Linear Search Algorithm

Linear search algorithm is a simple algorithm to find out if an element is present in an array. It sequentially searches the array and if the element is found then it returns true. Otherwise it returns false.

Main Idea

The steps used in the implementation of Linear Search are listed as follows -

- First, we have to traverse the array elements using a **for** loop.
- In each iteration of **for loop,** compare the search element with the current array element, and -
  - If the element matches, then return **true**.
  - If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return **false.**

Pseudocode

```
Algorithm Linear Search(a,n,x)
//a is an array of size n
//x is the element we are searching for
{
    for i:=1 to n do
    {
        If (a[i]==x) then
            return true;
    }
    return false;
}
```

Space Complexity: There are 4 variables: a, n, x, i. For a, we require n words and for the rest of the variables we need 1 word each. So, the Space complexity of linear search,

$S_{linear-search}(n) = n + 3$

That is: O(n) as 2n>=n+3 for all n>=3.

Time complexity:

Algorithm	s/e	Frequency			Total Steps		
		Best Case	Average Case	Worst Case	Best Case	Average Case	Worst Case
Algorithm Linear Search(a,n,x)	0	-	-	-	-	-	-
//a is an array of size n	0	-	-	-	-	-	-
//x is the element we are searching for	0	-	-	-	-	-	-
{	0	-	-	-	-	-	-
for i:=1 to n do	1	1	X	n	1	X	n
{	0	-	-	-	-	-	-
If (a[i]==x) then	1	1	X	n	1	X	n
return true;	1	1	1	1	1	1	1
}	0	-	-	-	-	-	-
return false;	1	-	-	-	-	-	-
}	0	-	-	-	-	-	-
Time Complexity, $T_{Linear-Search}$(n)=					3	2X+1	2n+1

Here, the Best case is when the searching element is found in the 1st index of the array. Worst case is if the searching element is the last element of the array. Average case lies in between.

In case of average case the highest value for X is n-1.

So, 2X+1=2 (n-1) + 1

=2n-2+1=2n-1

Big Oh notation for best case: O(1) [as the time complexity is constant]

Big Oh notation for Average case: O(n) as 3n>=2n-1 for all n>=1

Big Oh notation for Worst case: O(n) as 3n>=2n+1 for all n>=1

For detailed understanding you can follow this link: https://www.javatpoint.com/linear-search

Binary Search Algorithm

Binary search algorithm is used to search an element in an array if the array is sorted. For sorted array binary search performs better than linear search. Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Main Idea

1. Compare x with the middle element.

2. If x matches with the middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
4. Else (x is smaller) recur for the left half.

Pseudocode

```
Algorithm BinarySearch(a,low, high, x)
//a is an array of size n
//x is the element we are searching for
//low is the lowest position and high is the highest position of the array
{
   while (low<=high) do
   {
      mid = (low+ high) / 2;
      if (a[mid] == x) then
         return true;
      if (a[mid] > x) then
         high:=mid -1;
          else
             low:=mid+1;
   }
   return false;
}
```

Space Complexity: There are 5 variables: a, low, high, x, mid. If the size of the array is n (i.e., high is equal to n) then for a, we require n words and for the rest of the variables we need 1 word each. So, the Space complexity of binary search, $S_{binary-search}(n) = n + 4$

That is: O(n) as 2n>=n+4 for all n>=4.

Time complexity:

Here, in each iteration of the while loop. The size of the array (searching space) is getting half in size and it terminates when the size of the array (searching space) becomes 1. Here, the actual size of array always remains n but the size of the array where we are searching, that means the searching space, is decreasing at each iteration. So,

In 1st iteration the size of the array is: n

In 2nd iteration the size of the array is: n/2

In 3rd iteration the size of the array is: $n/4=n/2^2$

So, in kth iteration the size of the array is: $n/2^{k-1}$

In the worst case, $n/2^{k-1}=1$

- ⇨ $n=2^{k-1}$
- ⇨ log(n)=k-1

⇨ k=log(n)+1

So, the time complexity in worst case is log(n)+1 for binary search algorithm. That is, O(log(n)) as 2log(n)>=log(n)+1 for all n>=1.

