method.**
2. **Iterative Method.**

1. **<u>Recursive Algorithm (Divide and Conquer Technique)</u>**

Algorithm BinSrch (a, low, high, x)

//Given an array a [low : high]of elements in increasing

//order,1≤low≤high,determine whether x is present, and

//if so, return j such that x=a[j]; else return 0.

```
{
        if( low = =high ) then  // If small(P)
        {
                if( x=a[low] ) then return low;
                else return 0;
        }
        else
        {
        //Reduce p into a smaller subproblem.
                        mid:=  (low+high)/2
                       if( x = a[mid] ) then return mid;
                       else if ( x<a[mid] ) then
                return BinSrch(a, low, mid-1, x);
                       else
                return BinSrch(a, mid+1, high, x);
        }
}
```

2. Iterative Algorithm (Divide and Conquer Technique)

```
Algorithm BinSearch(a, n, x)
// a is an array of size n, x is the key element to be searched.
{       low:=1;  high:=n;
  while( low ≤ high)
  {
          mid:=(low+high)/2;
         if (x==a[mid])
         {
                  return     mid;
         }
          if( x < a[mid] )  then  high := mid-1;
            else  low := mid+1;
   }
  return 0;
}
```

Time complexity of Binary Search

➢ If the time for diving the list is a constant, then the computing time for binary search is described by the recurrence relation.

$$T(n) = \begin{cases} c_1 & n=1,\ c1 \text{ is a constant} \\ T(n/2) + c_2 & n>1,\ c2 \text{ is a constant} \end{cases}$$

Assume $n=2^k$, then

$T(n) = T(n/2) + c2$
$=T(n/4)+c2+c2$
$=T(n/8) +c2+c2+c2$
…..
…..
$= T(n / 2^k) + c2+c2+c2+$ …….k times
$= T(1)+ kc2$
$= c1+kc2 =c1+ logn*c2 =$ O(logn)

Successful searches:

Best	Average	Worst
O(1)	O(log n)	O(log n)

Unsuccessful searches:

Best	Average	Worst
O(log n)	O(log n)	O(log n)

Program for Iterative binary search : BinarySearch_iterative.java

```
import java.util.*;
class BinarySearch_iterative
{
  int binarySearch(int array[], int x, int low, int high)
  {

    // Repeat until the pointers low and high meet each other
    while (low <= high)
    {
      int mid = low + (high - low) / 2;

      if (array[mid] == x)
        return mid;

      if (array[mid] < x)
        low = mid + 1;

      else
        high = mid - 1;
    }

    return -1;
  }

  public static void main(String args[])
  {
BinarySearch_iterative ob = new BinarySearch_iterative ( );

        Scanner sc=new Scanner(System.in);

                System.out.println("enter array size");

                  int n = sc.nextInt();

          int array[]=new int[n];

     System.out.println("enter the elements of array ");

        for(int i=0;i<n;i++)
     {
            array[i] =sc.nextInt();
         }
        // Applying sort() method over to above array by passing the array as an argument
        Arrays.sort(array);
```

```
        // Printing the array after sorting
        System.out.println("sorted array[]:"+ Arrays.toString(array));

     System.out.println("Enter the key");
    int key=sc.nextInt();

    int result = ob.binarySearch(array, key, 0, n - 1);
    if (result == -1)
      System.out.println("Not found");
    else
      System.out.println("Element found at index " + result);
  }
}
```

Output:

Case-1:

```
enter array size
5
enter the elements of array
33
65
32
68
95
sorted array[]:[32, 33, 65, 68, 95]
Enter the key
59
Not found
```

Case-2:

```
enter array size
5
enter the elements of array
33
65
32
68
95
sorted array[]:[32, 33, 65, 68, 95]
enter array the key
68
Element found at index 3
```

Program for Recursive binary search: BinarySearch_recursive.java

```
import java.util.*;
class BinarySearch_recursive
{
  int binarySearch(int array[], int x, int low, int high) {

    if (high >= low) {
      int mid = low + (high - low) / 2;

      // If found at mid, then return it
      if (array[mid] == x)
        return mid;

      // Search the left half
      if (array[mid] > x)
        return binarySearch(array, x, low, mid - 1);

      // Search the right half
      return binarySearch(array, x, mid + 1, high);
    }

    return -1;
  }

  public static void main(String args[])
  {
    BinarySearch_recursive ob = new BinarySearch_recursive();

        Scanner sc=new Scanner(System.in);

        System.out.println("enter array size");

          int n = sc.nextInt();

       int array[]=new int[n];

  System.out.println("enter the elements of array ");

        for(int i=0;i<n;i++)
      {
          array[i] =sc.nextInt();
        }
        // Applying sort() method over to above array
        // by passing the array as an argument
```

```
        Arrays.sort(array);

        // Printing the array after sorting
        System.out.println("sorted array[]:"+ Arrays.toString(array));
  System.out.println("Enter the key");
    int key=sc.nextInt();


    int result = ob.binarySearch(array, key, 0, n - 1);
    if (result == -1)
      System.out.println("Element Not found");
    else
      System.out.println("Element found at index " + result);
  }
```

Output:

Case=1

```
enter array size5
enter the elements of array 15
35
25
95
65
sorted array[]:[15, 25, 35, 65, 95]
Enter the key
65
Element found at index 3
```

Case=2

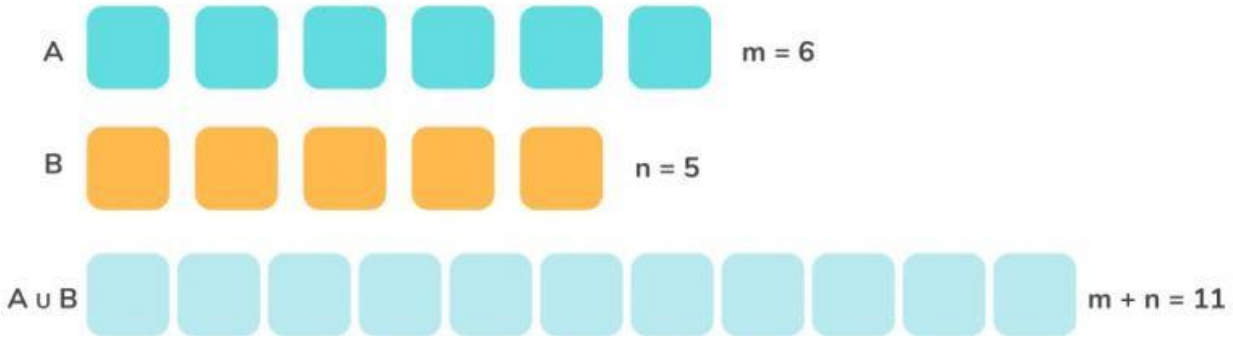
```
enter array size5
enter the elements of array 15
35
25
95
65
sorted array[]:[15, 25, 35, 65, 95]
Enter the key
30
Element Not found
```

Applications:

1. **Median of two sorted arrays.**
2. **Find the fixed point in a given array.**
3. **Find Smallest Common Element in All Rows.**
4. **Longest Common Prefix.**
5. **Koko Eating Bananas.**

1. Median of two sorted arrays:

- There are two sorted arrays **A** and **B** of sizes **m** and **n** respectively.
- Find the median of the two sorted arrays(The median of the array formed by merging both the arrays).
- **Median:** The middle element is found by ordering all elements in sorted order and picking out the one in the middle (or if there are two middle numbers, taking the mean of those two numbers).



Examples:

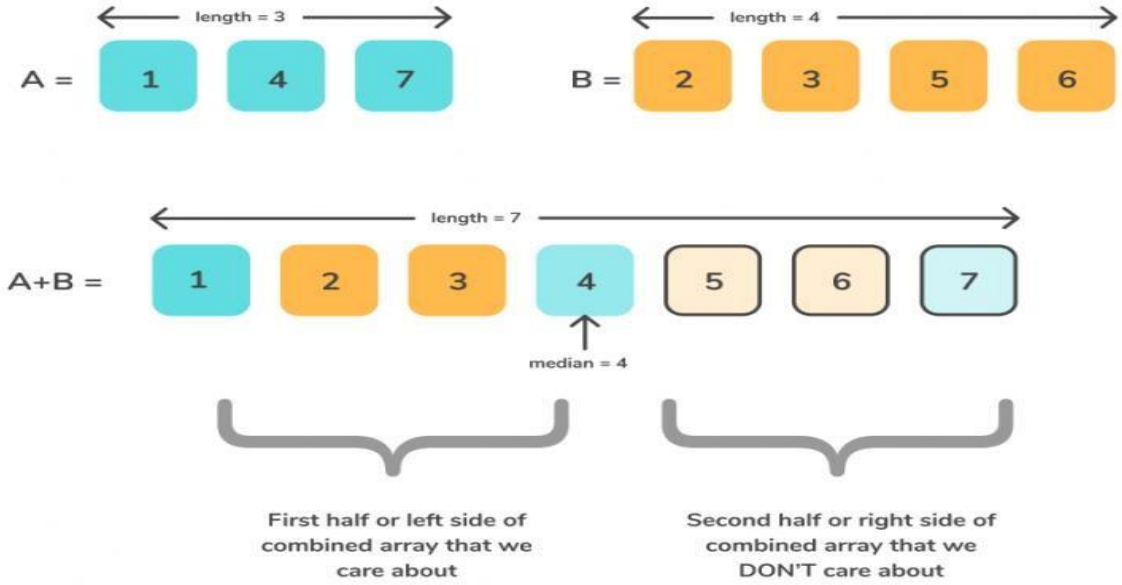
Input: A[] = {1, 4, 5}, B[] = {2, 3}
Output: 3
Explanation:
Merging both the arrays and arranging in ascending:
[1, 2, 3, 4, 5]
Hence, the median is 3

Input: A[] = {1, 2, 3, 4}, B[] = {5, 6}
Output: 3.5
Explanation:
Union of both arrays:
{1, 2, 3, 4, 5, 6}
Median = (3 + 4) / 2 = 3.5

