# K. J. Somaiya College of Engineering, Mumbai-77

(Autonomous College Affiliated to University of Mumbai)

## Department of Computer Engineering

**Batch:** **A1** **Roll No.:** **1711006**

**Experiment No.__2__**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

## Title: Implementation of Binary search/Max-Min algorithm

**Objective:** To learn the divide and conquer strategy of solving the problems of different types

**CO to be achieved:**

| Sr. No | Objective |
|---|---|
| CO 1 | Compare and demonstrate the efficiency of algorithms using asymptotic complexity notations. |
| CO 2 | Analyze and solve problems for divide and conquer strategy, greedy method, dynamic programming approach and backtracking and branch & bound policies. |
| CO 3 | Analyze and solve problems for different string matching algorithms. |

**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran," Fundamentals of computer algorithm", University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein," Introduction to algortihtms",2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://en.wikipedia.org/wiki/Binary_search_algorithm**

4. **https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Binary_search_algorithm.html**
5. **http://video.franklin.edu/Franklin/Math/170/common/mod01/binarySearchAlg.html**
6. **http://xlinux.nist.gov/dads/HTML/binarySearch.html**
7. **https://www.cs.auckland.ac.nz/software/AlgAnim/searching.html**

---

**Pre Lab/ Prior Concepts:**
Data structures

---

**Historical Profile:**

Finding maximum and minimum or Binary search are few problems those are solved with the divide-and-conquer technique. This is one the simplest strategies which basically works on dividing the problem to the smallest possible level.

Binary Search is an extremely well-known instance of divide-and-conquer paradigm. Given an ordered array of n elements, the basic idea of binary search is that for a given element , "probe" the middle element of the array. Then continue in either the lower or upper segment of the array, depending on the outcome of the probe until the required (given) element is reached.

---

**New Concepts to be learned:**
Number of comparisons, Application of algorithmic design strategy to any problem, Classical problem solving Vs Divide-and-Conquer problem solving.

---

**Algorithm IterativeBinarySearch**

```
int binary_search(int A[ ], int key, int imin, int imax)
//The algorithm takes as parameters an array A[1.. n] , the search key and lower-higher index pair of the array.
 // Output- The algorithm returns index of the search key in the given array, if it’s present.
{
  // continue searching while [imin, imax] is not empty
  WHILE (imax >= imin)
    {
          // calculate the midpoint for roughly equal partition
          int imid = midpoint(imin, imax);
          IF(A[imid] == key)
              // key found at index imid
              return imid;
            // determine which subarray to search
          ELSE If (A[imid] < key)
              // change min index to search upper subarray
              imin = imid + 1;
            ELSE
              // change max index to search lower subarray
              imax = imid - 1;
    }
  // key was not found
  RETURN KEY_NOT_FOUND;
```

}

The most important condition to appy this algorithm is that the input array needs to be sorted.

**The space complexity of Iterative Binary Search:**

The space complexity of Iterative Binary Search is O(1) as we need only 3 variables i.e. mid, start and end independent of input size.

**Algorithm RecursiveBinarySearch**

```
int binary_search(int A[], int key, int imin, int imax)
//The algorithm takes as parameters an array A[1.. n] , the search key and lower-higher index pair
of the array.
 // Output- The algorithm returns index of the search key in the given array, if it's present.
{
  // test if array is empty
  IF (imax < imin)
    // set is empty, so return value showing not found
    RETURN KEY_NOT_FOUND;
  ELSE    {
           // calculate midpoint to cut set in half
           int imid = midpoint(imin, imax);
            // three-way comparison
           IF (A[imid] > key)
                // key is in lower subset
                RETURN binary_search(A, key, imin, imid-1);
           ELSE IF (A[imid] < key)
                // key is in upper subset
                RETURN binary_search(A, key, imid+1, imax);
           ELSE
                // key has been found
                RETURN imid;
          }
}
```

**The space complexity of Recursive Binary Search:**

Best case(Ω): In the best case the element to be found will be at exactly mid position. So there would be no further recursive calls. Hence the best case space complexity will be Ω(1).

Worst case(O): In the worst case the element to be found will not be present. So the recursive calls will pile up until start = end. This will take log n steps if n is the size of input array. Therefore stack will grow upto log n size. Therefore worst case space complexity will be O(log n).

Average case(Ө): In average case the element to be found will be present but not exactly at the middle. So the recursive calls will pile up until arr[mid]= key. Therefore average case space complexity will be Ө (log n).

