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.

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.

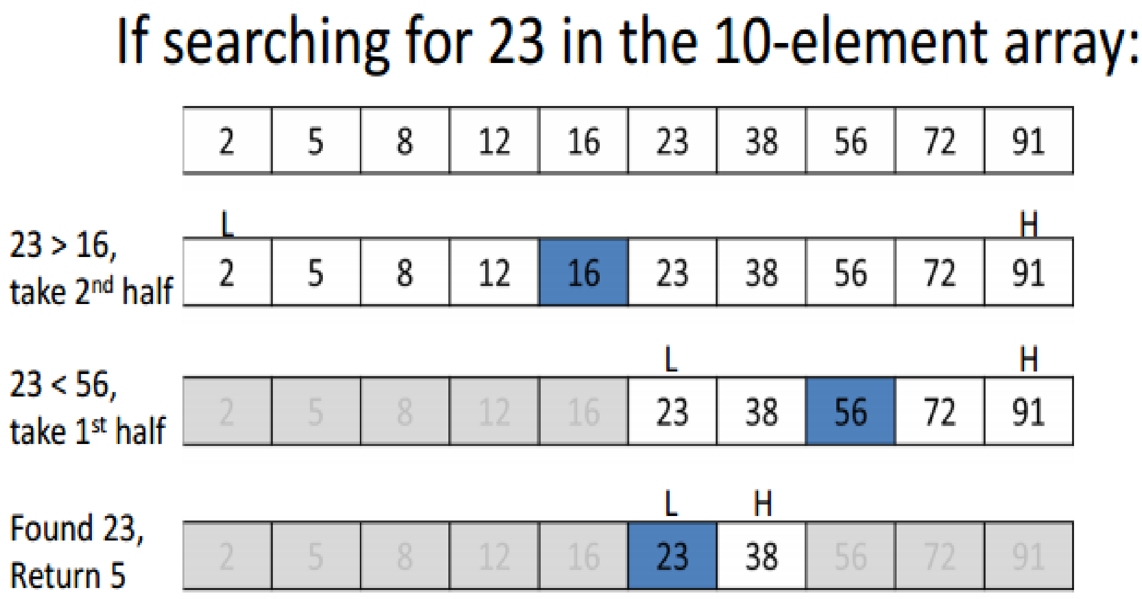
I. Binary Search

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)


If searching for 23 in the 10-element array:
2 5 8 12 16 23 38 56 72 91
L H
23 > 16, take 2nd half
2 5 8 12 16 23 38 56 72 91
L H
23 < 56, take 1st half
2 5 8 12 16 23 38 56 72 91
L H
Found 23, Return 5
2 5 8 12 16 23 38 56 72 91

Working:

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

1. Recursive Method
2. Iterative Method

1. Iterative Algorithm (Divide and Conquer Technique)

```
// a is an array of size n, x is the key element to be searched.
Algorithm BinSearch(a, n, x)
{
    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;
}
```

2. Recursive Algorithm (Divide and Conquer Technique)

```
/*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.*/
Algorithm BinSrch (a, low, high, x)
{
    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);
     }
}
```

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.

$T(n) = c_1$ n=1, c1 is a constant
$T(n/2) + c_2$ n>1, c2 is a constant

$$
\begin{aligned}
T(n) &= T(n/2) + c2 \\
&= T(n/4)+c2+c2 \\
&= T(n/8) +c2+c2+c2 \\
&\ldots.. \\
&\ldots.. \\
&= T(n / 2^k) + c2+c2+c2+ \ldots\ldots\ldots..k \text{ times} \\
&= T(1)+ kc2 \\
&= c1+kc2 =c1+ \log n*c2 = O(\log n)
\end{aligned}
$$

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 ele, 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] == ele)
				return mid;
			else if (array[mid] < ele)
				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();
		}
		// Sorting the array
		Arrays.sort(array);
		// Printing the array after sorting
		System.out.println("Sorted array[]: "+ Arrays.toString(array));
		System.out.println("Enter the search 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);
	}
}
```

Test Cases:

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 search key
59
Element 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 the search 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 = (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();
		}

		Arrays.sort(array);
		// Printing the array after sorting
		System.out.println("Sorted array[]: "+ Arrays.toString(array));
		System.out.println("Enter the search 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);
	}
}
```

Test Cases:

Case=1

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

Case=2

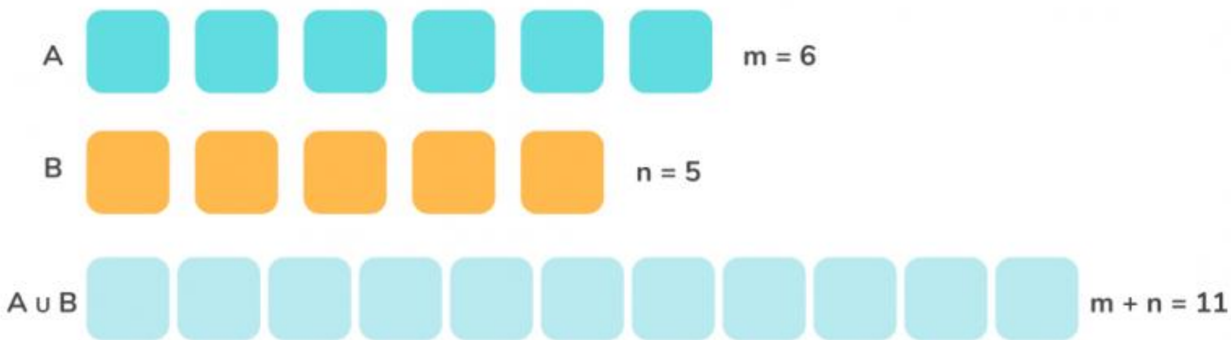
Enter array size5
Enter the elements of array 15
35
25
95
65
Sorted array[]:[15, 25, 35, 65, 95]
Enter the search key
30
Element not found

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

Constraints:

- **nums1.length == m**
- **nums2.length == n**
- **0 <= m <= 1000**
- **0 <= n <= 1000**
- **1 <= m + n <= 2000**
- **-106 <= nums1[i], nums2[i] <= 106**

