National University of Computer & Emerging Sciences, Karachi

Computer Science Department

Fall 2022, Lab Manual – 07

Course Code: CL-2001	**Course : Data Structures - Lab**
Instructor(s) :	**Abeer Gauher, Sobia Iftikhar**

LAB - 7

Sorting and Searching

Sorting

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

Quick Sort

Key Points:
Step:1 Consider the last element of the list as pivot (i.e., Element at last position in the list).
Step 2 - Define two variables low and high. Set low and high to first and last elements of the list respectively. and set pivot to arr[high]//last element.
Step 3 - Initialize the smaller element i to low-1// i = low-1 and next element j.
Set j= low.
Step 4 - Traverse the elements from j to high -1(second last element)
Step 4.1- If arr[j] < pivot then increment i and swap arr[i] and arr[j].
Step 6 - Repeat steps 3,4 & 4.1 until j recahes high -1.
Step 7 - Exchange the pivot element with arr[i + 1] element.
Step 8 - Return i when added 1.
arr[] = {10, 80, 30, 90, 40, 50, 70} Indexes: 0 1 2 3 4 5 6
low = 0, high = 6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1
Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same
j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]
j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30
j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]
j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot,
do i++ and swap arr[i] with arr[j] i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of the loop because j is now equal to high-1.

Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than
70 are before it and all elements greater than 70 are after it.

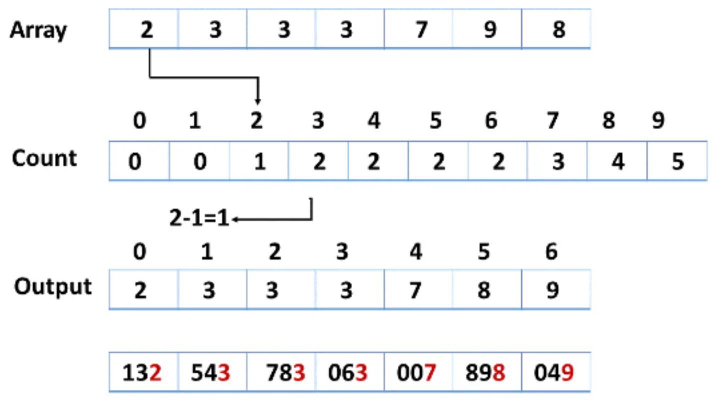
Radix Sort:

Let's start with [132, 543, 783, 63, 7, 49, 898]. It is sorted using radix sort, as illustrated in the figure below.

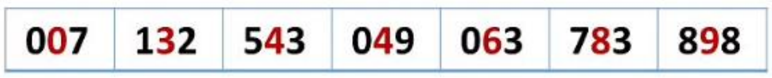
- Find the array's largest element, i.e., maximum. Consider A to be the number of digits in maximum. A is calculated because we must traverse all of the significant locations of all elements.

The largest number in this array [132, 543, 783, 63, 7, 49, 898] is 898. It has three digits. As a result, the loop should be extended to hundreds of places (3 times).

- Now, go through each significant location one by one. Sort the digits at each significant place with any stable sorting technique. You must use counting sort for this. Sort the elements using the unit place digits (A = 0).

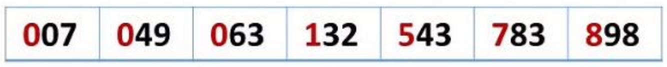


- Sort the elements now by digits in the tens place.



007	132	543	049	063	783	898

- Finally, sort the elements by digits in the hundreds place.



