**Batch: A1 Roll No.: 16010120015**

**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:**

|  |  |
| --- | --- |
| 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.

**Topic: Divide and Conquer**

**Theory:**  Given a function to compute on n inputs the divide-and-conquer strategy suggests splitting the inputs into k distinct subsets, 1< k ≤n, yielding k sub problems. These sub problems must be solved and then a method must be found to combine sub solutions into a solution of the whole. If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied. Often the sub problems resulting from a divide-and-conquer design are the same type as the original problem. For those cases the reapplication of the divide-and- conquer principle is naturally expressed by a recursive algorithm. Now smaller and smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced.

**Control Abstraction**:

Type DAndC(Problem P)

{

if small (P) return S(P);

else {

divide P into smaller instances P1, P2, …. ,Pk, k ≥1;

Apply DAndC to each of these sub problems;

Return combine(DAndC(P1), DAndC(P2),…., DAndC(Pk));

}

}

**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:**

**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 upper subset

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

**ELSE**

// key has been found

**RETURN** imid;

}

}

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

**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:**

**CODE:**

**#include<stdio.h>**

**int binary\_search(int a[],int s,int l,int h)**

**{**

**int mid;**

**mid=(l + h)/2;**

**if(l<=h)**

**{**

**if(s==a[mid])**

**return mid;**

**else if(s<mid)**

**return binary\_search(a,s,l,mid-1);**

**else**

**return binary\_search(a,s,mid+1,h);**

**}**

**else**

**return -1;**

**}**

**int main()**

**{**

**int a[50],n,i,s,low,high,index;**

**printf("\nENTER NUMBER OF ELEMENTS : ");**

**scanf("%d",&n);**

**printf("\nEnter %d elements: \n", n);**

**for(i=0;i<n;i++)**

**scanf("%d",&a[i]);**

**printf("\nENTER THE KEY ELEMENT TO BE SEARCHED: \n");**

**scanf("%d",&s);**

**low=0;**

**high=n-1;**

**index=binary\_search(a,s,low,high);**

**if(index==-1)**

**printf("\nELEMENT NOT FOUND!\n");**

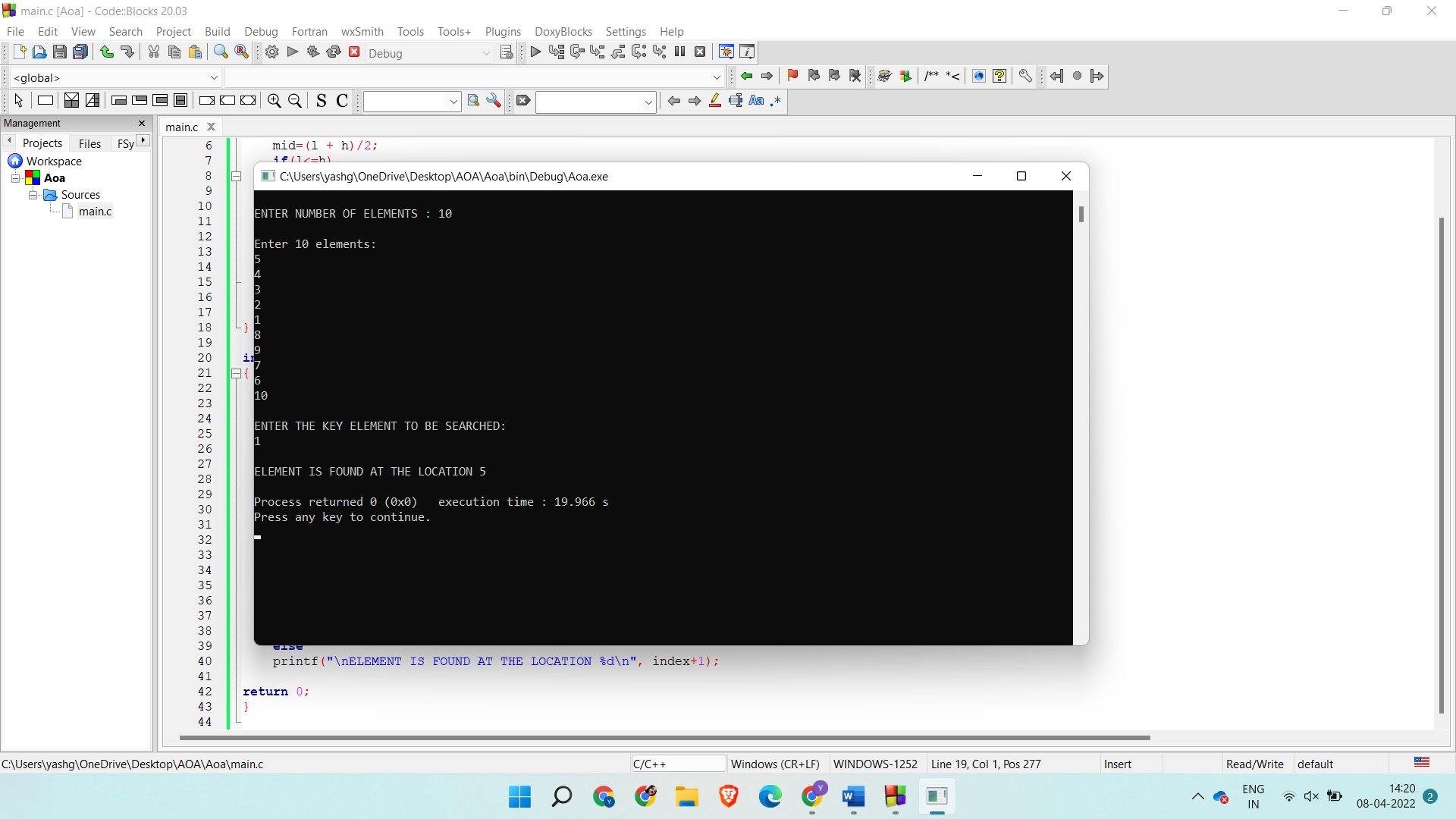
**else**

**printf("\nELEMENT IS FOUND AT THE LOCATION %d\n", index+1);**

**return 0;**

**}**

**OUTPUT:**



**Max-min**

**#include <stdio.h>**

**int a[50],max,min;**

**void maxmin(int i, int j)**

**{**

**int newmax,newmin,mid;**

**if (i==j)**

**max=min=a[i];**

**else**

**{**

**if (i==j-1)**

**{**

**if (a[i] < a[j])**

**{**

**max = a[j];**

**min = a[i];**

**}**

**else**

**{**

**max = a[i];**

**min = a[j];**

**}**

**}**

**else**

**{**

**mid = (i+j)/2;**

**maxmin(i,mid);**

**newmax=max;**

**newmin=min;**

**maxmin(mid+1,j);**

**if(max < newmax)**

**max = newmax;**

**if(min > newmin)**

**min = newmin;**

**}**

**}**

**}**

**void main()**

**{**

**int n,i;**

**printf("\nEnter number of elements ");**

**scanf("%d",&n);**

**printf("\nEnter %d elements: \n", n);**

**for(i=0;i<n;i++)**

**scanf("%d",&a[i]);**

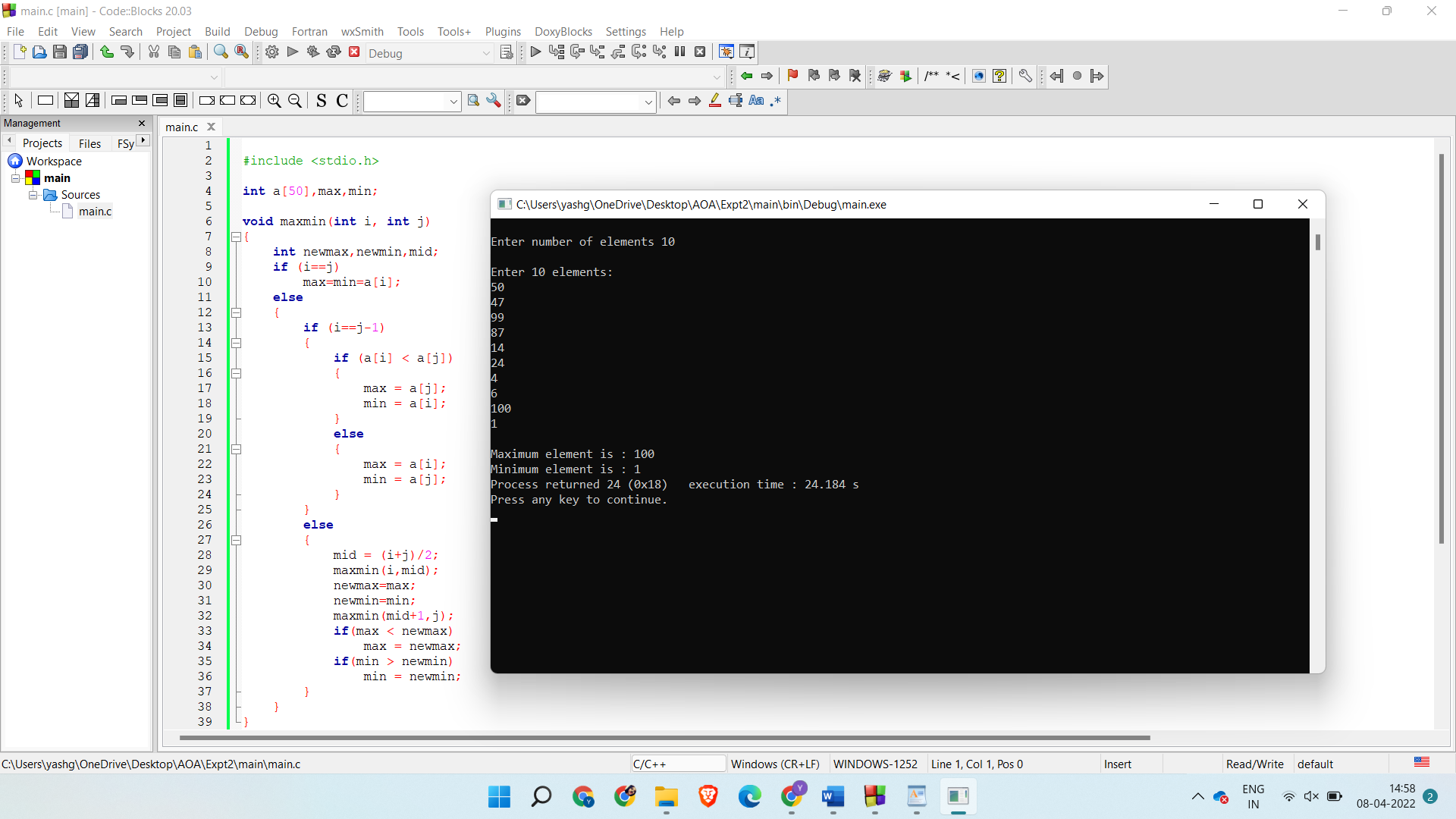
**maxmin(0,n-1);**

**printf("\nMaximum element is : %d ",max);**

**printf("\nMinimum element is : %d ",min);**

**}**

**Output:**



**CONCLUSION:**