**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 subproblems 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));

}

}



|  |
| --- |
| **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 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 Iterative 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.

{

// 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;

}

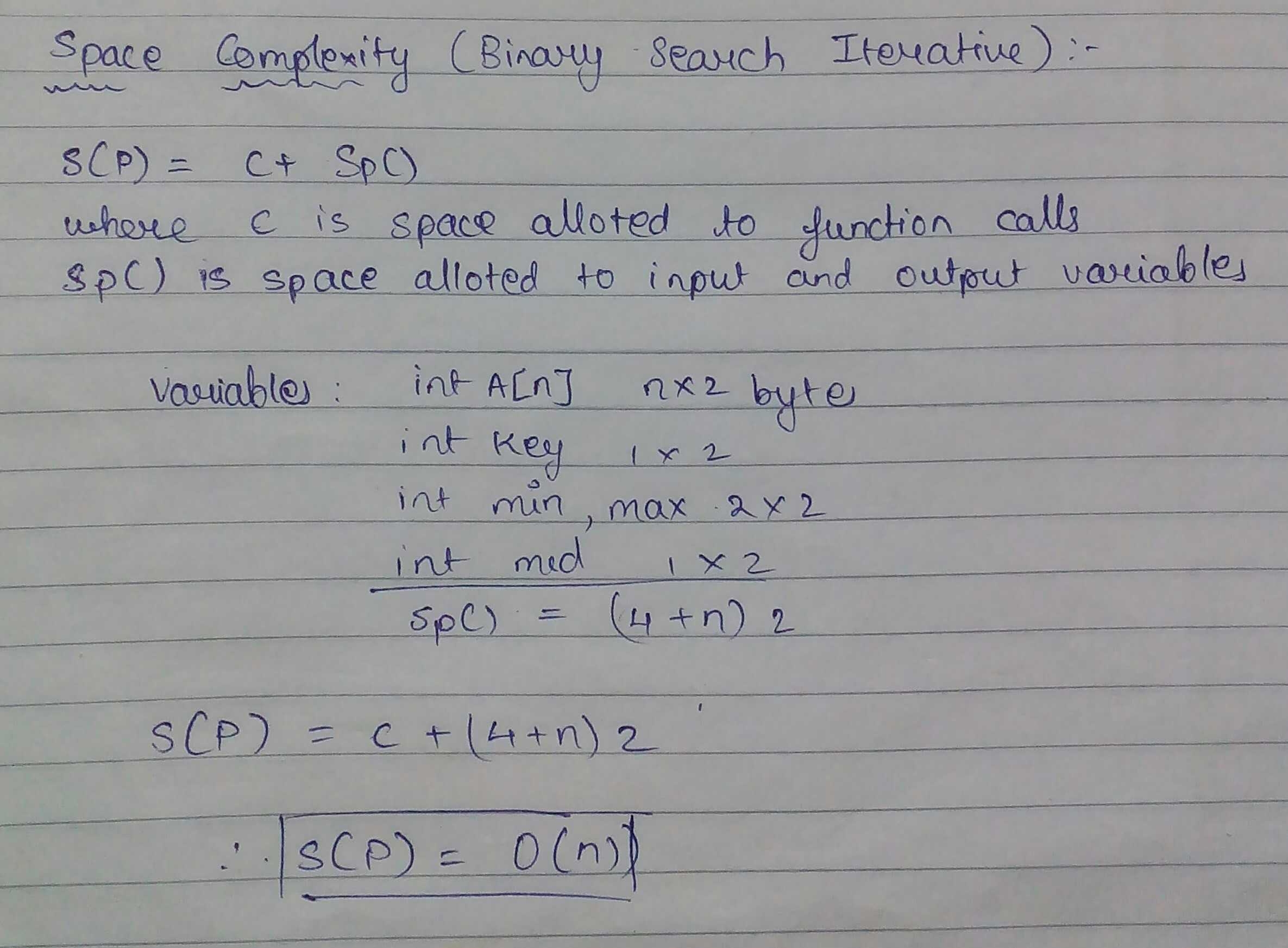
// key was not found

**RETURN** KEY\_NOT\_FOUND;

}

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

With iterative code, we are allocating one variable (O(1) space) plus a single stack frame for the call (O(1) space). The while loop doesn't ever allocate anything extra, either by creating new variables or object instances, or by making more recursive calls. So the only space that we need, is for the whole call, is the O(1) space taken up by the variable we create and the rest of the stack frame.