007	049	063	132	543	783	898

Implementation

```
radixSort(array)
  d <- maximum number of digits in the largest element
  create d buckets of size 0-9
  for i <- 0 to d
    sort the elements according to ith place digits using countingSort

countingSort(array, d)
  max <- find largest element among dth place elements
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique digit in dth place of elements and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

<u>Searching</u>

Searching techniques to be covered include the following:

- Linear
- Binary
- Interpolation

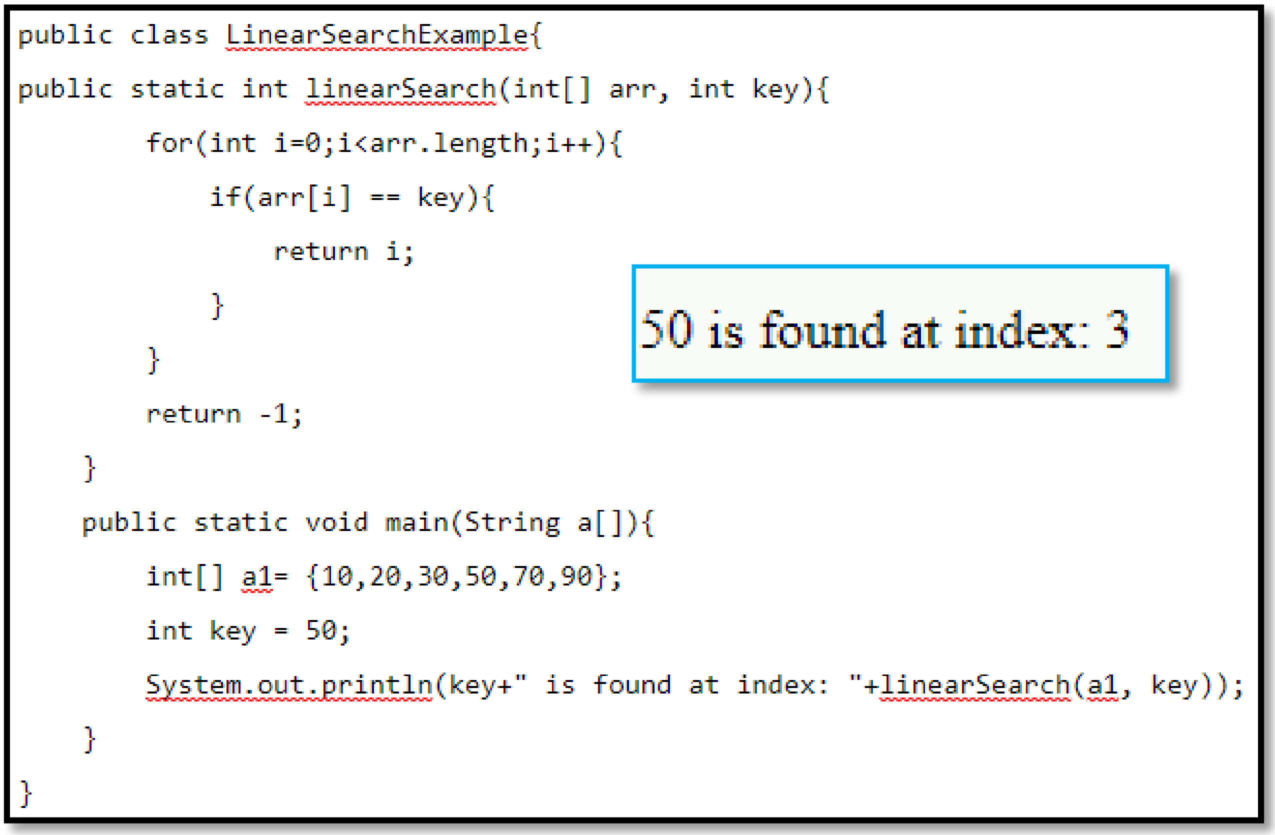
<u>Linear Searching:</u>

Linear search is used to search a key element from multiple elements.

Algorithm:

- Step 1: Traverse the array
- Step 2: Match the key element with array element
- Step 3: If key element is found, return the index position of the array element
- Step 4: If key element is not found, return.

Example:



```
public class LinearSearchExample{
public static int linearSearch(int[] arr, int key){
        for(int i=0;i<arr.length;i++){
            if(arr[i] == key){
                return i;
            }
        }
        return -1;
    }
    public static void main(String a[]){
        int[] a1= {10,20,30,50,70,90};
        int key = 50;
        System.out.println(key+" is found at index: "+linearSearch(a1, key));
    }
}
```

