Batch: C1	**Roll No.: 16010122257**
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.

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:

The space complexity of iterative binary search is O(1) .

It means that it only requires a constant amount of extra space, regardless of the size of the input array. It only needs two variables to keep track of the range of elements that are to be checked.

Algorithm Recursive Binary Search

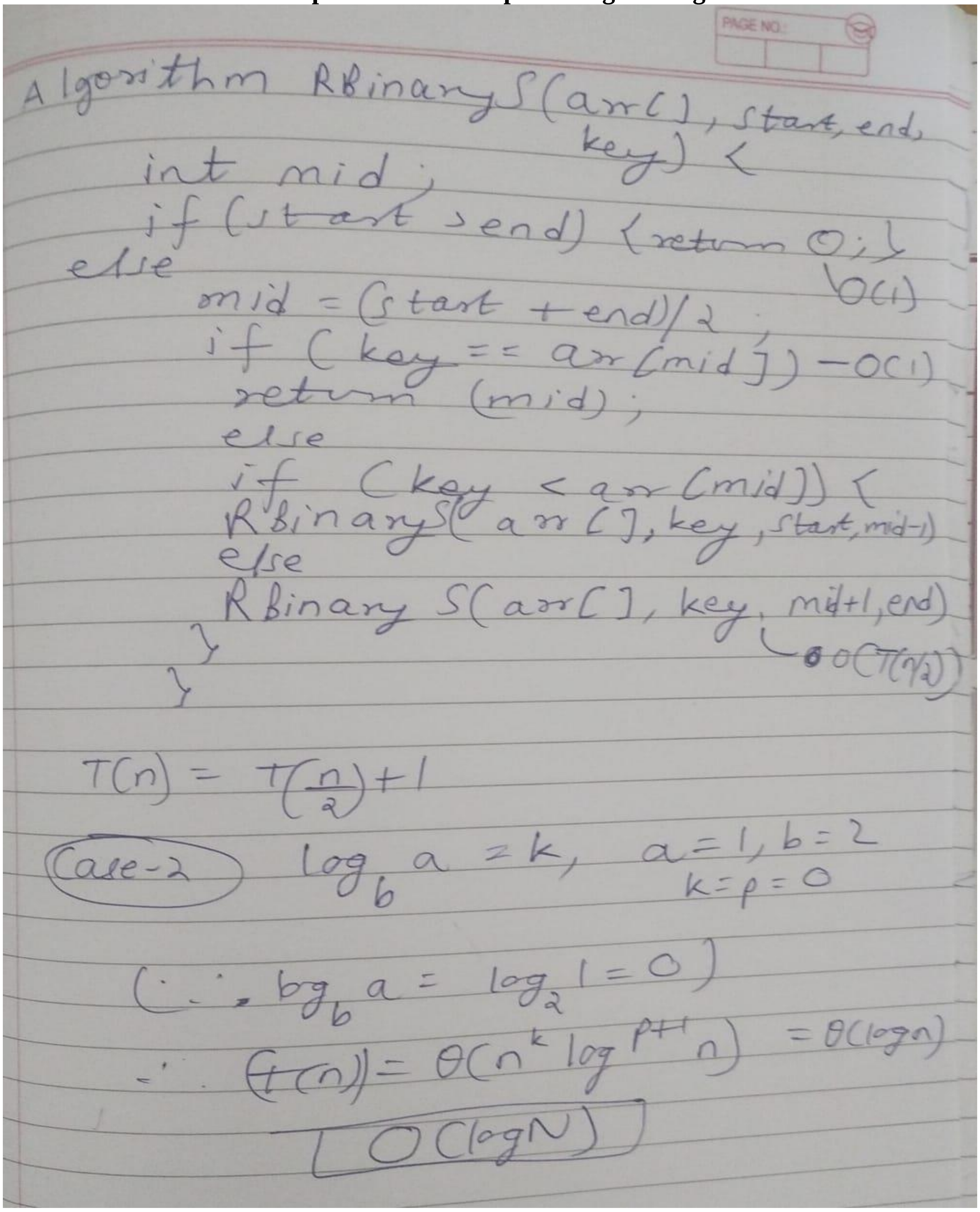
```
int binary_search(int A[], int key, int imin, int imax)
//The algorithm takes as parameters an array A[1.. n] , the search key and lower-higher index pair of the array.
 // Output- The algorithm returns index of the search key in the given array, if it's present.
{
  // test if array is empty
  IF (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:

The space complexity of recursive binary search is O(logN).

It means that it requires a logarithmic amount of extra space, proportional to the size of the input array. This is because in the worst case, there will be logN recursive calls and all these recursive calls will be stacked in memory.

The Time complexity of Binary Search:



PAGE NO.

