Batch: B1 **Roll No.: 1711072**

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:

Sr. No	Objective
CO 1	Compare and demonstrate the efficiency of algorithms using asymptotic complexity notations.
CO 2	Analyze and solve problems for divide and conquer strategy, greedy method, dynamic programming approach and backtracking and branch & bound policies.
CO 3	Analyze and solve problems for different string matching algorithms.

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 algortithms”,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:

It is O(1) i.e constant space complexity. This is because irrespective of the length of the array passed we can eventually find (or maybe not) the required key, by using certain variables.

Algorithm RecursiveBinarySearch

```
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:

If one uses a recursive approach, then at each stage, we have to make a recursive call. That means leaving the current invocation on the stack, and calling a new one. When we are *k* levels deep, we have got *k* lots of stack frame, so the space complexity ends up being O(k).

The Time complexity of Binary Search:

It simply comes down to dividing an array of size 'n' into halves at each iteration until we reach size of one. Hence, mathematically,

$1 = N / 2^x$

multiply by 2^x:

$2^x = N$

now do the $\log_2$ on both sides:

$\log_2(2^x) \quad = \log_2 N$

$x * \log_2(2) = \log_2 N$

$x * 1 \qquad = \log_2 N$

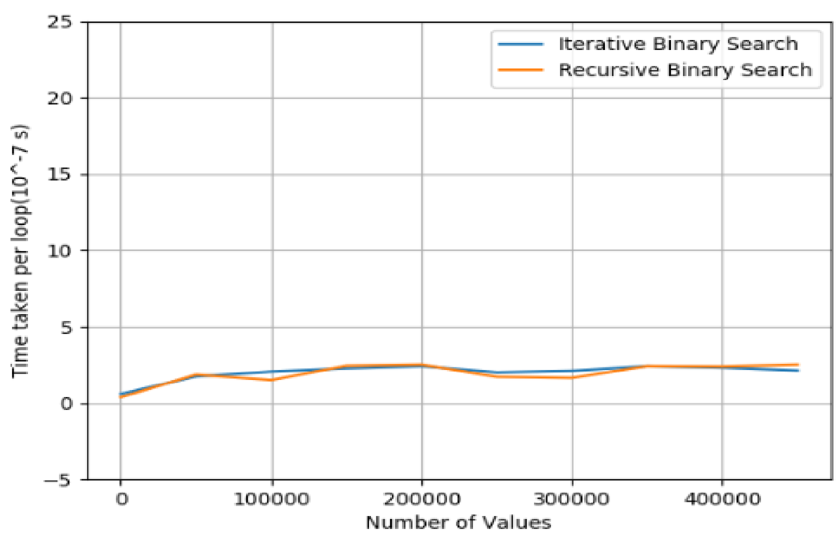
Thus, we finally end up with logarithmic time complexity O(log N).

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import bisect, time, random

x1=[]
y1=[]
x2=[]
y2=[]
inc,n=20,50
for i in range(10):
  arr = list(range(1,n+1))
  tic=time.time()
  #Iterative
  def BinarySearch(arr,x):
    val=bisect.bisect_left(arr,x)
    if val!= len(arr) and arr[val]==x:
      return val
    else:
      return -1
  tac=time.time()
  x1.append(len(arr))
  y1.append((tac-tic)*(10**5))
  tac=time.time()
  #Recursive
  def binary_search(arr, start, end, key):
    if not start < end:
        return -1

    mid = (start + end)//2
    if arr[mid] < key:
        return binary_search(arr, mid + 1, end, key)
    elif arr[mid] > key:
        return binary_search(arr, start, mid, key)
    else:
```

```
        return mid
    toe=time.time()
    x2.append(len(arr))
    y2.append((toe-tac)*(10**5))

    x=int(1)
    res=BinarySearch(arr,x)
    recursive=binary_search(arr,0,n,x)
    n=n+50
    if res!=-1:
      print("Element present at:",res+1)
    else:
      print("Not present")
    if recursive!=-1:
      print("Element found recursively at: ", res+1)
    else:
      print("Not present")

plt.ylim((-10,10))
plt.plot(x1, y1, label = 'Iterative Binary Search')
plt.plot(x2,y2, label='Recursive Binary Search')
plt.legend()
plt.grid()
plt.xlabel(' Number of Values ')
plt.ylabel(' Time taken per loop(10^-7 s) ')
plt.savefig('graph.png')
```

Graph:



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 subproblems.  Find where to split the set.
              int mid=(i+j)/2;
             // Solve the  subproblems.
              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:

Iterative simply requires O(1) as we obtain our min and max element by simply traversing through the array. Recursive ends up requiring O(k), where k is the depth of stack which we get after the recursive calls.