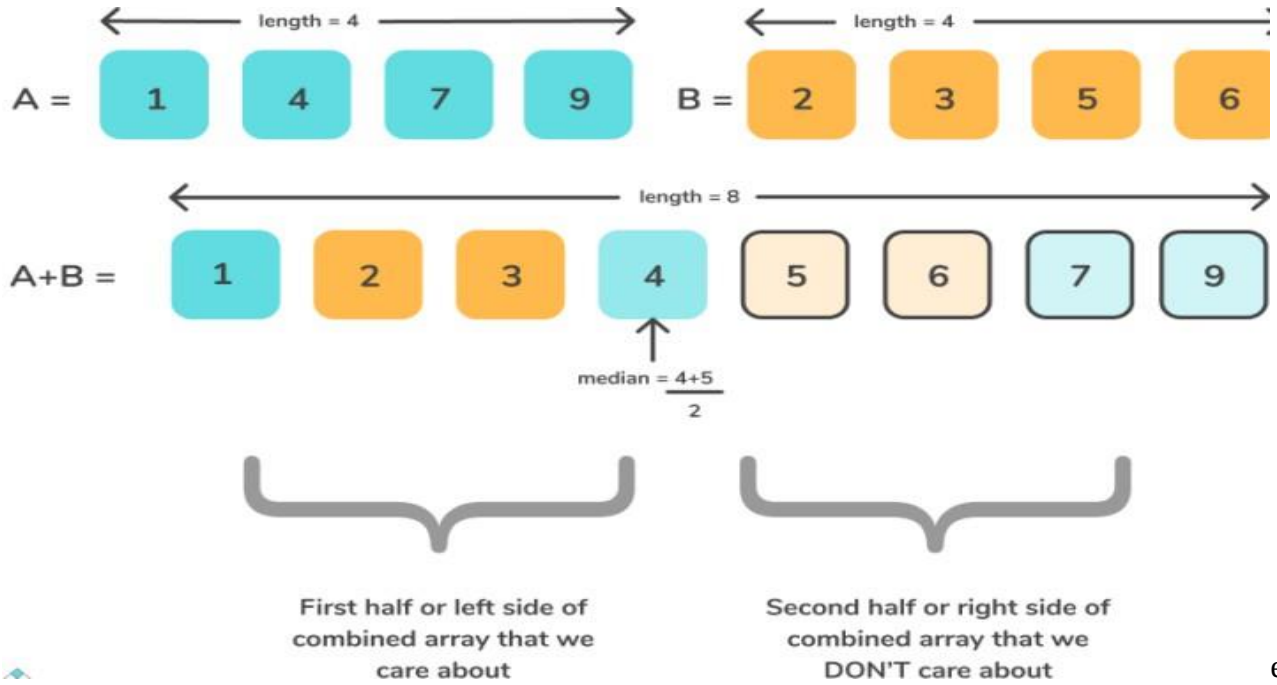
The overall Run time complexity should be O(log (m+n)).

Using Binary search:

The key idea to note here is that both the arrays are **sorted**. Therefore, this leads us to think of binary search. Let us try to understand the algorithm using an example:
A[ ] = {1, 4, 7} **B[ ]** = {2, 3, 5, 6}



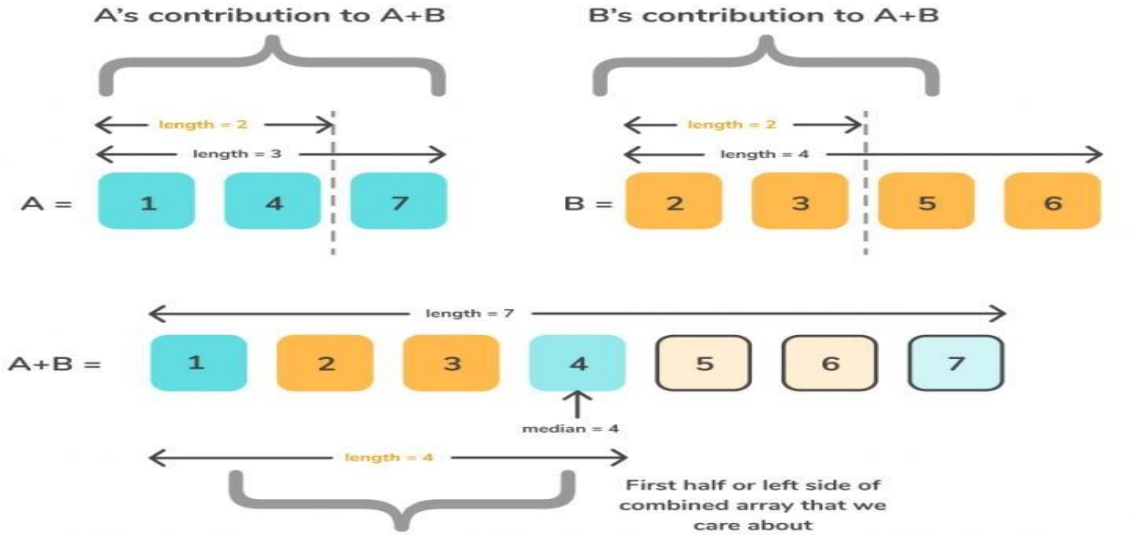
From the above diagram, it can be easily deduced that only the **first half** of the array is needed and the **right half** can be discard



ed.

➢ Similarly, for an even length merged array, ignore the **right half** and only the **left half** contributes to our final answer.

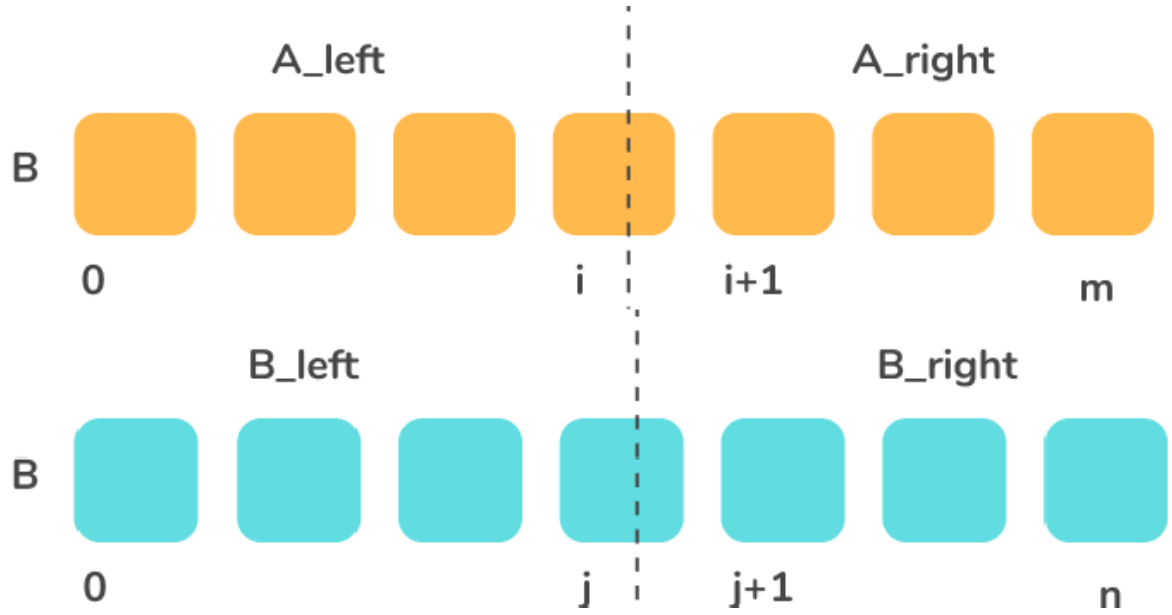
➢ Therefore, the motive of our approach is to find which of the elements from both the array helps in contributing to the final answer. Therefore, the **binary search** comes to the rescue, as it can discard a part of the array every time, the elements don't contribute to the median.



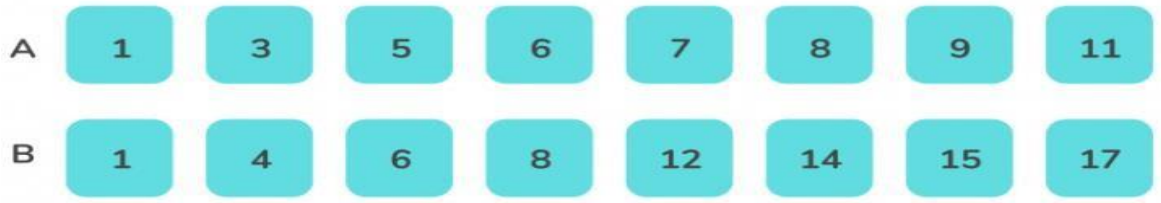
Algorithm

➢ The first array is of size **n**, hence it can be split into **n + 1** parts.

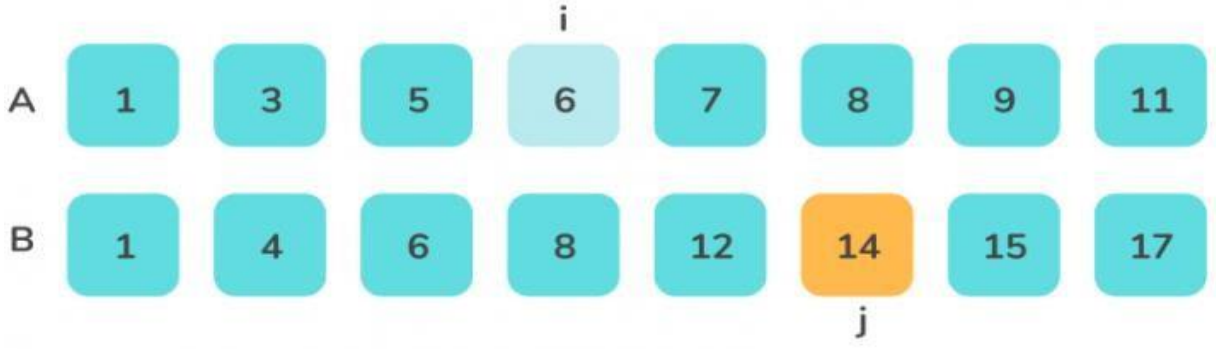
➢ The second array is of size **m**, hence it can be split into **m + 1** parts



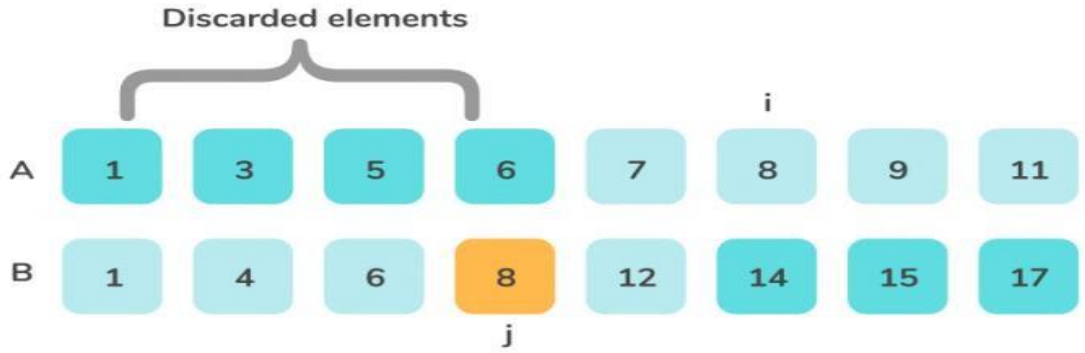
- As discussed earlier, we just need to find the elements contributing to the left half of the array.
- Since, the arrays are already sorted, it can be deduced that **A[i – 1] < A[i]** and **B[j – 1] < B[j]**.
- Therefore, we just need to find the index **i**, such that **A[i – 1] <= B[j]** and **B[j – 1] <= A[i].**
- Calculate **i =(low+hing)/2** (here we are considering low and high from A array)
- Consider **mid = (n + m +1) / 2** and check if this satisfies the above condition.
- Calculate j=mid-i
- If **A[i-1] <= B[j]** and **B[j-1]<=A[i]** satisfies the condition, return the index **i**.
- If **A[i-1] >B[j]**, increase the range towards the right. Hence update **high = i-1**.
- Similarly, if **A[i] <B[j-1]**, decrease the range towards left. Hence update **low = i+1**.



