UNIT-II: Searching Techniques

2.1 Searching
2.2 Linear search (Sequential search)
2.3 Ordered Linear Search
2.4 Time complexity
2.5 Binary search
2.6 Time complexity

Searching: Searching is a process of checking whether an element is present Or not in a given list.

Linear Search:

Linear search is a method for searching a particular value in a list by comparing it with one element at a time until a search is found to be successful or unsuccessful.

Consider an array a[ ] of n elements

The element we want to search is 'key'

Algorithm:

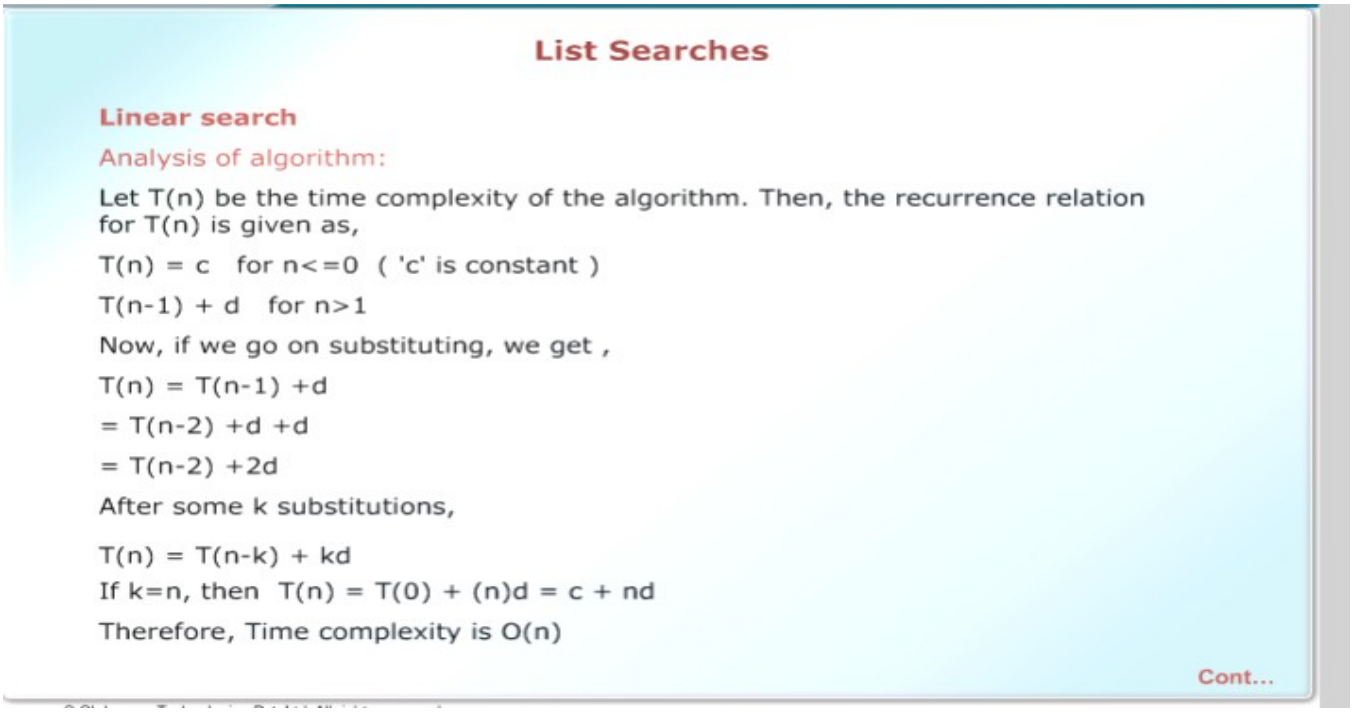
```
Algorithm Lsearch(a,n,key)
{
        for i=0 to n
           {
            if(a[i]==key)
            return i;
           }
        return -1;
}
```

Time complexity:

BEST CASE: This happens when the element to be searched is at First position, then the complexity is O (1).

WORST CASE: This happens when the element to be searched is at last position, then the complexity is O (n).

AVERAGE CASE: To find this we need to consider all possible positions of the key including no match and computer the average of the number of searches needed for each possibility. It is O (n)



This method is not effective for a large sized array.