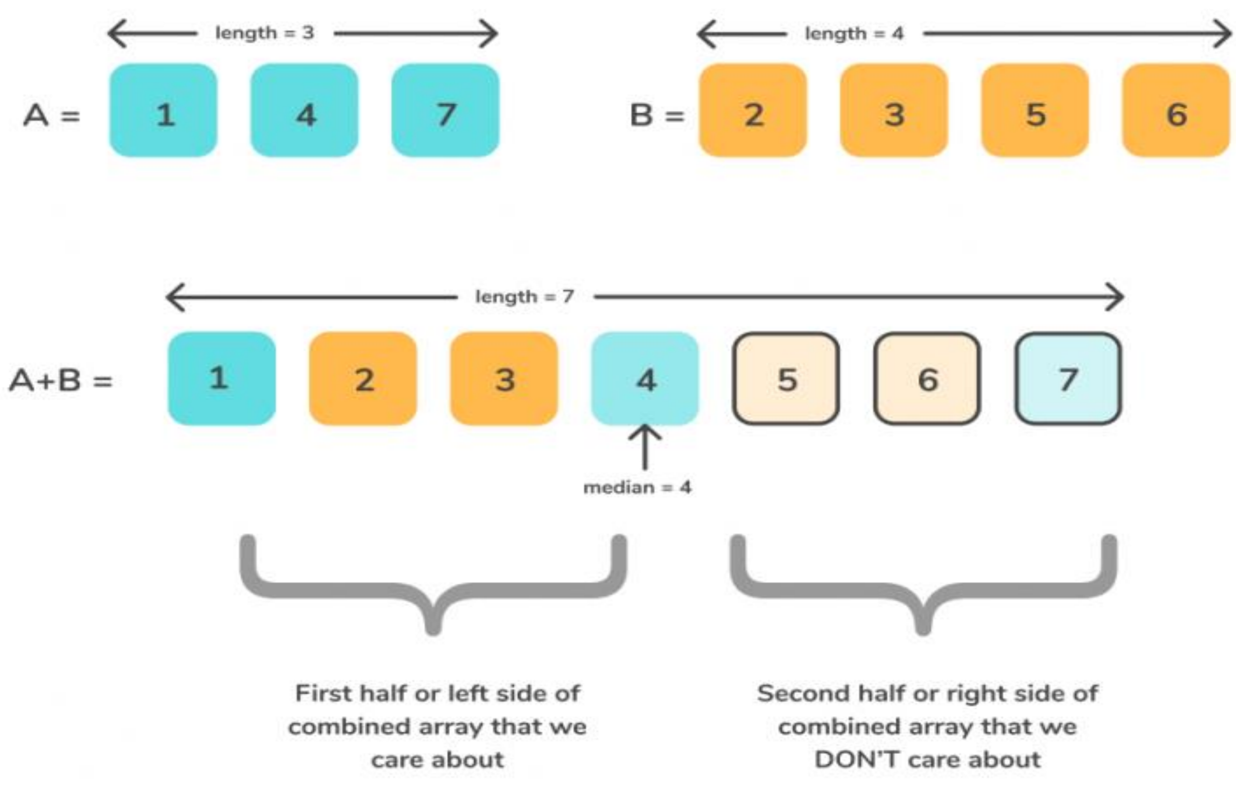
The overall run time complexity should be O(log (m+n)).

Using Binary search:

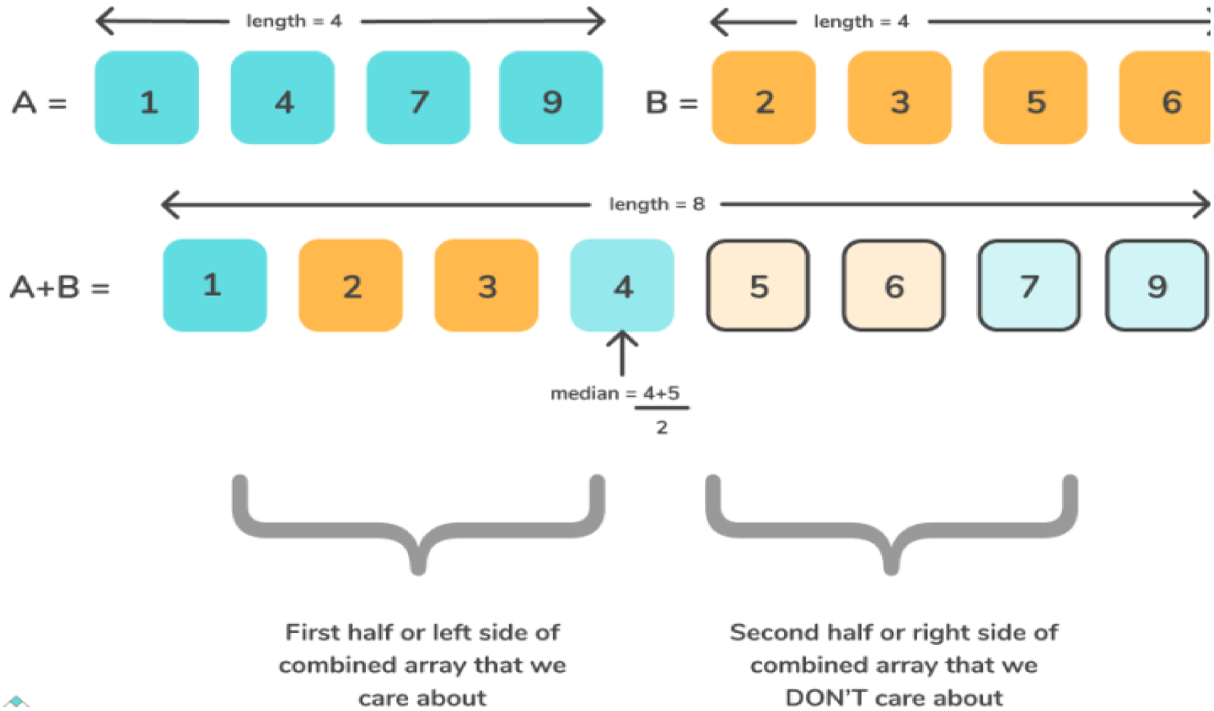
The key idea to note here is that both the arrays are **sorted**. Therefore, this leads us to think of **binary search**. Let us try to understand the algorithm using an example:

A[] = {1, 4, 7}

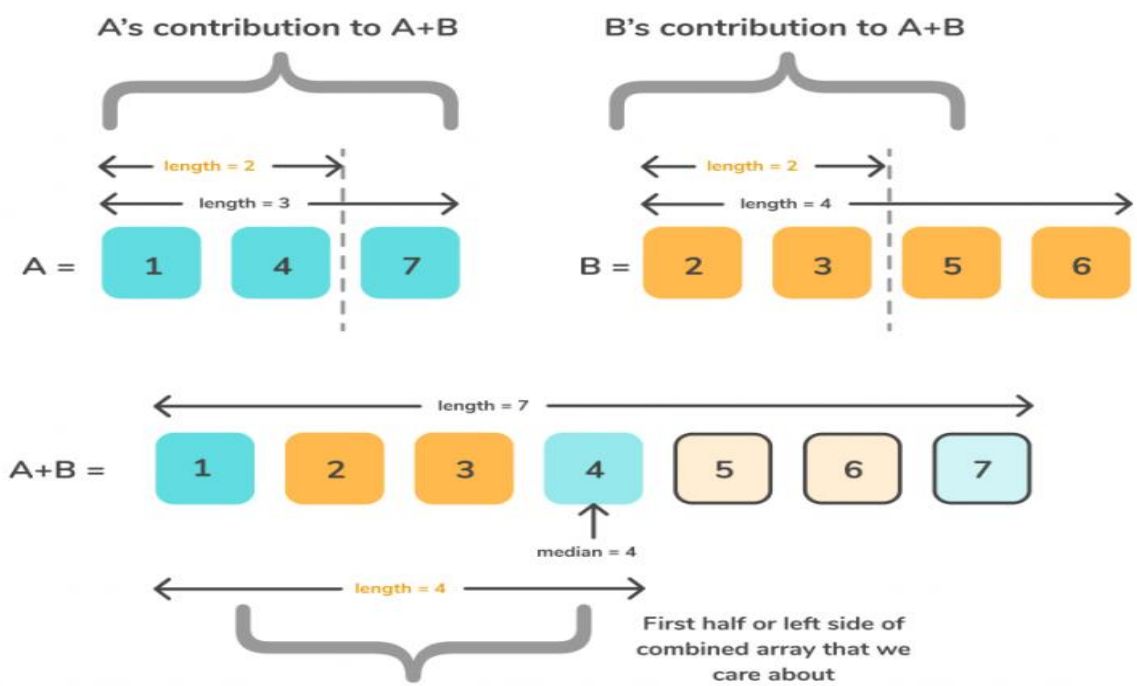
B[] = {2, 3, 5, 6}



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 discarded.