```
Algorithm RBinaryS(arr[], start, end,
                              key) {
     int mid;
     if (start > end) {return 0;}
                                  O(1)
else
       mid = (start + end)/2 ;
       if ( key == arr[mid]) — O(1)
        return (mid);
        else
        if  ( key < arr[mid]) {
        RBinaryS( arr[], key, start, mid-1)
        else
        RBinaryS(arr[], key, mid+1, end)
                                  O(T(n/2))
       }
     }
```

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Case-2 $\log_b a = k,\quad a=1, b=2$
$k = p = 0$

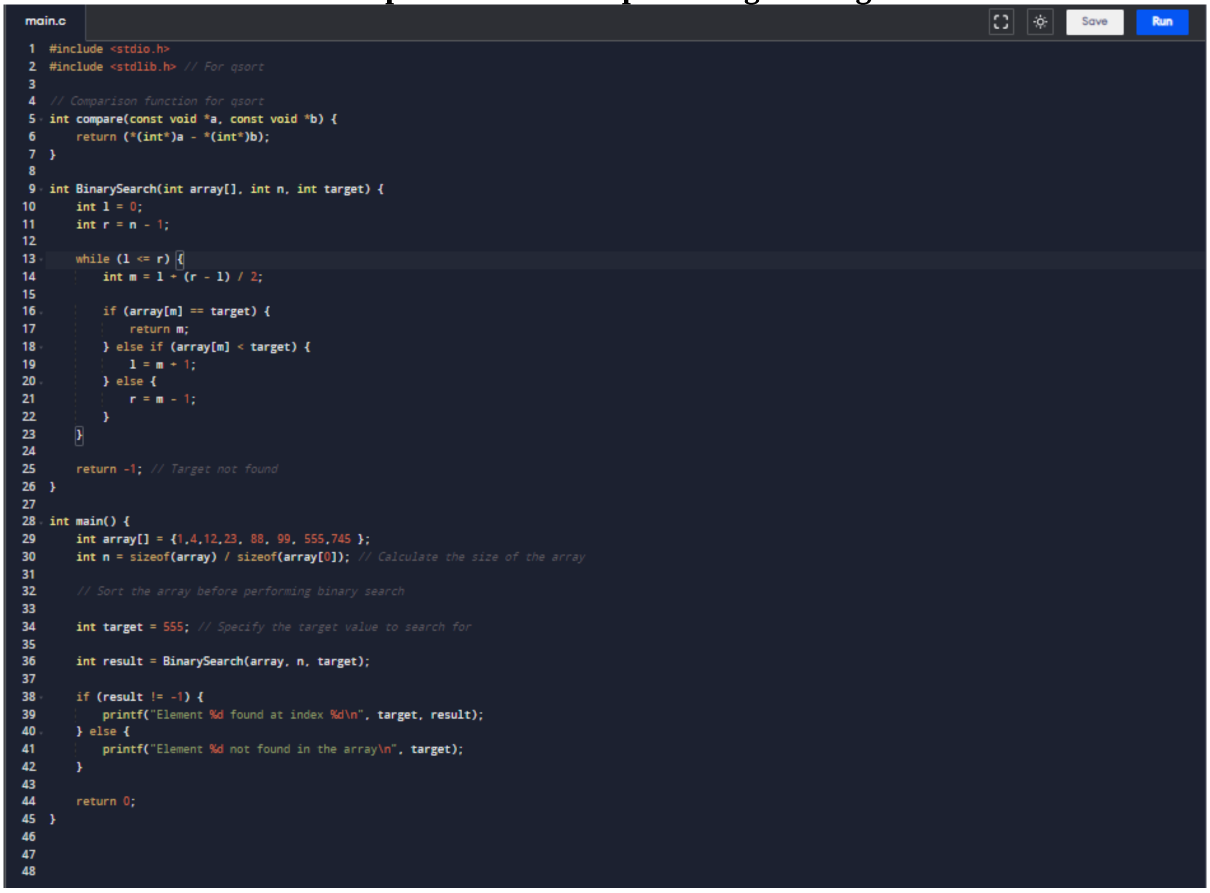
$(\because \log_b a = \log_2 1 = 0)$

$\therefore T(n) = \theta(n^k \log^{p+1} n) = \theta(\log n)$

$\boxed{O(\log N)}$

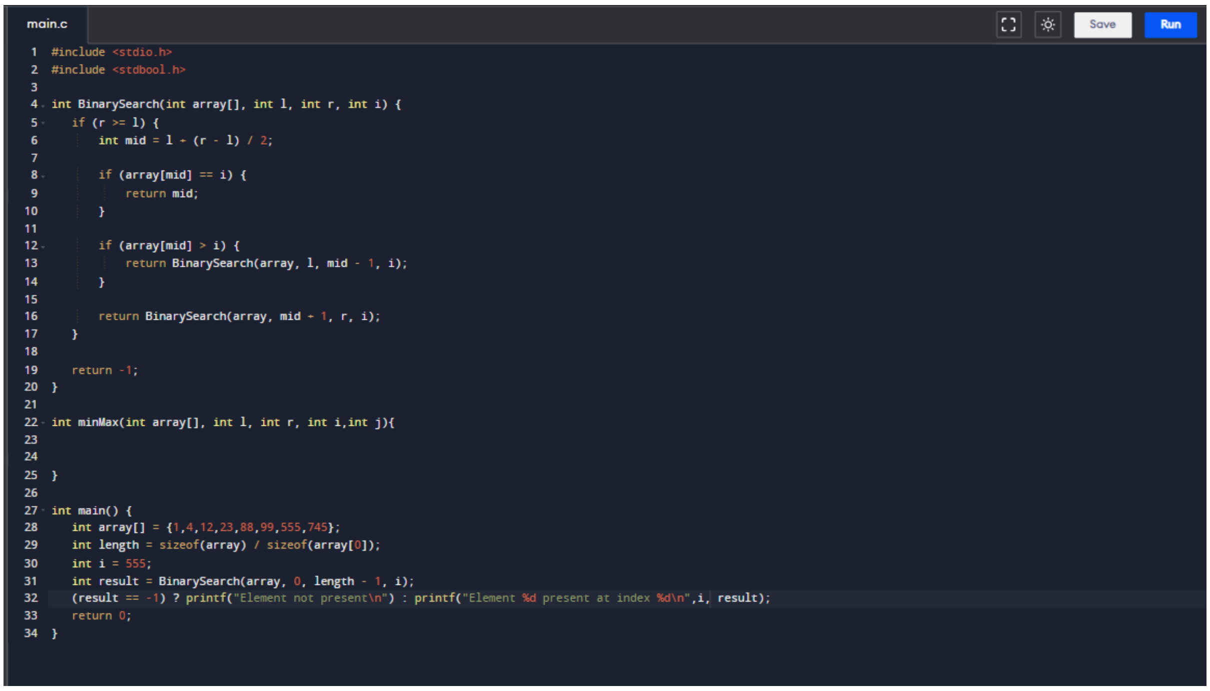
CODE:

Iterative:



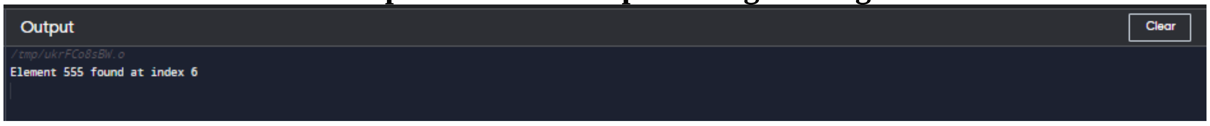
```c
#include <stdio.h>
#include <stdlib.h> // For qsort