i is the mid of array **A** as shown below



A[i-1] = 5 b[j-1] = 12, since B[j-1] > A[i], needs to increase



Max of left side will be either A[i-1] or B[j-1], in this case its A[i-1] = 7

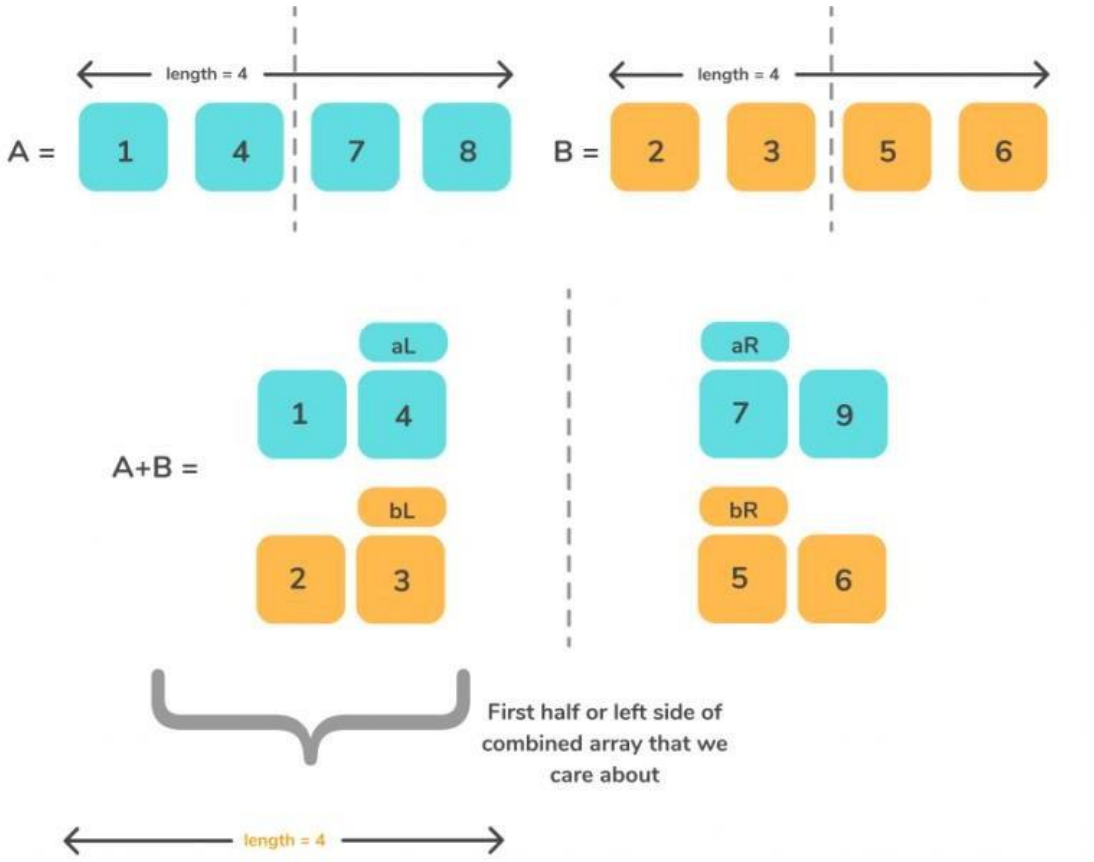
Min of right will be either A[i] or B[j], in this case both are equal which is 8

Since, sum of length is even, we return average of these two which is 7.5

:

Few corner cases to take care of :

- If the size of any of the arrays is **0**, return the median of the non-zero sized array.
- If the size of smaller array is **1**,:
- If the size of the larger array is also one, simply return the median as the mean of both the elements.
- Else, if size of larger array is odd, adding the element from first array will result in size even, hence median will be affected if and only if, the element of the first array lies between **, M / 2th** and **M/2 + 1**th element of **B[].**
- Similarly, if the size of the larger array is even, check for the element of the smaller array, **M/2th element** and **M / 2 + 1 th** element.
- If the size of smaller array is **2**,
- If a larger array has an odd number of elements, the median can be either the **middle** element **or** the median of elements of smaller array and **M/2 -1th** element or minimum of the second element of **A[]** and **M/2+ 1** th array.



Java program for Median Of Two Sorted Array (Binary search approach)

MedianOfTwoSortedArrays.java

```
import java.util.*;
public class MedianOfTwoSortedArrays
 {
   private static double findMedianSortedArrays(int[] nums1, int[] nums2)
   {
  // Check if num1 is smaller than num2 If not, then we will swap num1 with num2
       if (nums1.length > nums2.length)
       {
          return findMedianSortedArrays(nums2, nums1);
       }
       // Lengths of two arrays
       int m = nums1.length;
       int n = nums2.length;
       // Pointers for binary search
       int start = 0;
       int end = m;
       // Binary search starts from here
       while (start <= end)
          {
           // Partitions of both the array
           int partitionNums1 = (start + end) / 2;
           int partitionNums2 = (m + n + 1) / 2 - partitionNums1;

// Edge cases there are no elements left on the left side after partition
int maxLeftNums1 = partitionNums1 == 0 ? Integer.MIN_VALUE :nums1[partitionNums1 - 1];

// If there are no elements left on the right side after partition
 int minRightNums1 = partitionNums1 == m ? Integer.MAX_VALUE : nums1[partitionNums1];

 // Similarly for nums2
int maxLeftNums2 = partitionNums2 == 0 ? Integer.MIN_VALUE :nums2[partitionNums2 - 1];

int minRightNums2 = partitionNums2 == n ? Integer.MAX_VALUE :nums2[partitionNums2]
          // Check if we have found the match

     if (maxLeftNums1 <= minRightNums2 && maxLeftNums2 <= minRightNums1)
                {
             // Check if the combined array is of even/odd length
             if ((m + n) % 2 == 0)
           {
```

```
            return (Math.max(maxLeftNums1, maxLeftNums2)+Math.min(minRightNums1, minRightNu
    ms2)) / 2.0;
                }
            else {
                    return Math.max(maxLeftNums1, maxLeftNums2);
                }
             }
             // If we are too far on the right, we need to go to left side
             else if (maxLeftNums1 > minRightNums2) {
                end = partitionNums1 - 1;
             }
             // If we are too far on the left, we need to go to right side
             else {
                start = partitionNums1 + 1;
             }
          }
          // If we reach here, it means the arrays are not sorted
          throw new IllegalArgumentException();
       }
       public static void main(String[] args)
       {
          Scanner sc=new Scanner(System.in);
           System.out.println("enter the  size of the 1st array");
                 int n = sc.nextInt();
                 int nums1[]=new int[n];
            System.out.println("enter the elements of 1st array ");

          for(int i=0;i<n;i++)
          {
            nums1[i] =sc.nextInt();
           }
        System.out.println("enter the  size of the 2nd array");

            int m = sc.nextInt();
              int nums2[]=new int[m];
            System.out.println("enter the elements of 2nd array ");

          for(int i=0;i<m;i++)
          {
             nums2[i] =sc.nextInt();
           }
System.out.println("The Median of two sorted arrays is :" + findMedianSortedArrays(nums1, nums2));
          }
    }
```

Example:

Input=

enter the size of the 1st array
4
enter the elements of 1st array
1 4 7 8
enter the size of the 2nd array
4
enter the elements of 2nd array
2 3 5 6

Output=

The Median of two sorted arrays is :4.5

2. Find the fixed point in a given array:

- Given an array of n distinct integers sorted in ascending order, write a function that returns a Fixed Point in the array, if there is any Fixed Point present in array, else returns -1.
- Fixed Point in an array is an index i such that arr[i] is equal to i. Note that integers in array can be negative.

Example 1:
Input: [-10,-5,0,3,7]
Output: 3
Explanation:
For the given array, A[0] = -10, A[1] = -5, A[2] = 0, A[3] = 3, thus the output is 3.

Example 2:
Input: [0,2,5,8,17]
Output: 0
Explanation:
A[0] = 0, thus the output is 0.

Example 3:
Input: [-10,-5,3,4,7,9]
Output: -1
Explanation:
There is no such i that A[i] = i, thus the output is -1.

Note:
1 <= A.length < 10^4
-10^9 <= A[i] <= 10^9

Algorithm

The basic idea of binary search is to divide n elements into two roughly equal parts, and compare a[n/2] with x.

- If x=a[n/2], then find x and the algorithm stops;
- if x<a[n/2], as long as you continue to search for x in the left half of array a,
- if x>a[n/2], then as long as you search for x in the right half of array a.

Java program for Find the Fixed point in an array: Findfixedpoint.java

```
import java.util.*;
class Findfixedpoint
{
   static int fixedpoint(int arr[], int low, int high)
   {
      if (high >= low) {
         int mid = (low + high ) / 2;
         if (mid == arr[mid])
            return mid;
         int res = -1;
         if (mid + 1 <= arr[high])
            res = fixedpoint(arr, (mid + 1), high);
         if (res != -1)
            return res;
         if (mid - 1 >= arr[low])
            return fixedpoint(arr, low, (mid - 1));
      }

      /* Return -1 if there is no Fixed Point */
      return -1;
   }

   // main function
   public static void main(String args[])
   {
       Scanner sc=new Scanner(System.in);
       System.out.println("enter array size");
       int n = sc.nextInt();
        int array[]=new int[n];

  System.out.println("enter the elements of array ");

      for(int i=0;i<n;i++)
     {
        array[i] =sc.nextInt();
       }
```

```
        Arrays.sort(array);

        // Printing the array after sorting
        System.out.println("sorted array[]:"+ Arrays.toString(array));
         System.out.println("Fixed Point is " + fixedpoint(array, 0, n - 1));
    }
}
```

Example-1:

Input=

enter array size
10
enter the elements of array
11 30 50 0 3 100 -10 -1 10 102
sorted array[]:[-10, -1, 0, 3, 10, 11, 30, 50, 100, 102]

Output=

Fixed Point is 3.

Example-2:

Input=

enter array size
6
enter the elements of array
3 9 4 7 -5 -10
sorted array[]:[-10, -5, 3, 4, 7, 9]

Output=

Fixed Point is -1

Example-3:

Input=

enter array size
5
enter the elements of array
8 2 5 17 0
sorted array[]:[0, 2, 5, 8, 17]

Output=

Fixed Point is 0

3. Find Smallest Common Element in All Rows.

Given a matrix where every row is sorted in increasing order. Write a function that finds and returns a smallest common element in all rows. If there is no common element, then returns **-1**.

