**Batch: C1 Roll No.: 16010122221**

**Experiment No.\_\_\_\_\_\_\_**

**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:**

The time complexity of recursive binary search is O(log n) where n is the number of

elements in the sorted array. This means that in each iteration or recursive call, the search

gets reduced to half of the array size.

**Binary search code:**

#include <bits/stdc++.h>

using namespace std;

void binary(int \*arr, int ele, int n)

{

int l = 0, r = n - 1;

while (l <= r)

{

int mid = (l + r) / 2;

if (arr[mid] == ele)

{

cout << mid << endl;

break;

}

else if (arr[mid] > ele)

r = mid - 1;

else

l = mid + 1;

}

}

void search(int \*arr, int l, int r, int ele)

{

if (arr[(l + r) / 2] == ele)

{

cout << (l + r) / 2 << endl;

return;

}

if (l >= r)

{

cout << "Not Found" << endl;

return;

}

if (arr[(l + r) / 2] > ele)

search(arr, l, (l + r) / 2 - 1, ele);

else

search(arr, (l + r) / 2 + 1, r, ele);

}

int main()

{

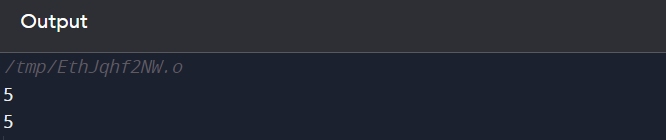
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

binary(arr, 6, 10);

search(arr, 0, 10, 6);

return 0;

}



**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;

}

}

#include <bits/stdc++.h>

using namespace std;

int recMax(int arr[], int len)

{

if (len == 1)

return arr[0];

return max(arr[len - 1], recMax(arr, len - 1));

}

int recMin(int arr[], int len)

{

if (len == 1)

return arr[0];

return min(arr[len - 1], recMin(arr, len - 1));

}

void minMax(int\* arr,int n){

int max=arr[0],min=arr[0];

for(int i=1;i<n;i++){

if(arr[i]>max)

max=arr[i];

if(arr[i]<min)

min=arr[i];

}

cout<<max<<" "<<min<<endl;

}

int main()

{

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

cout << recMax(arr, 10) << endl;

cout << recMin(arr, 10) << endl;

minMax(arr,10);

return 0;

}

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

Time complexity is O(n)

**Time complexity for Max-Min:**

Space complexity is O(1).

**Code:**

#include <bits/stdc++.h>

using namespace std;

int recMax(int arr[], int len)

{

if (len == 1)

return arr[0];

return max(arr[len - 1], recMax(arr, len - 1));

}

int recMin(int arr[], int len)

{

if (len == 1)

return arr[0];

return min(arr[len - 1], recMin(arr, len - 1));

}

void minMax(int\* arr,int n){

int max=arr[0],min=arr[0];

for(int i=1;i<n;i++){

if(arr[i]>max)

max=arr[i];

if(arr[i]<min)

min=arr[i];

}

cout<<max<<" "<<min<<endl;

}

int main()

{

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

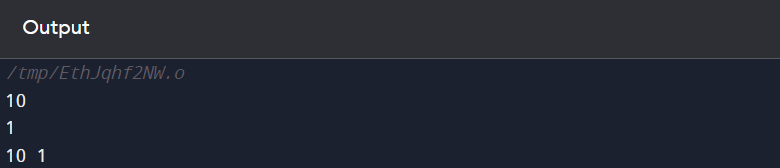
cout << recMax(arr, 10) << endl;

cout << recMin(arr, 10) << endl;

minMax(arr,10);

return 0;

}



**CONCLUSION:**

The divide and conquer strategy solves problems by dividing them into smaller subproblems

and combining their solutions. Binary search and min-max are two examples of this strategy

that can find an element or a pair of elements in an array efficiently.