****

**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:**

Best case - O (1) comparisons

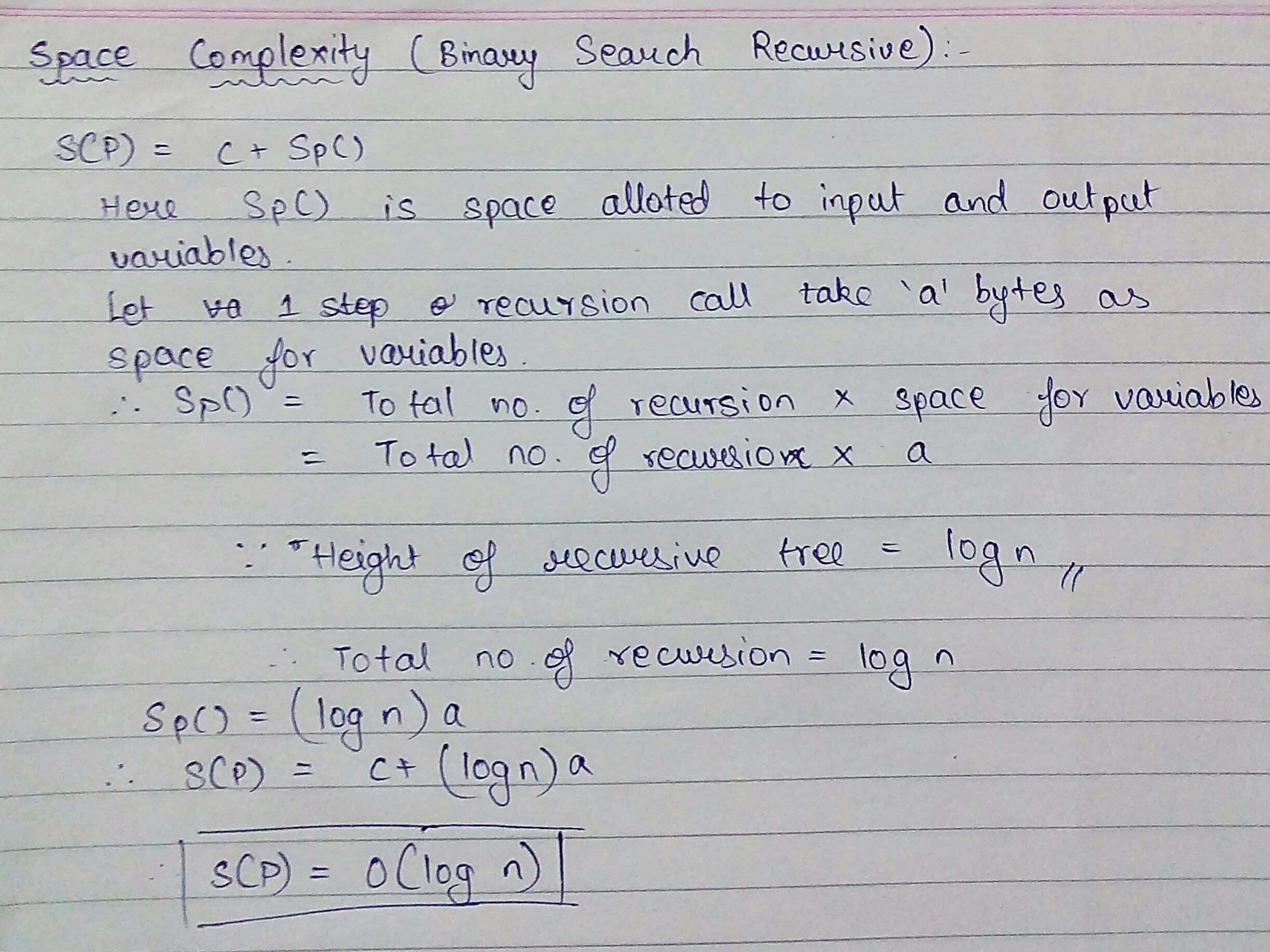
In the best case, the item X is the middle in the array A. A constant number of comparisons (actually just 1) are required.