Example-1:
Input: mat [4][5] = { {1, 2, 3, 4, 5},
{2, 4, 5, 8, 10},
{3, 5, 7, 9, 11},
{1, 3, 5, 7, 9}
};
Output: 5

Time complexity:

- A **O(m*n*n) simple solution** is to take every element of first row and search it in all other rows, till we find a common element.
- Time complexity of this solution is O(m*n*n) where m is number of rows and n is number of columns in given matrix.
- This can be improved to **O(m*n*logn)** if we **use Binary Search** instead of linear search.

Java Program for Find the Smallest Common Element Using Binary Search:SmallestCommonElement.java

```
import java.util.*;
class SmallestCommonElement
{
   private boolean binarySearch(int[] arr, int low, int high, int target)
{
      while(low <= high) {
         int mid = (low + high)/2;
         if(arr[mid] == target)
{
            return true;
         }
else if(arr[mid] < target)
 {
            low = mid+1;
         } else {
            high = mid-1;
         }
      }
```

```
        return false;
    }

    public int smallestCommonElement(int[][] mat)
     {
        if(mat.length == 1) return mat[0][0];
        for(int a : mat[0]) {
            int count = 0;
            for(int i=1; i<mat.length; i++) {
                if(binarySearch(mat[i], 0, mat[i].length-1, a))
                 {
                    count++;
                } else {
                    break;
                }
            }
            if(count == mat.length-1) return a;
        }
        return -1;
    }
    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        System.out.println( "enter size of two dimensional matrix" );
       System.out.println( "enter row size of matrix " );
       int m=sc.nextInt();
       System.out.println( "enter column size of matrix " );
        int n=sc.nextInt();
        int[][] arr = new int[m][n];
 System.out.println( "enter the elements " );
        for(int i=0;i<m;i++)
        for(int j=0;j<n;j++)
        arr[i][j] = sc.nextInt();
        System.out.println( "smallest comman element :"+new SmallestCommonElement().smallestCo
mmonElement(arr) );
    }
}
```

Exmaple-1:

input=

enter size of two dimensional matrix
enter row size of matrix
4
enter column size of matrix
5
enter elements
1 2 3 4 5
2 4 5 8 10
3 5 7 9 11
1 3 5 7 9

output=

smallest comman element :5

Exmaple-2:

input=

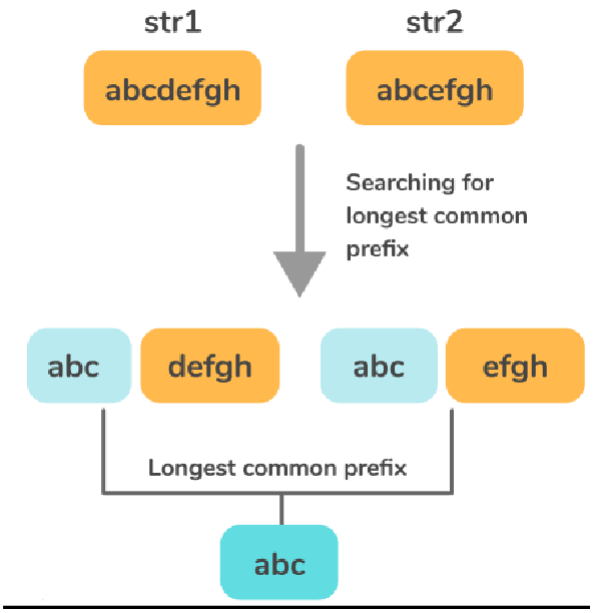
enter size of two dimensional matrix
enter row size of matrix
2
enter column size of matrix
2
enter elements for matrix
1 2
3 4

output=

smallest comman element :-1

4. Longest Common Prefix

- Given the array of strings S[], you need to find the longest string S which is the prefix of ALL the strings in the array.
- **Longest common prefix (LCP)** for a pair of strings S1 and S2 is the longest string S which is the prefix of both S1 and S2.
- For Example: longest common prefix of **"abcdefgh"** and **"abcefgh"** is **"abc"**.



Examples:
Input: S[] = {"abcdefgh", "abcefgh"}
Output: "abc"
Explanation: Explained in the image description above

Input: S[] = {"abcdefgh", "aefghijk", "abcefgh"}
Output: "a"

Binary Search Approach

Algorithm:

- Consider the string with the smallest length. Let the length be **L**.
- Consider a variable **low = 0** and **high = L – 1**.
- Perform binary search:
- Divide the string into two halves, i.e. **low – mid** and **mid + 1** to **high**.
- Compare the substring up to the **mid** of this smallest string to every other character of the remaining strings at that index.
- If the substring from **0** to **mid – 1** is common among all the substrings, update **low** with **mid + 1**, else update **high** with **mid – 1**
- If **low == high**, terminate the algorithm and return the substring from **0** to **mid.**

Java program for LongestCommonPrefix using Binary search approach

LongestCommonPrefix.java

```
import java.util.*;

class  LongestCommonPrefix
 {
   public String longestCommonPrefix(String[] strs)
    {
      if (strs == null || strs.length == 0)
      return "";
         return longestCommonPrefix(strs, 0 , strs.length - 1);
   }

   private String longestCommonPrefix(String[] strs, int l, int r)
   {
      if (l == r)
      {
         return strs[l];
      }
      else
      {
         int mid = (l + r)/2;
         String lcpLeft =  longestCommonPrefix(strs, l , mid);
         String lcpRight =  longestCommonPrefix(strs, mid + 1,r);
         return commonPrefix(lcpLeft, lcpRight);
     }
   }
```

```
    String commonPrefix(String left,String right)
     {
        int min = Math.min(left.length(), right.length());
        for (int i = 0; i < min; i++)
         {
            if ( left.charAt(i) != right.charAt(i) )
                return left.substring(0, i);
        }
        return left.substring(0, min);
    }

    public static void main(String args[])
{
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter Strings");
        String[] words = sc.nextLine().split(" ");

        System.out.println("Longest common Prefix is: "+new
LongestCommonPrefix().longestCommonPrefix(words));
    }
}
```

Example-1:

input:

Enter Strings
fly flower flow

output:

Longest common Prefix is: fl

Example-2:

input:

Enter Strings
f c i

output:

Longest common Prefix is:

5. Koko Eating Bananas:

- Koko loves to eat bananas. There are n piles of bananas, the **ith** pile has **piles[i]** bananas. The guards have gone and will come back in h hours.
- Koko can decide her bananas-per-hour eating speed of **k**. Each hour, she chooses some pile of bananas and eats **k** bananas from that pile. If the pile has less than **k** bananas, she eats all of them instead and will not eat any more bananas during this hour.
- Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.
- Return *the minimum integer* k *such that she can eat all the bananas within* h *hours.*

Example 1:
Input: piles = [3,6,7,11], h = 8
Output: 4
Example 2:
Input: piles = [30,11,23,4,20], h = 5
Output: 30
Example 3:
Input: piles = [30,11,23,4,20], h = 6
Output: 23

Input:
piles = [**30**,**11**,**23**,**4**,**20**], H = **6**
output:
23

Koko will eat bananas in this way to eat all bananas in 6 hours:

First hour: 23
Second hour: 7
Third hour: 11
Fourth hour: 23

Approach for Koko Eating Bananas

The first and the most important thing to solve this problem is to bring out observations. Here are a few observations for our search interval:

1. Koko must eat at least one banana per hour. So this is the minimum value of K. let's name it as **Start**
2. We can limit the maximum number of bananas Koko can eat in one hour to the maximum number of bananas in a pile out of all the piles. So this is the maximum value of K. let's name it as **End.**

Now we have our search interval. Suppose the size of the interval is **Length** and the number of piles is **n.** The naive approach could be to check for each value in the interval. if for that value of K Koko can eat all bananas in H hour successfully then pick the minimum of them. The time complexity for the naive approach will be Length*n in worst case.

We can improve the time complexity by using Binary Search in place of Linear Search.
The time complexity using the Binary Search approach will be log(Length)*n.

Time complexity:

- The time complexity of the above code is **O(n*log(W))** because we are performing a binary search between one and W this takes logW time and for each value, in the binary search, we are traversing the piles array.
- So the piles array is traversed logW times it makes the time complexity n*logW. Here n and W are the numbers of piles and the maximum number of bananas in a pile.

Space complexity:

The space complexity of the above code is **O(1)** because we are using only a variable to store the answer.

Java program for Kokoeatingbananas: Kokoeatingbanana.java

```
import java.util.*;
public class Kokoeatingbanana
{
public static int calculateTotalHours(int[] v, int hourly)
{
      int totalH = 0;
      int n = v.length;
      //find total hours:
      for (int i = 0; i < n; i++) {
         totalH += Math.ceil((double)(v[i]) / (double)(hourly));
      }
      return totalH;
```

```
    }

    public static int minEatingSpeed(int[] piles, int H)
{
        int low = 1, high = Arrays.stream(piles).max().getAsInt();
        //apply binary search:
        while (low <= high)
     {
            int mid = (low + high) / 2;
            int totalH = calculateTotalHours(piles, mid);
            if (totalH <= H)
           {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
        return low;
    }

  public static void main(String[] args) throws Exception
   {
         Scanner sc=new Scanner(System.in);
          int n = sc.nextInt();
          int arr[]=new int[n];

          for(int i=0;i<n;i++)
        {
          arr[i] =sc.nextInt();
          }
            int H = sc.nextInt();
    int ans= minEatingSpeed(arr, H);
    System.out.println(ans);
  }
}
input=
4
3 6 7 11
8
output=
4
```

II. Greedy Method:

- Among all the algorithmic approaches, the simplest and straightforward approach is the Greedy method.
- In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.
- Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit.
- This approach never reconsiders the choices taken previously
- This approach is mainly used to solve optimization problems.
- Greedy method is easy to implement and quite efficient in most of the cases.
- Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.
- In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

Control abstraction (pseudo code) for Greedy Method

```
Algorithm GreedyMethod (a, n)
{
// a is an array of n inputs
        Solution: =Ø;
        for i: =0 to n do
        {
        s: = select (a);
        if (feasible (Solution, s)) then
        {
         Solution: = union (Solution, s);
         }
         else reject (); // if solution is not feasible reject it.
         }
         return Solution;
 }
```

Three important activities:

1. A selection of solution from the given input domain is performed, i.e. s:= select(a).
2. The feasibility of the solution is performed, by using feasible '(solution, s)' and then all feasible solutions are obtained.
3. From the set of feasible solutions, the particular solution that minimizes or maximizes the given objection function is obtained. Such a solution is called optimal solution

Applications:

1. **Minimum product subset of an array**
2. **Best Time to Buy and Sell Stock**
3. **0/1 Knapsack Problem**
4. **Minimum cost spanning trees**
5. **Single source shortest path Problem**

1. Minimum product subset of an array

Given an array a, we have to find the minimum product possible with the subset of elements present in the array. The minimum product can be a single element also.

Examples:
Input : a[] = { -1, -1, -2, 4, 3 }
Output : -24
Explanation : Minimum product will be (-2 * -1 * -1 * 4 * 3) = -24

Input : a[] = { -1, 0 }
Output : -1
Explanation : -1(single element) is minimum product possible

Input : a[] = { 0, 0, 0 }
Output : 0

A simple solution is to generate all subsets, find the product of every subset and return the minimum product.