**The Time complexity of Binary Search:**

Best case(Ω): In the best case the element to be found will be at exactly mid position. So there would be no further recursive calls. Hence the best case time complexity will be Ω(1).

Average case (Ө): In average case the element to be found will be present but not exactly at the middle. So the recursive calls will pile up until arr[mid]= key.

By master method:

For binary search , the number of problems to be solved i.e. a=1 and no of subproblems i.e. b=2 and the no of outputs i.e. f(n)=1

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

$=T(n/2) + 1$

$n^{\log_b a} = n^{\log_2 1} = n^0=1$

i.e. $f(n)= n^{\log_b a}$

Therefore $T(n)= \Theta( n^{\log_b a} \log n)$

$= \Theta(\log n)$

So the average case time complexity is Ө(log n).

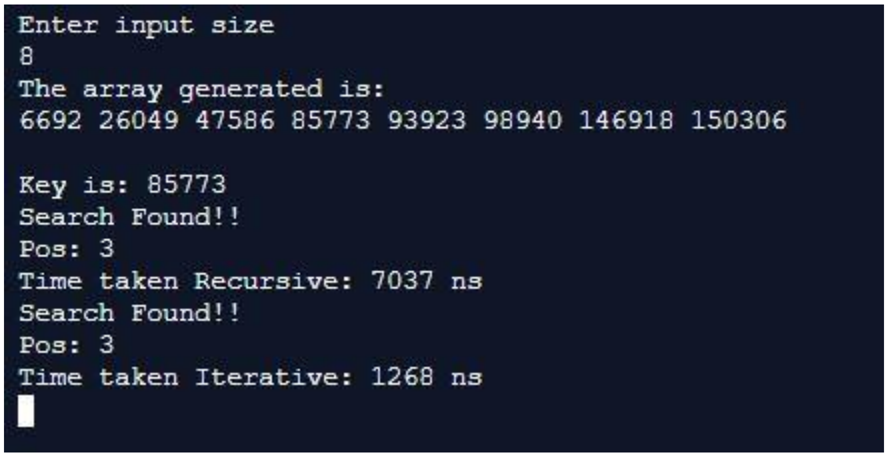
Worst case(O): In the worst case the element to be found will not be present. So the recursive calls will take place up until start = end. This will take $\log_2 n$ steps if n is the size of input array.Therefore worst case time complexity will be O(log n).

**Implementation of Binary Search :**

```
import java.util.Scanner;
import java.util.Random;

class BinarySearch{
      // Using Insertion Sort to sort the array before searching
      public static int[] sort(int arr[]){
            for(int i=1;i<arr.length;i++){
            for(int j=i;j>0;j--){
              if(arr[j]<arr[j-1]){
                   int temp=arr[j];
                    arr[j]=arr[j-1];
                    arr[j-1]=temp;
                  }else
                    break;
            }
            }
            return arr;
      }
      //Recursive function for Binary Search
      public static int binary_search(int arr[],int key,int start,int end){
     int mid=(start+end)/2;
    if(start==end){
      return -1;
    }
       else if(arr[mid]==key){
             return mid;
       }else if(key<arr[mid]){
             return binary_search(arr,key,start,mid);
       }else{
             return binary_search(arr,key,mid+1,end);
       }
      }
      public static void main(String args[]){
            Scanner sc=new Scanner(System.in);
            System.out.println("Enter input size"); // Taking input size from
the user
            int n=sc.nextInt();
            Random r=new Random();
            int arr[]=new int[n];
            for(int i=0;i<n;i++){
                  arr[i]=r.nextInt(20000);  // Generating a random array
            }
            arr=sort(arr); //Sorting the random array before searching
            /*System.out.println("The array generated is: ");
            for(int i:arr){
                  System.out.print(i+" ");
            }
            System.out.println();
            System.out.println();*/
    int key=r.nextInt(100000); // Generating random key to be searched
    System.out.println("Key is: "+key);
    //int key=sc.nextInt();  // Taking key from user
    long start_time,end_time;
      // Calling the recursive Binary Search Function
    start_time=System.nanoTime();
            int j=binary_search(arr,key,0,arr.length);
    end_time=System.nanoTime();
    if(j>=0){
      System.out.println("Search Found!!");
            System.out.println("Pos: "+j);
    }else{
      System.out.println("Search  not Found!!");
    }
     System.out.println("Time taken Recursive: "+(end_time-start_time)+"
ns");
      // *******Modifications********
      // Iterative approach of Binary Search
    int mid,start=0,end=arr.length;
    start_time=System.nanoTime();
    while(true){
      mid=(start+end)/2;
      if(start==end){
        mid=-1;
        break;
      }
          else if(arr[mid]==key){
                break;
          }else if(key<arr[mid]){
                 end=mid;
          }else{
                start=mid+1;
         }
    }
    end_time=System.nanoTime();
    if(mid==-1){
      System.out.println("Search  not Found!!");
    }else{
      System.out.println("Search Found!!");
            System.out.println("Pos: "+mid);
    }
     System.out.println("Time taken Iterative: "+(end_time-start_time)+"
ns");
      }

}
```