For detailed understanding you can follow this link:
https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm
To understand the time complexity of Binary Search Algorithm:
https://www.geeksforgeeks.org/complexity-analysis-of-binary-search/

Bubble Sort Algorithm

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.

Main Idea

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

If we have total n elements, then we need to repeat this process for n-1 times.

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

Pseudocode

```
Algorithm BubbleSort(a,n)
//a is an array of size n
{
  for i:= 0 to n-2 do
      {
            for j:= 1 to n-i-1 do
            {
                  if (a[j] > a[j+1]) then
                  {
                        t:=a[j];
                        a[j]:=a[j+1];
                        a[j+1]:=t;
                  }
            }
```

```
        }
}
```

Space Complexity: There are 5 variables: a, n, i, j, t. For a, we require n words and for the rest of the variables we need 1 word each. So, the Space complexity of Bubble sort algorithm, $S_{Bubble-Sort}(n) = n + 4$

That is: O(n) as 2n>=n+4 for all n>=4.

Time complexity:

Algorithm	s/e	Frequency	Total Steps
Algorithm BubbleSort(a,n)	0	-	-
//a is an array of size n	0	-	-
{	0	-	-
for i:= 0 to n-2 do	1	n	n
{	0	-	-
for j:= 1 to n-i-1 do	1	X+1	X+1
{	0	-	-
if (a[j] > a[j+1]) then	1	X	X
{	0	-	-
t:=a[j];	1	X	X
a[j]:=a[j+1];	1	X	X
a[j+1]:=t;	1	X	X
}	0	-	-
}	0	-	-
}	0	-	-
}	0	-	-
Time Complexity, $T_{Bubble-Sort}$(n)=			n+5X+1

As the time complexity is a function of n, we have to define x in terms of n.

Here, X is a series of (n-1)+(n-2)+….+1

So, X={(n-1)(n-1+1)}/2=$(n^2 - n)/2$

So, time Complexity, $T_{Bubble-Sort}$(n)= n+[{5($(n^2 - n)$}/2]+1=$(5n^2 - 3n + 2)/2$

Big Oh notation for Time Complexity here: O(n^2) as $3n^2$>=$(5n^2 - 3n + 2)/2$ for all n>=1.

Here, time complexity will remain same for best, average and worst cases.

For detailed understanding you can follow this link: https://www.javatpoint.com/bubble-sort

Selection Sort Algorithm

Selection 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. Selection Sort selects elements for the appropriate positions and thus eventually the array gets sorted.

Main Idea

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

Pseudocode

```
Algorithm SelectionSort(a,n)
//a is an array of size n
{
   for i:= 1 to n-1 do
         {
                    K:=i;
                    for j:= i+1 to n do
                    {
                    if (a[j] < a[k]) then
                    k:=j;
                    }
                    t:=a[k];
                    a[k]:=a[i];
                    a[i]:=t;

         }
}
```

Space Complexity: There are 6 variables: a, n, i, j, t, k. For a, we require n words and for the rest of the variables we need 1 word each. So, the Space complexity of Bubble sort algorithm, $S_{Selection-Sort}(n) = n + 5$

That is: O(n) as 2n>=n+5 for all n>=5.

Time complexity:

Algorithm	s/e	Frequency	Total Steps
Algorithm SelectionSort(a,n)	0	-	-
//a is an array of size n	0	-	-
{	0	-	-
for i:= 1 to n-1 do	1	n	n
{	0	-	-
K:=i;	1	n-1	n-1
for j:= i+1 to n do	1	X+1	X+1
{	0	-	-
if (a[j] < a[k]) then	1	X	X
k:=j;	1	X	X
}	0	-	-
t:=a[k];	1	n-1	n-1

a[k]:=a[i];	1	n-1	n-1
a[i]:=t;	1	n-1	n-1
}	0	-	-
}	0	-	-
Time Complexity, $T_{Selection-Sort}$(n)=			5n+3X-3

As the time complexity is a function of n, we have to define x in terms of n.