A better solution is to use the below facts.

->If there are even number of negative numbers and no zeros, the result is the product of all except the largest valued negative(closest to zero) number.
->If there are an odd number of negative numbers and no zeros, the result is simply the product of all.
->If there are zeros and positive, no negative, the result is 0. The exceptional case is when there is no negative number and all other elements positive then our result should be the first minimum positive number.

Complexity Analysis:
Time Complexity: O(n)
Auxiliary Space: O(1)

Java program to find Minimum product of a subset MinProductSubset.java

```
import java.util.*;

class MinProductSubset
{

   static int minProductSubset(int a[], int n)
   {
      if (n == 1)
         return a[0];

 /*Find count of negative numbers,count of zeros, maximum valued negative number, minimum valued positive number and product of non-zero numbers */

      int negmax = Integer.MIN_VALUE;
      int posmin = Integer.MAX_VALUE;
          int count_neg = 0;
        int count_zero = 0;
        int product = 1;

      for (int i = 0; i < n; i++) {

         // if number is zero,count it  but dont multiply

  if (a[i] == 0) {
            count_zero++;
            continue;
         }

   // count the negative numbers  and find the max negative number
         if (a[i] < 0) {
            count_neg++;
            negmax = Math.max(negmax, a[i]);
         }

         // find the minimum positive number
         if (a[i] > 0 && a[i] < posmin)
            posmin = a[i];

         product *= a[i];
      }

      // if there are all zeroes or zero is present but no negative number is present
```

```
 if (count_zero == n  || (count_neg == 0 && count_zero > 0))
          return 0;

       // If there are all positive
       if (count_neg == 0)
          return posmin;

       // If there are even number except
       // zero of negative numbers
       if (count_neg % 2 == 0 && count_neg != 0) {

          // Otherwise result is product of
          // all non-zeros divided by maximum
          // valued negative.
          product = product / negmax;
       }

       return product;
    }

 public static void main(String[] args)
    {

       Scanner sc=new Scanner(System.in);
       System.out.println("enter size of the array");
       int n=sc.nextInt();
          System.out.println("enter elements");
          int a[]=new int[n];
       for(int i=0;i<n;i++)
       {
          a[i]=sc.nextInt();
       }
       System.out.println(minProductSubset(a,  n));
}
}
```

case=1

input=

enter size of the array
5
enter elements
-1 -1 -2 4 3

output=

-24

2. Best Time to Buy and Sell Stock

Type I: At most one transaction is allowed

Given an array prices [] of length N, representing the prices of the stocks on different days, the task is to find the maximum profit possible for buying and selling the stocks on different days using transactions where at most one transaction is allowed.

Note: Stock must be bought before being sold.

Input: prices[] = {7, 1, 5, 3, 6, 4]
Output: 5
Explanation:
The lowest price of the stock is on the 2nd day, i.e. price = 1. Starting from the 2nd day, the highest price of the stock is witnessed on the 5th day, i.e. price = 6.
Therefore, maximum possible profit = 6 – 1 = 5.

Input: prices[] = {7, 6, 4, 3, 1}
Output: 0
Explanation: Since the array is in decreasing order, no possible way exists to solve the problem.

Time Complexity: O(N). Where N is the size of prices array.
Auxiliary Space: O(1). We do not use any extra space.

Java program for Best Time to Buy and Sell Stock with at most one transaction

BTBS_Atmostonetime.java

```
import java.util.*;

class BTBS_Atmostonetime
{
static int maxProfit(int prices[], int n)
{
 int minprice=Integer.MAX_VALUE;
 int max_profit = 0;

 for (int i = 0; i < n; i++)
 {
 if (prices[i]< minprice)
        minprice= prices[i];

 else if (prices[i] -minprice > max_profit)
```

```
  max_profit = prices[i] -minprice;
 }
 return max_profit;
}
public static void main(String args[])
{
        Scanner sc =new Scanner(System.in);
        System.out.println("Enter array size");

    int size=sc.nextInt();

         int prices[]=new int[size];

       System.out.println("Enter elements");

    for(int i=0;i<size;i++)
 {
     prices[i]=sc.nextInt();
     }
 int max_profit = maxProfit(prices,size);
 System.out.println("Maximum Profit is:" + max_profit);
}
}
```

Input=
Enter array size
6
Enter elements
7 1 5 3 6 4
Output=
Maximum Profit is:5

Input=
Enter array size
6
Enter elements
6 5 4 3 2 1
output
Maximum Profit is:0

Type II: Infinite transactions are allowed

Given an array price[] of length N, representing the prices of the stocks on different days, the task is to find the maximum profit possible for buying and selling the stocks on different days using transactions where any number of transactions are allowed.

input: prices[] = {7, 1, 5, 3, 6, 4}
Output: 7

Explanation:
Purchase on 2nd day. Price = 1.
Sell on 3rd day. Price = 5.
Therefore, profit = 5 – 1 = 4.
Purchase on 4th day. Price = 3.
Sell on 5th day. Price = 6.
Therefore, profit = 4 + (6 – 3) = 7.

Input: prices = {1, 2, 3, 4, 5}
Output: 4
Explanation:
Purchase on 1st day. Price = 1.
Sell on 5th day. Price = 5.
Therefore, profit = 5 – 1 = 4.

Approach: The idea is to maintain a boolean value that denotes if there is any current purchase ongoing or not. If yes, then at the current state, the stock can be sold to maximize profit or move to the next price without selling the stock. Otherwise, if no transaction is happening, the current stock can be bought or move to the next price without buying.

Java program for Best Time to Buy and Sell Stock with Infinite transactions :

BTBS_InfiniteTransactions.java

```
import java.util.*;
  class BTBS_InfiniteTransactions
{
      public static void main(String [] args)
      {
         Scanner sc = new Scanner(System.in);
         int n=sc.nextInt();
         int arr[] = new int[n];
         for(int i=0;i<n;i++){
            arr[i]=sc.nextInt();
         }
        System.out.println(prof(arr,n));
      }
     public static int prof(int arr[] , int n){
        int profit=0;
         for(int i=0;i<n-1;i++)
        {
            if(arr[i]<arr[i+1])
           {
               profit+=(arr[i+1]-arr[i]);
            }
         }
         return profit;
      }
  }
```

input=

7

7 1 5 3 6 4

Output=

7

3. 0/1 Knapsack Problem:

- In this problem we have a Knapsack that has a weight limit M.
- There are items i1, i2, ..., in each having weight w1, w2, … wn and some benefit (value or profit) associated with it p1, p2, ..., pn
- Our objective is to maximize the benefit such that the total weight inside the knapsack is at most M, and we are also allowed to take an item in fractional part.
- There are **n items in the store**

-weight of **ith item $w_i > 0$**

- Profit for **ith item $p_i > 0$ and**

-Capacity of the Knapsack is **M.**

- In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction ***xi of ith item.*** $0 \le xi \le 1$
- The ith item contributes the weight $xi.\ wi$ to the total weight in the knapsack and profit $xi*pi$ to the total profit.
- Hence, the objective of this algorithm is to $maximize\ \Sigma_{n=1\ to\ n}(xi.\ pi)$
- subject to constraint, $\Sigma_{n=1\ to\ n}(xi.\ wi) \le M$
- It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.
- Thus, an optimal solution can be obtained by $\Sigma_{n=1\ to\ n}(xi.\ wi) = M$
- In this context, first we need to sort those items according to the value of pi/wi, **so that** $(pi+1)/(wi+1) \le pi/wi$.
- **Here,** ***x is an array to store the fraction of items.***

Algorithm:

```
void GreedyKnapsack(float m, int n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
// size and x[1:n] is the solution vector.
{
   for (int i=1; i<=n; i++) x[i] = 0.0; // Initialize x.
   float U = m;
   for (i=1; i<=n; i++) {
      if (w[i] > U) break;
      x[i] = 1.0;
      U -= w[i];
   }
   if (i <= n) x[i] = U/w[i];
}
```

Analysis:

If the provided items are already sorted into a decreasing order of $\boldsymbol{pi/wi}$, then the while loop takes a time in ($\boldsymbol{n}$); Therefore, the total time including the sort is in

$\boldsymbol{O(n\ log\ n)}$.

Example 1:

- Let us consider that the capacity of the knapsack $\boldsymbol{W = 60}$ and the list of provided item are shown in below

Profits=(p1,p2,p3,p4)=(280,100,120,120)

Weigts=(w1,w2,w3,w4)=(40,10,20,24)

- As the provided items are not sorted based on **pi / wi. After sorting, the items are as** shown in the following table.

ITEM	B	A	C	D
PROFIT	100	280	120	120
WEIGHT	10	40	20	24
RATIO (Pi/Wi)	10	7	6	5

- After sorting all the items according to $\boldsymbol{pi/wi}$. **First all of *B is chosen as weight of B is less*** than the capacity of the knapsack.
- Next, item ***A is chosen, as the available capacity of the*** knapsack is greater than the weight of ***A.***
- ***Now, C is chosen as the next item. However,***the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C.***

  Hence, fraction of ***C (i.e. (60 − 50)/20) is chosen.***
- Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.
- Feasible solution is **(x1,x2,x3,x4)=(1,1,0.5,0)**
- The total weight of the selected items is $\boldsymbol{10 + 40 + 20 * (10/20) = 60}$
- And the total profit is $\boldsymbol{100 + 280 + 120 * (10/20) = 380 + 60 = 440}$
- This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Java Program for 0/1 knapsack : Greedy_Knapsack.java

```
import java.util.*;

class Greedy_Knapsack
{
   public static void main(String args[])
   {
      int m,i,j,k[],temp1;
      float max,temp;
      float  w[],p[],r[],x[],n,w1=0,sum=0;
      Scanner sc=new Scanner(System.in);
      System.out.println("enter kanpsack size ");
      n=sc.nextFloat();
      System.out.println("enter the no.of Items ");
      m=sc.nextInt();
      w=new float[m];
      p=new float[m];
      r=new float[m];
      x=new float[m];
      k=new int[m];
      System.out.println("enter the weigths of Items ");
      for(i=0;i<m;i++)
      {
         k[i]=i+1;
         w[i]=sc.nextInt();
      }
      System.out.println("enter the profits of Items");
      for(i=0;i<m;i++)
      {
         p[i]=sc.nextInt();
      }
      for(i=0;i<m;i++)
      {
         r[i]=p[i]/w[i];
      }
      for(i=0;i<m-1;i++)
      {
         for(j=0;j<m-i-1;j++)
         {
            if(r[j+1]>r[j])
            {
               temp=r[j];
```

```
                r[j]=r[j+1];
                r[j+1]=temp;

                temp=p[j];
                p[j]=p[j+1];
                p[j+1]=temp;

                temp=w[j];
                w[j]=w[j+1];
                w[j+1]=temp;

                temp1=k[j];
                k[j]=k[j+1];
                k[j+1]=temp1;
            }
        }
    }
    for(i=0;i<m;i++)
    {
        if(n>w[i]) {
            n=n-w[i];
            x[i]=1;
        }
        else if(n==0){
            x[i]=0;
        }
        else
        {
            x[i]=n/w[i];
            n=0;
        }
    }
    System.out.println("feasible solution is ");
    System.out.println("Itemno"+"\t"+"Weights"+"\t"+"Profits"+"\t"+"pi/wi Ratio"+"\t"+"Selecte
d");
    for(i=0;i<m;i++)
    {
        System.out.print(k[i]+"\t"+w[i]+"\t"+p[i]+"\t"+r[i]+"\t"+"\t"+  x[i]+"\n");
    }
    for(i=0;i<m;i++)
    {
        sum=sum+(x[i]*p[i]);
    }
```

```
        System.out.println("Optimal Solution: Maximum Profit is ");
        System.out.println(sum);
        /*for(i=0;i<m;i++)
        {
           w1=w1+w[i]*x[i];//for checing whether total equal to actual weight of the bag
        }*/
        //System.out.println(w1);
    }
}
```