// Comparison function for qsort
int compare(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

int BinarySearch(int array[], int n, int target) {
    int l = 0;
    int r = n - 1;

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

        if (array[m] == target) {
            return m;
        } else if (array[m] < target) {
            l = m + 1;
        } else {
            r = m - 1;
        }
    }

    return -1; // Target not found
}

int main() {
    int array[] = {1,4,12,23, 88, 99, 555,745 };
    int n = sizeof(array) / sizeof(array[0]); // Calculate the size of the array

    // Sort the array before performing binary search

    int target = 555; // Specify the target value to search for

    int result = BinarySearch(array, n, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

RECURSIVE:



```c
#include <stdio.h>
#include <stdbool.h>

int BinarySearch(int array[], int l, int r, int i) {
    if (r >= l) {
        int mid = l + (r - l) / 2;

        if (array[mid] == i) {
            return mid;
        }

        if (array[mid] > i) {
            return BinarySearch(array, l, mid - 1, i);
        }

        return BinarySearch(array, mid + 1, r, i);
    }

    return -1;
}

int minMax(int array[], int l, int r, int i,int j){


}

int main() {
    int array[] = {1,4,12,23,88,99,555,745};
    int length = sizeof(array) / sizeof(array[0]);
    int i = 555;
    int result = BinarySearch(array, 0, length - 1, i);
    (result == -1) ? printf("Element not present\n") : printf("Element %d present at index %d\n",i, result);
    return 0;
}
```

OUTPUT:



```
/tmp/ukrFCo8sBW.o
Element 555 found at index 6
```

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];
        }
}
```

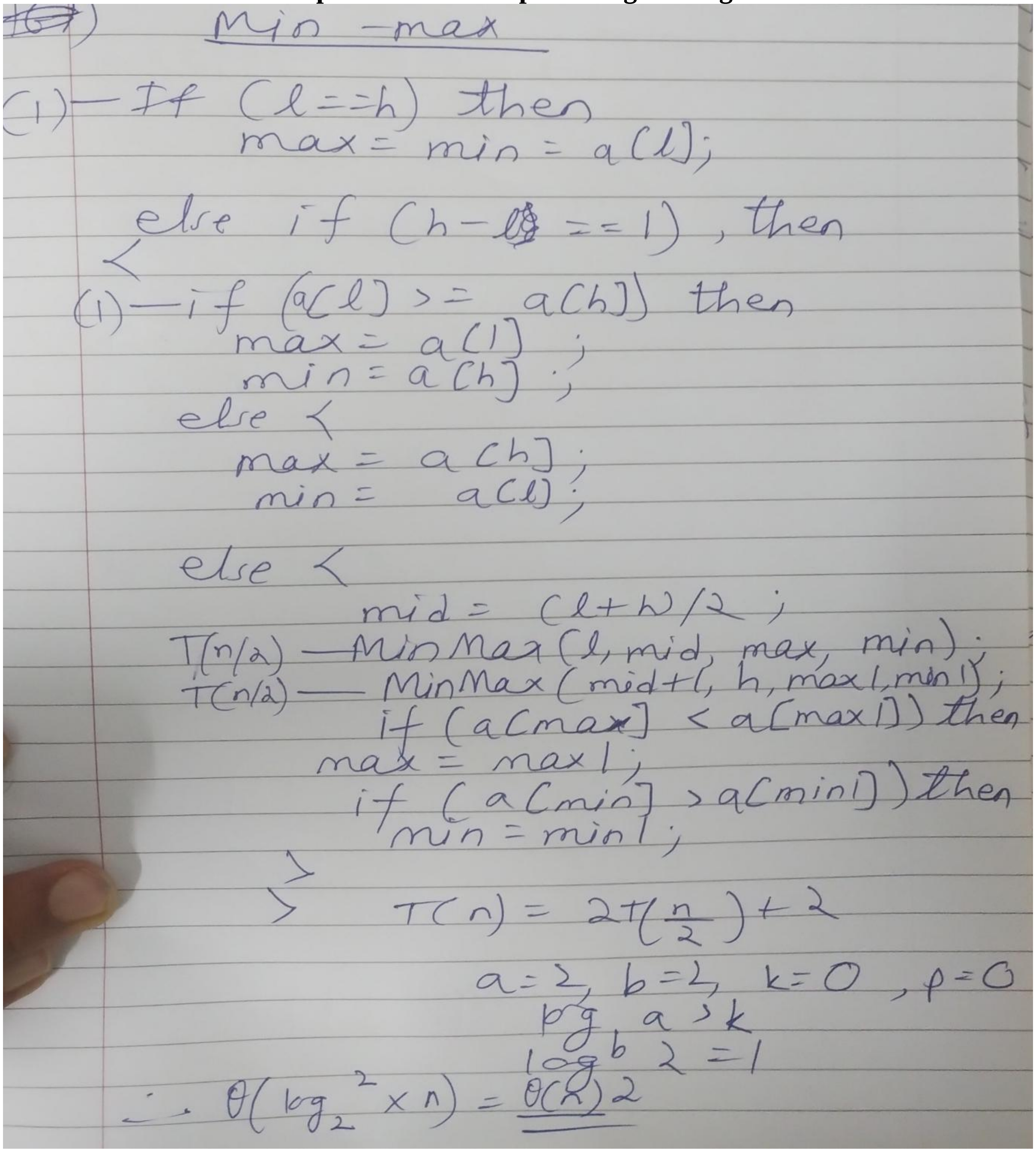
Algorithm: Recursive Max-Min

