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

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

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

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

**Example 2:**

Input: nums = [5,7,7,8,8,10], target = 6

Output: [-1,-1]

**Example 3:**

Input: nums = [], target = 0

Output: [-1,-1]

**Constraints:**

0 <= nums.length <= 10^5

-10^9 <= nums[i] <= 10^9

nums is a non-decreasing array.

-10^9 <= target <= 10^9

**Attempt 1: 2022-09-18**

**Solution 1: 5 min, use while(lo <= hi) template as same as L704 by using Find Target First/Last Occurrence**

class Solution {

    public int[] searchRange(int[] nums, int target) {

        int starting\_pos = findStartingPos(nums, target);

        int ending\_pos = findEndingPos(nums, target);

        return new int[] {starting\_pos, ending\_pos};

    }

    private int findStartingPos(int[] nums, int target) {

        int lo = 0;

        int hi = nums.length - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(nums[mid] >= target) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        if(lo >= nums.length || nums[lo] != target) {

            return -1;

        }

        return lo;

    }

    private int findEndingPos(int[] nums, int target) {

        int lo = 0;

        int hi = nums.length - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(nums[mid] > target) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        if(hi < 0 || nums[hi] != target) {

            return -1;

        }

        return hi;

    }

}

Space Complexity: O(1)

Time Complexity: O(logn)

**Solution 2: 10 min, use while(lo < hi) template as same as L278. First Bad Version**

class Solution {

    public int[] searchRange(int[] nums, int target) {

        int starting\_pos = findStartingPos(nums, target);

        int ending\_pos = findEndingPos(nums, target);

        return new int[] {starting\_pos, ending\_pos};

    }

    private int findStartingPos(int[] nums, int target) {

        int lo = 0;

        int hi = nums.length - 1;

        while(lo < hi) {

            int mid = lo + (hi - lo) / 2;

            if(nums[mid] >= target) {

hi = mid;

            } else {

                lo = mid + 1;

            }

        }

        if(lo >= nums.length || nums[lo] != target) {

            return -1;

        }

        return lo;

    }

    private int findEndingPos(int[] nums, int target) {

        int lo = 0;

        int hi = nums.length - 1;

        while(lo < hi) {

            int mid = lo + (hi - lo) / 2 + 1;

            if(nums[mid] > target) {

                hi = mid - 1;

            } else {

                lo = mid;

            }

        }

        if(lo >= nums.length || nums[lo] != target) {

            return -1;

        }

        return lo;

    }

}

Space Complexity: O(1)

Time Complexity: O(logn)

**while(lo < hi) template refer to**

<https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/discuss/14699/Clean-iterative-solution-with-two-binary-searches-(with-explanation)>

The problem can be simply broken down as two binary searches for the beginning 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:

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

If A[mid] > target, it means the range must begins on the ***left*** of mid (j = mid-1)

If A[mid] = target, then the range must begins ***on the left of or at*** mid (j= mid)

**Note: For left most we start comparison between if A[mid] < target, for easily remember: {left, <} -> {i=mid+1}**

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;

Together with case 1 is:

1. If A[mid] < target, i = mid + 1;

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:

If A[mid] > target, then the range must begins on the ***left*** of mid (j = mid-1)

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

If A[mid] = target, then the range must begins ***on the right of or at*** mid (i= mid)

**Note: For right most we start comparison between if A[mid] > target, for easily remember: {right, >} -> {j=mid-1}**

Again, we can merge condition 2 and 3 into:

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

Together with case 1 is:

1. If A[mid] > target, then j = mid - 1;

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!

**Right Biased Trick**

**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.

**Test out by input: [5,7,7,8,8,10], if keep as mid = (i + j) / 2 and not '+ 1' will get TLE because of infinite loop for while(i < j)**

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 stuck.

    }

    ret[1] = j;

    return ret;

}

**以下是在** [L278.First Bad Version (Ref.L34,L704)](note://D1F4A021B45746B29CF97EE2191C924D) **中关于while(lo < hi)的模版的解释：**

**How while(lo < hi) work with hi = mid and lo = mid + 1 to find first bad version ?**

**Refer to**

<https://leetcode.com/problems/first-bad-version/discuss/71386/An-clear-way-to-use-binary-search>

public int firstBadVersionLeft(int n) {

int i = 1;

int j = n;

while (i < j) {

int mid = i + (j - i) / 2;

if (isBadVersion(mid)) {

j = mid;

} else {

i = mid + 1;

}

}

return i;

}

**Step by Step**

Template:

while lo < hi:

  mid = (lo+hi)/2

  if function(mid):

    hi = mid

  else:

    lo = mid+1

return lo

=================================================================================

e.g

isBadVersion result -> FFTTTT

index start from 1  -> 123456

Find left most 'T'

while(lo < hi) -> ending condition lo == hi, means lo and hi overlap at same position ... hi ...

