Binary Search: Explained - Tutorial

Binary Search: Explained

Mark as Completed

This is the very first article of the Binary Search series. Until now, we have learned the linear search algorithm. Now, in this article, we will discuss another search algorithm i.e. the Binary Search algorithm. The flow of this article will be the following:

- **A real-life example of Binary Search**
- **Coding problem example**
- **Iterative code implementation of Binary Search**
- **Recursive code implementation of Binary Search**
- **Time complexity**
- **Overflow case**

A real-life example of Binary Search:

Problem statement: Assume there is a dictionary and we have to find the word “raj”.

Method 1: One of the many ways is to check every possible page of the entire dictionary and see if we can find the word “raj”. This technique is known as linear search. *Basically, we can traverse from the first till the end to find the target value in the search space i.e. the entire dictionary in our example.*

Method 2: In this case, we will optimize our search by using the property of a dictionary i.e. a dictionary is always in the sorted order.

- We will first try to open the dictionary in such a way that it is roughly divided into two parts. Then, we will check the left page. Now, assume the words on the left page starts with ‘s’. We can certainly say that our target word i.e. “raj” definitely comes before the words start with ‘s’. So, now, we need not search in the entire dictionary rather we will only search in the left half.
- Now, we will do the same thing with the left half. First, we will divide it into 2 halves and then try to locate which half contains the word “raj”. Eventually, after certain steps, we will end up finding the word “raj”.

This is a typical real-life example of binary search.

Note:

- Binary search is only applicable in a sorted search space. The sorted search space does not necessarily have to be a sorted array. It can be anything but the search space must be sorted.

- In binary search, we generally divide the search space into two equal halves and then try to locate which half contains the target. According to that, we shrink the search space size.

Coding problem example:

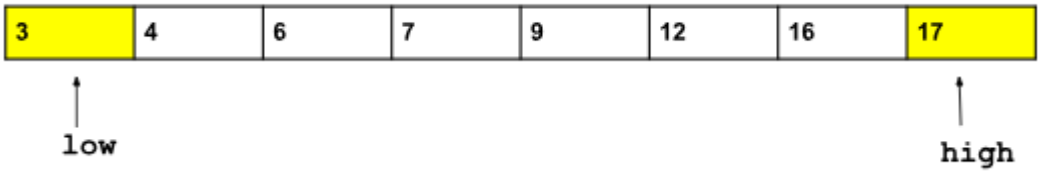
Problem statement: You are given a sorted array of integers and a target, your task is to search for the target in the given array. Assume the given array does not contain any duplicate numbers.

Let's say the given array is = {3, 4, 6, 7, 9, 12, 16, 17} and target = 6.

Solution:

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Practice: Solve Problem
▼ Iterative Implementation
▼ Algorithm / Intuition ›
We will use a couple of pointers i.e. **low** and **high** to apply binary search. Initially, the low pointer should point to the first index and the high pointer should point to the last index.



Search space: The entire area between the low and the high pointer(*including them*) is considered the search space. Here, the search space is sorted.

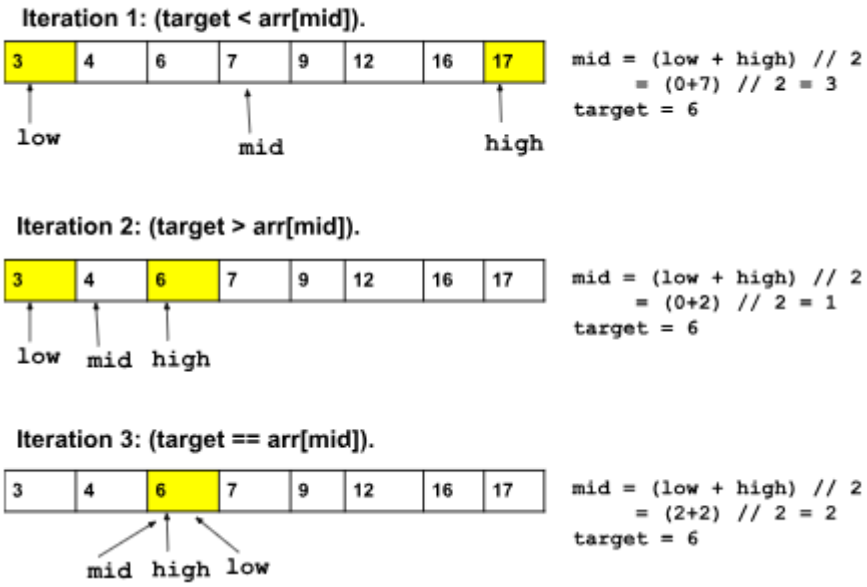
Algorithm:

Now, we will apply the binary search algorithm in the given array:

- **Step 1: Divide the search space into 2 halves:**In order to divide the search space, we need to find the middle point of it. So, we will take a **'mid'** pointer and do the following:
  mid = (low+high) // 2 ('//' refers to integer division)
- **Step 2: Compare the middle element with the target:**In this step, we can observe 3 different cases:
  - **If arr[mid] == target:** We have found the target. From this step, we can return the index of the target possibly.
  - **If target > arr[mid]:** This case signifies our target is located on the right half of the array. So, the next search space will be the right half.
  - **If target < arr[mid]:** This case signifies our target is located on the left half of the array. So, the next search space will be the left half.

