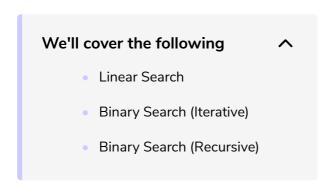
Binary Search

In this lesson, you will learn about the Binary Search algorithm and its implementation in Python.



In this lesson, we take a look at the well-known Binary Search algorithm in Python. **Binary Search** is a technique that allows you to search an ordered list of elements using a divide-and-conquer strategy. It's also an algorithm you'll want to know very well before you step into your technical interview. Now before we dive into discussing binary search, let's talk about linear search.

Linear Search

Linear search is when you iterate through an array looking for your target element. Essentially, it means sequentially scanning all the elements in the array one by one until you find your target element.

Let's see how we do this in Python:

```
def linear_search(data, target)
for i in range(len(data)):
    if data[i] == target:
        return True
    return False

linear_search(data, target)
```

The for loop on line 2 starts from i equal o and runs until i equal len(data) - 1. If in any iteration data[i] equals target, we return True to

indicate that we have found target in data. On the other hand, if the for loop terminates and the condition on line 3 never comes out to be True,

False is returned from the function (line 5). In the worst case, we might have to scan an entire array and not find what we are looking for. Thus, the worst-case runtime of a linear search would be O(n).

This is where binary search comes into play. Binary search is more efficient than the linear search. Let's find out how.

Binary Search (Iterative)

Binary search assumes that the array on which the search will take place is sorted in ascending order. In binary search, the target element is compared with the middle element of the array following which the next chunk of the array to be searched is decided. If the target matches the middle element, we are successful. Otherwise, since the array is sorted, if the target is smaller than the middle element, it could only be in the left half of the array. Alternatively, if the target is greater than the middle element, it could be in the right half of the array. So, we exclude one half of the array from the further search and repeat the same strategy to the remaining half.

Let's jump to the code below so you get a clearer idea of binary search.

```
def binary_search_iterative(data, target):
    low = 0
    high = len(data) - 1

while low <= high:
        mid = (low + high) // 2
        if target == data[mid]:
            return True
        elif target < data[mid]:
            high = mid - 1
        else:
            low = mid + 1
        return False</pre>
```

binary_search_iterative(data, target)

data and target are the input parameters to binary_search_iterative function. data is the array in which we are searching, and target is the value that we are searching for. On lines 2-3, low and high are initialized to 0 and len(data) - 1 respectively. Based on the assumption that data is a sorted list, low and high have been assigned as the indices for the minimum and the

maximum values in data.

Next, the while loop on line 5 will run until low is less than or equal to high. On line 6, mid is calculated by dividing the sum of low and high by 2 and getting the floored value because of the // operator. As specified before, target will be compared to the middle element, which is what happens on line 7. If target is equal to data[mid] (the middle element), it implies target exists in data and True is returned from the function as an indication. On the other hand, if target is less than the middle element, it means that target is somewhere in the first half of the array as the array is sorted. Therefore, we set high to mid - 1, i.e., the upper bound of the chunk of the array to be searched will be at a position to the left of mid. In contrast, if target is greater than data[mid], target must be in the second half of the array, so the lower bound (low) is set to mid + 1.

In general, we keep dividing the array into halves in the binary search instead of iterating through all the elements to search for the target element. This implies that it takes O(logn) steps to divide into halves until we reach a single element. As a result, the worst-case time complexity of a binary search is O(logn).

Binary Search (Recursive)

Now that we have implemented binary search iteratively, let's go ahead and learn how to implement the algorithm recursively:

```
def binary_search_recursive(data, target, low, high):
    if low > high:
        return False
    else:
        mid = (low + high) // 2
        if target == data[mid]:
            return True
        elif target < data[mid]:
            return binary_search_recursive(data, target, low, mid-1)
        else:
            return binary_search_recursive(data, target, mid+1, high)</pre>
```

binary_search_recursive(data, target, low, high)

In the recursive approach, in addition to data and target, low and high are also passed as input parameters to binary_search_recursive. This is to help us code our base case. The base case for this recursive function will be when low

becomes greater than high. If the base case turns out to be True, False is returned from the function to end the recursive calls (lines 2-3). On the other hand, if low is less than or equal to high, execution jumps to line 5 where mid is calculated in the same way as in the iterative function. If target is equal to data[mid], True is returned (line 7). If not, then the condition on line 8 is evaluated. If target is less than data[mid], we make a recursive call to binary_search_recursive and pass mid - 1 which is the high in the scope of the next recursive call. This will reduce the search span as it will be halved with each recursive call. Similarly, if target is greater than data[mid], low needs to be adjusted and so we pass mid + 1 to the recursive call on line 11 which is low in the next recursive call.

We keep dividing the array into halves with recursive calls until the base case is reached. As every recursive call takes constant time, the worst-case time complexity of the recursive approach is also O(logn).

Below is an executable code with all the functions that we have implemented in this lesson in a sample test case:

```
# Linear Search
                                                                                          G
def linear_search(data, target):
        for i in range(len(data)):
                if data[i] == target:
                        return True
        return False
# Iterative Binary Search
def binary_search_iterative(data, target):
        low = 0
        high = len(data) - 1
        while low <= high:
                mid = (low + high) // 2
                if target == data[mid]:
                        return True
                elif target < data[mid]:</pre>
                        high = mid - 1
                else:
                        low = mid + 1
        return False
# Recursive Binary Search
def binary_search_recursive(data, target, low, high):
        if low > high:
                return False
                mid = (low + high) // 2
                if target == data[mid]:
                        return True
                elif target < data[mid]:</pre>
```

In the next lesson, we look at a problem and solve it using a binary search.