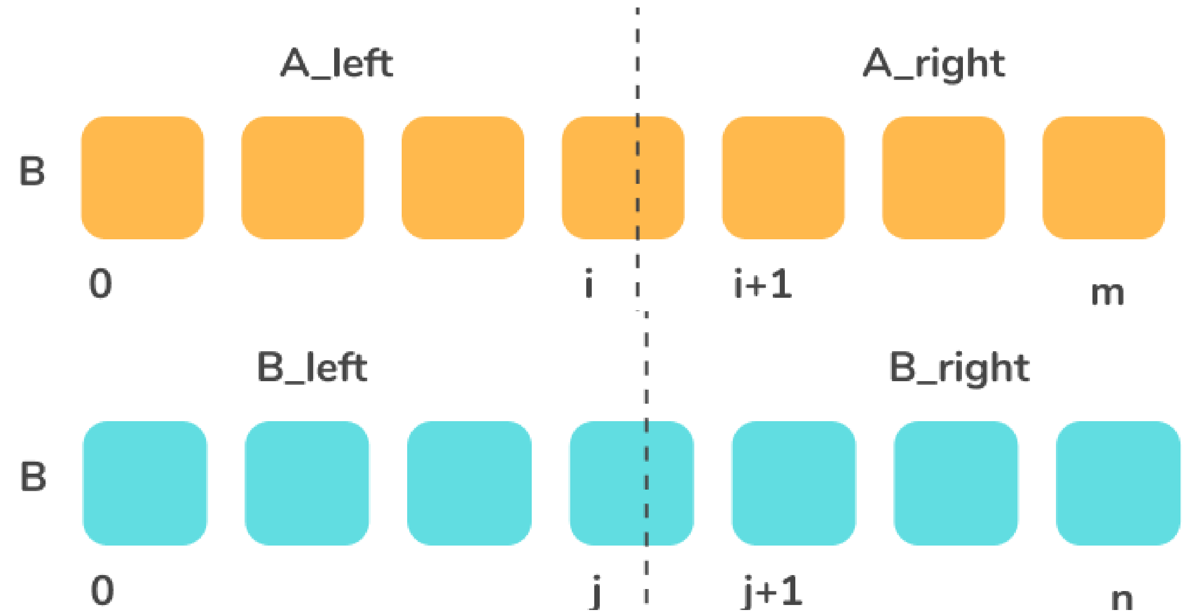


- Similarly, for an even length merged array, ignore the **right half** and only the **left half** contributes to our final answer.
- Therefore, the motive of our approach is to find which of the elements from both the array helps in contributing to the final answer. Therefore, the **binary search** comes to the rescue, as it can discard a part of the array every time, the elements don't contribute to the median.

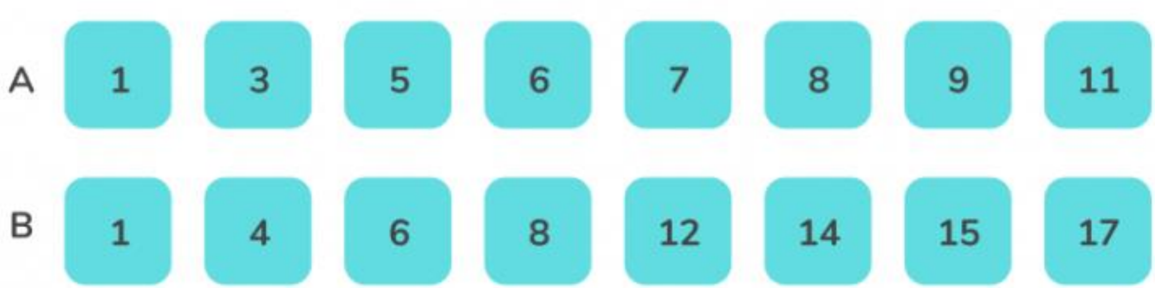


Algorithm

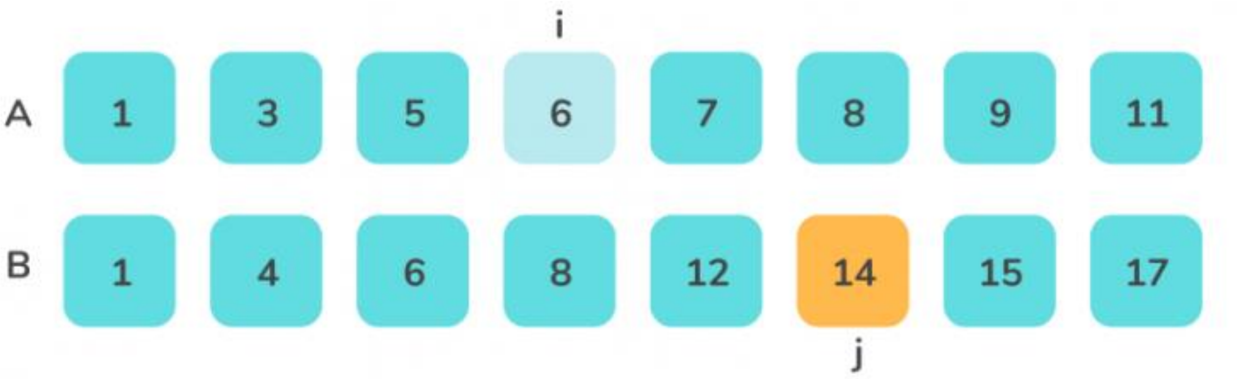
- The first array is of size **n**, hence it can be split into **n + 1** parts.
- The second array is of size **m**, hence it can be split into **m + 1** parts



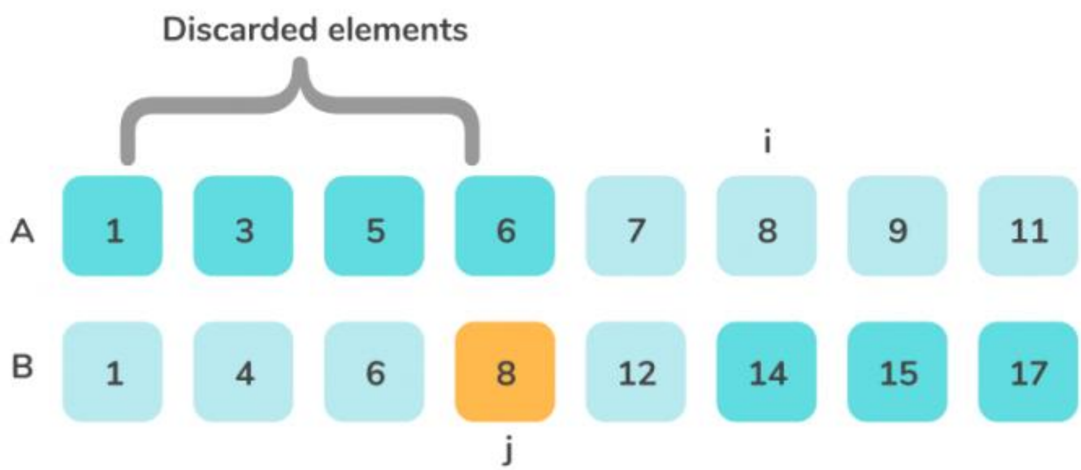
- As discussed earlier, we just need to find the elements contributing to the left half of the array.
  - Since, the arrays are already sorted, it can be deduced that
    A[i – 1] < A[i] and **B[i – 1] < B[i]**.
  - Therefore, we just need to find the index **i**, such that
    A[i – 1] <= B[j] and **B[j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



