**Experiment No: 1 Date:- 02-04-2021**

**AIM:** Iterative and Recursive Binary Search Algorithm using Divide and Conquer & estimating its step count

**THEORY:**

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until its narrowed down the possible locations to just one.

Its time complexity is O(logn)

Binary Search follows divide and conquer approach in which, the list is divided into two halves and the item to be searched is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match. I.E if the element to be searched is smaller than the middle element of the list, then the search will further continue in the first half of the list. Likewise, search continues in the latter half of the list if the middle element is smaller than the item to be searched.

The major difference between the recursive and iterative versions of the binary search algorithm is that the space complexity of the iterative version is O(1) whereas the space complexity of the recursive version is O(logn). Hence the iterative version is a little more efficient than the recursive counterpart.

**ALGORITHMS:**

* **Iterative Binary Search**

AlgorithmBinSearch(a,n,x)

// Given an array a[l:n] of elementsin nondecreasing

// order,n >0,determine whether x is present, and

// if so ,return j such that x = a[j]; else return 0.

{

low :=1;high :=n;

**while** (low<=high) do

{

mid:=[(low+high)/2];

**if** (x <a[mid])then high :=mid

**else if** (a >a[mid]) then low :=mid+ 1;

**else**

return mid;

}

return0;

}

* **Recursive Binary Search**

AlgorithmBinSrch(a,i,l,x)

Given an array a[i:l] of elements in nondecreasing

order,1<i <I,determine whether x is present, and

if so, return j such that x = a[j]; else return 0.

{

**if** (l=i) **then** // If Small(P)

{

**if** (x = a[i]) **then** return i;

**else** return0;

}

**else**

{ //ReduceP into a smaller subproblem.

mid:=[(i+l)/2];

**if** (x = a[mid]) **then** **return** mid

**else if**(x <a[mid]) **then**

**return** BinSrch(a, low, mid-1,x);

**else return** BinSrch(a,mid+1,1,x);

}

}

**Recursive Tree call Diagram:**

**Time Complexity:**

The time complexity of Binary Search can be written as

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

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is :

According to Master’s theorem

a=1,b=2

n^(log(base b)a)=n^(0) =1

T(n) = n^(0)\*U(n)

U(n)->h(n)=F(n)/(n^0)

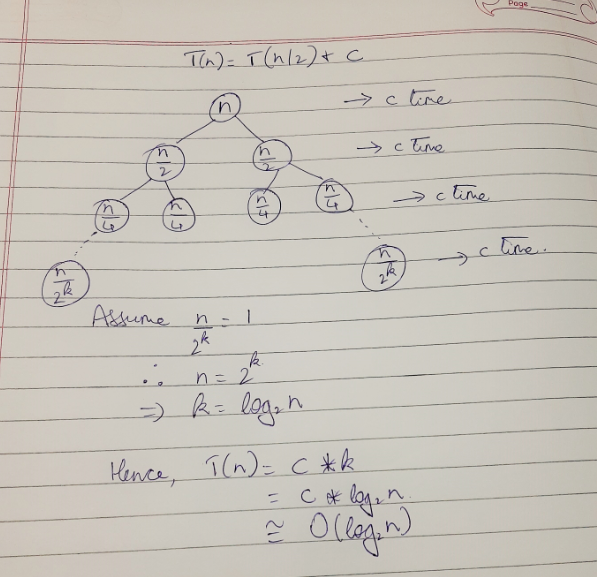
= c/(n^0) =>c=>r=0 =

=>i=0

(Log(n))^i+1/(i+1) =(Log(n))

Therefore, the time complexity is T(n) = 1\*Log(n)=0(Log(n))

Shown below, is the recursive tree diagram:



**Space Complexity:**

In case of iterative binary search, the space complexity is just O(1), owing to the mid variable and in case of recursive binary search the space complexity is O(logn).

**Problem Tracing:**

Consider sorted array a={1,2,3,4,5}. Let x=4(element to be searched)

Here size of array n=5.

