

| **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:**

| CO 2 | Describe various algorithm design strategies to solve different problems and analyse Complexity. |
| --- | --- |



**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 space complexity of Iterative Binary Search:**

// package Codes.Binary;

import java.time.\*;

import java.util.\*;

import java.lang.Math;

class Binary {

// Returns index of x if it is present in arr[],

// else return -1

int binarySearch(int arr[], int x)

{

int l = 0, r = arr.length - 1;

while (l <= r) {

int m = l + (r - l) / 2;

if (arr[m] == x)

return m;

if (arr[m] < x)

l = m + 1;

else

r = m - 1;

}

for (int k = 0; k < arr.length; k++) {

System.out.print(arr[k]+ " ");

}

System.out.println();

// if we reach here, then element was

// not present

return -1;

}

// Driver method to test above

public static void main(String args[])

{

Binary ob = new Binary();

Scanner sc = new Scanner(System.in);

System.out.println("Enter length of array: ");

int N=sc.nextInt();

int arr[];

arr=new int[N];

for(int i=0;i<N;i++)

{

double x=Math.random()\*300;

int num= (int)x;

arr[i]= num;

}

// int arr[] = { 2, 3, 4, 10, 40 };

// int n = arr.length;

System.out.println("Enter element to be searched: ");

int x = sc.nextInt();

long start = System.nanoTime();

int result = ob.binarySearch(arr, x);

if (result == -1){

System.out.println("Element not present");

}

else{

System.out.println("Element found at "+ "index " + result);

}

long end = System.nanoTime();

long execution = end - start;

System.out.println("Execution time: " + execution + "nanoseconds");

}

}

**Algorithm Recursive Binary Search**

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 🡪 higher subset

**RETURN** binary\_search(A, key, imid+1, imax);

**ELSE**

// key has been found

**RETURN** imid;

}

}

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

// package Codes.Binary;

import java.time.\*;

import java.util.\*;

import java.lang.Math;

class BinarySearchRecursive {

static void selectionSort(int arr[], int n)

{

int i, j, min\_idx;

for (i = 0; i < n - 1; i++)

{

// Find the minimum element in unsorted array

min\_idx = i;

for (j = i + 1; j < n; j++)

if (arr[j] < arr[min\_idx])

min\_idx = j;

// Swap the found minimum element with the first element

int temp = arr[min\_idx];

arr[min\_idx]= arr[i];

arr[i] = temp;

}

int k;

for (k = 0; k < n; k++) {

System.out.print(arr[k]+ " ");

}

System.out.println();

}

int binarySearch(int arr[], int l, int r, int x){

if(r>=l){

int mid= l+(r-l)/2;

if(arr[mid]==x)

{

return mid;

}

if(arr[mid]>x){

return binarySearch(arr, l, mid-1, x);

}

return binarySearch(arr, mid+1, r, x);

}

return -1;

}

public static void main(String args[])

{

BinarySearchRecursive object = new BinarySearchRecursive();

Scanner sc = new Scanner(System.in);

System.out.println("Enter length of Array: ");

int N=sc.nextInt();

int arr[];

arr=new int[N];

for(int i=0;i<N;i++)

{

double x=Math.random()\*300;

int num= (int)x;

arr[i]= num;

}

System.out.println("Enter element to search: ");

int x=sc.nextInt();

long start = System.nanoTime();

selectionSort(arr, N);

int result = object.binarySearch(arr,0,N-1,x);

if(result ==-1){

System.out.println("Element not present :)");

}

else{

System.out.println("Element is at "+result+ " position");

}

long end = System.nanoTime();