Program:

```
#include<iostream.h>
#include<conio.h>
#include<malloc.h>
class Data
{
 private:
    int i,n,*ary;
 public:
```

```
    void read_data();
    int Lsearch_data(int);
};


 void Data :: read_data()
 {
  cout<<"\n Enter the array size : ";
  cin>>n;
  ary = (int *)malloc(n * sizeof(int));
  cout<<"\n Enter " << n << " Elements : ";
  for(i=0;i<n;i++)
  cin>>ary[i];
 }

 int Data :: Lsearch_data(int k)
 {
   for(i=0;i<n;i++)
   if(k==ary[i])
    return i;
    return -1;
 }

void main()
{
 Data obj;
 int key,p;
 clrscr();
 obj.read_data();
 cout<<"\n Enter the element to search  : ";
 cin>>key;
 p = obj.Lsearch_data(key);

  if(p==-1)
   cout<<"\n Element not found " ;
  else
   cout<<"\n Element found at position " << p ;
}
```

Execution: Enter array size 6

Enter 6 elements 15 23 34 12 45 27

Enter the element to search 12

Element found at the position 4

Ordered Linear Search

Here the input array is considered to be a sorted in ascending order, hence searching becomes simple.

If the element to be searched is greater than the last OR less than the first element simply we can skip the procedure

```
#include<iostream.h>
#include<conio.h>
#include<malloc.h>
class Data
{
 private:
    int i,n,*ary;
 public:
    void read_data();
    int OLsearch_data(int);
};
 void Data :: read_data()
 {
  cout<<"\n Enter the array size : ";
  cin>>n;
  ary = (int *)malloc(n * sizeof(int));
  cout<<"\n Enter " << n << " Elements in Ascending order : ";
  for(i=0;i<n;i++)
  cin>>ary[i];
 }
 int Data :: OLsearch_data(int k)
 {
if((k<ary[i])||(k>ary[n-1]))
   return -1;
```

```
   for(i=0;i<n;i++)
{
    if(ary[i]==k)
     return i;
}
return -1;
}
void main()
{
 Data obj;
 int key,p;
 clrscr();
 obj.read_data();
 cout<<"\n Enter the element to search  : ";
 cin>>key;
 p = obj.OLsearch_data(key);
  if(p==-1)
   cout<<"\n Element not found " ;

else
   cout<<"\n Element found at position " << p ;
  getch();
}
```

Sequential Search

Array contains the key element to be searched, in its n^{th} position.
Searching starts by checking from first position to $n-1^{th}$ position, and if the element is not found it is identified to be unsuccessful search.

```
#include<iostream.h>
#include<conio.h>
#include<malloc.h>
class Data
{
 private:
    int i,n,*ary;
 public:
    void read_data();
```

```
    int Ssearch_data(int);
};
 void Data :: read_data()
 {
  cout<<"\n Enter the array size : ";
  cin>>n;
  ary = (int *)malloc(n+1 * sizeof(int));
  cout<<"\n Enter " << n << " Elements : ";
  for(i=0;i<n;i++)
  cin>>ary[i];
 }
 int Data :: Ssearch_data(int k)
 {
   ary[n] = k,i=0;
   while(ary[i]!=k)
    i++;
   if(i == n)
    return -1;
   else
    return i;
 }
void main()
{
 Data obj;
 int key,p;
 clrscr();
 obj.read_data();
 cout<<"\n Enter the element to search  : ";
 cin>>key;
 p = obj.Ssearch_data(key);
  if(p==-1)
   cout<<"\n Element not found " ;
  else
   cout<<"\n Element found at position " << p ;
 getch();
}
```

BINARY SEARCH

Binary search works on sorted arrays
It works as follows:

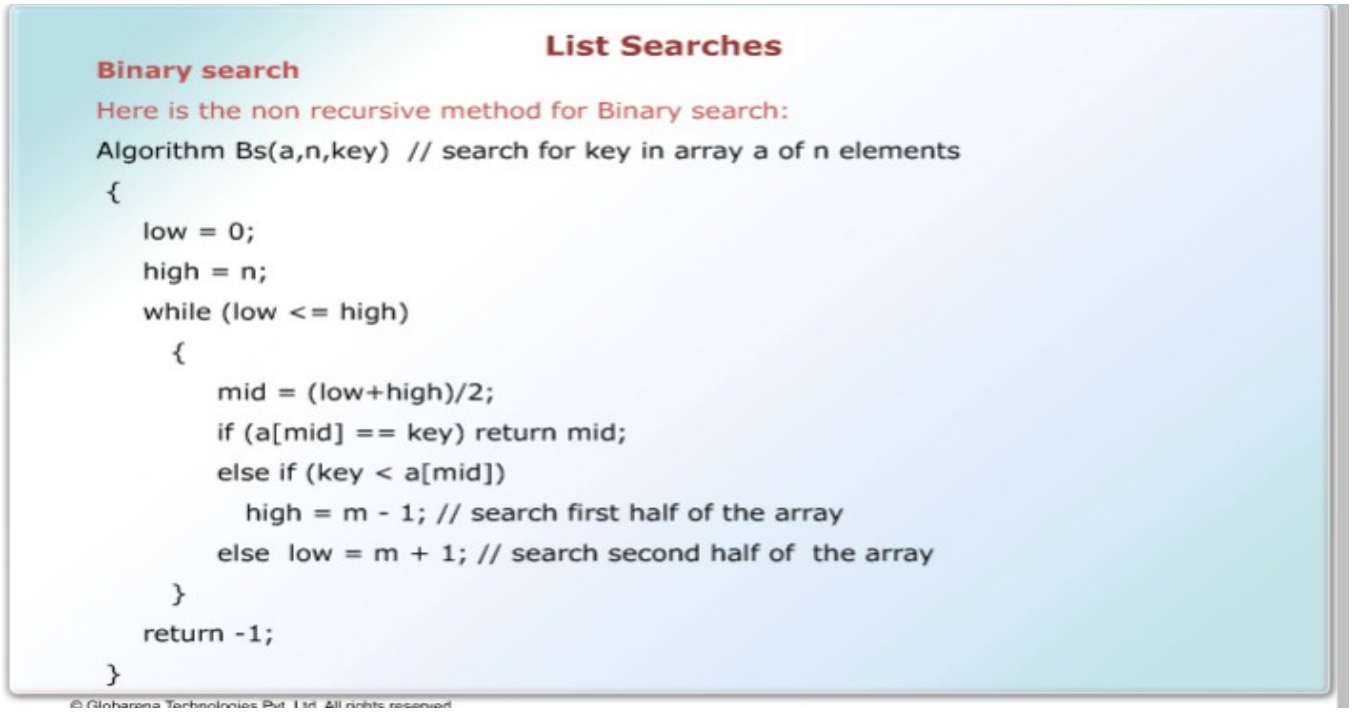
The middle element is located and compared with the element we are searching for