* Using iterative binsearch

Int pos = binsearch(a,n,x) //inside main count=1 for assignment

//inside binsearch func

Int low=0,high=n-1; //count=3 for 2 assignments

Iteration 1: 0<=4

Count=4 //while loop condition

Mid = (0+4)/2=2 //count=5 for assignment

if condition is false //count=6

else if condition also false //count=7

**else** condition **true**:

hence low=mid+1=2+1=3 //count=8

Iteration 2: 3<=4

Count=9 //while loop condition

Mid = (3+4)/2=3 //count=10 for assignment

**if** condition is **true:** //count=11

**//found x at mid i.e at 3**

return mid //count=12 for return

**final count=12**

* Using recursive binsearch

Int pos = binsearch(a,0,n-1,x) //inside main count=1 for assignment

//inside binsearch func

low=0,high=n-1,x=4; //Assigned due to function params

count=2 //if condition

//if condition is false, since 0!=4. Hence else statement gets executed.

Mid=(low+high)/2=(0+4)/2=2 //count=3

Count=4 //if condition

//if condition is false x!=a[mid] i.e 4!=3

Count=5 //else If condition

//else if condition is false since 3<4 ie a[mid]<x

Count=6 //else condition gets executed and count increments for return

**//Re-entering the function**

low=3,high=4,x=4 //function call assignments

count=7 //if condition

//if condition is false, since 3!=4. Hence else statement gets executed.

Mid=(low+high)/2=(3+4)/2=3 //count=8

Count=9 //if condition

**//found x at mid i.e. at 3**

return mid //count=10 for return

**final count=10**

PROGRAM IMPLEMENTATION:

* Iterative

#include<iostream>

using namespace std;

int binarysearch(int \*,int,int);

int count=0;

int main()

{

int n,x,\*a,p;

cout<<"Enter number of elements\n";

count++;

cin>>n;

count++;

a = new int[n];

count++;

cout<<"Enter "<<n<<" elements in ascending order\n";

count++;

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

{

count++; //for

cin>>a[i];

count++;

}

count++;//for

cout<<"Enter element to be searched\n";

count++;

cin>>x;

count++;

p=binarysearch(a,n,x);

count++; //for assignment

count++; //if

if(p==-1)

cout<<"Element does not exist\n";

else

cout<<"Element is present at position "<<p<<endl;

cout<<"Count="<<count<<endl;

return 0;

}

int binarysearch(int \*a,int n,int x)

{

int low=0,high=n-1;

count+=2; //2 assignments

while(low<=high)

{

count++; //while

int mid = (low+high)/2;

count++; //mid

count++; //if

if(x==a[mid])

{ count++;

return mid+1;

}

else if(x<a[mid])

{

high=mid-1;

count++; //assign

}

else

{

count++; //else if

low=mid+1;

count++; //for assign

}

}

count++; //while

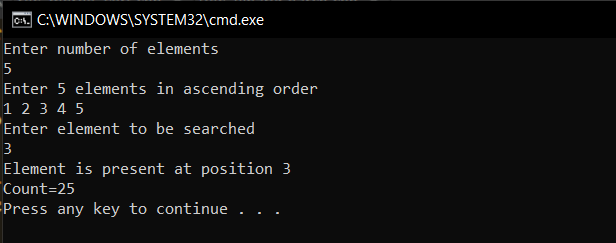
count++; //return

return -1;

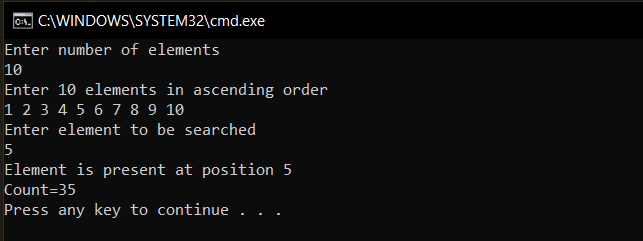
}

OUTPUTS:

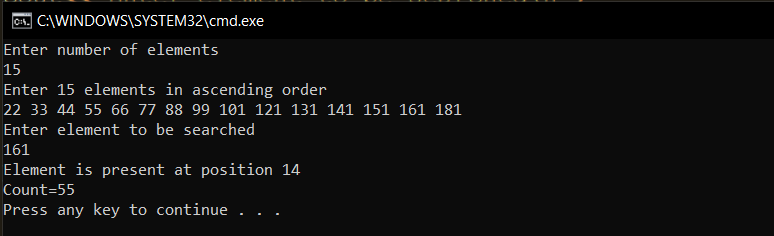
1. When n=5 and x==a[mid]



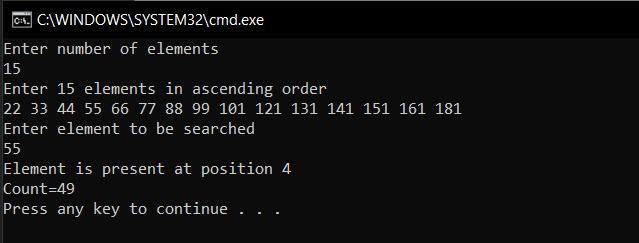
1. When n=10 and x==a[mid]



