

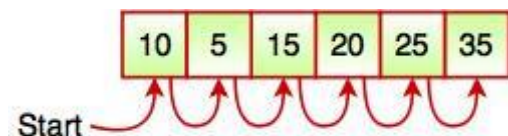
Searching

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. **Sequential Search:** In this, the list or array is traversed sequentially and every element is checked. For example: [Linear Search](#).
2. **Interval Search:** These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the centre of the search structure and divide the search space in half. For Example: [Binary Search](#).

1. Linear Search

- Linear search is also called as Sequential Search.
- Linear search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Linear search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.



The above figure shows how Linear search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Linear search is applied on the unsorted or unordered list when there are fewer elements in a list.

```
list = ["apple", "banana", "mango"]
```

Given a list search for an item in it, ex: banana

```
target = "mango"
for fruit in list:
    if fruit == target:
        return True
return False
```

This is a basic approach. Another approach for this would be,

```
>>> fruits = ["apple", "banana", "mango"]
>>> "mango" in fruits
True
>>>
```

This is called **LINEAR SEARCH**.

Best case time complexity in linear search would be **O (1)** and the **worst time complexity** would be **O (n)**.

One of the best things we can do with a computer is to search for data. It was lot of work for people to search information from the old registries and get information or update information about a particular data. So, how to improve searching process?

Just as we search in Dictionary, we can do it even in computers. In Linear search, we search every element from the beginning. but what if, we could skip or jump some elements. This is the concept behind **BINARY SEARCH**

Linear Search is a good, but its worst-case time complexity is $O(n)$. We want to make searching faster. So, the better time complexity would be $O(\log n)$. if we have a number as 10^5 , we will get a smaller value if we use **Binary Search**.

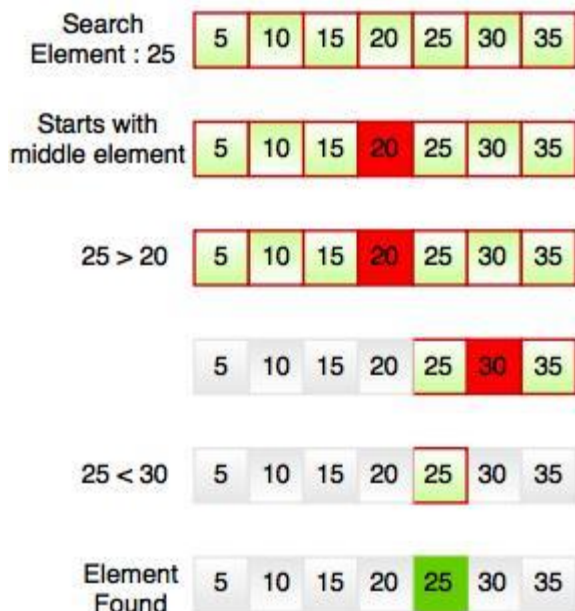
2. Binary Search

- Binary Search is used for searching an element in a sorted array.
- It is a fast search algorithm with run-time complexity of $O(\log n)$.
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.

5	10	15	20	25	30
---	----	----	----	----	----

- The above array is sorted in ascending order. As we know **binary search is applied on sorted lists only for fast searching**.

For example, if searching an element 25 in the 7-element array, following figure shows how binary search works:

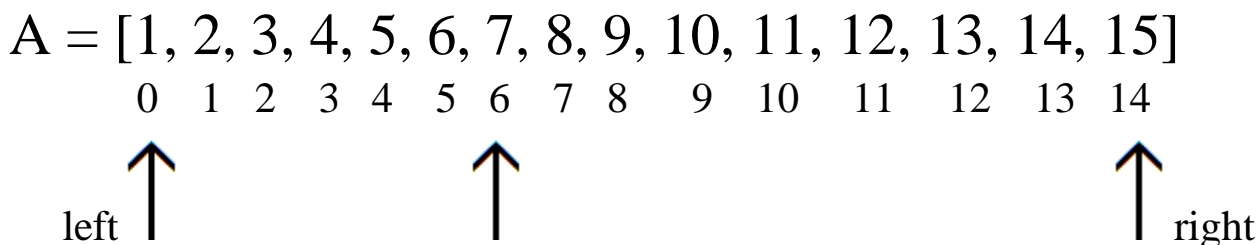


Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.

Examples:

Ex: 1

target is 3.







so we have one variable (left) at 0 and the another variable (right) is having index at pointer 15. The mid value for this would be, $(\text{left} + \text{right}) // 2 = 7$.

If the index is 6, the value at 6th index would be 7. So comparing the value, will the target be towards the left side or towards the right side? Since the target is 3, the answer will now be towards the left of the mid value. So we can simply put the right at 5th index.

A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Step 1: left    mid  right

Step 2: left   mid  right




In this, left is at 0 and right is 5. So, mid will be $(0+5)//2 = 2$. In the 2nd index the value is 3, target found.

Ex:2

IN the below list, the property of Binary Search is getting satisfied. The target value is 47 and mid value would be, $(0+5)//2 = 2$.

A = [1, 4, 15, 22, 33, 47]




0 1 2 3 4 5

Step 1: left   mid  right

Our answers would lie towards the right from the mid value.