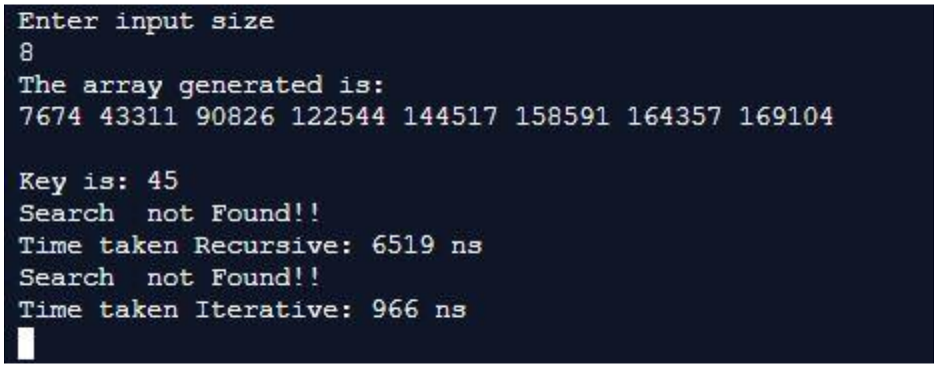
**Output:**

1) Demonstration that algorithm works properly on a small input set:



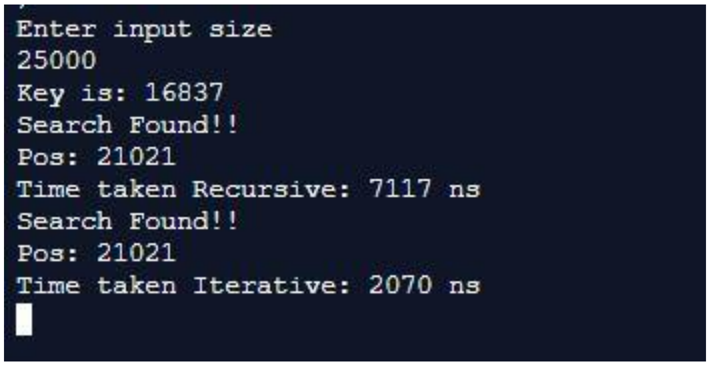
```
Enter input size
8
The array generated is:
6692 26049 47586 85773 93923 98940 146918 150306

Key is: 85773
Search Found!!
Pos: 3
Time taken Recursive: 7037 ns
Search Found!!
Pos: 3
Time taken Iterative: 1268 ns
```

2) Demonstration that algorithm returns Search not found when element not present on a small input set:



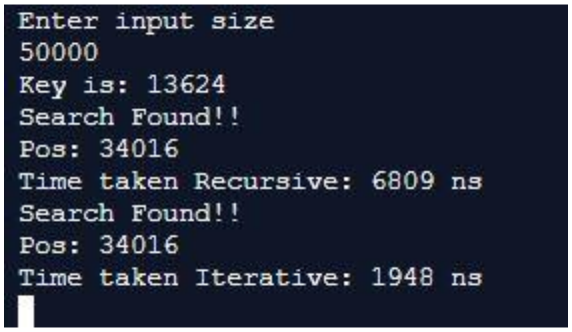
```
Enter input size
8
The array generated is:
7674 43311 90826 122544 144517 158591 164357 169104

Key is: 45
Search  not Found!!
Time taken Recursive: 6519 ns
Search  not Found!!
Time taken Iterative: 966 ns
```

3) Input size as 25000:



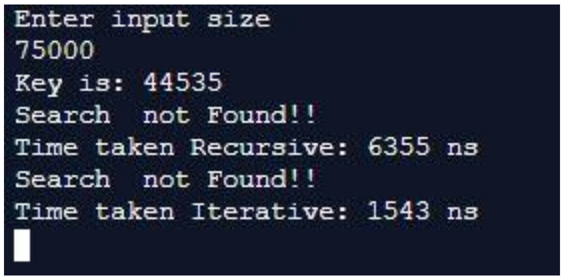
```
Enter input size
25000
Key is: 16837
Search Found!!
Pos: 21021
Time taken Recursive: 7117 ns
Search Found!!
Pos: 21021
Time taken Iterative: 2070 ns
```

4) Input size as 50000:



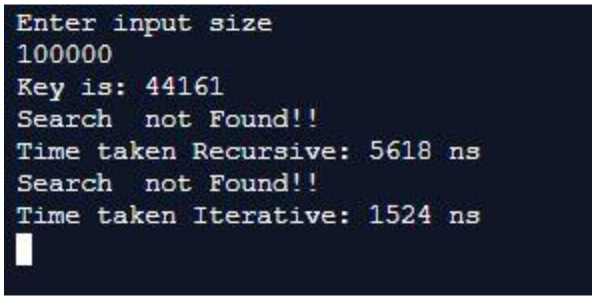
```
Enter input size
50000
Key is: 13624
Search Found!!
Pos: 34016
Time taken Recursive: 6809 ns
Search Found!!
Pos: 34016
Time taken Iterative: 1948 ns
```

5) Input size as 75000:



```
Enter input size
75000
Key is: 44535
Search  not Found!!
Time taken Recursive: 6355 ns
Search  not Found!!
Time taken Iterative: 1543 ns
```

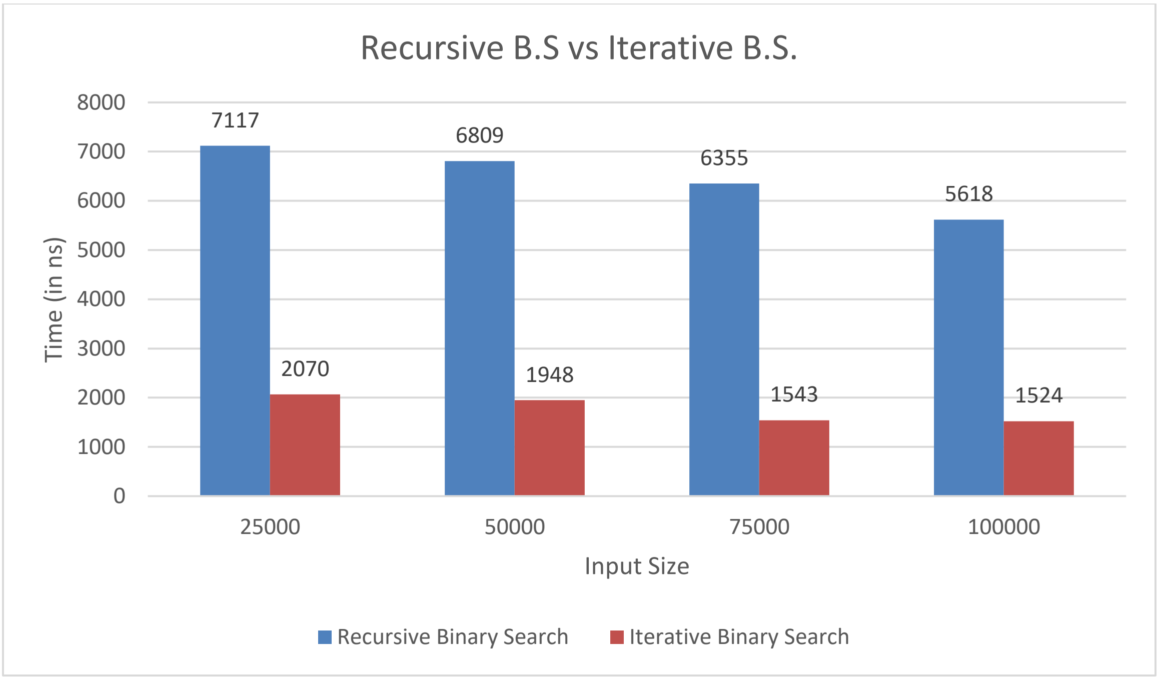
6) Input size as 100000:



