

AIM: Implementation of searching algorithms: Linear Search and Binary search

1) Linear Search:

THEORY:

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order. We start at one end and check every element until the desired element is not found. Here is simple approach is to do Linear Search:

- Start from the leftmost element of array and one by one compare the element we are searching for with each element of the array.
- If there is a match between the element we are searching for and an element of the array, return the index.
- If there is no match between the element we are searching for and an element of the array, return -1.

Linear Search Complexities:

Time Complexity: $O(n)$

Space Complexity: $O(1)$

Linear Search Applications:

For searching operations in smaller arrays (<100 items).

Algorithm:

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if $i > n$ then go to step 7

Step 3: if $A[i] = x$ then go to step 6

Step 4: Set i to $i + 1$

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Pseudocode:

```
procedure linear_search (list, value)

    for each item in the list
        if match item == value
            return the item's location
        end if
    end for

end procedure
```

Example:

The following steps are followed to search for an element $k = 1$ in the list below.



Array to be searched for

1. Start from the first element, compare k with each element x .

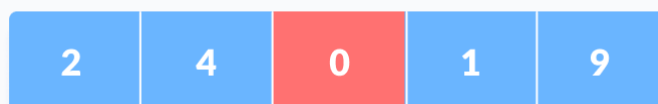
$k = 1$



↑
 $k \neq 2$



↑
 $k \neq 4$



↑
 $k \neq 0$

Compare with each element

2. If $x == k$, return the index.



↑
 $k = 1$

Element found

3. Else, return not found.


SOURCE CODE:

```
#include <iostream>
using namespace std;

int linear_search( int arr[], int n, int s )
{
    for(int i=0; i < n; i++)
    {
        if(arr[i] == s)
        {
            return i;
        }
    }
    return -1;
}

int main()
{
    int n;
    cout << "Enter number of elements for an array \n";
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter element "<<i+1<<"\n";
        cin >> arr[i];
    }
    int s;
    cout<< "Enter element to search \n";
    cin>>s;
    int rs = linear_search(arr, n, s);
    if(rs==-1)
    {
        cout<<"Element is not found!!";
    }
    else
    {
        cout<<"Element is found at position "<<rs+1;
    }
    return 0;
}
```

OUTPUT:

 "C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\linearsearch.exe"

```
Enter number of elements for an array
6
Enter element 1
5
Enter element 2
-9
Enter element 3
4
Enter element 4
7
Enter element 5
1
Enter element 6
2
Enter element to search
1
Element is found at position 5
Process returned 0 (0x0)   execution time : 22.124 s
Press any key to continue.
```

2) Binary Search:

THEORY:

Binary Search is a searching algorithm for finding an element's position in a sorted array. The element is always searched in the middle of a portion of an array. Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first. Binary Search Algorithm can be implemented in two ways:

1. Iterative Method
2. Recursive Method

Time Complexities:

- Best case complexity: $O(1)$
- Average case complexity: $O(\log n)$
- Worst case complexity: $O(\log n)$

Space Complexity: $O(1)$.

Binary Search Applications:

- In libraries of Java, .Net, C++ STL, while debugging, the binary search is used to pinpoint the place where the error happens.

Binary Search Algorithm:

1) Iteration Method:

do until the pointers low and high meet each other.

```
mid = (low + high)/2
```

```
if (x == arr[mid])
```

```
    return mid
```

```
else if (x > arr[mid]) // x is on the right side
```

```
    low = mid + 1
```

```
else // x is on the left side
```

```
    high = mid - 1
```

2) Recursive Method:

```
binarySearch(arr, x, low, high)
```

```
if low > high
```

```
    return False
```

```
else
```

```
    mid = (low + high) / 2
```

```
    if x == arr[mid]
```

```
        return mid
```

```
    else if x > arr[mid] // x is on the right side
```

```
        return binarySearch(arr, x, mid + 1, high)
```

```
    else // x is on the left side
```

```
        return binarySearch(arr, x, low, mid - 1)
```

Example:

The array in which searching is to be performed is:



Initial array

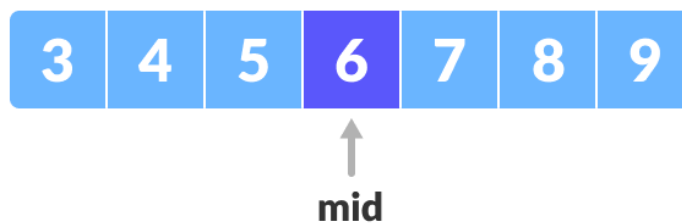
Let $x = 4$ be the element to be searched.

Set two pointers low and high at the lowest and the highest positions respectively.



Setting pointers

Find the middle element `mid` of the array ie. $\text{arr}[(\text{low} + \text{high})/2] = 6$.



Mid element

If $x == \text{mid}$, then return mid. Else, compare the element to be searched with m.

If $x > \text{mid}$, compare x with the middle element of the elements on the right side of `mid`. This is done by setting `low` to `low = mid + 1`.

Else, compare x with the middle element of the elements on the left side of `mid`. This is done by setting `high` to `high = mid - 1`.



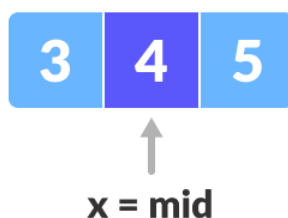
Finding mid element

Repeat steps 3 to 6 until low meets high.



Mid element

$x = 4$ is found.



Founded

SOURCE CODE:

```
#include <iostream>
using namespace std;

void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int binary_search( int arr[], int l, int r, int s )
{
    if (r>=1)
    {
        int mid = (l + r)/2;

        if(arr[mid] == s)
        {
            return mid;
        }
        else if(arr[mid] < s)
        {
            return binary_search(arr, mid + 1, r, s);
        }
        else
        {
            return binary_search(arr, l, mid - 1, s);
        }
    }
    else
    {
        return -1;
    }
}

int * bubble_sort( int A[ ], int n )
{
    int temp;
    for(int k = 0; k< n-1; k++)
    {
        // (n-k-1) is for ignoring comparisons of elements which have already been compared in
        // earlier iterations

        for(int i = 0; i < n-k-1; i++)
        {
            if(A[ i ] > A[ i+1 ] )
            {
                // here swapping of positions is being done.
            }
        }
    }
}
```


```
        temp = A[ i ];
        A[ i ] = A[ i+1 ];
        A[ i + 1] = temp;
    }

}

return A;
}

int main()
{
    int n;
    cout << "Enter number of elements for an array \n";
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter element "<<i+1<<"\n";
        cin >> arr[i];
    }
    int s;
    cout<< "Enter element to search \n";
    cin>>s;
    int *ar = bubble_sort(arr,n);
    cout<< "Sorted array is \n";
    printArray(ar,n);
    int rs = binary_search(ar, 0, n-1, s);
    if(rs==-1)
    {
        cout<<"Element is not found!!";
    }
    else
    {
        cout<<"Element is found at position "<<rs+1;
    }
    return 0;
}
```


OUTPUT:

 "C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\binarysearch.exe"

```
Enter number of elements for an array
5
Enter element 1
6
Enter element 2
4
Enter element 3
3
Enter element 4
9
Enter element 5
1
Enter element to search
6
Sorted array is
1 3 4 6 9
Element is found at position 4
Process returned 0 (0x0)    execution time : 15.344 s
Press any key to continue.
```

CONCLUSION:

I have learned the searching algorithms such as linear and binary. It is observed that binary search is more optimized than linear search.