A = [1, 4, 15, 22, 33, 47]

0 1 2 3 4 5

Step 1: left   mid  right

Step 2:  left   right

Now, the 3rd index would be our left and right value will be same.

The mid value for this would be $(3+5)//2 = 4$. The value at 4 is 33.

Since 47 is greater than 33, the value would lie towards the right side.

So, the solution space has shrunk further and the left and right value at this point would be at 5th index.

A = [1, 4, 15, 22, 33, 47]

0 1 2 3 4 5

Step 1: left ↑ mid ↑ right ↑

Step 2: left ↑ right ↑

Step 3: left ↑ right

$(5+5) // 2 = 5$. The value at 5th index is 47, target found.

CODE:

```
"""
We can apply binary search as everything is in
ascending order.
"""
def binarySearch (A, target):
    left = 0
    right = len(A) - 1

    while left <= right:
        mid = (left + right) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            right = mid - 1
        else:
            left = mid + 1

    return None

if __name__ == "__main__":
    A = [1,22,44,55,66,77,88,120,135,151]
    target = 88

    print (binarySearch (A, target))
```

Time Complexity for this would-be $O(\log n)$, because there is **division** ($n/2$)

The same approach can work for strings as well. Since **apple** comes first then **apples** in dictionary.

Leetcode: 704. Binary Search

class Solution:

```
def search(self, nums: List[int], target: int) -> int:
```

```
    left = 0
```

```
    right = len(nums) - 1
```

```
    while left <= right:
```

```
        mid = (left + right) // 2
```

```
        if nums[mid] == target:
```

```
            return mid
```

```
        elif nums[mid] > target:
```

```
            right = mid - 1
```

```
        else:
```

```
            left = mid + 1
```

```
    return None
```

Q) Given an Array where some elements are repeated and are sorted.

Eg: A = [1,2,2,3,3,3,3,4,4,4,4]

Target = 3 and the lower bound of 3. Lower bound means, value equal to or *just* less than the target.

arr = [1,4,4,13,13,15].

The lower bound of 3 in “arr” would be 0 because,

Equal to = the search element is 3, but we do not have 3 in the “arr”.

Just less than = value less than 3 and closest to 3 in the given array is 1 and its index is 0.

So, the lower bound of 3 in the “arr” is 0.

[1, 1, 1, 1, 2, 2, 8, 15, 17]

0 1 2 3 4 5 6 7 8

Lower bound of 4: We do not have 4 in the array, so the just less than value for 4 in the array would be 2 at the index 5.

Lower bound of 8 would be 6 because we have a value of 6 in the array already present and so its index would be the lower bound value.

- Lower bound is just smaller value in left or lower index same value.

[1, 1, 5, 7, 7, 7, 8, 15]; target is 7.

0 1 2 3 4 5 6 7

The lower bound of 7 in this is 3. We have the target value 7 already present in the array. The *brute-force* way for this would be doing the linear search using for loop.

```
def findLowerBound(A, target):
    prev = 0
    for i in range(len(A)):
        if A[i] > target:
            return prev
        prev = i
```

In [3, 3, 4, 4, 5, 5, 6]

0 1 2 3 4 5 6

The lower bound of 1 will be -1, because there is no value lesser than 1 in the given array. LB of 2 will also be none.

LB of 3 = 0

LB of 4 = 2

LB of 5 = 4

LB of 6 = 6

Let's have a variable prev = -1. CODE would be,

```
def findLowerBound (A, target):
    prev = 0
    for i in range(len(A)):
        if A[i] > target:
            return prev
        prev = i
```

[3, 3, 4, 5, 7, 8, 15] target = 2

0 1 2 3 4 5 6

The lower bound for target = 2, will be -1, because $2 < 3$.

For target = 5, the code will compare each value from the 0th index and to see if the target is equal to or greater than the target. And for every iteration the value of **prev** is now changing to element that is checked. So, the lower bound of 5 would be 3.

[3, 3, 3, 4, 4, 5, 7, 8] target = 6

Initially, prev = -1. It will check if $A[i] > \text{target}$, if no, then prev will become 1 and will check for the next value until it reaches the 6th index value(7). The value at 6th index is greater than 6, then $A[i] > \text{target}$ will become true and it will return the prev value.

If $A[i] == \text{target}$, then the change in the code would be,

```
def findLowerBound(A, target):
    prev = 0
    for i in range(len(A)):
        if A[i] == target:
            return i
        elif A[i] > target:
            return prev
    prev = i
```

Q) Given a sorted array A having integers and a target X. Find the no. of occurrence of x.

1 1 1 1 1 1 4 4 4 4 4 5 5 5 6 6 6

Target = 5. No of occurrences = 3

count = 0

for i in range (0, len(a)):

if a[i] == target:

count += 1

H. W

How to use upper bound to solve this problem?

CC_Week6_Day1

<https://leetcode.com/problems/binary-search/>

Implement lower bound and upper bound with $O(n)$ time complexity

Asg_Week6_Day1

Solve using binary search.

<https://leetcode.com/problems/search-insert-position/>

Implement it using lower bound upper bound: (No binary search)

<https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/>

<https://www.geeksforgeeks.org/searching-algorithms/>

<https://www.geeksforgeeks.org/the-ubiquitous-binary-search-set-1/>

<https://www.geeksforgeeks.org/find-a-peak-in-a-given-array/>