Linear Search

- In computer science, a **linear search** or **sequential search** is a method for finding an element within a list.
- It sequentially checks each element of the list until a match is found or the whole list has been searched.
- A linear search runs in at worst linear time and makes at most *n* comparisons, where *n* is the length of the list.
- If each element is equally likely to be searched, then linear search has an average case of $n+1/\boxed{2}$ comparisons, but the average case can be affected if the search probabilities for each element vary.
- Linear search is rarely practical because other search algorithms and schemes, such as the binary search algorithm and hash tables, allow significantly faster searching for all but short lists.
- Linear search is used on a collections of items.
- It relies on the technique of traversing a list from start to end by exploring properties of all the elements that are found on the way.

For example, consider an array of integers of size

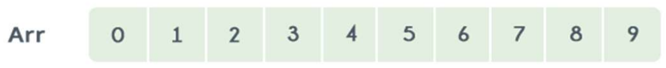
- You should find and print the position of all the elements with value.
- Here, the linear search is based on the idea of matching each element from the beginning of the list to the end of the list with the integer, and then printing the position of the element if the condition is `True'.

Implementation:

The pseudo code for this example is as follows:

```
for(start to end of array)
{
   if (current_element equals to 5)
   {
      print (current_index);
   }
}
```

For example, consider the following image:



- If you want to determine the positions of the occurrence of the number in this array.
- To determine the positions, every element in the array from start to end, i.e., from index to index will be compared with number, to check which element matches the number

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

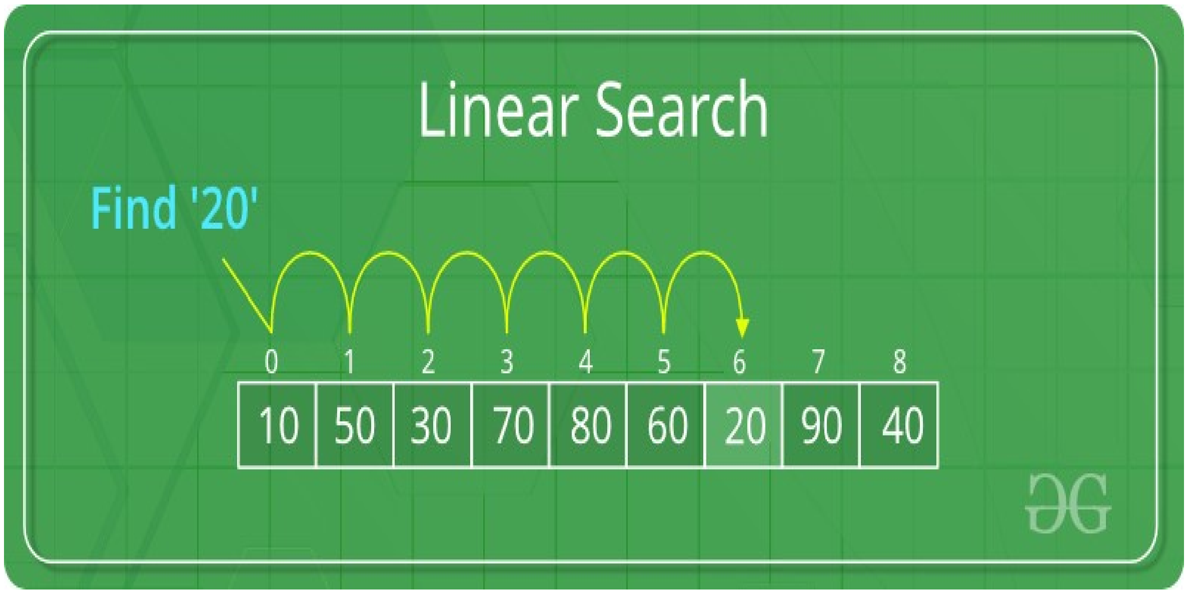
Examples :

Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 110;
Output : 6
Element x is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 175;
Output : -1
Element x is not present in arr[].

