Linear search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Sample:

```
int  linearSearch(int ara[],int x)
{
        for(int i=0; i<n; i++)
                if(x==ara[i])
                        return i;
        return -1;
}
```

Analysis:

Suppose have a Array of 5 element,

Ara={10,15,35,5,25}

Let x=20, we want know the existence of x in the array so the procedure is given below:

<u>Step 1:</u>

Compare x with 1st element of the array

10	15	35	5	25

ara[0], x!=ara[0] ,go to the next step

<u>Step 2:</u>

10	**15**	35	5	25

Compare x with 2nd element of the array

ara[1], x!=ara[1] , go to the next step

<u>Step 3:</u>

10	15	**35**	5	25

Compare x with 3rd element of the array

ara[1], x!=ara[2] , go to the next step

<u>Step 4:</u>

10	15	35	**5**	25

Compare x with 4th element of the array

ara[1], x!=ara[3] , go to the next step

Step 5:

10	15	35	5	25

Compare x with 5th element of the array

X==ara[4] ,so function will return index of ara[4]

Time complexity

Best case:

We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be: Θ(1)

Worst case:

For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be : Θ(n).

Average case:

For average case Sum all the calculated values and divide the sum by total number of inputs.

$$= \frac{1+2+3\ldots\ldots+n}{n}$$

$$= \frac{\frac{n(n+1)}{2}}{n}$$

$$= \frac{n+1}{2}$$

Ignoring the constant co-efficient, we can say that the complexity in average case of linear search is : O(n).

Binary search

Binary search is another search technique. It is efficient than linear search while time complexity of linear search is O(n) but time complexity of Binary search is O(logn).

Let us say, we want to look for a particular number in an array of few integers, sorted in increasing order. Now, just take the middle number and check if the number you are looking for, is greater or lesser than this number. If it is greater, then you can eliminate all elements to the 'left' of this number (since all of them are guaranteed to be smaller and your number will not be present in this set). Else, eliminate the right. After this step, you have just one half of the initial set remaining to search in. Continue the process with this new reduced set. Keep doing this, until you find the number you are looking for.

Imagine we have a (sorted) array such as this:

{1, 2, 3, 5, 7, 9, 11, 15}. There are 8 elements in the array.

By using log base 2 of 8, we know we need to divide the array three times (Log base(2) of 8 = 3) before we have a list of individual numbers. If there are 16 elements in the array you will need to divide 4 times and so on, and that is why the binary search has a worst-case time complexity of log(n)

Example:

```
#include <bits/stdc++.h>
using namespace std;

int BS(int arr[], int left, int right, int x)
{
    while (left <= right) {
        int mid = left + (right - l) / 2;
        if (arr[mid] == x)
            return mid;

        if (arr[mid] < x)  left = mid + 1;
        else  right = mid - 1;
    }
      return 0;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int res = BS(arr, 0, n - 1, x);

    if (res == 0) cout << "Element is not present in array"<<endl;
    else  cout << "Element is present at index " << result<<endl;

    return 0;
}
```

Bubble Sort

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

Sample:

```
void bubble(int ara[])
{
   for(int i=0; i<n; i++)
            for(int j=0; j<(n-i-1); j++)
              if(ara[j]>ara[j+1])
                 swap(ara[j],ara[j+1]);
}
```

Analysis:

Let’s consider an array: (5, 1, 4, 2, 8)

We want to sort it in ascending order:

So,

1st Iteration:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

2nd Iteration:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

3rd Iteration:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Time Complexity

In Bubble Sort, (n-1) comparisons will be done in the 1st pass, (n-2) in 2nd pass, (n-3) in 3rd pass and so on. So the total number of comparisons will be,

(n-1) + (n-2) + (n-3) + (n-4) + …… + 3 + 2 + 1

= (n-1)*n/2

Ignoring the constant co-efficient, we can say that the complexity is: O(n^2)

By using the following process we have to do same number operation for best case, worst case and average case.

So complexity is: O(n^2)

Optimized Bubble sort

Sample :

```
void bubbleOpt(int ara[])
{
        bool flg;
       for(int i=0; i<n; i++)
        {
         flg=true;
         for(int j=0; j<(n-i-1); j++)
                if(ara[j]>ara[j+1])
                        swap(ara[j],ara[j+1]),flg=false;
        if(!flg)
         break;

       }
}
```

It can be optimized by stopping the algorithm if inner loop didn't cause any swap.

<u>Worst and Average Case Time Complexity</u>:

8	5	4	2	1

O(n*n). Worst case occurs when array is reverse sorted.

<u>Best Case Time Complexity</u>:

1	**2**	**4**	**5**	**8**

O(n). Best case occurs when array is already sorted.

Insertion Sort

The first step involves the comparison of the element in question with its adjacent element.

1. And if at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position.
2. The above procedure is repeated until all the element in the array is at their apt position.

Let us now understand working with the following example:

Consider the following array: 25, 17, 31, 13, 2

1st Iteration:
(**25**, 17, 31, 13, 2) -> (**25**, 17, 31, 13, 2)

2nd Iteration:
(25, **17**, 31, 13, 2) -> (17, 25, 31, 13, 2)

3rd Iteration:
(17, 25, **31**, 13, 2) -> (17, 25, 31, 13, 2)

4th Iteration:
(17, 25, 31, **13**, 2) -> (17, 25, **13**, 31, 2)
(17, 25, **13**, 31, 2) -> (17, **13**, 25, 31, 2)
(17, **13**, 25, 31, 2) -> (**13**, 17, 25, 31, 2)

5th Iteration:
(13, 17, 25, 31, **2**) -> (13, 17, 25, **2**, 31)
(13, 17, 25, **2**, 31) -> (13, 17, **2**, 25, 31)
(13, 17, **2**, 25, 31) -> (13, **2**, 17, 25, 31)
(13, **2**, 17, 25, 31) -> (**2**, 13, 17, 25, 31)

Time complexity for insertion sort is O(n2)

Example:

```
#include<bits/stdc++.h>

using namespace std;

void insertion(int ara[],int n)

{

   int i,j,x;

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

   {

      x=ara[i];

      j=i-1;

      while(j>=0 && ara[j]>x)

         ara[j+1]=ara[j],j--;

      ara[j+1]=x;

   }

}

int main()

{

   int ara[]={5,1,4,2,8};

   int n=sizeof(ara)/sizeof(ara[0]);

   cout<<"Element before sorting: ";

   for(int i=0;i<n;i++) cout<<ara[i]<<' ';

   insertion(ara,n);

   cout<<endl<<"Element before sorting: ";

   for(int i=0;i<n;i++) cout<<ara[i]<<' ';

   return 0;

}
```