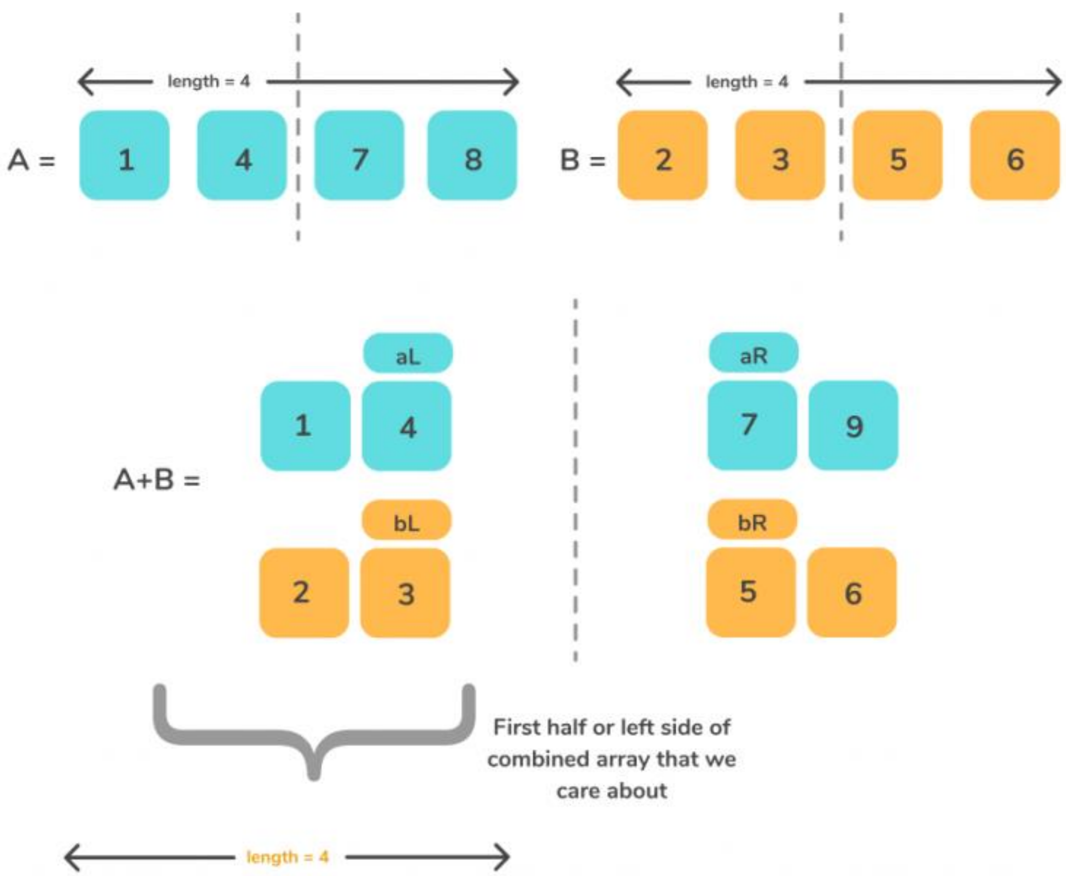
Max of left side will be either A[i-1] or B[j-1], in this case its A[i-1] = 7

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 MedianOfTwoSortedArray(Binary search approach)

MedianSortedArrays.java

```
import java.util.*;
class MedianSortedArrays
{
	public double findMedianSortedArrays(int[] arr1, int[] arr2)
	{
		// Swap the arrays if arr1 length is greater than arr2 length
		if(arr1.length > arr2.length)
			findMedianSortedArrays(arr2, arr1);
		int x = arr1.length;
		int y = arr2.length;
		int low = 0;
		int high = x;


		while(low <= high)
		{
		  int partitionX = (low + high)/2;
		  int partitionY = (x + y + 1)/2 - partitionX;

		// If partitionX = 0, it means nothing is there on left side. Use -INF for max
		// Edge cases:
		// If partitionX = 0, it means nothing is there on left side. Use -INF for maxLeftX
		// If partitionY = length of input, it means nothing is there on right side. Use
		                                                        +INF for minRightX
		  int maxLeftX = (partitionX == 0) ? Integer.MIN_VALUE : arr1[partitionX - 1];
		  int minRightX = (partitionX == x) ? Integer.MAX_VALUE : arr1[partitionX];
		  int maxLeftY = (partitionY == 0) ? Integer.MIN_VALUE : arr2[partitionY - 1];
		  int minRightY = (partitionY == y) ? Integer.MAX_VALUE : arr2[partitionY];
		  if(maxLeftX <= minRightY && maxLeftY <= minRightX)
		  {
			// Array partitioning is @ correct place
			// Now get max of left elements and min of right elements to get the median
			// in case of even length combined array size or
			// get max of left for odd length combined array size
			if((x + y) %2 == 0)
			{
				return ((double)(Math.max(maxLeftX, maxLeftY) +
						Math.min(minRightX, minRightY)))/2;
			}
			else
			{
				return (double)Math.max(maxLeftX, maxLeftY);
			}
		  }
```

```
                    else if(maxLeftX > minRightY)
                    // We are too far on right side for partitionX. Need to move left
                    {
                            high = partitionX - 1;
                    }
                    else    // We are too far on left side for partitionX. Need to move right
                    {
                            low = partitionX + 1;
                    }
                }
                return -1;
        }
        public static void main(String[] args)
        {
                Scanner sc=new Scanner(System.in);
                int m=sc.nextInt();
                int n=sc.nextInt();
                int[] A= new int[m];
                int[] B=new int[n];
                for(int i=0;i<m;i++)
                        A[i]= sc.nextInt();
                for(int i=0;i<n;i++)
                        B[i]= sc.nextInt();
                System.out.println( new MedianSortedArrays().findMedianSortedArrays(A,B) );
        }
}
```

Sample Input = 4 6
32 45 70 95
40 50 59 67 73 84
Sample Output = 63.0

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: **Fixedpoint.java**

```
import java.util.*;
class FixedPoint
{
	public int fixedPoint(int[] A)
	{
		int lft = 0, rt = A.length - 1;
		while (lft < rt)
		{
			int mid = (lft + rt) / 2;
			if (A[mid] - mid < 0)
				lft = mid + 1;
			else
				rt = mid;
		}
		return A[lft] == lft ? lft : -1;
	}
	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();
		Arrays.sort(arr);
		// Printing the array after sorting
		System.out.println("sorted array[]:"+ Arrays.toString(arr));
		System.out.println(new FixedPoint().fixedPoint(arr));
	}
}
```

Test Case1:

```
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]
Fixed Point is 3
```

Test Case2:

enter array size
6
enter the elements of array
3
9
4
7
-5
-10
sorted array[]:[-10, -5, 3, 4, 7, 9]
Fixed Point is -1

Test Case3:

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

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

8 and 1 are present in all rows.

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 oflinear search.