Worst case - O (log n) comparisons

In the worst case, the item X does not exist in the array A at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving can be done ceiling (lg n) times. Thus, ceiling (lg n) comparisons are required.

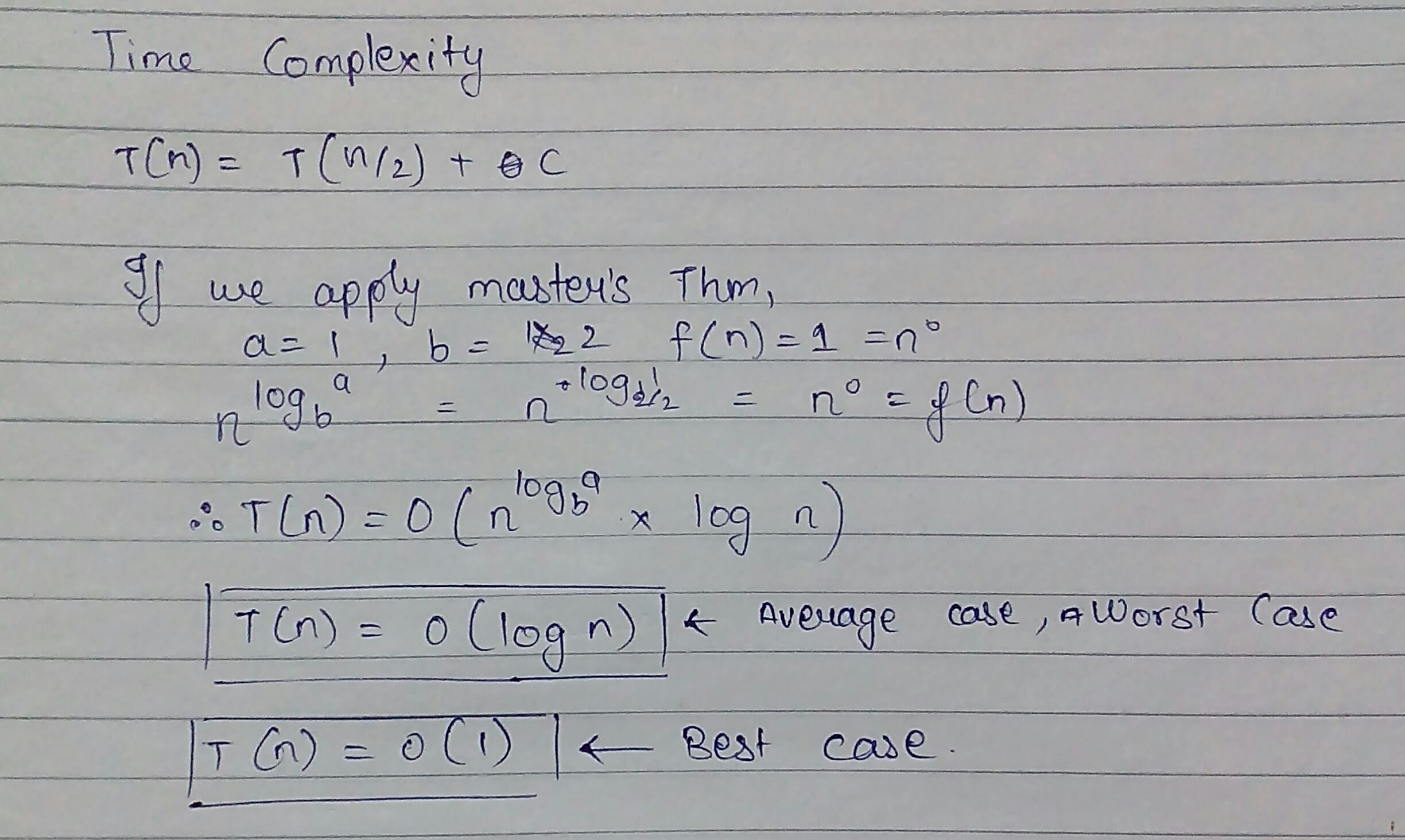
Average case - O (log n) comparisons

To find the average case, take the sum over all elements of the product of number of comparisons required to find each element and the probability of searching for that element. To simplify the analysis, assume that no item which is not in A will be searched for, and that the probabilities of searching for each element are uniform.



**The Time complexity of Binary Search:**

Binary search has a best case efficiency of O(1) and worst case (average case) efficiency of O(log n).

****

**Algorithm Straight Max-Min:**

**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;

}

**Space complexity of Max-Min Algorithm:**

Space complexity =input + extra space

=2n + logn (stack used for recursion)

=O(n)

Space used by stack is number of levels or distinct function calls made by program.

**Time complexity for Max-Min Algorithm:**

Time Complexity: O(n)

Total number of comparisons: let number of comparisons be T(n). T(n) can be written as follows:

Algorithmic Paradigm: Divide and Conquer

T(n) = T(floor(n/2)) + T(ceil(n/2)) + 2

T (2) = 1

T (1) = 0

If n is a power of 2, then we can write T(n) as:

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

After solving above recursion, we get

T(n) = 3/2n -2

Thus, the approach does 3/2n -2 comparisons if n is a power of 2. And it does more than 3/2n -2 comparisons if n is not a power of 2.

**Implementation Code:**

Max-Min

import java.util.\*;  
  
class Mm  
{   
 public static int M=-999,m=999,m2,M2;  
   
 public static void main(String[] args)  
 {  
 Scanner sc=new Scanner(System.in);  
 Random rand=new Random();  
 int i;  
 System.out.print("Enter the no. of inputs: ");  
 int n=sc.nextInt();  
 int a[]=new int[n];  
 for(i=0;i<n;i++)  
 {  
 a[i]=rand.nextInt(100);  
 }  
 for(i=0;i<n;i++)  
 {  
 System.out.print(a[i]+" ");  
 }  
 Mm obj=new Mm();  
 obj.find(a,0,n-1,Mm.M,Mm.m);  
 obj.disp(Mm.M,Mm.m);  
 }  
   
 public void find(int a[],int si, int ei, int M, int m)  
 {  
 if(si==ei)  
 {  
 Mm.m2=a[si];  
 Mm.M2=a[si];  
 if(Mm.m2<Mm.m)  
 {  
 Mm.m=Mm.m2;  
 }  
 if(Mm.M2>Mm.M)  
 {  
 Mm.M=Mm.M2;  
 }  
 }  
 else if(si==ei-1)  
 {  
 if(a[si]<a[ei])  
 {  
 Mm.m2=a[si];  
 Mm.M2=a[ei];  
 }  
 else  
 {  
 Mm.M2=a[si];  
 Mm.m2=a[ei];  
 }  
 if(Mm.m2<Mm.m)  
 {  
 Mm.m=Mm.m2;  
 }  
 if(Mm.M2>Mm.M)  
 {  
 Mm.M=Mm.M2;  
 }  
 }  
 else  
 {  
 int mid=(si+ei)/2;  
 find(a,si,mid,Mm.M,Mm.m);  
 find(a,mid+1,ei,Mm.M,Mm.m);  
 }  
 }  
   
 public void disp(int M,int m)  
 {  
 System.out.println("Max: "+Mm.M+" Min: "+Mm.m);  
 }  
   