```
50 is found at index: 3
```

Binary Searching

Linear search is a basic technique. In this technique, the array is traversed sequentially and each element is compared to the key until the key is found or the end of the array is reached.
Linear search is used rarely in practical applications. Binary search is the most frequently used technique as it is much faster than a linear search.

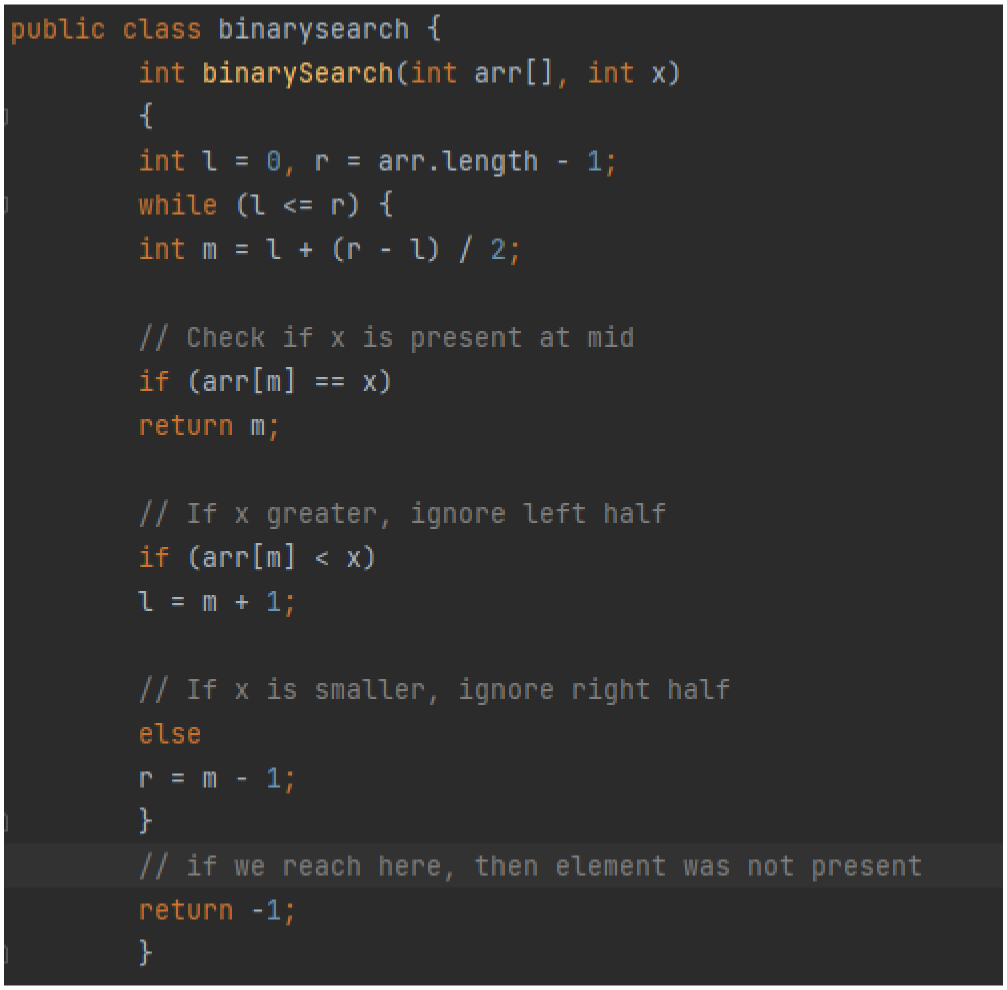
Algorithm For Binary Search In Java

In the binary search method, the collection is repeatedly divided into half and the key element is searched in the left or right half of the collection depending on whether the key is less than or greater than the mid element of the collection.

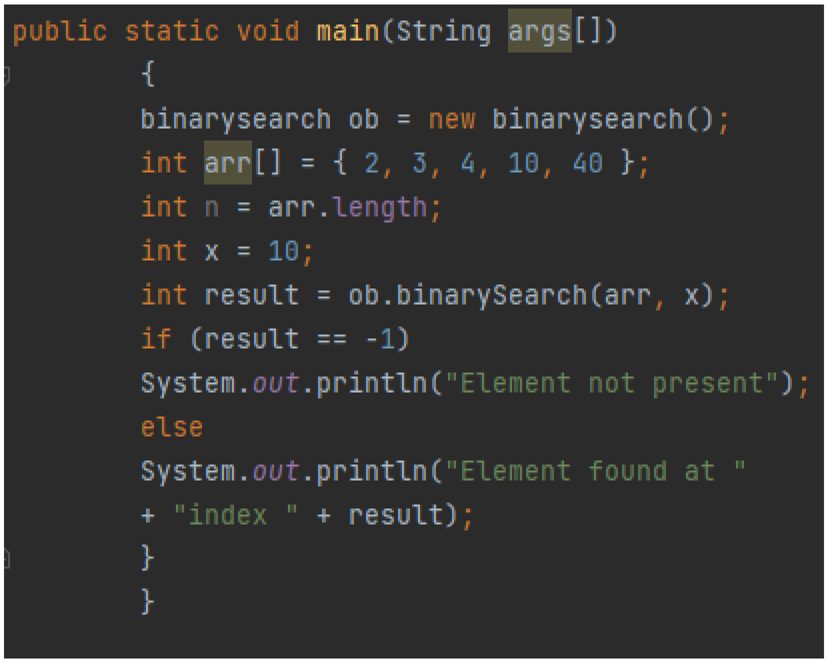
A simple Binary Search Algorithm is as follows:

1. Calculate the mid element of the collection.
2. Compare the key items with the mid element.
3. If key = middle element, then we return the mid index position for the key found.
4. Else If key > mid element, then the key lies in the right half of the collection. Thus repeat steps 1 to 3 on the lower (right) half of the collection.
5. Else key < mid element, then the key is in the upper half of the collection. Hence you need to repeat the binary search in the upper half.

Example:

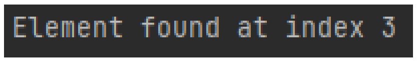


