Linear Search

In computer science, linear search or sequential search is a method for finding a particular value in a list, that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Linear search is the simplest search algorithm; it is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list; and so is its expected cost, if all list elements are equally likely to be searched for. Therefore, if the list has more than a few elements, other methods (such as binary search or hashing) will be faster, but they also impose additional requirements.

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

$$\begin{cases} n & \text{if } k = 0 \\ \dfrac{n+1}{k+1} & \text{if } 1 \leq k \leq n. \end{cases}$$

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

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

(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).

Pseudocode

Forward iteration:

This pseudocode describes a typical variant of linear search, where the result of the search is supposed to be either the location of the list item where the desired value was found; or an invalid location Λ, to indicate that the desired element does not occur in the list.

```
for each item in the list:
        if that item has the desired value,
        stop the search and return the item's location.
return Λ.
```

In this pseudocode, the last line is executed only after all list items have been examined with none matching.

If the list is stored as an array data structure, the location may be the index of the item found (usually between 1 and n, or 0 and n−1). In that case the invalid location Λ can be any index before the first element (such as 0 or −1, respectively) or after the last one (n+1 or n, respectively).

If the list is a simply linked list, then the item's location is its reference, and Λ is usually the null pointer. R

Recursive version

Linear search can also be described as a recursive algorithm:

```
LinearSearch(value, list)
        if the list is empty, return Λ;
        else
                if the first item of the list has the desired value,
                        return its location;
                else:
                        return LinearSearch(value, remainder of the list)
```

Binary Search Algorithm:

In computer science, a binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.
A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time. A binary search is a dichotomic divide and conquer search algorithm.

Overview
Searching a sorted collection is a common task. A dictionary is a sorted list of word definitions. Given a word, one can find its definition. A telephone book is a sorted list of people's names, addresses, and telephone numbers. Knowing someone's name allows one to quickly find their telephone number and address.
If the list to be searched contains more than a few items (a dozen, say) a binary search will require far fewer comparisons than a linear search, but it imposes the requirement that the list be sorted. Similarly, a hash search can be faster than a binary search but imposes still greater requirements. If the contents of the array are modified between searches, maintaining these requirements may even take more time than the searches. And if it is known that some items will be searched for much more often than others, and it can be arranged that these items are at the start of the list, then a linear search may be the best. More generally, algorithm allows searching over argument of any monotonic function for a point, at which function reaches the arbitrary value (enclosed between minimum and maximum at the given range).

Examples Example:
L = 1 3 4 6 8 9 11. X = 4.
Compare X to 6. It's smaller. Repeat with L = 1 3 4.
Compare X to 3. It's bigger. Repeat with L = 4.

Compare X to 4. It's equal. We're done, we found X.

Each iteration of (1)-(4) the length of the list we are looking in gets cut in half.
Therefore, the total number of iterations cannot be greater than logN.

Number guessing game
This rather simple game begins something like "I'm thinking of an integer between forty and sixty inclusive, and to your guesses I'll respond 'Higher', 'Lower', or 'Yes!' as might be the case." Supposing that N is the number of possible values (here, twenty-one as "inclusive" was stated), then at most $\lfloor \mathrm{Log}_2 N \rfloor$ questions are required to determine the number, since each question halves the search space. Note that one less question (iteration) is required than for the general algorithm, since the number is already constrained to be within a particular range.
Even if the number to guess can be arbitrarily large, in which case there is no upper bound N, the number can be found in at most $2\lfloor \mathrm{Log}_2 K \rfloor + 1$ steps (where k is the (unknown) selected number) by first finding an upper bound by repeated doubling.

For example, if the number were 11, the following sequence of guesses could be used to find it: 1 (Higher), 2 (Higher), 4 (Higher), 8 (Higher), 16 (Lower), 12 (Lower), 10 (Higher). Now we know that the number must be 11 because it is higher than 10 and lower than 12. One could also extend the method to include negative numbers; for example the following guesses could be used to find −13: 0, −1, −2, −4, −8, −16, −12, −14. Now we know that the number must be −13 because it is lower than −12 and higher than −14.

Recursive
A straightforward implementation of binary search is recursive. The initial call uses the indices of the entire array to be searched. The procedure then calculates an index midway between the two indices, determines which of the two subarrays to search, and then does a recursive call to search that subarray. Each of the calls is tail recursive, so a compiler need not make a new stack frame for each call. The variables *imin* and *imax* are the lowest and highest inclusive indices that are searched.

```
binary_search(int A[], int key, int imin, int imax)
{
        // test if array is empty
if (imax < imin)
        // set is empty, so return value showing not found
        return KEY_NOT_FOUND;
else
{
         // calculate midpoint to cut set in half
imid = midpoint(imin, imax);
        // three-way comparison
if (A[imid] > key)
        // key is in lower subset
        return binary_search(A, key, imin, imid-1);
else if (A[imid] < key)
         // key is in upper subset
        return binary_search(A, key, imid+1, imax);
else
        // key has been found
        return imid;
}
```

It is invoked with initial imin and imax values of 0 and N-1 for a zero based array of length N.
The number type "int" shown in the code has an influence on how the midpoint calculation can be implemented correctly. With unlimited numbers, the midpoint can be calculated as "(imin + imax) / 2". In practical programming, however, the calculation is often performed with numbers of a limited range, and then the intermediate result "(imin + imax)" might overflow. With limited numbers, the midpoint can be calculated correctly as "imin + ((imax - imin) / 2)".

Sorting

Sorting is the process of arranging or ordering a collection of items such that each item and its successor satisfy a prescribed relationship. The items can be simple values, such as integers and reals, or more complex types, such as student records or dictionary entries. In either case, the ordering of the items is based on the value of a sort key. The key is the value itself when sorting simple types or it can be a specific component or a combination of components when sorting complex types.

Insertion sort

This is an in-place comparison-based sorting algorithm. A sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element that is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

Selection Sort

Selection sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works?

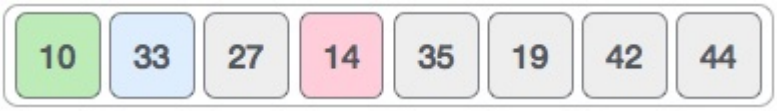
Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −

Now, let us learn some programming aspects of selection sort.

Algorithm

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Bubble Sort

Bubble sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

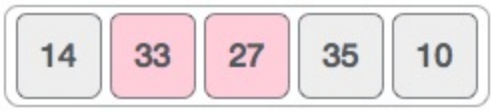
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

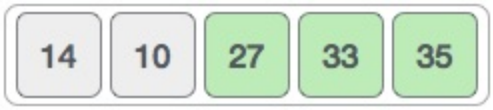
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

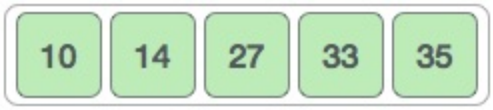
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)
   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for
   return list
   end BubbleSort
```