

# Brute Force Algorithms

## Brute Force Algorithms

- A brute force algorithm solves a problem through exhaustion: it goes through all possible choices until a solution is found.
- The time complexity of a brute force algorithm is often proportional to the input size.
- Brute force algorithms are simple and consistent, but very slow.

```
# pseudocode that prints all divisors of n  
by brute force
```

```
define printDivisors, n  
    for all numbers from 1 to n  
        if the number is a divisor of n  
            print the number
```

## Searching for smallest or largest value using linear search

Linear search can be used to search for the smallest or largest value in an unsorted list rather than searching for a match. It can do so by keeping track of the largest (or smallest) value and updating as necessary as the algorithm iterates through the dataset.

```
Create a variable called max_value_inc  
Set max_value_index to the index of the first element  
For each element in the search list  
    if element is greater than the element at max_value_index  
        Set max_value_index equal to the index of the current element  
return max_value_index
```



## Linear Search best case

For a list that contains  $n$  items, the best case for a linear search is when the target value is equal to the first element of the list. In such cases, only one comparison is needed. Therefore, the best case performance is  $O(1)$ .

## Linear Search Complexity

Linear search runs in linear time and makes a maximum of  $n$  comparisons, where  $n$  is the length of the list. Hence, the computational complexity for linear search is  $O(N)$ . The running time increases, at most, linearly with the size of the items present in the list.

## Linear Search expressed as a Function

A linear search can be expressed as a function that compares each item of the passed dataset with the target value until a match is found.

The given pseudocode block demonstrates a function that performs a linear search. The relevant index is returned if the target is found and -1 with a message that a value is not found if it is not.

```
For each element in the array
  if element equal target value then
    return its index
if element is not found, return
  "Value Not Found" message
```



## Return value of a linear search

A function that performs a linear search can return a message of success and the index of the matched value if the search can successfully match the target with an element of the dataset. In the event of a failure, a message as well as -1 is returned as well.

```
For each element in the array
  if element equal target value then
    print success message
    return its index
if element is not found
  print Value not found message
  return -1
```



## Modification of linear search function

A linear search can be modified so that all instances in which the target is found are returned. This change can be made by not 'breaking' when a match is found.

```
For each element in the searchList
    if element equal target value then
        Add its index to a list of occurrences
    if the list of occurrences is empty
        raise ValueError
    otherwise
        return the list occurrences
```



## Linear search

*Linear search* sequentially checks each element of a given list for the target value until a match is found. If no match is found, a linear search would perform the search on all of the items in the list.

For instance, if there are  $n$  number of items in a list, and the target value resides in the  $n-5$ th position, a linear search will check  $n-5$  items total.

## Linear search as a part of complex searching problems

Despite being a very simple search algorithm, linear search can be used as a subroutine for many complex searching problems. Hence, it is convenient to implement linear search as a function so that it can be reused.

## Linear Search Best and Worst Cases

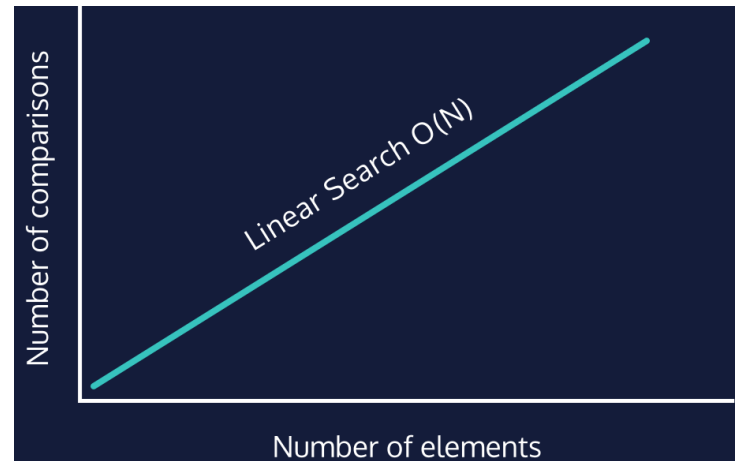
The best-case performance for the Linear Search algorithm is when the search item appears at the beginning of the list and is  $O(1)$ . The worst-case performance is when the search item appears at the end of the list or not at all. This would require  $N$  comparisons, hence, the worse case is  $O(N)$ .

## Linear Search Average Runtime

The Linear Search Algorithm performance runtime varies according to the item being searched. On average, this algorithm has a Big-O runtime of  $O(N)$ , even though the average number of comparisons for a search that runs only halfway through the list is  $N/2$ .

## Linear Search Runtime

The Linear Search algorithm has a Big-O (worst case) runtime of  $O(N)$ . This means that as the input size increases, the speed of the performance decreases linearly. This makes the algorithm not efficient to use for large data inputs.



## Throwing Exception in Linear Search

The linear search function may throw a `ValueError` with a message when the target value is not found in the search list. Calling the linear search function inside a `try` block is recommended to catch the `ValueError` exception in the `except` block.

```
def linear_search(lst, match):  
    for idx in range(len(lst)):  
        if lst[idx] == match:  
            return idx  
    else:  
        raise ValueError("{0} not in  
list".format(match))  
  
recipe = ["nori", "tuna", "soy sauce",  
          "sushi rice"]  
ingredient = "avocado"  
  
try:  
    print(linear_search(recipe, ingredient))  
except ValueError as msg:  
    print("{0}".format(msg))
```

## Find Maximum Value in Linear Search

The Linear Search function can be enhanced to find and return the maximum value in a list of numeric elements. This is done by maintaining a variable that is compared to every element and updated when its value is smaller than the current element.

```
def find_maximum(lst):  
    max = None  
    for el in lst:  
        if max == None or el > max:  
            max = el  
    return max  
  
test_scores = [88, 93, 75, 100, 80, 67,  
71, 92, 90, 83]  
  
print(find_maximum(test_scores)) # returns  
100
```

## Linear Search Multiple Matches

A linear search function may have more than one match from the input list. Instead of returning just one index to the matched element, we return a list of indices. Every time we encounter a match, we add the index to the list.

```
def linear_search(lst, match):  
    matches = []  
    for idx in range(len(lst)):  
        if lst[idx] == match:  
            matches.append(idx)  
    if matches:  
        return matches  
    else:  
        raise ValueError("{0} not in  
list".format(match))  
  
scores = [55, 65, 32, 40, 55]  
print(linear_search(scores, 55))
```

## Raise Error in Linear Search

A Linear Search function accepts two parameters:

- 1) input list to search from
- 2) target element to search for in the input 1:



If the target element is found in the list, the function returns the element index. If it is not found, the function raises an error. When implementing in Python, use the `raise` keyword with `ValueError()`.

```
def linear_search(lst, match):  
    for idx in range(len(lst)):  
        if lst[idx] == match:  
            return idx  
    raise ValueError('Sorry, {0} is not  
found.'.format(match))
```