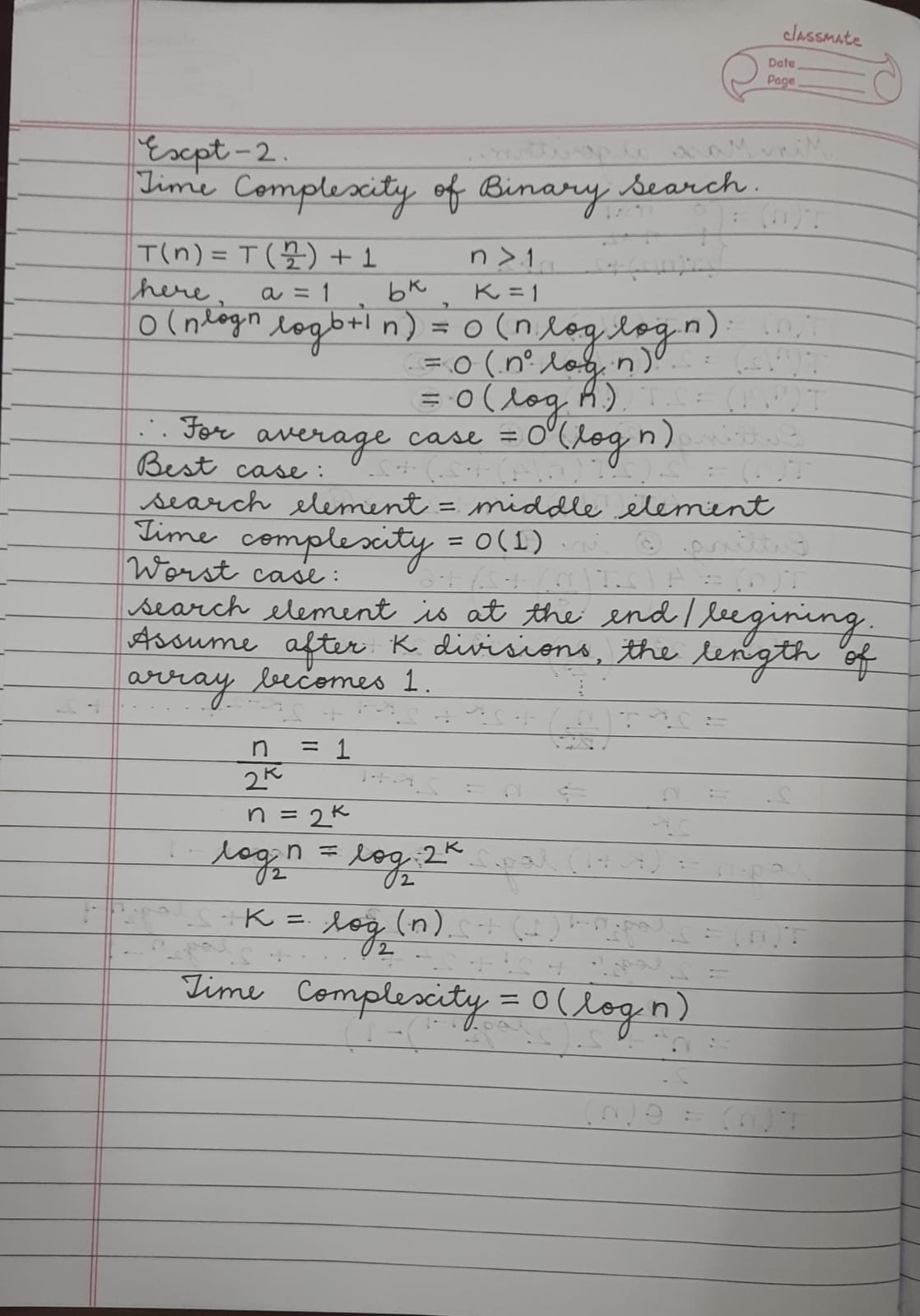
long execution = end - start;

System.out.println("Execution time: " + execution + " nanoseconds");

}

}

**The Time complexity of Binary Search:**

****

**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 sub problems. Find where to split the set.

int mid=(i+j)/2;

// solve the sub problems.

MaxMin(i, mid, max, min);

MaxMin(mid+1, j, max1, min1);

// Combine the solutions.