Here, X is a series of (n-1)+(n-2)+(n-3)+….+1

So, X={(n-1)(n-1+1)}/2=$(n^2 - n)/2$

So, time Complexity, $T_{Bubble-Sort}$(n)= 5n+{$3(n^2 - n)/2$}-3=$(3n^2 + 7n - 6)/2$

Big Oh notation for Time Complexity here: O(n^2) as $2n^2$>=$(3n^2 + 7n - 6)/2$ for all n>=1.

Here, time complexity will remain same for best, average and worst cases.

For detailed understanding you can follow this link: https://www.javatpoint.com/selection-sort

Insertion Sort Algorithm

Insertion sort is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order.

Main Idea

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array.

Pseudocode

```
Algorithm InsertionSort(a,n)
//a is an array of size n
{
   for i:= 2 to n do
        {
                 k:=a[i];
                 j:=i-1;
                 while ( j>= 1 and a[j]>k) do
                 {
                          a[j+1]:=a[j];
                          j:=j-1;
```

```
                }
                A[j+1]:=k;

        }
}
```

Space Complexity: There are 5 variables: a, n, i, j, k. For a, we require n words and for the rest of the variables we need 1 word each. So, the Space complexity of Bubble sort algorithm, $S_{Insertion-Sort}(n) = n + 4$

That is: O(n) as 2n>=n+4 for all n>=4

Time complexity:

Algorithm	s/e	Frequency	Total Steps
Algorithm InsertionSort(a,n)	0	-	-
//a is an array of size n	0	-	-
{	0	-	-
for i:= 2 to n do	1	n	n
{	0	-	-
k:=a[i];	1	n-1	n-1
j:=i-1;	1	n-1	n-1
while (j>= 1 and a[j]>k) do	1	X+1	X+1
{	0	-	-
a[j+1]:=a[j];	1	X	X
j:=j-1;	1	X	X
}	0	-	-
A[j+1]:=k;	1	n-1	n-1
}	0	-	-
}	0	-	-
Time Complexity, $T_{Insertion-Sort}$(n)=			4n+3X-2

As the time complexity is a function of n, we have to define x in terms of n.

Here, in worst case, X is a series of 1+2+3+…..+n-1

So, X={(n-1)(n-1+1)}/2=$(n^2 - n)/2$

So, time Complexity, $T_{Bubble-Sort}$(n)= 4n+$\{3(n^2 - n)/2\}$-2=$(3n^2 + 5n - 4)/2$

Big Oh notation for Time Complexity here: O(n^2) as $2n^2$>=$(3n^2 + 5n - 4)/2$for all n>=1.

For detailed understanding you can follow this link:
https://www.interviewbit.com/tutorial/insertion-sort-algorithm/

Time Complexities for the above-mentioned algorithms

Algorithm	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(logn)	O(logn)
Bubble Sort	O(n^2)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)

What is stability of sorting algorithms?

A sorting algorithm is said to be stable if two objects with equal value appear in the same order in sorted output as they appear in the input array to be sorted. Bubble sort and insertion sort is stable but selection sort is not.

What is in-place and not-in-place sorting algorithms?

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting**. Bubble sort is an example of in-place sorting.

However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called **not-in-place sorting**. Merge-sort is an example of not-in-place sorting.

All the above mentioned sorting algorithms are in-place sorting algorithms.

How to decide when to use linear search and when to use binary search?

As binary search algorithm has better performance than linear search, so we would like to use binary search always. But binary search can only be used when the array is sorted. So, we will use linear search if the array is not sorted. Otherwise, will use binary search algorithm.

Which algorithm should I use among Bubble Sort, Insertion Sort and Selection sort?

We will prefer insertion sort; because it has order of n time complexity in best case. We will not choose selection sort or bubble sort; because it has time complexity of order n^2 in all cases. Space complexity is not a good measure for comparing these algorithms as they all are in-place algorithms. Bubble sort and selection sort can be used when the array is already almost sorted and the size of the array is short. With some little modification, bubble sort can also have order n time complexity in best case. However, Insertion sort is much superior among three.