Order-agnostic Binary Search (easy)

We'll cover the following

Problem Statement

Given a sorted array of numbers, find if a given number 'key' is present in the array. Though we know that the array is sorted, we don't know if it's sorted in ascending or descending order. You should assume that the array can have duplicates.

Write a function to return the index of the 'key' if it is present in the array, otherwise return -1.

Example 1:

```
Input: [4, 6, 10], key = 10
Output: 2
```

Example 2:

```
Input: [1, 2, 3, 4, 5, 6, 7], key = 5
Output: 4
```

Example 3:

```
Input: [10, 6, 4], key = 10
Output: 0
```

Example 4:

```
Input: [10, 6, 4], key = 4
Output: 2
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```java
class BinarySearch {

  public static int search(int[] arr, int key) {
    // TODO: Write your code here
    return -1;
  }

  public static void main(String[] args) {
    System.out.println(BinarySearch.search(new int[] { 4, 6, 10 }, 10));
    System.out.println(BinarySearch.search(new int[] { 1, 2, 3, 4, 5, 6, 7 }, 5));
    System.out.println(BinarySearch.search(new int[] { 10, 6, 4 }, 10));
    System.out.println(BinarySearch.search(new int[] { 10, 6, 4 }, 4));
  }
}
```

Run Save Reset

Solution

To make things simple, let's first solve this problem assuming that the input array is sorted in ascending order. Here are the set of steps for **Binary Search**:

1. Let's assume `start` is pointing to the first index and `end` is pointing to the last index of the input array

(let's call it `arr`). This means:

```
int start = 0;
int end = arr.length - 1;
```

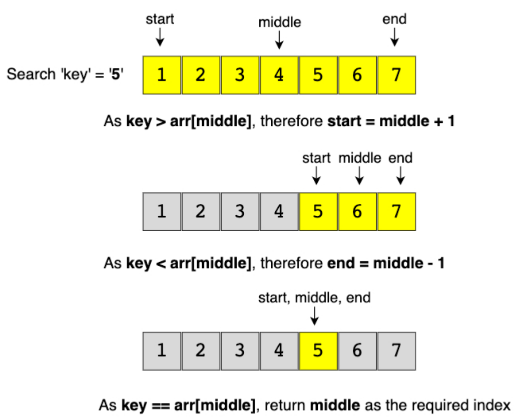
2. First, we will find the `middle` of `start` and `end`. An easy way to find the middle would be: $middle = (start + end)/2$. For **Java and C++**, this equation will work for most cases, but when `start` or `end` is large, this equation will give us the wrong result due to integer overflow. Imagine that `end` is equal to the maximum range of an integer (e.g. for Java: `int end = Integer.MAX_VALUE`). Now adding any positive number to `end` will result in an integer overflow. Since we need to add both the numbers first to evaluate our equation, an overflow might occur. The safest way to find the middle of two numbers without getting an overflow is as follows:

```
middle  = start + (end-start)/2
```

> The above discussion is not relevant for **Python**, as we don't have the integer overflow problem in pure Python.

3. Next, we will see if the 'key' is equal to the number at index `middle`. If it is equal we return `middle` as the required index.
4. If 'key' is not equal to number at index `middle`, we have to check two things:
   - If `key < arr[middle]`, then we can conclude that the `key` will be smaller than all the numbers after index `middle` as the array is sorted in the ascending order. Hence, we can reduce our search to `end = mid - 1`.
   - If `key > arr[middle]`, then we can conclude that the `key` will be greater than all numbers before index `middle` as the array is sorted in the ascending order. Hence, we can reduce our search to `start = mid + 1`.
5. We will repeat steps 2-4 with new ranges of `start` to `end`. If at any time `start` becomes greater than `end`, this means that we can't find the 'key' in the input array and we must return '-1'.

Here is the visual representation of **Binary Search** for the Example-2:



If the array is sorted in the descending order, we have to update the step 4 above as:

- If `key > arr[middle]`, then we can conclude that the `key` will be greater than all numbers after index `middle` as the array is sorted in the descending order. Hence, we can reduce our search to `end = mid - 1`.
- If `key < arr[middle]`, then we can conclude that the `key` will be smaller than all the numbers before index `middle` as the array is sorted in the descending order. Hence, we can reduce our search to `start = mid + 1`.

Finally, how can we figure out the sort order of the input array? We can compare the numbers pointed out by `start` and `end` index to find the sort order. If `arr[start] < arr[end]`, it means that the numbers are sorted in ascending order otherwise they are sorted in the descending order.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```java
class BinarySearch {

  public static int search(int[] arr, int key) {
    int start = 0, end = arr.length - 1;
    boolean isAscending = arr[start] < arr[end];
    while (start <= end) {
      // calculate the middle of the current range
      int mid = start + (end - start) / 2;

      if (key == arr[mid])
        return mid;

      if (isAscending) { // ascending order
        if (key < arr[mid]) {
          end = mid - 1; // the 'key' can be in the first half
        } else { // key > arr[mid]
          start = mid + 1; // the 'key' can be in the second half
        }
      } else { // descending order
        if (key > arr[mid]) {
          end = mid - 1; // the 'key' can be in the first half
        } else { // key < arr[mid]
          start = mid + 1; // the 'key' can be in the second half
        }
      }
    }
    return -1; // element not found
  }
```

Run | Save | Reset

Time complexity

Since, we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(logN)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.

Back
Introduction

Next
Ceiling of a Number (medium)

Mark as Completed

Report an Issue Ask a Question