**IF** (max < max1) max = max1;

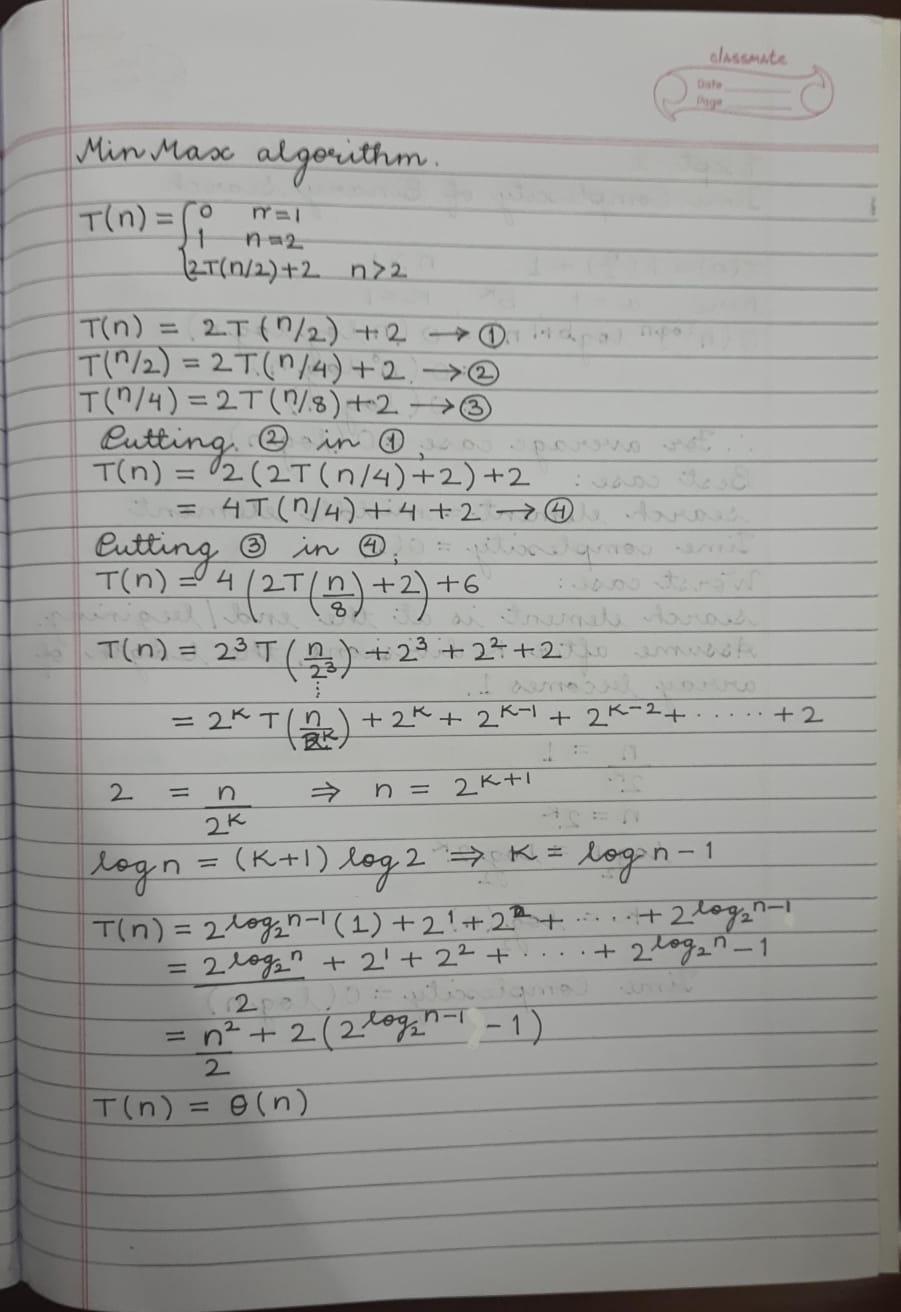
**IF** (min > min1) min = min1;

}

}

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

**Time complexity for Max-Min:**



**CONCLUSION:**

We learnt how to solve algorithms using the divide and conquer strategy. We also understood and implemented binary search and min-max algorithm iteratively and recursively.