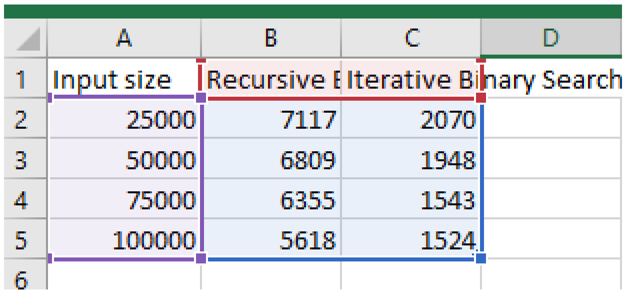
```
Enter input size
100000
Key is: 44161
Search  not Found!!
Time taken Recursive: 5618 ns
Search  not Found!!
Time taken Iterative: 1524 ns
```

**Comparison of Recursive Binary Search and Iterative Binary Search on different Input Sizes:**

The bar graph below shows the time taken by recursive binary search and iterative binary search for various input sizes. The comparison is not made using line graphs as the search time generally depends on the key and slightly on the size of the input in worst case and the key to be searched here is randomized. Therefore it won't be correct to compare time taken using line graph.



Excel Sheet:



| | A | B | C | D |
|---|---|---|---|---|
| 1 | Input size | Recursive E | Iterative Binary Search | |
| 2 | 25000 | 7117 | 2070 | |
| 3 | 50000 | 6809 | 1948 | |
| 4 | 75000 | 6355 | 1543 | |
| 5 | 100000 | 5618 | 1524 | |
| 6 | | | | |

From the bar graph above it is clear that recursive algorithm for binary search takes more time as compared to iterative. This is because time is spent on function calls during recursion.

**Algorithm StraightMaxMin:**

```
 VOID  StraightMaxMin (Type a[], int n, Type& max, Type& min)
// Set max to the maximum and min to the minimum of a[1:n].
{   max = min = a[1];
FOR (int i=2; i<=n; i++){
IF (a[i]>max)  then max = a[i];
IF (a[i]<min) min = a[i];
}
}
```

**Algorithm: Recursive Max-Min**

```
 VOID MaxMin(int i, int j, Type& max, Type& min)
// A[1:n] is a global array. Parameters i and j are  integers, 1 <= i <= j <= n.
 //The effect is to set  max and min to the largest and smallest values in a[i:j], respectively.
     {
        IF (i == j) max = min = a[i]; // Small(P)
        ELSE IF (i == j-1) { // Another case of Small(P)
                IF (a[i] < a[j])
                       max = a[j]; min = a[i];
                ELSE { max = a[i]; min = a[j];
                }
          ELSE {      Type max1, min1;
  // If P is not small   divide P into subproblems.  Find where to split the set.
                int mid=(i+j)/2;
               // Solve the  subproblems.
                MaxMin(i, mid, max, min);
               MaxMin(mid+1, j, max1, min1);
          // Combine the solutions.
            IF (max < max1) max = max1;
            IF (min > min1) min = min1;
    }
  }
```

**The space complexity of Max-Min:**

**Recursive:**

In case of divide and conquer we will be splitting the array into half until we get a subarray of 2 or 1 element. Therefore the recursive call will go into $\log_2 n$ levels deep in the stack. Therefore max space acquired will be O(log n).

**Iterative:**

Since we will be using only two variables max and min while iterating through the array the space complexity is O(1).