```
public class binarysearch {
        int binarySearch(int arr[], int x)
        {
        int l = 0, r = arr.length - 1;
        while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
        return m;

        // If x greater, ignore left half
        if (arr[m] < x)
        l = m + 1;

        // If x is smaller, ignore right half
        else
        r = m - 1;
        }
        // if we reach here, then element was not present
        return -1;
        }
```



```
public static void main(String args[])
        {
        binarysearch ob = new binarysearch();
        int arr[] = { 2, 3, 4, 10, 40 };
        int n = arr.length;
        int x = 10;
        int result = ob.binarySearch(arr, x);
        if (result == -1)
        System.out.println("Element not present");
        else
        System.out.println("Element found at "
        + "index " + result);
        }
        }
```

Output:



```
Element found at index 3
```

Interpolation Search:

Interpolation search is an improved variant of binary search. This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.

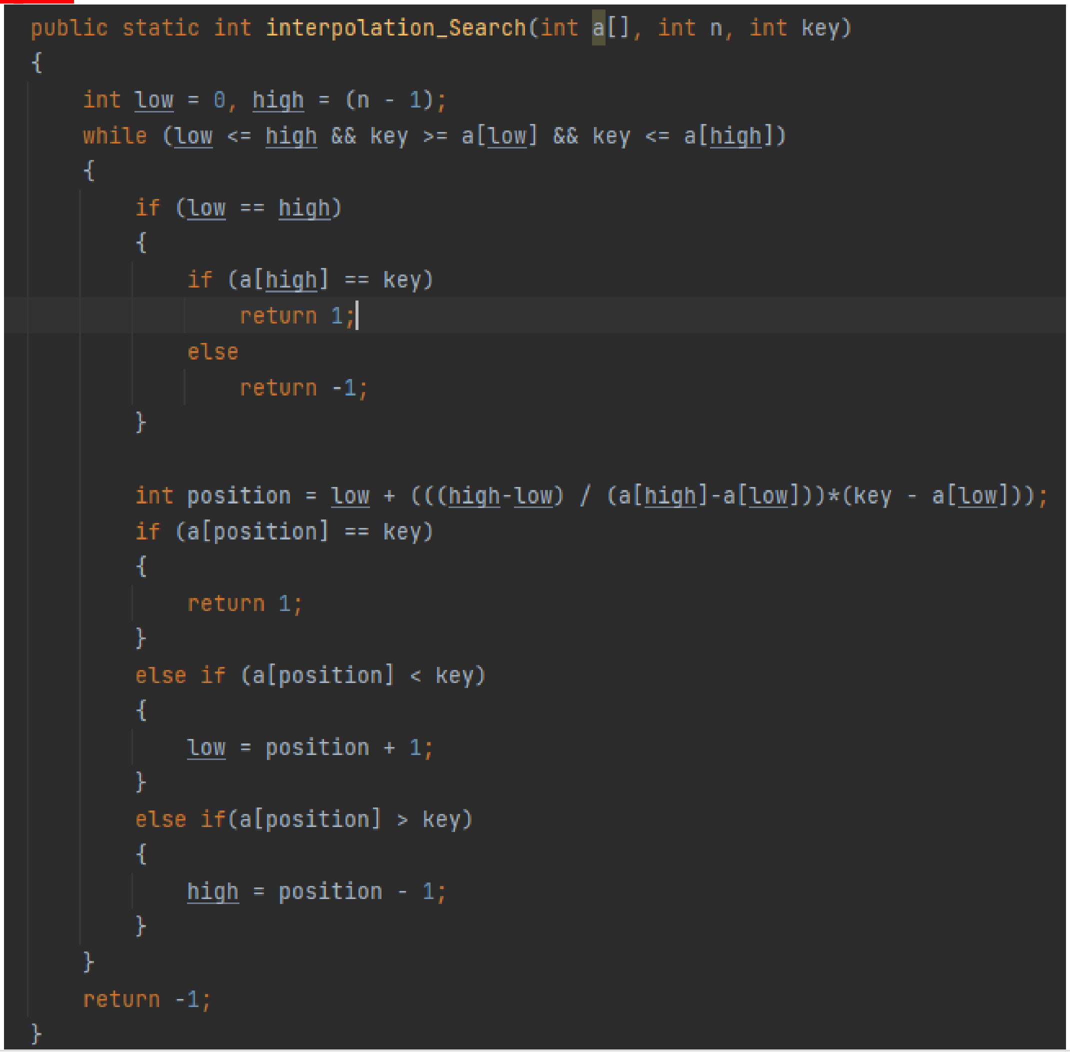
Position Probing in Interpolation Search

Interpolation search finds a particular item by computing the probe position.

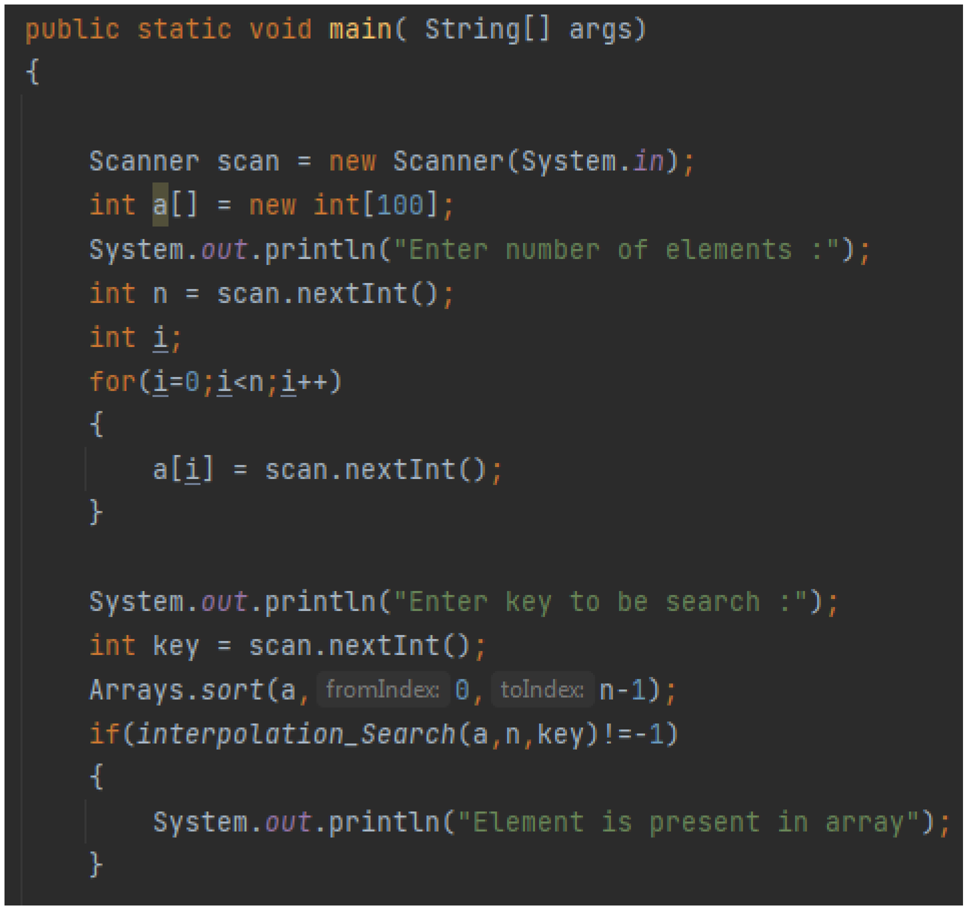
Algorithm

Step 1 − Start searching data from middle of the list.
Step 2 − If it is a match, return the index of the item, and exit.
Step 3 − If it is not a match, probe position.
Step 4 − Divide the list using probing formula and find the new middle.
Step 5 − If data is greater than middle, search in higher sub-list.
Step 6 − If data is smaller than middle, search in lower sub-list.
Step 7 − Repeat until match.

Example:

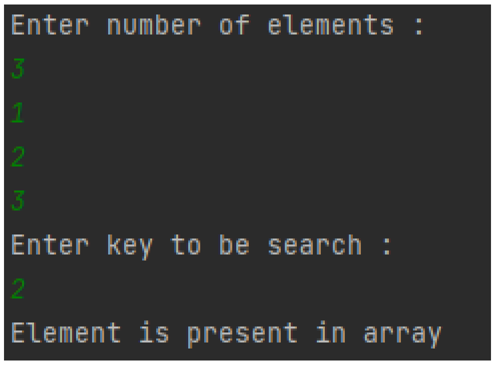