1. When n=15 and x>a[mid]



1. When n=15 and x<a[mid]



* Recursive

#include<iostream>

using namespace std;

int binarysearch(int \*,int,int,int);

int count=0;

int main()

{

int n,x,\*a,p;

cout<<"Enter number of elements\n";

count++;

cin>>n;

count++;

a = new int[n];

count++;

cout<<"Enter "<<n<<" elements in ascending order\n";

count++;

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

{

count++;//for

cin>>a[i];

count++;

}

count++;//for

cout<<"Enter element to be searched\n";

count++;

cin>>x;

count++;

p=binarysearch(a,0,n-1,x);

count++; //for assignment

count++; //if

if(p==-1)

cout<<"Element does not exist\n";

else

cout<<"Element is present at position "<<p<<endl;

cout<<"Count="<<count<<endl;

return 0;

}

int binarysearch(int \*a,int low, int high,int x)

{

count++; //if

if(low>high)

{

count++;

return -1;

}

int mid = (low+high)/2;

count++;

count++;//if

if(x==a[mid])

{ count++;

return mid+1;

}

else if(x<a[mid])

{

count++;//return

return binarysearch(a,low,mid-1,x);

}

else

{

count++; //for else if condition

count++;//return

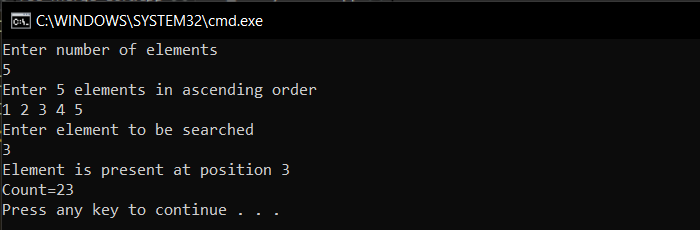
return binarysearch(a,mid+1,high,x);

}

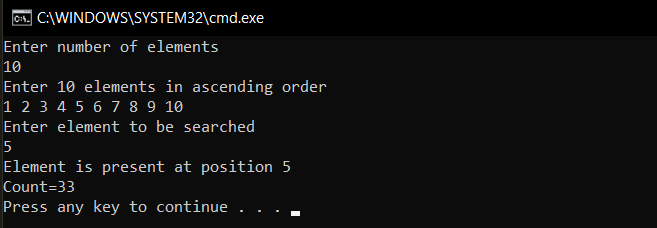
}

Output:

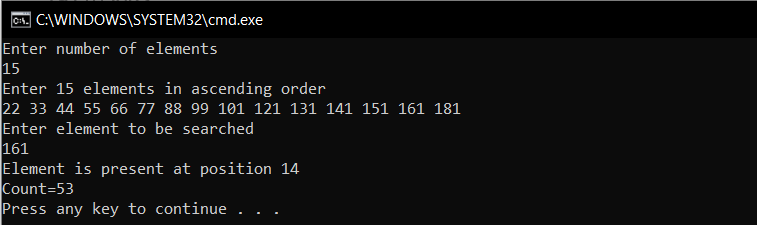
1. When n=5 and x==a[mid]



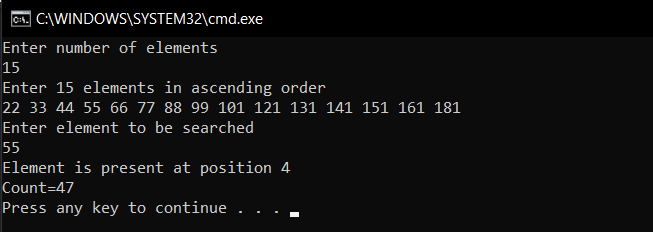
1. When n=10 and x==a[mid]



1. When n=15 and x>a[mid]



1. When n=15 and x<a[mid]



Conclusion: All programs were successfully run and executed with main emphasis on the complexities of Binary Search Algorithm. The following was also observed.

1. The iterative version of the Binary Search Algorithm has a space complexity of O(1), whereas its recursive counterpart has a space complexity of O(logn).
2. There is a trade off between space and time as the recursive version performs more efficiently than the iterative version in terms of number of steps taken, although it requires more space.