Output:

```
enter kanpsack size
60
enter the no.of Items
4
enter the weigths of Items
40 10 20 24
enter the profits of Items
280 100 120 120
feasible solution is
Itemno    Weights Profits   pi/wi Ratio     Selected
  2          10.0    100.0      10.0          1.0
  1          40.0    280.0      7.0           1.0
  3          20.0    120.0      6.0           0.5
  4          24.0    120.0      5.0           0.0
Optimal Solution: Maximum Profit is
440.0
```

Graph

A graph can be represented as a collection of vertices with edges joining them. The most popular types of graphs:

- **Undirected Graph:** An undirected graph is one where the edges do not indicate in the same direction, making it bidirectional instead of unidirectional. It's also feasible to consider it as a graph with V vertices and E edges, each uniting two separate vertices.
- **Connected Graph:** A connected graph is one when there is invariably a path from one vertex to another. Also, we can say that a graph is connected if we can go to any vertex from any different vertex by pursuing edges in either direction.
- **Directed Graph:** A graph is considered a directed graph if all the edges present between any nodes or vertices have a defined direction.

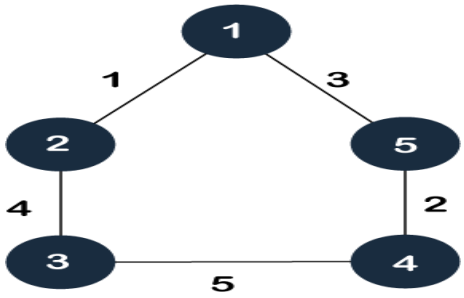
Spanning Tree:

- A spanning tree is known as a subgraph of an undirected connected graph that possesses all of the graph's edges or vertices with the rarest feasible edges. If a vertex is missing, then it is not a spanning tree.
- A complete undirected graph possesses $n^{(n-2)}$ number of spanning trees, so if we have n = 4, the highest number of potential spanning trees is equivalent to $4^{4-2} = 16$. Thus, 16 spanning trees can be constructed from a complete graph with 4 vertices.

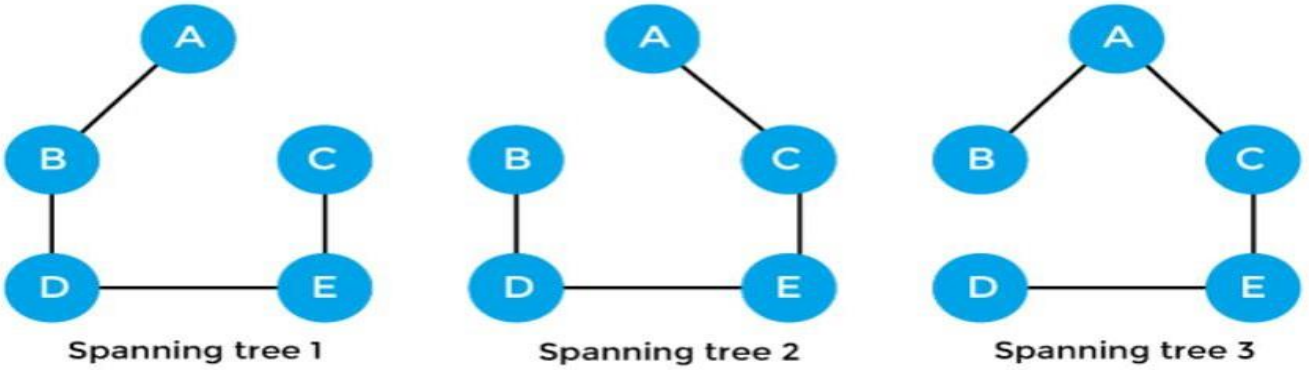
Example of Spanning Tree:

Now, let's walk through the below example and try to understand the spanning tree and how it works:

Let the real graph be:



The following are some examples of spanning trees that can be generated from the graph above:



Properties of Spanning Tree

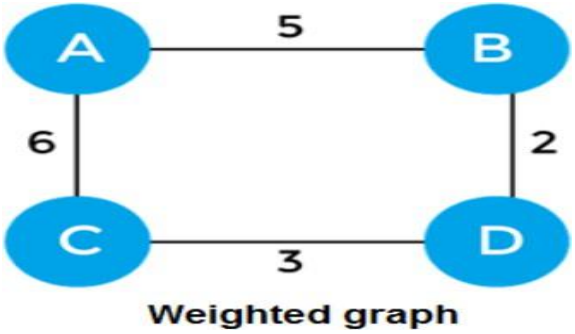
- There can be more than one spanning tree of a connected graph G.
- A spanning tree does not have any cycles or loop.
- A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
- A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
- There can be a maximum $\mathbf{n^{n-2}}$ number of spanning trees that can be created from a complete graph.
- A spanning tree has **n-1** edges, where 'n' is the number of nodes.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

Minimum Spanning tree

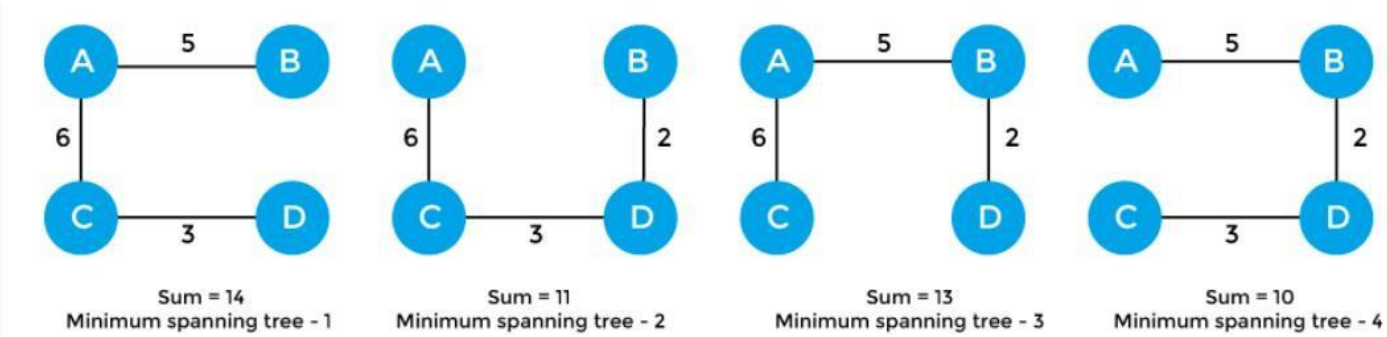
- A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum.
- The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.
- In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

Example of minimum spanning tree

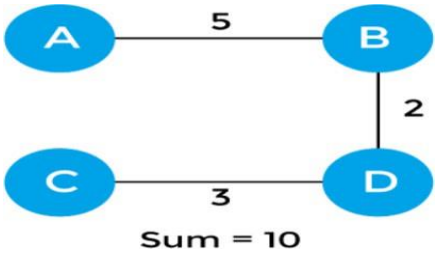
Let's understand the minimum spanning tree with the help of an example.



The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are –



So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is



Real-World Application of Minimum Spanning Tree

- The building or Connecting of roads among cities or villages at a minimal cost.
- Network service providers for finding the minimal cost to provide service to every user.
- Cluster Analysis of data sets
- Protocols to avoid network cycles
- Maximum bottleneck paths
- A real-time face tracking algorithm
- Dithering algorithms
- Prime Factorial

Algorithms for Minimum spanning tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

- Prim's Algorithm
- Kruskal's Algorithm

4. Kruskal's Algorithm:

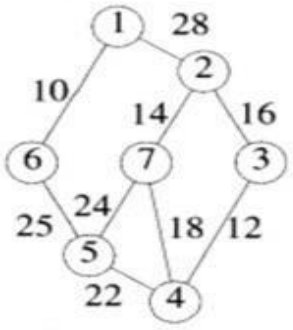
- Kruskal's algorithm is a realistic algorithm that provides the shorted path from one place to another in terms of edge weights or cost. This produces output exactly the same as prim's algorithm but the process or algorithm differs.
- It greedily picks n-1 edges one by one from the graph that has the minimum cost such that no cycle is created. So Kruskal's Algorithm takes up a connected and undirected graph and returns its minimum spanning tree.

Working Procedure:

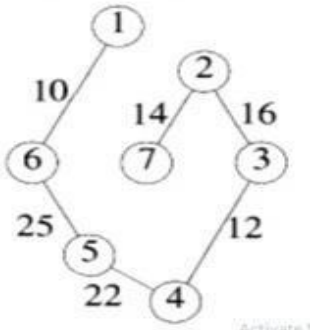
Kruskal's Algorithm: Add edges in increasing weight, skipping those whose addition would create a cycle.
How would we check if adding an edge {u, v} would create a cycle?

- ► Would create a cycle if u and v are already in the same component.
- ► We start with a component for each node.
- ► Components merge when we add an edge.
- ► Need a way to: check if u and v are in same component and to merge two components into one.