Java program for Find the Smallest Common Element:

SmallestCommonElement.java

```
import java.util.*;
class SmallestCommonElement
{
	private static 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 static int smallestCommonElement(int[][] mat)
        {
                if(mat.length == 1)
                    return mat[0][0];
        // Get each element of array 1. Compare with each element of remaining array elements
                for(int ele : mat[0])
                {
                    int count = 1;
                    for(int i = 1; i < mat.length; i++)
                    {
                        if(binarySearch(mat[i], 0, mat[i].length-1, ele))
                            count++;
                        else
                            break;
                    }


                    if(count == mat.length)
                        return ele;
                }
                return -1;
        }
        public static void main(String[] args)
        {
                Scanner sc=new Scanner(System.in);
                int m=sc.nextInt();
                int n=sc.nextInt();
                int[][] arr = new int[m][n];
                for(int i=0;i<m;i++)
                    for(int j=0;j<n;j++)
                        arr[i][j] = sc.nextInt();
                System.out.println(smallestCommonElement(arr));
        }
}
```

Sample Test Cases :

Case 1:

Input :=
4 5
1 2 3 4 5

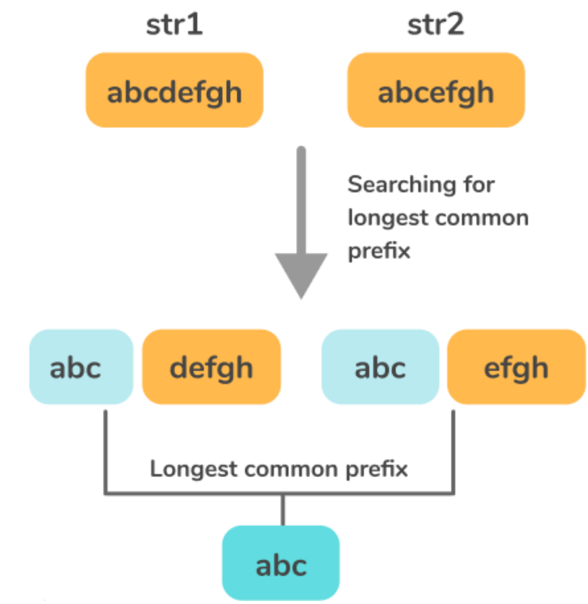
5 6 7 8 9
4 5 6 7 8
2 3 4 5 6
output :=5

Case 2:
input :=
5 4
10 30 45 46
1 2 10 20
10 12 13 14
2 3 4 10
7 8 9 10
output =10

4. Longest Common Prefix

Problem Statement:

- 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 upto 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

LCP_BS.java

```
/*
Time complexity: O(NM log M)
N = Number of strings
M = Length of the longest string

Space complexity:
O(M)
M = Length of the longest string
*/
import java.util.*;
class LCP_BS
{
	// A Function to find the string having the minimum length
	static int findMinLength(String arr[], int n)
	{
		int min = Integer.MAX_VALUE;
		for (int i = 0; i < n; i++)
		{
			if (arr[i].length() < min)
			{
				min = arr[i].length();
			}
		}
		return min;
	}


	public static String longestCommonPrefix(String[] strs, int n)
	{
		if (strs == null || strs.length == 0) return "";

		int index = findMinLength(strs, n);
		String prefix;
		int prefixLen = -1;

		int low = 0, high = index;
		while (low <= high)
		{
			int mid = (low + high) / 2;
			if(isCommon(strs, mid))
			{
				low = mid+1;
				prefixLen = mid;
			}
			else
```

```
                {
                        high = mid-1;
                }
        }
        prefix = strs[0].substring(0, prefixLen);
        return prefix.toString();
    }

    public static boolean isCommon(String[] str, int len)
    {
        String pre = str[0].substring(0, len);
        for(int i = 1; i < str.length; i++)
        {
                if(!str[i].startsWith(pre))
                {
                        return false;
                }
        }
        return true;
    }

    public static void main(String args[])
    {
        // hello hell her he
        // genesis general generic
        Scanner sc= new Scanner(System.in);
        String[] words = sc.nextLine().split(" ");
        System.out.println(longestCommonPrefix(words, words.length));
    }
}
```

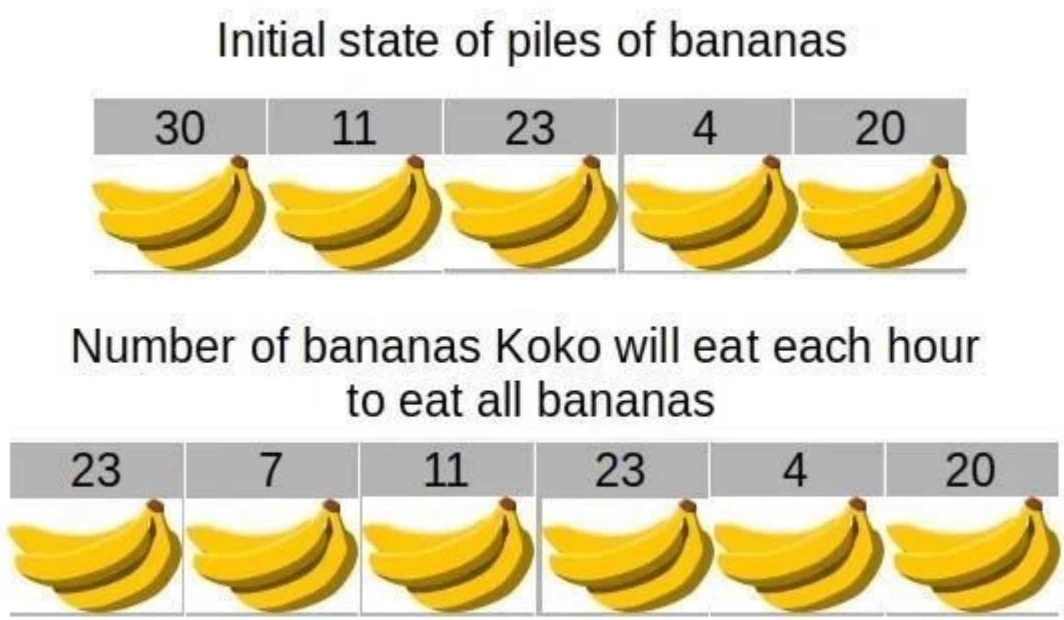
input:
flower flow flight
output: fl

5. Koko Eating Bananas:

- In the problem " Koko Eating Bananas" we are given an array of size n which contains the number of bananas in each pile. In one hour Koko can eat at most K bananas. If the pile contains less than K bananas in that case if Koko finishes all bananas of that pile then she cannot start eating bananas from another pile in the same hour.
- Koko wants to eat all bananas within H hours. We are supposed to find the minimum value of K.

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.

<u>Java program for Kokoeatingbananas</u>: KokoEatingBananas_BS.java

```
import java.util.*;
public class KokoEatingBananas_BS
{
	public static int minEatingSpeed(int[] piles, int hrs)
	{
		// Initalize the left and right boundaries
		int left = 1, right = 1;
		for (int pile : piles)
		{
			right = Math.max(right, pile);
		}

		while (left < right)
		{
			// Get the middle index between left and right boundary indexes.
			// hourSpent stands for the total hour Koko spends.
			int middle = (left + right) / 2;
			int hourSpent = 0;
```

```
                // Iterate over the piles and calculate hourSpent.
                // We increase the hourSpent by ceil(pile / middle)
                for (int pile : piles)
                {
                        hourSpent += Math.ceil((double) pile / middle);
                }

            // Check if middle is a workable speed, and cut the search space by half.
                if (hourSpent <= hrs)
                {
                        right = middle;
                }
                else
                {
                        left = middle + 1;
                }
            }

            // Once the left and right boundaries coincide, we find the target value,
            // that is, the minimum workable eating speed.
            return right;
        }

        public static void main(String args[])
        {
            Scanner sc = new Scanner(System.in);
            String [] str = sc.nextLine().split(" ");

            int[] boxes = new int[str.length];
            for (int i = 0; i < str.length; i++)
            {
                boxes[i] = Integer.valueOf(str[i]);
            }

            int hours = sc.nextInt();
            System.out.println(minEatingSpeed(boxes, hours));
        }
}
```

Sample Input-1:
----
4 8 9 13
8

Sample Output-1:

5

Sample Input-2:
----
15 18 12 17 22
7

Sample Output-2:

17

GREEDY METHOD

Optimization problem is a problem which requires minimum results or maximum results.