```
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:

Space complexity is O(1).

Time complexity for Max-Min:



~~6~~ 7) Min - max

```
(1)— If (l==h) then
          max = min = a[l];

     else if (h−l == 1), then
   <
(1)—if (a[l] >= a[h]) then
          max = a[l] ;
          min = a[h] ;
     else <
          max = a[h] ;
          min = a[l] ;

     else <
               mid = (l+h)/2 ;
T(n/2)—MinMax(l, mid, max, min);
T(n/2)—MinMax(mid+1, h, max1, min1);
               if (a[max] < a[max1]) then
          max = max1 ;
               if (a[min] > a[min1]) then
               min = min1 ;
     >
     >
```

$T(n) = 2T(\frac{n}{2}) + 2$

$a = 2,\ b = 2,\ k = 0,\ p = 0$

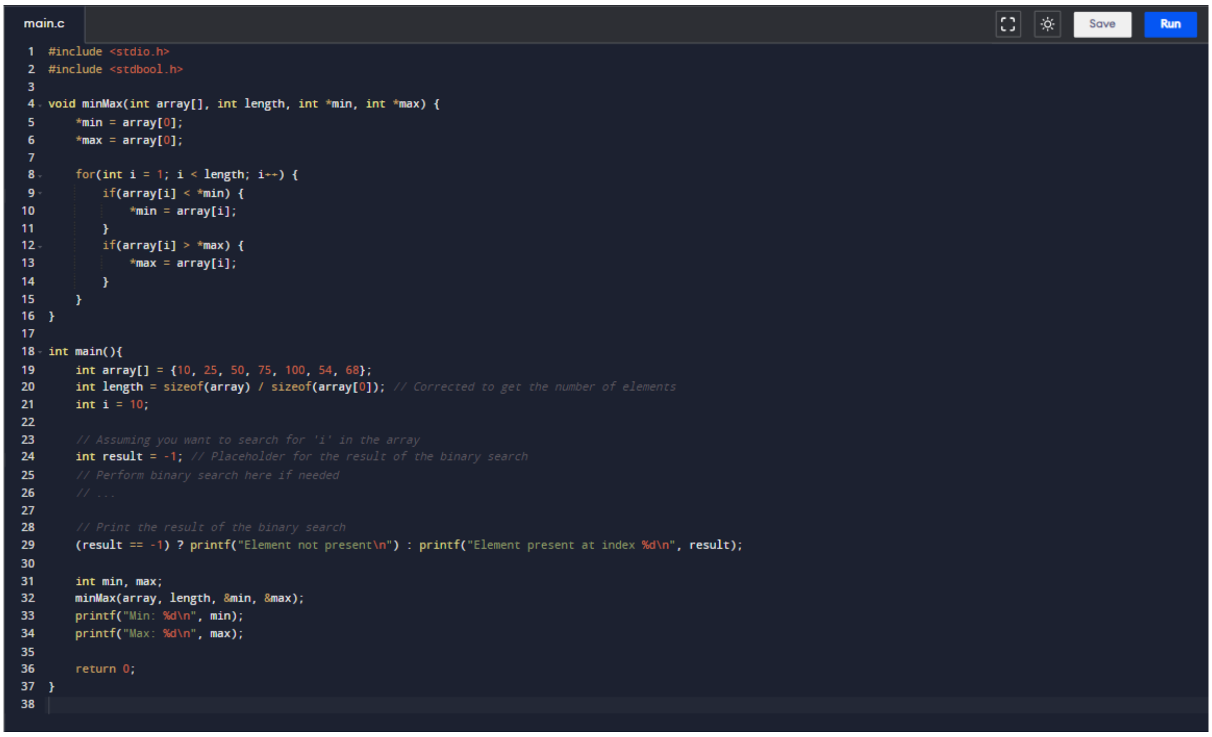
pg, $a > k$

$\log^b 2 = 1$

$\therefore\ \theta(\log_2^2 \times n) = \underline{\theta(n)2}$

CODES:

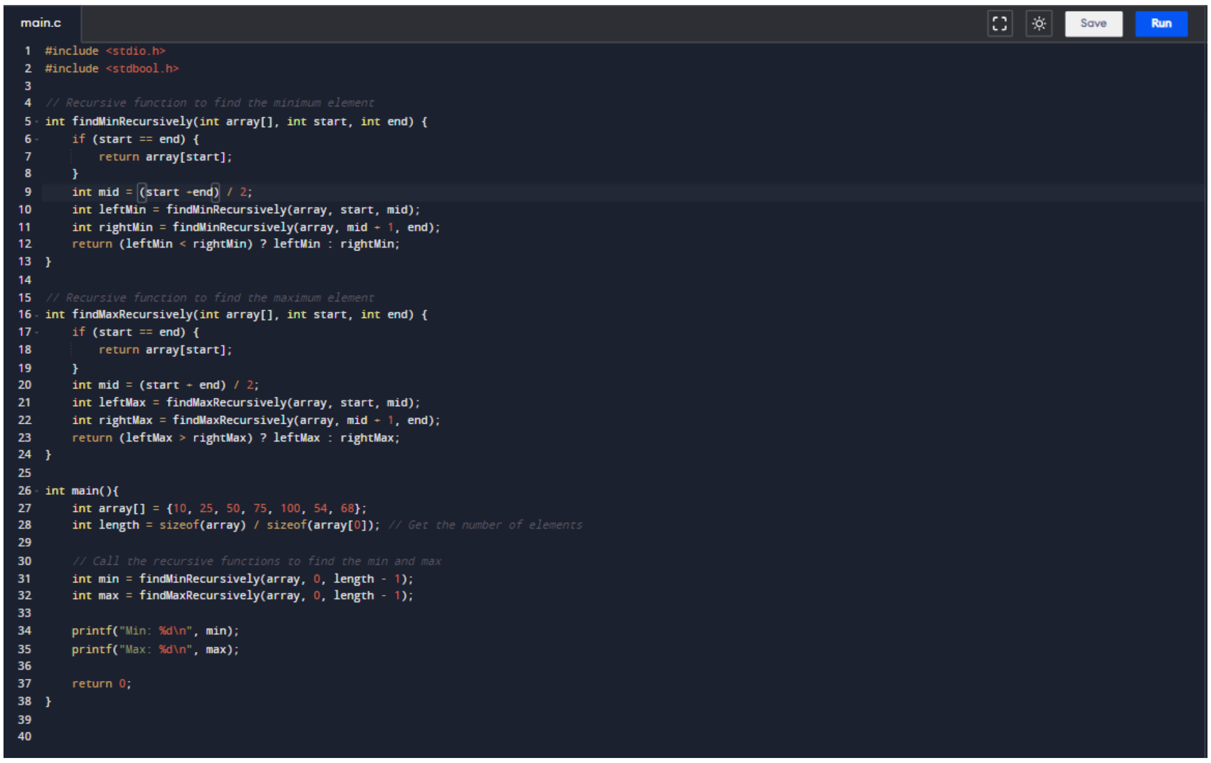
ITERATIVE:



main.c

```c
#include <stdio.h>
#include <stdbool.h>

void minMax(int array[], int length, int *min, int *max) {
    *min = array[0];
    *max = array[0];

    for(int i = 1; i < length; i++) {
        if(array[i] < *min) {
            *min = array[i];
        }
        if(array[i] > *max) {
            *max = array[i];
        }
    }
}

int main(){
    int array[] = {10, 25, 50, 75, 100, 54, 68};
    int length = sizeof(array) / sizeof(array[0]); // Corrected to get the number of elements
    int i = 10;

    // Assuming you want to search for 'i' in the array
    int result = -1; // Placeholder for the result of the binary search
    // Perform binary search here if needed
    // ...

    // Print the result of the binary search
    (result == -1) ? printf("Element not present\n") : printf("Element present at index %d\n", result);

    int min, max;
    minMax(array, length, &min, &max);
    printf("Min: %d\n", min);
    printf("Max: %d\n", max);

    return 0;
}

```

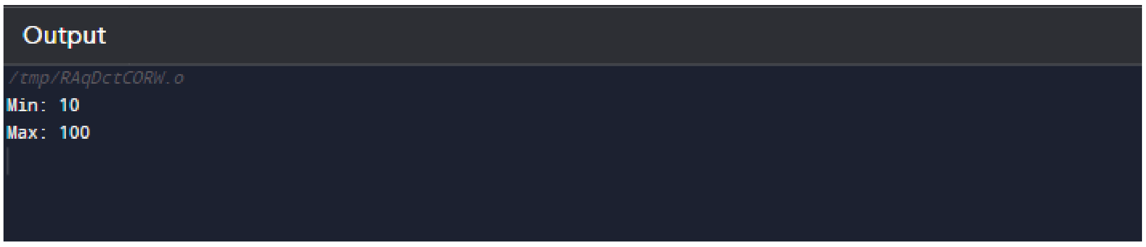
RECURSIVE:



main.c

```c
#include <stdio.h>
#include <stdbool.h>

// Recursive function to find the minimum element
int findMinRecursively(int array[], int start, int end) {
    if (start == end) {
        return array[start];
    }
    int mid = (start +end) / 2;
    int leftMin = findMinRecursively(array, start, mid);
    int rightMin = findMinRecursively(array, mid + 1, end);
    return (leftMin < rightMin) ? leftMin : rightMin;
}

// Recursive function to find the maximum element
int findMaxRecursively(int array[], int start, int end) {
    if (start == end) {
        return array[start];
    }
    int mid = (start + end) / 2;
    int leftMax = findMaxRecursively(array, start, mid);
    int rightMax = findMaxRecursively(array, mid + 1, end);
    return (leftMax > rightMax) ? leftMax : rightMax;
}

int main(){
    int array[] = {10, 25, 50, 75, 100, 54, 68};
    int length = sizeof(array) / sizeof(array[0]); // Get the number of elements

    // Call the recursive functions to find the min and max
    int min = findMinRecursively(array, 0, length - 1);
    int max = findMaxRecursively(array, 0, length - 1);

    printf("Min: %d\n", min);
    printf("Max: %d\n", max);

    return 0;
}


```

OUTPUT:



Output

```
/tmp/RAqDctCORW.o
Min: 10
Max: 100
```

CONCLUSION:

The divide and conquer approach tackles problems by breaking them down into smaller, more manageable subproblems, then integrating their solutions. Binary search and min-max are prime instances of this strategy, adept at efficiently locating an element or pair of elements within an array.