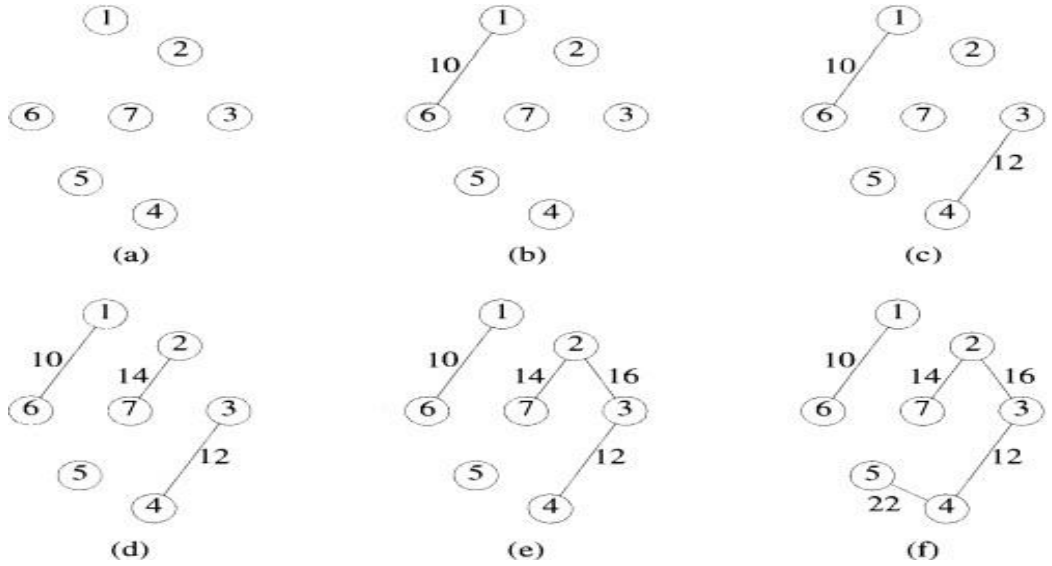
Example Graph:



Graph



Resultant Graph



Pseudo Code:

```
1   Algorithm Kruskal(E, cost, n, t)
2   // E is the set of edges in G. G has n vertices. cost[u, v] is the
3   // cost of edge (u, v). t is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for i := 1 to n do parent[i] := -1;
8       // Each vertex is in a different set.
9       i := 0; mincost := 0.0;
10      while ((i < n - 1) and (heap not empty)) do
11      {
12          Delete a minimum cost edge (u, v) from the heap
13          and reheapify using Adjust;
14          j := Find(u); k := Find(v);
15          if (j ≠ k) then
16          {
17              i := i + 1;
18              t[i, 1] := u; t[i, 2] := v;
19              mincost := mincost + cost[u, v];
20              Union(j, k);
21          }
22      }
23      if (i ≠ n - 1) then write ("No spanning tree");
24      else return mincost;
25  }
```

Algorithms for find and union

```
Find(v)   // path compression
   if parent[v] ≠ v
      parent[v] := Find(parent[v])
   return parent[v]

Union(a, b)   // union by rank
   ra := Find(a)
   rb := Find(b)
   if ra = rb
      return false        // already in same set; adding edge would form a cycle
   if rank[ra] < rank[rb]
      parent[ra] := rb
   else if rank[ra] > rank[rb]
      parent[rb] := ra
   else
      parent[rb] := ra
      rank[ra] := rank[ra] + 1
   return true
```

Time Complexity: O(ElogE)+O(E*4*alpha), ElogE for sorting and E*4*alpha for findParent operation 'E' times

Space Complexity: O(N). Parent array+Rank Array

Java Program for Kruskal's algorithm: Kruskals_Algo.java

```
import java.util.*;

class Node
{
   private int u;
   private int v;
   private int weight;

   Node(int _u, int _v, int _w)
   {
      u = _u; v = _v; weight = _w;
   }
   int getV()
   {
      return v;
   }
   int getU()
   {
      return u;
   }
   int getWeight()
   {
      return weight;
      }
}

class SortComparator implements Comparator<Node>
{
   @Override
   public int compare(Node node1, Node node2)
   {
      if (node1.getWeight() < node2.getWeight())
         return -1;
      if (node1.getWeight() > node2.getWeight())
         return 1;
      return 0;


   }
}
```

```
class Kruskals_Algo
{
   private int findPar(int u, int parent[])
   {
      if(u==parent[u]) return u;
      return parent[u] = findPar(parent[u], parent);
   }
   private void union(int u, int v, int parent[], int rank[])
   {
      u = findPar(u, parent);
      v = findPar(v, parent);
      if(rank[u] < rank[v])
      {
         parent[u] = v;
      }
      else if(rank[v] < rank[u])
      {
         parent[v] = u;
      }
      else
      {
         parent[v] = u;
         rank[u]++;
      }
   }
   void KruskalAlgo(ArrayList<Node> adj, int N)
   {
      Collections.sort(adj, new SortComparator());
      int parent[] = new int[N];
      int rank[] = new int[N];

      for(int i = 0;i<N;i++)
      {
         parent[i] = i;
         rank[i] = 0;
      }

      int costMst = 0;
      ArrayList<Node> mst = new ArrayList<Node>();
      for(Node it: adj)
      {
         if(findPar(it.getU(), parent) != findPar(it.getV(), parent))
         {
```

```
            costMst += it.getWeight();
            mst.add(it);
            union(it.getU(), it.getV(), parent, rank);
          }
        }
        System.out.println("minimum cost is:"+costMst);
        System.out.println("Spanning Tree is");
        for(Node it: mst)
        {
          System.out.println(it.getU() + " - " +it.getV());
        }
    }
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);

        //show custom message
        System.out.println("Enter number of vertices: ");

        //store user entered value into variable v
       int  n = sc.nextInt();
        ArrayList<Node> adj = new ArrayList<Node>();

         for(int i = 0; i < n; i++)
         {
            System.out.println("Enter weight for edge");
            int x = sc.nextInt();

            System.out.println("Enter source value for edge");
            int y = sc.nextInt();

            System.out.println("Enter destination value for edge");
            int z= sc.nextInt();

              adj.add(new Node(x, y,z ));

        }

        Test obj = new Test();
        obj.KruskalAlgo(adj, n);

    }
}
```

input=

```
Enter number of vertices:
5
Enter weight for edge
0
Enter source value for edge
1
Enter destination value for edge
2
Enter weight for edge
0
Enter source value for edge
3
Enter destination value for edge
6
Enter weight for edge
1
Enter source value for edge
3
Enter destination value for edge
8
Enter weight for edge
1
Enter source value for edge
2
Enter destination value for edge
3
Enter weight for edge
1
Enter source value for edge
4
Enter destination value for edge
5
```

output=

```
minimum cost is:16
Spanning Tree is
0 - 1
1 - 2
1 - 4
0 - 3
```

5. Prim's Algorithm:

➢ Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

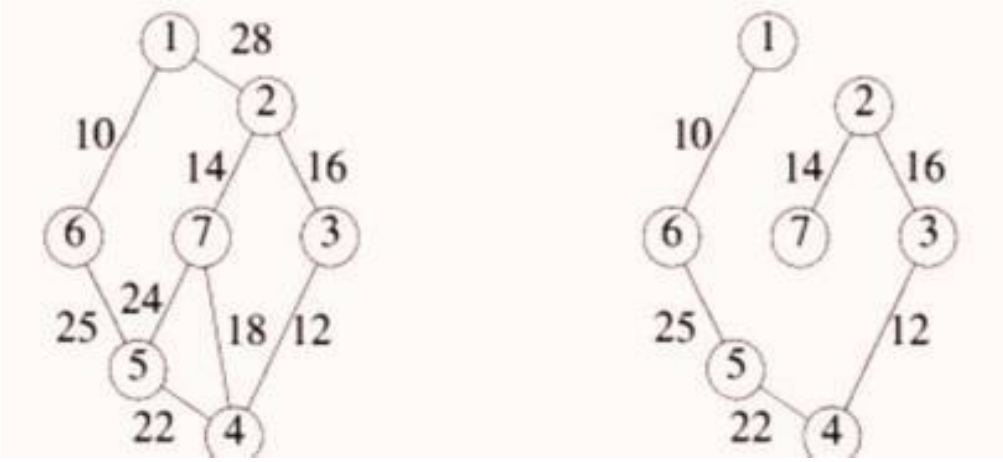
Step 1 - Remove all loops and parallel edges.

Step 2 - Choose any arbitrary node as root node.

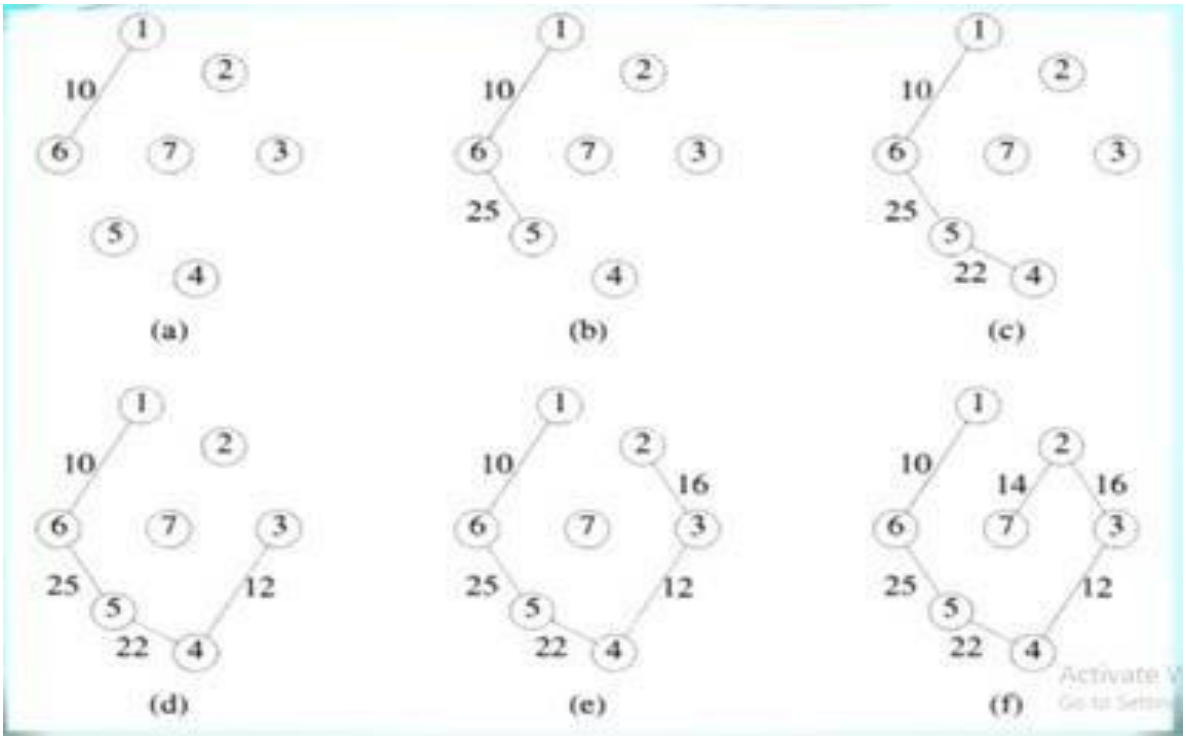
Step 3 - Check outgoing edges and select the one with less cost.