Greedy method is used for solving optimization problem.

Greedy method says that, our problem should be solved in **stages**.

In each **stage** we consider **one input** for given problem. If that input is feasible, then that will be included in the solution.

By including all those feasible inputs, we will get an optimal solution.

1. Fractional Knapsack Problem

Given the weights and profits of **N** items, in the form of **{profit, weight}** put these items in a knapsack of capacity **W** to get the maximum total profit in the knapsack.

In **Fractional Knapsack**, we can break items for maximizing the total value of the knapsack.

The basic idea of the greedy approach is to calculate the ratio profit/weight for each item and sort the item on the basis of this ratio.

Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it).

This will always give the maximum profit because, in each step it adds an element such that this is the maximum possible profit for that much weight.

Algorithm:

1. Input profits and weights of N items
2. Calculate the ratio (**profit/weight**) for each item.
3. Sort all the items in decreasing order of the ratio.
4. Initialize **res = 0**, curr_cap = given_cap.
   [Here, curr_cap is current capacity, given_cap is given capacity]
5. Do the following for every item "**I**" in the sorted order:
   5.1. If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result
   5.2. Else add the current item as much as we can and break out of the loop.
6. Return **res**.

Example:

Consider the example: **arr[] = [[100, 20], [60, 10], [120, 30]], W = 50.**

The sorted array will be **{{60, 10}, {100, 20}, {120, 30}}.**

[Initially sort the array based on the profit/weight ratio.]

Iteration:

- For i = 0, weight = 10 which is less than W. So, add this element in the knapsack.
  profit = 60 and remaining W = 50 – 10 = 40.
- For i = 1, weight = 20 which is less than W. So, add this element too.
  profit = 60 + 100 = 160 and remaining W = 40 – 20 = 20.
- For i = 2, weight = 30 is greater than W. So, add 20/30 fraction = 2/3 fraction of the element.
  Therefore profit = 2/3 * 120 + 160 = 80 + 160 = 240 and remaining W becomes 0.

Time Complexity: O(N * logN)

Auxiliary Space: O(N)

```
// Java program to solve fractional Knapsack Problem
import java.lang.*;
import java.util.*;
public class FractionalKnapSack
{
	// Function to get maximum value
	private static double getMaxValue(int[][] arr, int capacity)
	{
		// Sorting items by profit/weight ratio;
		Arrays.sort(arr, new Comparator<int[]>()
		{
			@Override
			public int compare(int[] item1, int[] item2)
			{
				double cpr1	= ((double)item1[0] / item1[1]);
				double cpr2	= ((double)item2[0] / item2[1]);

				if (cpr1 < cpr2)
					return 1;
				else
					return -1;
```

```
                }
            });
            double totalProfit = 0d;
            for (int[] i : arr)
            {
                int curProf = i[0];
                int curWt = i[1];
                if (capacity - curWt >= 0)
                {
                    // This weight can be picked whole
                    capacity = capacity - curWt;
                    totalProfit += curProf;
                }
                else
                {
                    // Partial weight can be picked (can not pick whole weight)
                    double fraction = ((double)capacity / (double)curWt);
                    totalProfit += (curProf * fraction);
                    capacity = (int)(capacity - (curWt * fraction));
                    break;
                }
            }
            return totalProfit;
        }
        public static void main(String[] args)
        {
            Scanner sc = new Scanner(System.in);
            int N = sc.nextInt();
            int[][] arr = new int[N][2];
            for (int i=0;i<N ;i++ )
            {
                arr[i][0] = sc.nextInt();
                arr[i][1] = sc.nextInt();
            }
            int capacity = sc.nextInt();
            double maxValue = getMaxValue(arr, capacity);
            System.out.println(maxValue);
        }
}
```

Sample Test Cases:
TC1:
Input:
4
2 1
4 3
7 5
10 7
8
Output: 12

TC2:
Input:
4
10 2
10 4
12 6
18 9
15
Output: 38.0

TC3:
Input:
4
18 6
20 3
14 5
18 9
21
Output: 66.0

2. 0/1 Knapsack Problem

- Given the weights and profits of **N** items, in the form of **{profit, weight}** put these items in a knapsack of capacity **W** to get the maximum total profit in the knapsack.
- The constraint here is we can either put an item completely into the bag or cannot put it at all
  [It is not possible to put a part of an item into the bag]
- Its either the item is added to the knapsack or not. That is why, this method is known as the **0-1**
- Hence, in case of 0-1 Knapsack, the value of x_i can be either **0** or **1**, where other constraints remain the same.

Example:

Let us consider that the capacity of the knapsack is W = 8 and the items are as shown in the following table.

Item	**A**	**B**	**C**	**D**
Profit	2	4	7	10
Weight	1	3	5	7

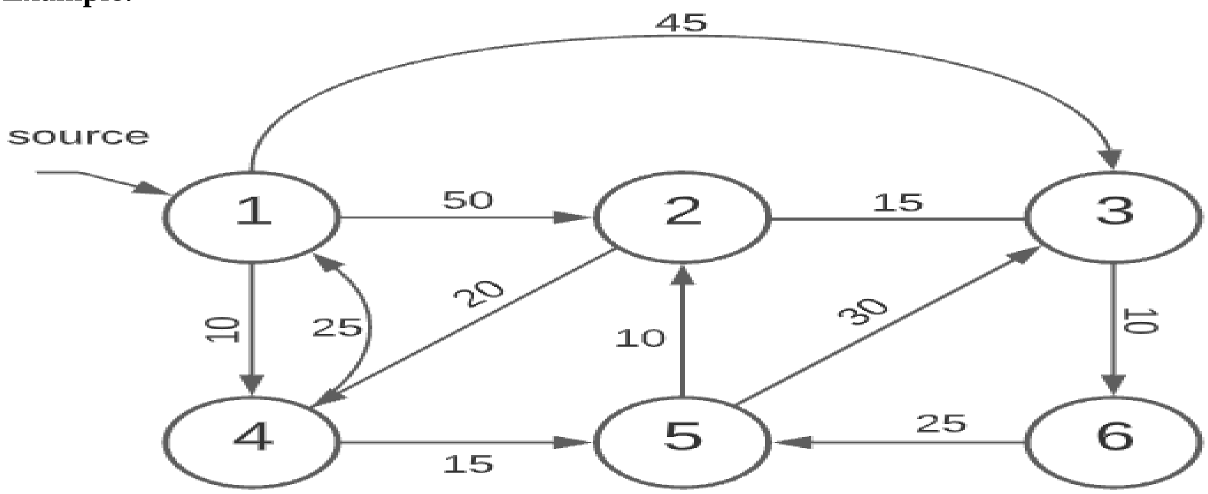
Following are differences between the 0/1 knapsack problem and the Fractional Knapsack problem

Sr. No	0/1 knapsack problem	Fractional knapsack problem
1.	The 0/1 knapsack problem is solved using dynamic programming approach.	Fractional knapsack problem is solved using a greedy approach.
2.	The 0/1 knapsack problem has not an optimal structure.	The fractional knapsack problem has an optimal structure.
3.	In the 0/1 knapsack problem, we are not allowed to break items.	Fractional knapsack problem, we can break items for maximizing the total value of the knapsack.
4.	0/1 knapsack problem, finds a most valuable subset item with a total value less than equal to weight.	In the fractional knapsack problem, finds a most valuable subset item with a total value equal to the weight.
5.	In the 0/1 knapsack problem we can take objects in an integer value.	In the fractional knapsack problem, we can take objects in fractions in floating points.