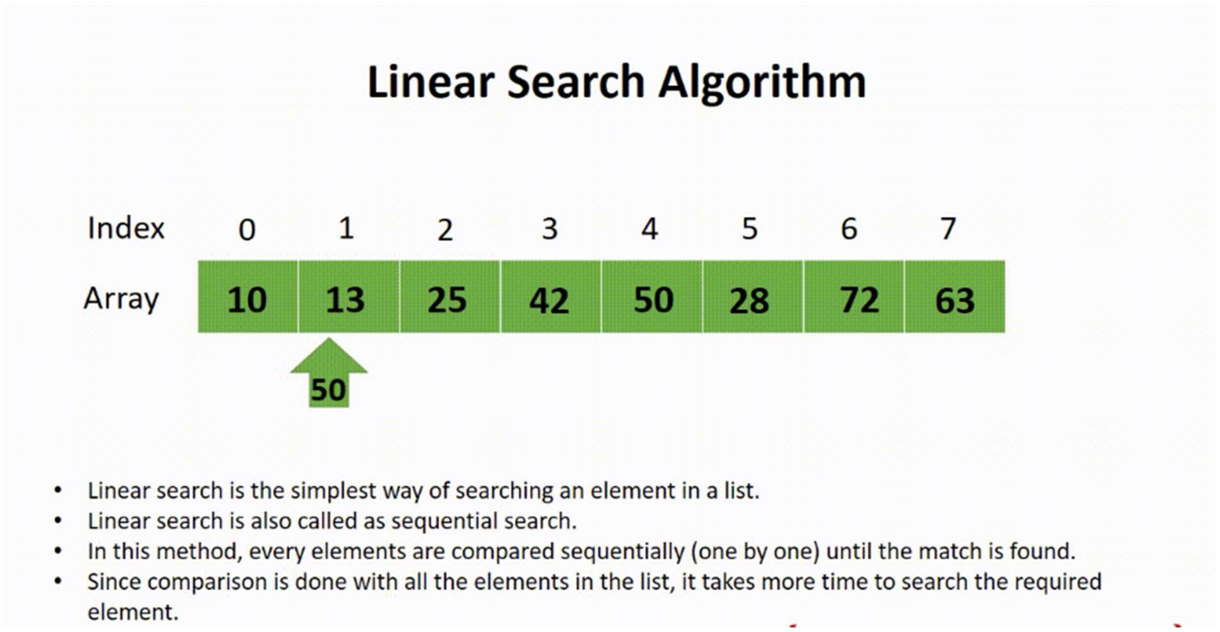
A simple approach is to do a **linear search**, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.



Example:

Linear Search Algorithm Visualisation:



- Linear search is the simplest way of searching an element in a list.
- Linear search is also called as sequential search.
- In this method, every elements are compared sequentially (one by one) until the match is found.
- Since comparison is done with all the elements in the list, it takes more time to search the required element.

```
// C++ code to linearly search x in arr[]. If x
// is present then return its location, otherwise
// return -1

#include <iostream>
using namespace std;

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

// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    // Function call
    int result = search(arr, n, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```
Output

```
Element is present at index 3
```

The **time complexity** of the above algorithm is O(n).

Improve Linear Search Worst-Case Complexity

1. if element Found at last O(n) to O(1)
2. if element Not found O(n) to O(n/2)

Below is the implementation:

```
// C++ program for linear search
#include<bits/stdc++.h>
using namespace std;

void search(vector<int> arr, int search_Element)
{
    int left = 0;
    int length = arr.size();
    int position = -1;
      int right = length - 1;

    // Run loop from 0 to right
    for(left = 0; left <= right;)
    {

        // If search_element is found with
        // left varaible
        if (arr[left] == search_Element)
        {

            position = left;
            cout << "Element found in Array at "
                 << position + 1 << " Position with "
            << left + 1 << " Attempt";

         break;
      }

      // If search_element is found with
      // right varaible
      if (arr[right] == search_Element)
      {
         position = right;
         cout << "Element found in Array at "
            << position + 1 << " Position with "
            << length - right << " Attempt";

         break;
      }
      left++;
      right--;
   }

   // If element not found
   if (position == -1)
      cout << "Not found in Array with "
          << left << " Attempt";
}

// Driver code
int main()
{
   vector<int> arr{ 1, 2, 3, 4, 5 };
   int search_element = 5;

   // Function call
   search(arr, search_element);
}

// This code is contributed by mayanktyagi1709
```
Output

```
Element found in Array at 5 Position with 1 Attempt
```

Complexity of Linear Search Algorithms

Class	Search algorithm
Worst-case performance	$O(n)$
Best-case performance	$O(1)$

Average performance	*O*(*n/2*)
Worst-case space complexity	*O*(1) iterative

Analysis

- For a list with *n* items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed.
- The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case *n* comparisons are needed.
- If the value being sought occurs *k* times in the list, and all orderings of the list are equally likely, the expected number of comparisons is

n	if	k=0
(n+1)/(k+1)	if	1≤k≤1

- For example, if the value being sought occurs once in the list, and all orderings of the list are equally likely, the expected number of comparisons is (n+1)/2.
- However, if it is *known* that it occurs once, then at most *n* - 1 comparisons are needed, and the expected number of comparisons is
  (n+2)(n-1)/2n

  (for example, for *n* = 2 this is 1, corresponding to a single if-then-else construct).

- Either way, asymptotically the worst-case cost and the expected cost of linear search are both O(*n*).

Non-uniform probabilities

- The performance of linear search improves if the desired value is more likely to be near the beginning of the list than to its end.
- Therefore, if some values are much more likely to be searched than others, it is desirable to place them at the beginning of the list.
- In particular, when the list items are arranged in order of decreasing probability, and these probabilities are geometrically distributed, the cost of linear search is only O(1).

Application

- Linear search is usually very simple to implement, and is practical when the list has only a few elements, or when performing a single search in an un-ordered list.
- When many values have to be searched in the same list, it often pays to pre-process the list in order to use a faster method.
- For example, one may sort the list and use binary search, or build an efficient search data structure from it.

- Should the content of the list change frequently, repeated re-organization may be more trouble than it is worth.
- As a result, even though in theory other search algorithms may be faster than linear search (for instance binary search), in practice even on medium-sized arrays (around 100 items or less) it might be infeasible to use anything else
- On larger arrays, it only makes sense to use other, faster search methods if the data is large enough, because the initial time to prepare (sort) the data is comparable to many linear searches.