                                                                                          lo

Round 1:

lo=1,hi=6 -> mid=3

isBadVersion(3)=T -> hi=mid=3 (hi=mid means include 'mid')

Round 2:

lo=1,hi=3 -> mid=2

isBadVersion(2)=F -> lo=mid+1=3 (lo=mid+1 means skip 'mid')

Now lo == hi while loop end, and hi can equal to mid, which means include the final answer,

return either hi or lo is fine

**How while(lo < hi) work with hi = mid and lo = mid + 1 to find last bad version ?**

**Refer to**

<https://leetcode.com/problems/first-bad-version/discuss/71386/An-clear-way-to-use-binary-search>

It is obvious that the version would looks like the following:

FFTTTT (The first two are correct version, the rest are bad ones)

To find the right most T we need to notice that since we are looking for the first bad version not the last correct version, we need to return the (position when the binary search stop) + 1

public int firstBadVersion(int n) {

if (isBadVersion(1)) {

return 1;

}

int i = 1;

int j = n;

while (i < j) {

int mid = i + (j - i) / 2 + 1;

if (!isBadVersion(mid)) {

j = mid - 1;

} else {

i = mid;

}

}

return i;

}

**Step by Step**

e.g

isBadVersion result -> FFTTTT

index start from 1  -> 123456

Find right most 'T'

while(lo < hi) -> ending condition lo == hi, means lo and hi overlap at same position ... hi ...

                                                                                          lo

Round 1:

lo=1,hi=6 -> mid=lo+(hi-lo)/2+1=1+(6-1)/2+1=4

isBadVersion(4)=T -> lo=mid=4 (lo=mid means include 'mid')

Round 2:

lo=4,hi=6 -> mid=lo+(hi-lo)/2+1=4+(6-4)/2+1=6

isBadVersion(6)=T -> lo=mid=6 (lo=mid means include 'mid')

Now lo == hi while loop end, and lo can equal to mid, which means include the final answer,

return either hi or lo is fine

**Solution 3: 10 min, use while(lo + 1 < hi) template from JiuZhang**

class Solution {

    public int[] searchRange(int[] nums, int target) {

        int starting\_pos = findStartingPos(nums, target);

        int ending\_pos = findEndingPos(nums, target);

        return new int[] {starting\_pos, ending\_pos};

    }

    private int findStartingPos(int[] nums, int target) {

        if(nums == null || nums.length == 0) {

            return -1;

        }

        int lo = 0;

        int hi = nums.length - 1;

        while(lo + 1 < hi) {

            int mid = lo + (hi - lo) / 2;

            if(nums[mid] < target) {

                lo = mid;

            } else {

                hi = mid;

            }

        }

        if(nums[lo] == target) {

            return lo;

        }

        if(nums[hi] == target) {

            return hi;

        }

        return -1;

    }

    private int findEndingPos(int[] nums, int target) {

        if(nums == null || nums.length == 0) {

            return -1;

        }

        int lo = 0;

        int hi = nums.length - 1;

        while(lo + 1 < hi) {

            int mid = lo + (hi - lo) / 2;

            if(nums[mid] == target) {

                lo = mid;

            } else if(nums[mid] > target) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // Check on 'end' first for requirement about find the last position

        // Test case: when two duplicate '2' as input, we suppose return second 2 first,

        // which means check 'hi' rather than 'lo' first

        // Input: [2,2] and 2

        // Output: [0,0]

        // Expected: [0,1]

        if(nums[hi] == target) {

            return hi;

        }

        if(nums[lo] == target) {

            return lo;

        }

        return -1;

    }

}

Space Complexity: O(1)

Time Complexity: O(logn)

**Refer to**

<https://github.com/lampardchelsea/hello-world/blob/master/lintcode/BinarySearch/VideoExamples/FirstPositionOfTarget.java>

<https://github.com/lampardchelsea/hello-world/blob/master/lintcode/BinarySearch/VideoExamples/LastPositionOfTarget.java>

**Refer to**

[L278.First Bad Version (Ref.L34,L704)](note://D1F4A021B45746B29CF97EE2191C924D)

[L704.Binary Search](note://484532A5D2CB4B98A28218674438D5D8)