Single Source Shortest Paths

We have a graph **G (V, E)**.
We assume one vertex as a source vertex.
We must find the shortest path from source vertex to all the remaining vertices of the graph.

Example:



We assume vertex 1 as the source vertex.
We must find shortest paths from 1 to remaining vertices.

In the first step we identify the vertex which is of shortest distance from 1 i.e., **1 – 4 cost is 10**

Now we identify the vertices that can be reached from 4 i.e., 4 – 5 cost is 15
Now 1 – 5 direct edge cost is ∞ because there is no edge between 1 and 5 but the edge cost of 1 – 4 and 4 – 5 is 10+15=25.
Therefore, edge cost of **1 – 5 is 25**.

Now we identify the vertices that can be reached from 5 i.e., 5 – 2 and 5 – 3

Now 1 – 2 direct edge cost 50 but edge cost between 1 – 5 and 5 – 2 is 25+10=35
Therefore, edge cost of **1 – 2 is 35**

Now 1 – 3 direct edge cost is 45 but edge cost between 1 – 5 and 5 – 3 is 25+30=55
Therefore, edge cost of **1 – 3 is 45**

Now identify vertices that can be reached from 2 is 3
Now direct edge cost between 1 – 3 is 45 but edge cost between 1 – 2 and 2 – 3 is 35+15=50
Therefore, edge cost of **1 – 3 is 45**

Now we identify the vertices that can be reached from 3 i.e., 3 – 6.
Now 1 – 6 direct edge cost is ∞ but edge cost of 1 – 3 and 3 – 6 is $45 + 10 = 55$
Therefore, edge cost of **1 – 6 is 55**

Solution:

1 – 4 => 10, 1 – 5 => 25, 1 – 2 => 35, 1 – 3 => 45, 1 – 6 =>55

/* A Java program for Dijkstra's single source shortest path algorithm. The program is for adjacency matrix representation of the graph */

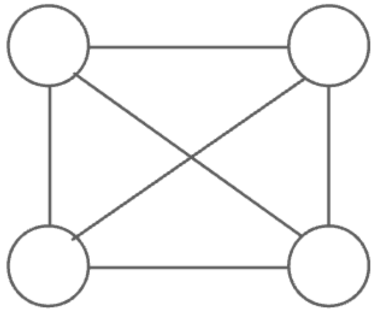
```
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
	/* 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);
    }
```

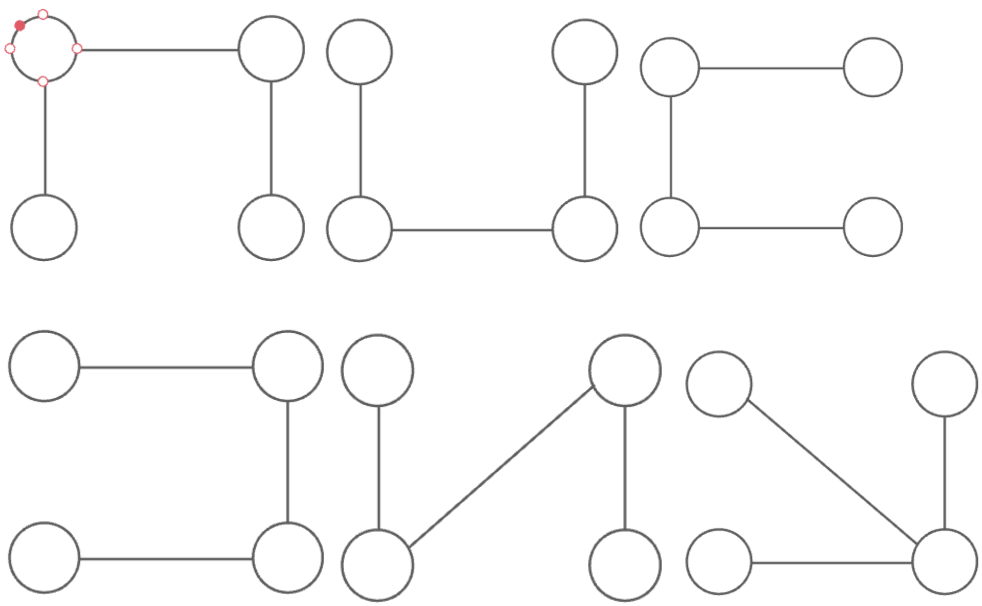
```
        // Driver method
        public static void main(String[] args)
        {
                int graph[][] = new int[][] {   { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                                                { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                                                { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                                                { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                                                { 0, 0, 0, 9, 0, 10, 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 } };
                ShortestPath t = new ShortestPath();
                t.dijkstra(graph, 0);
        }
}
```

Minimum Cost Spanning Trees

Let G=(V, E) be an undirected connected graph. A subgraph t=(v,E') of G is a spanning tree of G iff t is a tree. Eg. Graph



Spanning trees are



The spanning tree with minimum edge cost sum is minimum cost spaming tree.

Since the identification of a minimum cost spanning tree involves the selection of a subset of edge the problem fits the subset paradigm.

There are 2 methods to find minimum cost spamming trees

1) Prim’s algorithm
2) Kruskal’s algorithm

<u>Prim’s algorithm</u> :- builds the tree edge by edge, the next edge to include is chosen according to some optimization criteria. The criterion is to choose an edge that results in a minimum increase in the sum of costs of edges so far included.

<u>Pseudo code for the algorithm is as follows.</u>

1) The algorithm will start with a tree that includes only a minimum cost edge of G. Then edges are added to the tree one by one. The next edge (i, j) to be added is such that i is a vertex already included in the tree, j is a vertex not yet included, and the cost of(i, j), cost[i, j] is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree.

2) To determine this edge (I, j) efficiently, we associate with each vertex j not yet included in the tree and a value near[j], where near[j] is a vertex in the tree such that cost[j, near [j]] is

3) Minimum among all choices for near[j]. We define near[j] as 0 for all vertices j that are already in the tree. The next edge to include is defined by vertex j such that near[j] ≠ 0 (j in not part of tree) and cost[j, near[j]) is minimum.

<u>PROGRAM</u> **MST.java**

```
import java.io.*;
import java.lang.*;
import java.util.*;
class MST {
	// Number of vertices in the graph
	static int V;
	/*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[])
	{
		// Initialize min value
		int min = Integer.MAX_VALUE, min_index = -1;
		for (int v = 0; v < V; v++)
		{
			System.out.println("v " + v + " mstSet[v] " + mstSet[v] + " key[v] " +
key[v] + " min " + min);
			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[][])
        {
            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[][])
        {
            // 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  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);
                // 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);
        }
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                int r = sc.nextInt();
                V = r;
                int[][] graph = new int[r][r];
                for(int i=0;i<r;i++){
                        for(int j=0;j<r;j++)
                                graph[i][j]=sc.nextInt();
                }
                MST t = new MST();
                t.primMST(graph);
        }
}
```

Test Cases

case =1
input =
5
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0

output =
Edge Weight
0 - 1 2
1 - 2 3
0 - 3 6
1 - 4 5

case =2

input = 5
0 6 5 0 13
6 0 12 9 5
5 12 0 0 0
0 9 0 0 7
13 5 0 7 0
output =
Edge Weight
0 - 1 6
0 - 2 5
4 - 3 7
1 - 4 5

Kruskal's Algorithm:

This mechanism selects the set of edges considering the next minimum cost edge every time, in such a way that no cycle occurs and by the time n-1 edges are considered a minimum cost spanning tree is generated. The set of edges considered may not be a tree all the time.