- **Step 3: Trim down the search space:**Based on the probable location of the target we will trim down the search space.
    - If the target occurs on the left, we should set the high pointer to mid-1. Thus the left half will be the next search space.
    - Similarly, if the target occurs on the right, we should set the low pointer to mid+1. Thus the right half will be the next search space.

The above steps will continue until either ***we found the target*** or ***the search space becomes invalid i.e. high < low***. By definition of search space, it will lose its existence if the high pointer is appearing before the low pointer.
Dry-run:



Note: If the target is not present in the array, low and high will cross each other.

Note: *For a better understanding, please watch the video at the bottom of the page.*

Iterative implementation:

- Initially, the pointers low and high will be 0 and n-1(*where n = size of the given array*) respectively.
- Now inside a loop, we will perform the 3 steps discussed above in the algorithm section.
- The loop will run until either we fount the target or any of the pointers crosses the other.

▼ Code ›

```
def binarySearch(nums: [int], target: int):
    n = len(nums)  # size of the array
    low = 0
    high = n - 1

    # Perform the steps:
    while low <= high:
        mid = (low + high) // 2
        if nums[mid] == target:
            return mid
        elif target > nums[mid]:
            low = mid + 1
        else:
            high = mid - 1
    return -1

if __name__ == "__main__":
    a = [3, 4, 6, 7, 9, 12, 16, 17]
    target = 6
    ind = binarySearch(a, target)
    if ind == -1:
        print("The target is not present.")
    else:
        print("The target is at index:", ind)
```

Output: The target is at index: 2

▼ Complexity Analysis ›

Time Complexity:

In the algorithm, in every step, we are basically dividing the search space into 2 equal halves. This is actually equivalent to dividing the size of the array by 2, every time. After a certain number of divisions, the size will reduce to such an extent that we will not be able to divide that anymore and the process will stop. The number of total divisions will be equal to the time complexity.

Let's derive the number of divisions mathematically,

```
If a number n can be divided by 2 for x times:
        2^x = n
Therefore, x = logn (base is 2)
```

So the overall time complexity is O(logN), where N = size of the given array.

▼ Recursive Approach
▼ Algorithm / Intuition ›

Recursive implementation:

Pre-requisite: Recursion section

Approach:

Assume, the recursive function will look like this: **binarySearch(nums, low, high)**. It basically takes 3 parameters i.e. the array, the low pointer, and the high pointer. In each recursive call, we will change the value of low and high pointers to trim down the search space. Except for this, the rest of the steps will be the same.

The steps are as follows:

1. **Step 1: Divide the search space into 2 halves:**In order to divide the search space, we need to find the middle point of it. So, we will take a **'mid'** pointer and do the following:
   mid = (low+high) // 2 ('//' refers to integer division)
2. **Step 2: Compare the middle element with the target and trim down the search space:**In this step, we can observe 3 different cases:
   1. **If arr[mid] == target:** We have found the target. From this step, we can return the index of the target, and the recursion will end.
   2. **If target > arr[mid]:** This case signifies our target is located on the right half of the array. So, the next recursion call will be **binarySearch(nums, mid+1, high)**.
   3. **If target < arr[mid]:** This case signifies our target is located on the left half of the array. So, the next recursion call will be **binarySearch(nums, low, mid-1)**.
3. **Base case:** The base case of the recursion will be **low > high**. If **(low > high)**, the search space becomes invalid which means the target is not present in the array.

Note: *For a better understanding, please watch the video at the bottom of the page.*

▼ Code ›

```
def binarySearch(nums: [int], low: int, high: int, target: int):
    if low > high:
        return -1  # Base case

    # Perform the steps:
    mid = (low + high) // 2
    if nums[mid] == target:
        return mid
    elif target > nums[mid]:
        return binarySearch(nums, mid + 1, high, target)
    return binarySearch(nums, low, mid - 1, target)

def search(nums: [int], target: int):
    return binarySearch(nums, 0, len(nums) - 1, target)

if __name__ == "__main__":
    a = [3, 4, 6, 7, 9, 12, 16, 17]
    target = 6
    ind = search(a, target)
    if ind == -1:
        print("The target is not present.")
    else:
        print("The target is at index:", ind)
```

Output: The target is at index: 2

▼ Complexity Analysis ›

Time Complexity:

In the algorithm, in every step, we are basically dividing the search space into 2 equal halves. This is actually equivalent to dividing the size of the array by 2, every time. After a certain number of divisions, the size will reduce to such an extent that we will not be able to divide that anymore and the process will stop. The number of total divisions will be equal to the time complexity.

Let’s derive the number of divisions mathematically,

```
If a number n can be divided by 2 for x times:
        2^x = n
Therefore, x = logn (base is 2)
```

So the overall time complexity is O(logN), where N = size of the given array.