```
public static int interpolation_Search(int a[], int n, int key)
{
    int low = 0, high = (n - 1);
    while (low <= high && key >= a[low] && key <= a[high])
    {
        if (low == high)
        {
            if (a[high] == key)
                return 1;
            else
                return -1;
        }

        int position = low + (((high-low) / (a[high]-a[low]))*(key - a[low]));
        if (a[position] == key)
        {
            return 1;
        }
        else if (a[position] < key)
        {
            low = position + 1;
        }
        else if(a[position] > key)
        {
            high = position - 1;
        }
    }
    return -1;
}
```



```
public static void main( String[] args)
{

    Scanner scan = new Scanner(System.in);
    int a[] = new int[100];
    System.out.println("Enter number of elements :");
    int n = scan.nextInt();
    int i;
    for(i=0;i<n;i++)
    {
        a[i] = scan.nextInt();
    }

    System.out.println("Enter key to be search :");
    int key = scan.nextInt();
    Arrays.sort(a, fromIndex: 0, toIndex: n-1);
    if(interpolation_Search(a,n,key)!=-1)
    {
        System.out.println("Element is present in array");
    }
```

<u>Output:</u>



```
Enter number of elements :
3
1
2
3
Enter key to be search :
2
Element is present in array
```

Lab Tasks:

1. a) You are required to develop an algorithm that finds the majority element in an array A of size N. A majority element is an element that appears more than N/2 times.

   Example:
   3,3,4,2,4,4,2,4,4 has a majority element (4), whereas the array
   3,3,4,2,4,4,2,4 has no majority element.

   b) Sort the array in ascending order and find if an element exists using interpolation search.

2. Use the formula (a*m+i*b) % 100,000 for i between 0 and 999 and a = 47, m = 2743, and b = 5923 to fill the array with "random" numbers less than 100,000. Use the sequential search to search for an item and also count the number of comparisons for the sequential search. Array size can be of your choice but should be atleast 10.

3. Find the first and the last occurrence of a given number in a sorted array using binary search.
   Input:

   array = [2, 5, 5, 5, 6, 6, 8, 9, 9, 9]
   target = 5
   Output:
   The first occurrence of element 5 is located at index 1
   The last occurrence of element 5 is located at index 3

4. You are given a deck of cards. A deck of cards contain a total of 52 cards in general. You are required to implement a sorting algorithm on only 26 cards.
   Step 1: Take all the unsorted set of cards.
   Step 2: Take a random card from the set and consider it as the PIVOT element.
   And then the recurring procedure starts,
   <= pivot number —— on to your left side.
   > pivot number —— on to your right side.
   You can either declare or create your own array with unsorted elements.

5. You have recently attempted a Data Structure quiz marked out of 20. You are required to do the following:

   1. Create an array of size 10 atleast.
   2. The numbers in the ranges should range from 1 to 20 which means the array should contain both single and double digit numbers.
   3. Sort the array using the appropriate algorithm.