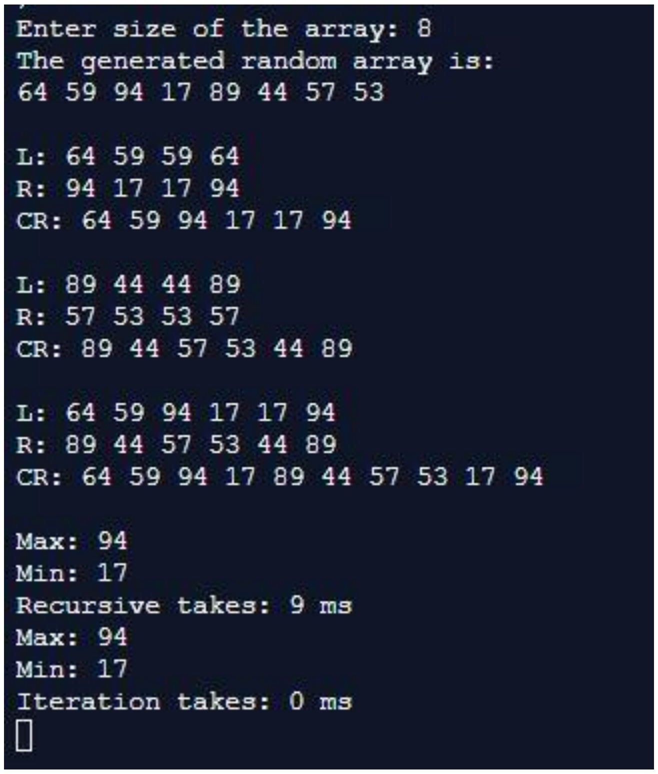
**Implementation:**

```
import java.util.Scanner;
import java.util.Random;
class MaxMin{
      // Recursive (Divide and Conquer) Algo for Max Min
  public static int [] MaxMin(int arr[],int i,int j){
    int maxMin[]=new int[2];
      if(i==j){
        maxMin[0]=arr[i];
        maxMin[1]=arr[i];
        return maxMin;
      }else if(i==j-1){
        if(arr[i]<=arr[j]){
          maxMin[0]=arr[i];
          maxMin[1]=arr[j];
        }else{
          maxMin[0]=arr[j];
          maxMin[1]=arr[i];
        }
         return maxMin;
      }else{
        int mid=(i+j)/2;
        int l[]=MaxMin(arr,i,mid);
        int r[]=MaxMin(arr,mid+1,j);
        //*****Modifications (Print recursive order)*****
       /* System.out.print("L: ");
        for(int k=i;k<=mid;k++){
          System.out.print(arr[k]+" ");
        }
        System.out.println(""+l[0]+" "+l[1]);
        System.out.print("R: ");
        for(int k=mid+1;k<=j;k++){
          System.out.print(arr[k]+" ");
        }
        System.out.println(""+r[0]+" "+r[1]);
        System.out.print("CR: ");// CR is combined result of left and right
part
        for(int k=i;k<=j;k++){
             System.out.print(arr[k]+" ");
           }*/
            maxMin[0]=(l[0]<r[0])?l[0]:r[0];
            maxMin[1]=(l[1]>r[1])?l[1]:r[1];
           /* System.out.println(""+maxMin[0]+" "+maxMin[1]);
            System.out.println();*/
            return maxMin;
        }
    }
    public static void main(String[] args) {
         Scanner sc=new Scanner(System.in);
         System.out.print("Enter size of the array: ");
         int n=sc.nextInt(); // Taking size as input from the user
         Random r=new Random();
         int arr[]=new int[n];
         for(int i=0;i<arr.length;i++){
           arr[i]=r.nextInt(100); // Generating random numbers for the array
         }
        /*System.out.println("The generated random array is:");
         for(int i: arr){
           System.out.print(i+" ");
         }
         System.out.println();
         System.out.println();*/
         long start_time,end_time;
         start_time=System.currentTimeMillis();
         int maxMin[]=MaxMin(arr,0,arr.length-1);
         end_time=System.currentTimeMillis();
         System.out.println("Max: "+maxMin[1]);
         System.out.println("Min: "+maxMin[0]);
         System.out.println("Recursive takes: "+(end_time-start_time)+" ms");

           //*****Iterative approach to find Max Min******
         int max=arr[0];
         int min=arr[0];
         start_time=System.currentTimeMillis();
         for(int i=0;i<arr.length;i++){
           if(arr[i]<=min){
             min=arr[i];
           }else if(arr[i]>max){
             max=arr[i];
           }
         }
         end_time=System.currentTimeMillis();
         System.out.println("Max: "+max);
         System.out.println("Min: "+min);
         System.out.println("Iteration takes: "+(end_time-start_time)+" ms");
    }
   }
```

**Output:**

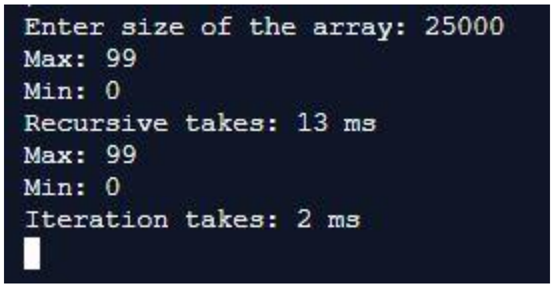
1) Demonstration that algorithm is working correctly along with recursion order for small input set

   The last two elements of each row stand for min and max respectively for the remaining elements of the subarray.CR is the combined result of left and right subarrays.