The generalized Kruskal's algorithm is a follows

1) Sort edges by ascending edge weight

2) Walk through the sorted edges and look at the two nodes the edge belongs to. If the nodes are already unified we dont include this edge, otherwise we include it and unify the nodes

3) The algorithm terminates when every edge has been processed or all the vertices have been unified

Program for Kruskal's algorithm to find Minimum Spanning Tree of a given connected, undirected and weighted graph

```
import java.util.*;
class Graph
{
	// A class to represent a graph edge
	class Edge implements Comparable<Edge>
	{
		int src, dest, weight;

		// Comparator function used for sorting edges based on their weight
		public int compareTo(Edge compareEdge)
		{
			return this.weight - compareEdge.weight;
		}
		public String toString()
		{
			return src + " " + dest + " " + weight;
		}
	};

	// A class to represent a subset for union-find
	class subset
	{
		int parent, rank;
	};

	int V, E; // V-> no. of vertices & E->no.of edges
	Edge edge[]; // collection of all edges
```

```
// Creates a graph with V vertices and E edges
Graph(int v, int e)
{
        V = v;
        E = e;
        edge = new Edge[E];
        for (int i = 0; i < e; ++i)
                edge[i] = new Edge();
}

// A utility function to find set of an element i (uses path compression technique)
int find(subset subsets[], int i)
{
        // find root and make root as parent of i (path compression)
        if (subsets[i].parent != i)
                subsets[i].parent = find(subsets, subsets[i].parent);

        return subsets[i].parent;
}

// A function that does union of two sets of x and y (uses union by rank)
void Union(subset subsets[], int x, int y)
{
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);

        // Attach smaller rank tree under root of high rank tree (Union by Rank)
        if (subsets[xroot].rank   < subsets[yroot].rank)
                subsets[xroot].parent = yroot;
        else if (subsets[xroot].rank > subsets[yroot].rank)
                subsets[yroot].parent = xroot;
        // If ranks are same, then make one as root and increment its rank by one
        else {
                subsets[yroot].parent = xroot;
                subsets[xroot].rank++;
        }
}

void KruskalMST()
{
        // This will store the resultant MST
        Edge result[] = new Edge[V];

        // An index variable, used for result[]
        int e = 0;

        // An index variable, used for sorted edges
```

```
int i = 0;
for (i = 0; i < V; ++i)
        result[i] = new Edge();

System.out.println("Before sort " + Arrays.deepToString(edge));

/* Step 1: Sort all the edges in non-decreasing order of their weight.
If we are not allowed to change the given graph,
we can create a copy of array of edges */
Arrays.sort(edge);

System.out.println("After sort " + Arrays.deepToString(edge));

// Allocate memory for creating V subsets
subset subsets[] = new subset[V];
for (i = 0; i < V; ++i)
        subsets[i] = new subset();

// Create V subsets with single elements
for (int v = 0; v < V; ++v)
{
        subsets[v].parent = v;
        subsets[v].rank = 0;
}

i = 0; // Index used to pick next edge

// Number of edges to be taken is equal to V-1
while (e < V - 1)
{
        // Step 2: Pick the smallest edge. And increment the index for next iteration
        Edge next_edge = edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        /* If including this edge doesn't cause cycle, include it in result and increment the
        index of result for next edge*/
        if (x != y)
        {
                result[e++] = next_edge;
                Union(subsets, x, y);
        }
        // Else discard the next_edge
}

// print the contents of result[] to display the built MST
System.out.println("Following are the edges in " + "the constructed MST");
```

```
        int minimumCost = 0;
        for (i = 0; i < e; ++i)
        {
            System.out.println(result[i].src + " -- " + result[i].dest+ " == " + result[i].weight);
            minimumCost += result[i].weight;
        }
        System.out.println("Minimum Cost Spanning Tree " + minimumCost);
    }

    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        int V = sc.nextInt();
        int E = sc.nextInt();

        Graph graph = new Graph(V, E);
        for(int i = 0; i < E; i++)
        {
            graph.edge[i].src = sc.nextInt();
            graph.edge[i].dest = sc.nextInt();
            graph.edge[i].weight = sc.nextInt();
        }

        graph.KruskalMST();
    }
}
```

Test Cases:

case =1
input =4 5
0 1 3
0 2 5
1 2 4
2 3 1
1 3 5
output =8

case =2
input =6 12
0 1 4
1 2 5
0 2 4
1 3 3
2 3 7
0 3 6
1 4 7

2 4 8
3 4 3
5 4 8
5 3 7
1 5 9
output =21

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.

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

Solution is as follows

1. If there are even number of negative numbers and no zeros, the result is the product of all except the largest valued negative number.
2. If there are an odd number of negative numbers and no zeros, the result is simply the product of all.
3. 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.

```
import java.util.*;
class MinProductSubset
{
        static int minProductSubset(int a[], int n)
        {
                if (n == 1)
                        return a[0];
                int negmax = Integer.MIN_VALUE;
                int posmin = Integer.MAX_VALUE;
                int count_neg = 0, 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);
                String[] arr=sc.next().split(",");
                int n=arr.length;
                int a[] = new int[arr.length];

                for(int i=0;i<arr.length;i++)
                    a[i]=Integer.parseInt(arr[i]);
                System.out.println(minProductSubset(a, n));
            }
}
```

Test cases:

<u>case =1</u>
input =2,3,-2,4
output =-48
<u>case =2</u>
input =-2,0,-3
output =-3
<u>case=3</u>

input=-4,2,-3,4,-5
output =-480

Best Time To Buy and Sell Stock

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.

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.

This problem can be solved using the greedy approach. To maximize the profit, we have to minimize the buy cost and we have to sell it at maximum price.

Follow the steps below to implement the above idea:

1. Declare a **buy** variable to store the buy cost and **max_profit** to store the maximum profit.
2. Initialize the **buy** variable to the first element of the **prices array**.
3. Iterate over the **prices** array and check if the current price is minimum or not.
   a. If the current price is minimum then buy on this **ith** day.
   b. If the current price is **greater** than the previous buy then make profit from it and maximize the **max_profit.**
4. Finally, return the **max_profit**

```
import java.util.*;
class BuyAndSellStock
{
	public int maxProfit(int prices[])
	{
		int minprice = Integer.MAX_VALUE;
		int maxprofit = 0;
		for (int i = 0; i < prices.length; i++)
		{
			if (prices[i] < minprice)
			minprice = prices[i];
			else if (prices[i] - minprice > maxprofit)
			maxprofit = prices[i] - minprice;
		}
```

```
		return maxprofit;
	}
	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		int n=sc.nextInt();
		int ar[]=new int[n];
		for(int i=0;i<n;i++)
		ar[i]=sc.nextInt();
		System.out.println(new BuyAndSellStock().maxProfit(ar));
	}
}
```