**Batch: B3 Roll No.: 121**

**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**
8. **https://www.knowledgehut.com/blog/programming/time-complexity-of-binary-search**
9. **https://www.javatpoint.com/daa-binary-search**
10. **https://www.tutorialspoint.com/design\_and\_analysis\_of\_algorithms/design\_and\_analysis\_of\_algorithms\_max\_min\_problem.htm**

**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:**

1. The space complexity of Iterative Binary Search is O(1).
2. Two variables are required to keep track of the number of elements that need to be checked. Additional data is not necessary.

**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:**

1. The space complexity of Recursive Binary Search is O(logn).
2. In the worst-case scenario, there will be logn number of recursive calls, and all these calls are stored in memory. Recursive calls will be stored in memory if there are any comparisons. An average-case complexity analysis shows that O(logn), which is the average memory, will likewise be stacked into memory.

**The Time complexity of Binary Search:**

**Input:** an array A of size n, already sorted in ascending or descending order.

**Output:** analyse to search an element item in the stored array of size n.

**Logic:** Let T(n) = number of comparisons of an item with n elements in a sorted array.

Set BEG = 1 and END = n

Find mid = (BED + END)/2

Compare the search item with the mid item

**Case 1:** Item = A[mid], then LOCATION = mid. It is the best case and T(n) = 1.

**Case 2:** Item ≠ A[mid], then the array will be split into two equal parts of size n/2.

Again, the midpoint of the half-sorted array is found and compared with the search element.

The process is repeated until the search element is found.

(Time to compare the search element with the mid element, then with the selected half of the array)

Repeating the same process i times,

**Stopping Condition:** T(1) = 1

At least there will be only one term left that’s why that term will compare out, and only one comparison will be done that’s why T(1) = 1 is the last term of the equation and it will be equal to 1.

**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 equal to the size of the recursion call stack, which is equal to the height of the recursion tree i.e., O(logn).

**Time complexity for Max-Min:**

Let T(n) be the number of comparisons made by Max\_Min(x,y), where the number of elements n = y – x + 1.

If T(n) represents the numbers, then the recurrence relation can be represented as

Let us assume that n is in the form of power of 2. Hence, n = 2k where k is the height of the recursion tree.

So,

i.e., the time complexity is O(n).

**CONCLUSION:**

Thus, in this experiment, the concept of Divide and Conquer Approach, in the case of Binary Search and Max-Min Algorithm, has been implemented. It has been found that although Recursive method is easier to debug by programmers, it is slower than iterative method. This is because the recursion method involves stack memory where all the recursion calls are stored in memory, each one occupying space, whereas the iteration method involves the use of a few additional variables which take up far less space. Less space taken means less time required for processing. Thus, iterative method, which takes less space, is faster than recursive method for both Binary Search implementation and Min-Max implementation.