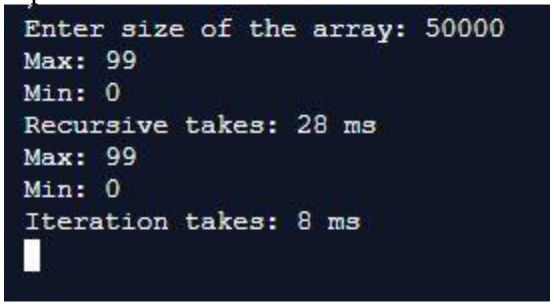
```
Enter size of the array: 8
The generated random array is:
64 59 94 17 89 44 57 53

L: 64 59 59 64
R: 94 17 17 94
CR: 64 59 94 17 17 94

L: 89 44 44 89
R: 57 53 53 57
CR: 89 44 57 53 44 89

L: 64 59 94 17 17 94
R: 89 44 57 53 44 89
CR: 64 59 94 17 89 44 57 53 17 94

Max: 94
Min: 17
Recursive takes: 9 ms
Max: 94
Min: 17
Iteration takes: 0 ms
```

2) Input size 25000



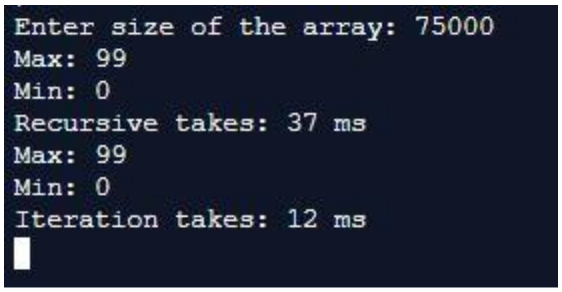
```
Enter size of the array: 25000
Max: 99
Min: 0
Recursive takes: 13 ms
Max: 99
Min: 0
Iteration takes: 2 ms
```

3) Input size 50000



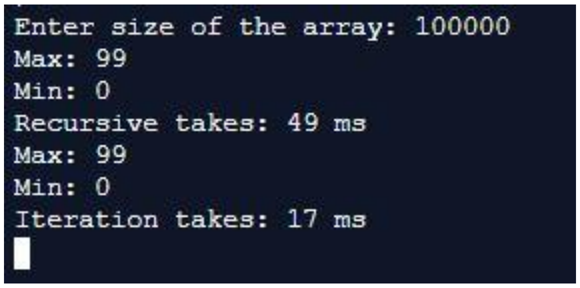
```
Enter size of the array: 50000
Max: 99
Min: 0
Recursive takes: 28 ms
Max: 99
Min: 0
Iteration takes: 8 ms
```

4) Input size 75000



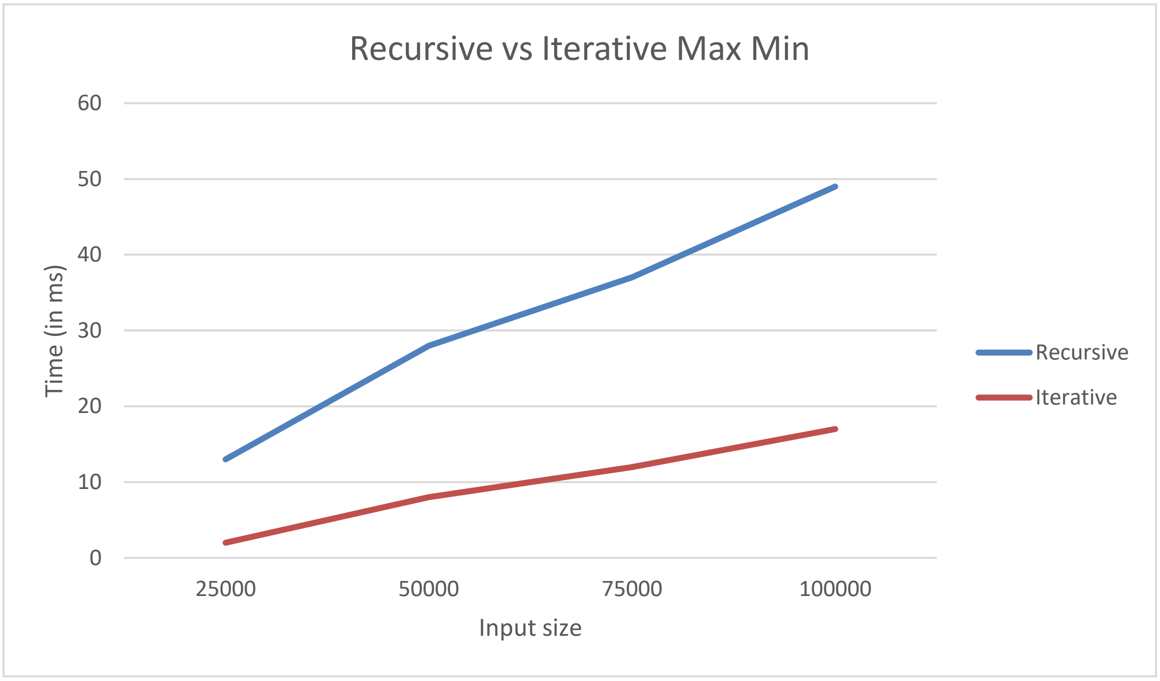
```
Enter size of the array: 75000
Max: 99
Min: 0
Recursive takes: 37 ms
Max: 99
Min: 0
Iteration takes: 12 ms
```