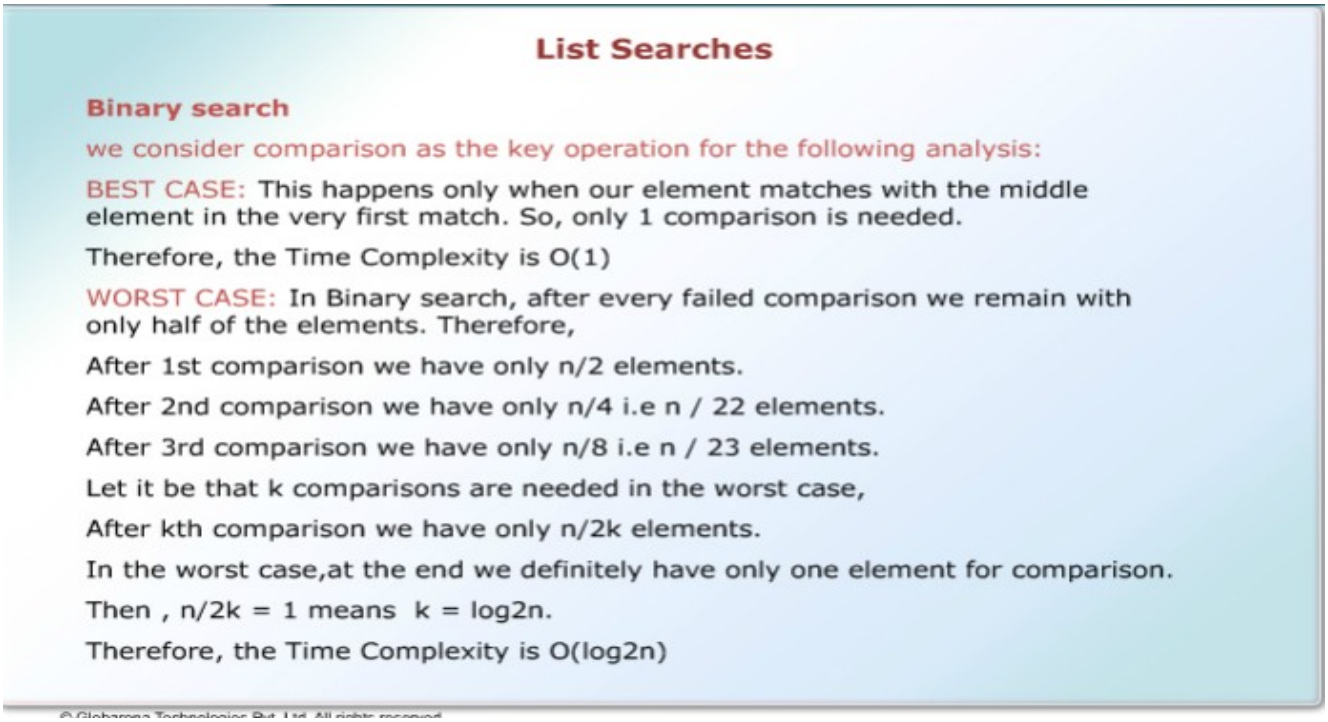
1. If they are equal, the key is found at corresponding index

2. If not the problem is divided in to two halves depending on the following condition

If the key is less than the middle element of the array, then the first half of the array is searched

If the key is greater than the middle element of the array, then the second half of the array is searched





Time complexity of Binary search:

$$T(n)=2T(n/2)+n$$
$$=2\{2T(n/2)+n/2\}+n,=>4T(n/2)+2n$$
$$=4\{2T(n/2)+n/2\}+2n=>8T(n/2)+3n$$
$$\ldots\ldots$$
$$\ldots\ldots$$
$$=2^iT(n/2^i)+in$$

Substitute $i=\log_2 n$(as i is maximum number of divisions possible)

$$=2^{\log_2 n}T(n/2^{\log_2 n})+{}^{\log_2 n}.n$$

$$=nT(n/n)+n\log_2 n$$

$$=nT(1)+n\log_2 n$$

=0+ $nlog_2n$

= $nlog_2n$

T(n)= log_2n n=1 i.e to search one element)

```
#include<iostream.h>
#include<conio.h>
#include<malloc.h>
class Data
{
 private:
     int i,n,*ary;
 public:
     void read_data();
     int Bsearch_data(int);
};
 void Data :: read_data()
 {
  cout<<"\n Enter the array size : ";
  cin>>n;
  ary = (int *)malloc(n * sizeof(int));
  cout<<"\n Enter " << n << " Elements IN INCREASING ORDER  : ";
  for(i=0;i<n;i++)
  cin>>ary[i];
 }
 int Data :: Bsearch_data(int k)
 {
  int p=-1,l,m,h;
   l=0;h=n-1;
    do
     {
      m = (l+h)/2;
      if(k==ary[m])
```

```
      {
        p = m;
        break;
      }
     else if(k<ary[m])
      h = m-1;
     else
      l = m+1;
    }while(l<=h);
   return p;
 }
void main()
{
 Data obj;
 int key,p;
 clrscr();
 obj.read_data();
 cout<<"\n Enter the element to search  : ";
 cin>>key;
 p = obj.Bsearch_data(key);
  if(p==-1)
   cout<<"\n Element not found " ;
  else
   cout<<"\n Element found at position " << p ;
 getch();
}
```

EXAMPLE:

The list of element are 3,14,39,42,55,70,74,98 and searching for k=70 in the list

Low=0 high=7

Mid= (low+high)/2 i.e. 3

As key>a[3] the problem is reduced to a[mid+1] to a[high]

Low=4 high=7

Mid= (low+high)/2 i.e. 5

As key==a[Mid] returns mid position as output