Time complexity for Max-Min:

Iterative algorithm simply requires O(n) as we have to either way to traverse the entire list, be it best or worst case.
Recursive algorithm eventually requires O(n) but more specifically it requires:

Each call to partition performs a constant amount of work, plus 2 additional recursive calls, each with *half* of the input index range. We can thus construct a *recurrence relation* for the time complexity function:
$T(n) = 2T(n/2) + C$
This expands to a geometric series C * (1 + 2 + 4 + ...), which continues for log n terms (because at each level of recursion the input size halves, so it decreases geometrically to the stopping condition n = 2).

$T(n) = 2*T(n/2) + 2$

$T(n) = 2^k * T(n/ 2^k) + 2^k + 2^{k-1} + \ldots + 2$

$k = \log n - 1$

$T(n) = n/2 + 2(n/2 - 1)$

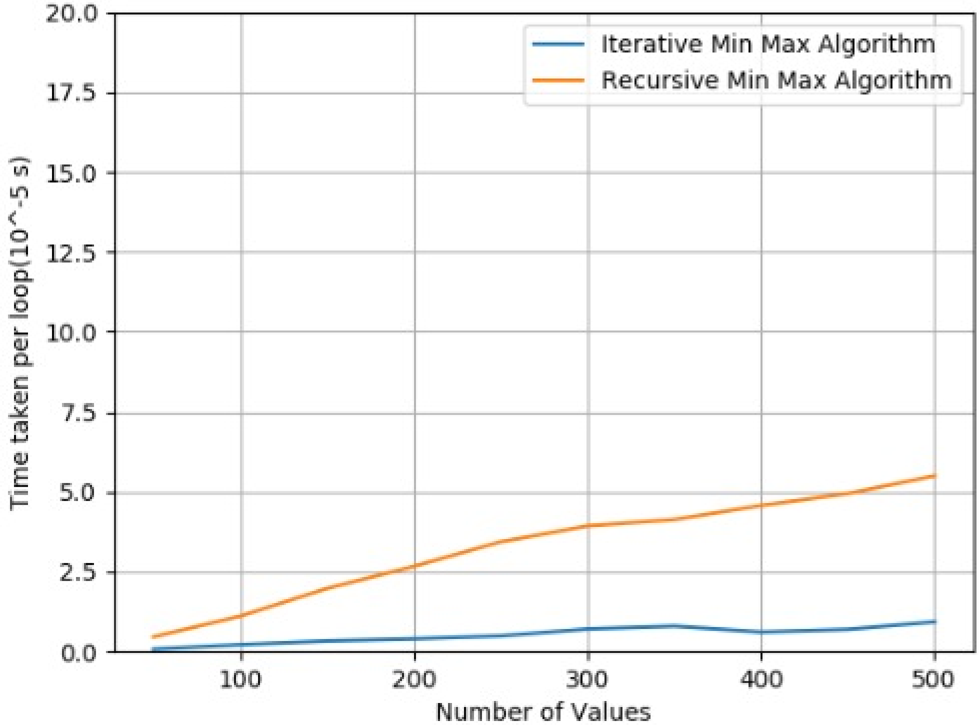
$T(n) = 3n/2 - 2$

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import time, random

x1=[]
x2=[]
y1=[]
y2=[]
inc, n=20,50
mini,maxi,tic,tac,toe=0,0,0,0,0
for i in range(10):
  arr=list(range(1,n+1))

  #Iterative
  def IterativeMinMax():
    tic=time.time()
```

```
    mini,maxi=arr[0],arr[0]
    for j in range(n):
      if(mini>arr[j]):
        mini=arr[j]
      else:
        maxi=arr[j]
    tac=time.time()
    x1.append(len(arr))
    y1.append((tac-tic)*(10**4))
    return mini,maxi

    #Recursive
  def RecursiveMinMax(low,high):
    if(low==high):
      maxi=mini=arr[high]
      return maxi,mini
    elif(low==high-1):
      return max(arr[low],arr[high]),min(arr[low],arr[high])
    else:
      mid=(low+high)//2
      maxi1,mini1=RecursiveMinMax(low,mid)
      maxi2,mini2=RecursiveMinMax(mid+1,high)
      return max(maxi1,maxi2),min(mini1,mini2)

  minimum,maximum=IterativeMinMax()
  tac=time.time()
  recmax,recmin=RecursiveMinMax(0,n-1)
  toe=time.time()
  x2.append(len(arr))
  y2.append((toe-tac)*(10**4))
  print("Iterative Minimum, Maximum: ", minimum, maximum)
  print("Recursive Minimum, Maximum: ", recmin, recmax)
  n=n+50

plt.ylim((0,25))
plt.plot(x1, y1, label = 'Iterative Min Max Algorithm')
plt.plot(x2,y2, label='Recursive Min Max Algorithm')
plt.legend()
plt.grid()
plt.xlabel(' Number of Values ')
plt.ylabel(' Time taken per loop(10^-5 s) ')
plt.savefig('graph.png')
```

Graph:



CONCLUSION: In this way, we successfully implemented divide and conquer approach by implementing min-max algorithm.