5) Input size 100000

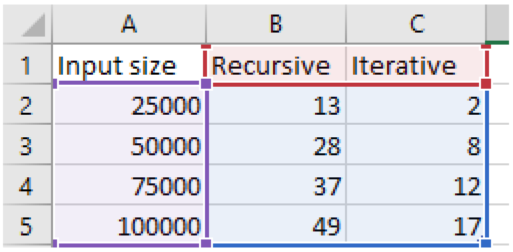


```
Enter size of the array: 100000
Max: 99
Min: 0
Recursive takes: 49 ms
Max: 99
Min: 0
Iteration takes: 17 ms
```

**Comparison of Recursive (Divide and Conquer) Max Min and Iterative Max Min on different Input Sizes:**



Excel Sheet:



| | A | B | C |
|---|---|---|---|
| 1 | Input size | Recursive | Iterative |
| 2 | 25000 | 13 | 2 |
| 3 | 50000 | 28 | 8 |
| 4 | 75000 | 37 | 12 |
| 5 | 100000 | 49 | 17 |

From the above graph it is clear that recursive max min takes more time as compared to iterative as time is spent on recursive function calls.

**Time complexity for Max-Min:**

In case of Max-Min, the number of problems to be solved at each level i.e. a=2 , the number of subproblems i.e. b=2 and the number of outputs at each level i.e. f(n)=2

Also T(n)=0 , n=1

=1 ,n=2

= 2T(n/2) +2 , n>2

Best case: In best case, there will be only 2 numbers or 1 number so the comparison can be directly made or in case of 1 number both max and min will be same.Or the array is sorted. Therefore **best case time complexity will be $\Omega(1)$ .**

Average Case: In average case we need to divide the array into subarrays until we get a subarray of 1 or 2 elements and then merge.

By master method:

$n^{\log_b a} = n^{\log_2 2} = n$

$f(n) < n$

Therefore, $T(n) = \Theta(n^{\log_b a})$

$= \Theta(n)$

**Average case time complexity is $\Theta(n)$**

Worst Case: The worst case will be same as the Average case, so **the time complexity for worst case will be $O(n)$.**

## CONCLUSION:

**Binary Search:** Binary search requires that the array is sorted for the element to search in it. Both the approaches i.e. recursive and iterative have the same time complexity i.e. O(log n), however the former takes more time on implementation as some time is spent on recursive method calls.

The space complexity of Recursive Binary Search is O(log n) as to find an element the recursive call stack may fill up upto log n levels whereas in case of Iterative approach the space complexity is only O(1) as only 3 variables i.e. start,mid and last is required.

**Time Complexity: Recursive Binary Search = Iterative Binary Search**
**Space Complexity: Recursive Binary Search > Iterative Binary Search**
**O(log n) O(1)**

**Max-Min:** Both the approaches of Max-Min i.e. recursive and iterative have the same time complexity i.e. O(n), however the former takes more time on implementation as some time is spent on recursive method calls.

The space complexity of Recursive Max-Min is O(log n) whereas that of iterative is O(1).

**Time Complexity: Recursive Max-Min = Iterative Max-Min**
**Space Complexity: Recursive Max-Min > Iterative Max-Min**
**O(log n) O(1)**