Search for a Range

Given a sorted array of integers, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of$O(\log n)$.

If the target is not found in the array, return `[-1, -1]`.

For example,
Given `[5, 7, 7, 8, 8, 10]` and target value 8,
return `[3, 4]`.

Solution 1

The problem can be simply broken down as two binary searches for the begining and end of the range, respectively:

First let's find the left boundary of the range. We initialize the range to [i=0, j=n-1]. In each step, calculate the middle element [mid = (i+j)/2]. Now according to the relative value of A[mid] to target, there are three possibilities:

1. If A[mid] < target, then the range must begins on the***right*** of mid (hence i = mid+1 for the next iteration)
2. If A[mid] > target, it means the range must begins on the***left*** of mid (j = mid-1)
3. If A[mid] = target, then the range must begins ***on the left of or at*** mid (j= mid)

Since we would move the search range to the same side for case 2 and 3, we might as well merge them as one single case so that less code is needed:

2*. If A[mid] >= target, j = mid;

Surprisingly, 1 and 2* are the only logic you need to put in loop while (i < j). When the while loop terminates, the value of i/j is where the start of the range is. Why?

No matter what the sequence originally is, as we narrow down the search range, eventually we will be at a situation where there are only two elements in the search range. Suppose our target is 5, then we have only 7 possible cases:

```
case 1: [5 7] (A[i] = target < A[j])
case 2: [5 3] (A[i] = target > A[j])
case 3: [5 5] (A[i] = target = A[j])
case 4: [3 5] (A[j] = target > A[i])
case 5: [3 7] (A[i] < target < A[j])
case 6: [3 4] (A[i] < A[j] < target)
case 7: [6 7] (target < A[i] < A[j])
```

For case 1, 2 and 3, if we follow the above rule, since mid = i => A[mid] = target in these cases, then we would set j = mid. Now the loop terminates and i and j both point to the first 5.

For case 4, since A[mid] < target, then set i = mid+1. The loop terminates and both i and j point to 5.

For all other cases, by the time the loop terminates, A[i] is not equal to 5. So we can easily know 5 is not in the sequence if the comparison fails.

In conclusion, when the loop terminates, if A[i]==target, then i is the left boundary of the range; otherwise, just return -1;

For the right of the range, we can use a similar idea. Again we can come up with several rules:

1. If A[mid] > target, then the range must begins on the***left*** of mid (j = mid-1)
2. If A[mid] < target, then the range must begins on the***right*** of mid (hence i =

   mid+1 for the next iteration)
3. If A[mid] = target, then the range must begins ***on the right of or at*** mid (i= mid)

Again, we can merge condition 2 and 3 into:

```
2* If A[mid] <= target, then i = mid;
```

However, the terminate condition on longer works this time. Consider the following case:

```
[5 7], target = 5
```

Now A[mid] = 5, then according to rule 2, we set i = mid. This practically does nothing because i is already equal to mid. As a result, the search range is not moved at all!

The solution is by using a small trick: instead of calculating mid as mid = (i+j)/2, we now do:

```
mid = (i+j)/2+1
```

Why does this trick work? When we use mid = (i+j)/2, the mid is rounded to the lowest integer. In other words, mid is always *biased* towards the left. This means we could have i == mid when j - i == mid, but we NEVER have j == mid. So in order to keep the search range moving, you must make sure the new i is set to something different than mid, otherwise we are at the risk that i gets stuck. But for the new j, it is okay if we set it to mid, since it was not equal to mid anyways. Our two rules in search of the left boundary happen to satisfy these requirements, so it works perfectly in that situation. Similarly, when we search for the right boundary, we must make sure i won't get stuck when we set the new i to i = mid. The easiest way to achieve this is by making mid *biased* to the right, i.e. mid = (i+j)/2+1.

All this reasoning boils down to the following simple code:

```
vector<int> searchRange(int A[], int n, int target) {
    int i = 0, j = n - 1;
    vector<int> ret(2, -1);
    // Search for the left one
    while (i < j)
    {
        int mid = (i + j) /2;
        if (A[mid] < target) i = mid + 1;
        else j = mid;
    }
    if (A[i]!=target) return ret;
    else ret[0] = i;

    // Search for the right one
    j = n-1;  // We don't have to set i to 0 the second time.
    while (i < j)
    {
        int mid = (i + j) /2 + 1;    // Make mid biased to the right
        if (A[mid] > target) j = mid - 1;
        else i = mid;                // So that this won't make the search range s
tuck.
    }
    ret[1] = j;
    return ret;
}
```

written by stellari original link here

Solution 2

```java
public class Solution {
    public int[] searchRange(int[] A, int target) {
        int start = Solution.firstGreaterEqual(A, target);
        if (start == A.length || A[start] != target) {
            return new int[]{-1, -1};
        }
        return new int[]{start, Solution.firstGreaterEqual(A, target + 1) - 1};
    }

    //find the first number that is greater than or equal to target.
    //could return A.length if target is greater than A[A.length-1].
    //actually this is the same as lower_bound in C++ STL.
    private static int firstGreaterEqual(int[] A, int target) {
        int low = 0, high = A.length;
        while (low < high) {
            int mid = low + ((high - low) >> 1);
            //low <= mid < high
            if (A[mid] < target) {
                low = mid + 1;
            } else {
                //should not be mid-1 when A[mid]==target.
                //could be mid even if A[mid]>target because mid<high.
                high = mid;
            }
        }
        return low;
    }
}
```

written by kxcf original link here

Solution 3

```
public class Solution {
    public int[] searchRange(int[] A, int target) {
        int[] range = {A.length, -1};
        searchRange(A, target, 0, A.length - 1, range);
        if (range[0] > range[1]) range[0] = -1;
        return range;
    }

    public void searchRange(int[] A, int target, int left, int right, int[] range
) {
        if (left > right) return;
        int mid = left + (right - left) / 2;
        if (A[mid] == target) {
            if (mid < range[0]) {
                range[0] = mid;
                searchRange(A, target, left, mid - 1, range);
            }
            if (mid > range[1]) {
                range[1] = mid;
                searchRange(A, target, mid + 1, right, range);
            }
        } else if (A[mid] < target) {
            searchRange(A, target, mid + 1, right, range);
        } else {
            searchRange(A, target, left, mid - 1, range);
        }
    }
}
```

written by xuanaux original link here