}

Binary Search

import java.util.\*;

class Bin

{

public static int flag=0;

public static void main(String[] args)

{

Scanner sc=new Scanner(System.in);

int i;

System.out.println("Enter the no. of inputs: ");

int n=sc.nextInt();

int a[]=new int[n];

System.out.println("Enter sorted array: ");

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

{

a[i]=sc.nextInt();

}

System.out.println("Entered array is: ");

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

{

System.out.print(a[i]+" ");

}

System.out.println("\n Enter the no. to be searched: ");

int s=sc.nextInt();

Bin obj=new Bin();

obj.find(a,0,n-1,s);

}

public void find(int a[],int si,int ei,int s)

{

if(si>ei)

{

System.out.println("Not found");

}

else

{

int mid=(si+ei)/2;

if(s==a[mid])

{

System.out.println("Found at position "+(mid+1));

Bin.flag=1;

}

else if(s<a[mid])

{

ei=mid-1;

find(a,si,ei,s);

}

else

{

si=mid+1;

find(a,si,ei,s);

}

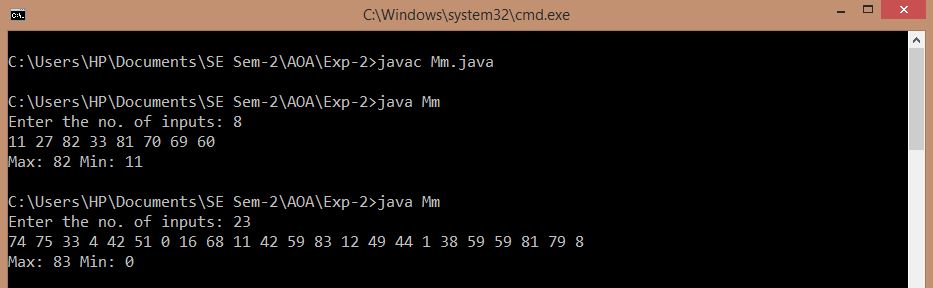
}

}

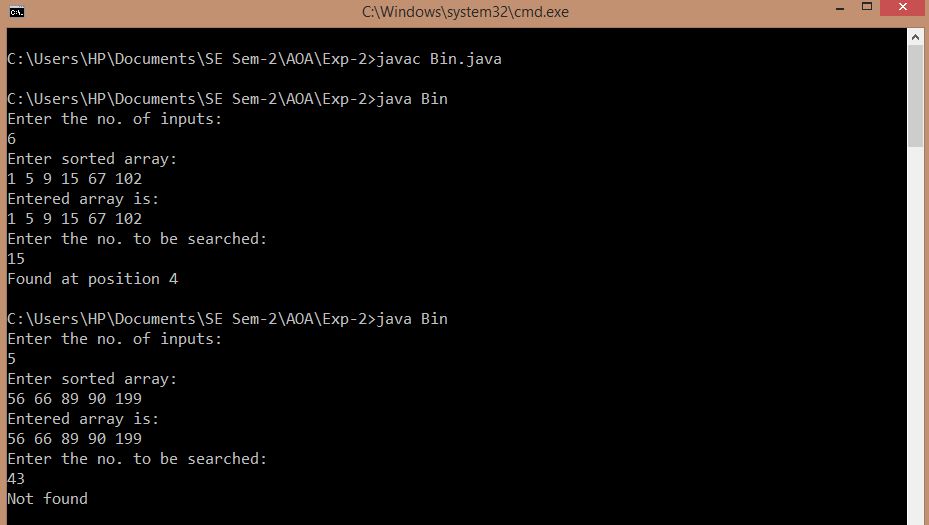
}

**Sample Output:**

Max-Min



Binary Search



**CONCLUSION:**

Therefore, divide and conquer strategy and its applications ere studied and a program to find the maximum and minimum among given numbers and recursive binary search was implemented using this strategy. Further, the time and space complexity of the algorithms of the above mentioned programs was also calculated.