Step 4 - Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree

Step 5 - Keep repeating step 4 until we get a minimum spanning tree



Graph **Resultant Graph**



Pseudo Code:

```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```

How to implement Prim's Algorithm?

Follow the given steps to utilize the **Prim's Algorithm** mentioned above for finding MST of a graph:

- Create a set **mstSet** that keeps track of vertices already included in MST.
- Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.
- While **mstSet** doesn't include all vertices
    - Pick a vertex **u** that is not there in **mstSet** and has a minimum key value.
    - Include **u** in the **mstSet**.
    - Update the key value of all adjacent vertices of **u**. To update the key values, iterate through all adjacent vertices.

- For every adjacent vertex **v**, if the weight of edge **u-v** is less than the previous key value of **v**, update the key value as the weight of **u-v**.
- The idea of using key values is to pick the minimum weight edge from the cut. The key values are used only for vertices that are not yet included in MST, the key value for these vertices indicates the minimum weight edges connecting them to the set of vertices included in MST.

Time Complexity:

Data structure used for the minimum edge weight	**Time Complexity**
Adjacency matrix, linear searching	$O(\|V\|^2)$
Adjacency list and binary heap	**O(|E| log |V|)**
Adjacency list and Fibonacci heap	O(|E|+ |V| log |V|)

Java Program for Prims algorithm using Adjacency List: Prims_Algo.java

```
import java.io.*;
import java.lang.*;
import java.util.*;

class Prims_Algo
{
   // Number of vertices in the graph
   private static final int V = 5;

   // A utility function to find the vertex with minimum
   // key value, from the set of vertices not yet included
   // in MST
   int minKey(int key[], Boolean mstSet[],int V)
   {
      // Initialize min value
      int min = Integer.MAX_VALUE, min_index = -1;

      for (int v = 0; v < V; v++)
         if (mstSet[v] == false && key[v] < min)
          {
            min = key[v];
            min_index = v;
          }
      return min_index;
   }

   // A utility function to print the constructed MST  stored in parent[]
```

```
    void printMST(int parent[], int graph[][],int V)
    {
       System.out.println("Edge \tWeight");
       for (int i = 1; i < V; i++)
          System.out.println(parent[i] + " - " + i + "\t"  + graph[i][parent[i]]);
    }

   // Function to construct and print MST for a graph/ represented using adjacency matrix
  //representation

    void primMST(int graph[][],int x)
    {
      int V=x;
       // Array to store constructed MST
       int parent[] = new int[V];

       // Key values used to pick minimum weight edge in  cut
       int key[] = new int[V];

       // To represent set of vertices included in MST
       Boolean mstSet[] = new Boolean[V];

       // Initialize all keys as INFINITE
       for (int i = 0; i < V; i++)
       {
          key[i] = Integer.MAX_VALUE;
          mstSet[i] = false;
       }

       // Always include first 1st vertex in MST. Make key 0 so that this vertex is
       // picked as first vertex
       key[0] = 0;

       // First node is always root of MST
       parent[0] = -1;

       // The MST will have V vertices
       for (int count = 0; count < V - 1; count++)
         {
          // Pick the minimum key vertex from the set of vertices not yet included in MST

          int u = minKey(key, mstSet,V);

          // Add the picked vertex to the MST Set
```

```
            mstSet[u] = true;

            // Update key value and parent index of the adjacent vertices of the picked vertex.
            // Consider only those vertices which are not yet included in MST

             for (int v = 0; v < V; v++)

               // graph[u][v] is non zero only for adjacent  vertices of m mstSet[v] is false for
               // vertices not yet included in MST Update   the key only if graph[u][v] is smaller
               // than key[v]
               if (graph[u][v] != 0 && mstSet[v] == false && graph[u][v] < key[v])
                {
                   parent[v] = u;
                   key[v] = graph[u][v];
                }
          }

          // Print the constructed MST
          printMST(parent, graph,V);
      }
    public static void main(String[] args)
    {
         Prims_Algo g = new Prims_Algo();
        Scanner sc=new Scanner(System.in);
        System.out.println("enter number vertices");

        int  V=sc.nextInt();
        System.out.println("enter row  size of the matrix");
        int x=sc.nextInt();
        System.out.println("enter column  size of the matrix");
        int y=sc.nextInt();

        int G[][]= new int[x][y];
         System.out.println("adjacency matrix is");
      for (int i=0;i<x;i++)
      {
         for(int j=0;j<y;j++)
         {
            G[i][j]=sc.nextInt();
         }
      }

      g. primMST (G, V);
    }
}
```

input=

```
enter number vertices
5
enter row  size of the matrix
5
enter column  size of the matrix
5
adjacency matrix is
0 9 75 0 0
9 0 95 19 42
75 95 0 51 66
0 19 51 0 31
0 42 66 31 0
```

output=

```
Edge : Weight
0 - 1 :  9
1 - 3 :  19
3 - 4 :  31
3 - 2 :  51
```

6. Single source shortest path Problem:

The **Single Source Shortest Path** (SSSP) problem is about finding the shortest path from a single source node to all other nodes in a weighted graph. The most famous algorithm to solve this problem using a **greedy** approach is **Dijkstra's Algorithm**.

Dijkstra's Algorithm Overview:

Dijkstra's algorithm works for graphs with non-negative weights and follows a greedy approach by selecting the closest (smallest tentative distance) node that hasn't been processed yet. It keeps expanding the path from the source node, ensuring the shortest possible paths are calculated for all nodes.

Steps:

1. **Initialization**:

   - Set the distance to the source node as 0 (since the distance from the source to itself is 0).
   - Set the distance to all other nodes as infinity (since they are initially unreachable).
   - Create a set of all unprocessed nodes.

2. **Greedy Selection**:

   - Choose the unprocessed node with the smallest known distance from the source. This node is considered processed (the shortest path to it is finalized).

3. **Relaxation:**

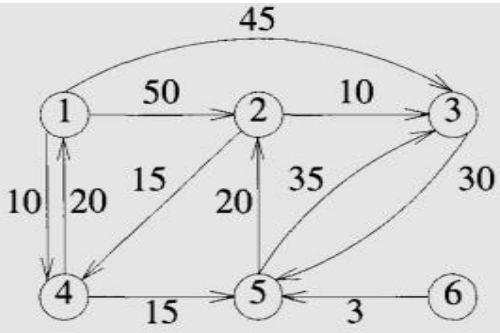
   - For each neighboring node of the currently processed node, calculate its tentative distance from the source through the current node.
   - If this new distance is smaller than the previously recorded distance, update the distance for the neighboring node.

4. **Repeat:**

   - Repeat steps 2 and 3 until all nodes have been processed or there are no more nodes with a finite distance.

5. **Result:**

   - After the algorithm finishes, you'll have the shortest path from the source to every other node.



(a) Graph

	Path	Length
1)	1, 4	10
2)	1, 4, 5	25
3)	1, 4, 5, 2	45
4)	1, 3	45

(b) Shortest paths from 1

Pseudo Code:

```
1   Algorithm ShortestPaths(v, cost, dist, n)
2   // dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
3   // path from vertex v to vertex j in a digraph G with n
4   // vertices. dist[v] is set to zero. G is represented by its
5   // cost adjacency matrix cost[1 : n, 1 : n].
6   {
7       for i := 1 to n do
8       { // Initialize S.
9           S[i] := false; dist[i] := cost[v, i];
10      }
11      S[v] := true; dist[v] := 0.0; // Put v in S.
12      for num := 2 to n − 1 do
13      {
14          // Determine n − 1 paths from v.
15          Choose u from among those vertices not
16          in S such that dist[u] is minimum;
17          S[u] := true; // Put u in S.
18          for (each w adjacent to u with S[w] = false) do
19              // Update distances.
20              if (dist[w] > dist[u] + cost[u, w])) then
21                  dist[w] := dist[u] + cost[u, w];
22      }
23  }
```

Time Complexity:

- Using a simple implementation with an array: **$O(V^2)$**, where V is the number of vertices.
- Using a priority queue (min-heap): **$O((V + E) \log V)$**, where V is the number of vertices and E is the number of edges.

Java Program for Single source shortest path (Dijkstra's Algorithm):

Dijkstra's_Shortestpath_algo.java

```
import java.util.*;
import java.io.*;
import java.lang.*;
import java.util.*;

class Dijkstra's_Shortestpath_algo
{
   // A utility function to find the vertex with minimum distance value, from the set of
  // vertices not yet  included in shortest path tree

 static final int V = 9;
   int minDistance(int dist[], Boolean sptSet[])
   {
      // Initialize min value
      int min = Integer.MAX_VALUE, min_index = -1;

      for (int v = 0; v < V; v++)
         if (sptSet[v] == false && dist[v] <= min) {
            min = dist[v];
            min_index = v;
         }

      return min_index;
   }

   // A utility function to print the constructed distance
   // array
   void printSolution(int dist[])
   {
      System.out.println(
         "Vertex \t\t Distance from Source");
      for (int i = 0; i < V; i++)
         System.out.println(i + " \t\t " + dist[i]);
   }

  // Function that implements Dijkstra's single source  shortest path algorithm for a graph      represented
using  adjacency matrix representation



void dijkstra(int graph[][], int src)
```

```
{
    int dist[] = new int[V]; // The output array.
  //dist[i] will hold the shortest distance from src to i

    // sptSet[i] will true if vertex i is included in
// shortest path tree or shortest distance from src     To i is finalized

    Boolean sptSet[] = new Boolean[V];

    // Initialize all distances as INFINITE and stpSet[]  as false
    for (int i = 0; i < V; i++) {
       dist[i] = Integer.MAX_VALUE;
       sptSet[i] = false;
    }

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++)
     {
       // Pick the minimum distance vertex from the set of vertices not yet processed.
      //u is always equal to src in first iteration.
       int u = minDistance(dist, sptSet);

       // Mark the picked vertex as processed
       sptSet[u] = true;

       // Update dist value of the adjacent vertices of  the picked vertex.

   for (int v = 0; v < V; v++)

          // Update dist[v] only if is not in sptSet, there is an edge from u to v, and total
          // weight of path from src to v through u is  smaller than current value of dist[v]

if (!sptSet[v] && graph[u][v] != 0  && dist[u] != Integer.MAX_VALUE && dist[u] + graph[u][v] <
dist[v])
            {
                    dist[v] = dist[u] + graph[u][v];
            }
    }

    // print the constructed distance array
    printSolution(dist);
}
```

```
    public static void main(String[] args)
    {
       Scanner sc=new Scanner(System.in);
          System.out.println("enter row  size of the matrix");
          int x=sc.nextInt();
          System.out.println("enter column  size of the matrix");
          int y=sc.nextInt();

          int graph[][]= new int[x][y];

       System.out.println("adjacency matrix is");
       for (int i=0;i<x;i++)
       {
          for(int j=0;j<y;j++)
          {
             graph[i][j]=sc.nextInt();
          }
       }

  Dijkstra's_Shortestpath_algo t = new Dijkstra's_Shortestpath_algo ();

          // Function call
          t.dijkstra(graph, 0);
    }
  }
```

Input:

```
enter row  size of the matrix
9
enter column  size of the matrix
9
adjacency matrix is
0 4 0 0 0 0 0 8 0
4 0 8 0 0 0 1 1 0
0 8 0 7 0 4 0 0 2
0 0 7 0 9 1 4 0 0
0 0 0 9 0 1 0 0 0
0 0 4 14 10 0 2 0 0
0 0 0 0 0 2 0 1 6
8 11 0 0 0 0 1 0 7
0 0 2 0 0 0 6 7 0
```

Output:

Vertex	Distance from Source
0	0
1	4
2	11
3	18
4	17
5	7
6	5
7	5
8	11