<https://leetcode.com/problems/search-insert-position/>

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

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

**Example 1:**

Input: nums = [1,3,5,6], target = 5

Output: 2

**Example 2:**

Input: nums = [1,3,5,6], target = 2

Output: 1

**Example 3:**

Input: nums = [1,3,5,6], target = 7

Output: 4

**Constraints:**

1 <= nums.length <= 10^4

-10^4 <= nums[i] <= 10^4

nums contains **distinct** values sorted in **ascending** order.

-10^4 <= target <= 10^4

**Attempt 1: 2022-10-02**

**Solution 1: 5min, Binary Search Find Upper Boundary (template based on while(lo <= hi), refer L704.Binary Search)**

class Solution {

    // Convert problem into find the most recent value position smaller

    // or equal to target

    // 1. If find the value equal to target, return position

    // 2. If not find, return most recent smaller value's position plus 1

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

        int len = nums.length;

        int lo = 0;

        int hi = len - 1;

        while(lo <= hi) {

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

            if(nums[mid] == target) {

                return mid;

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

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // 'hi' is the upper boundary, add '1' is the insert position

        return hi + 1;

    }

}

Space Complexity: O(1)

Time Complexity: O(logn)

**Solution 2: 10min, refer to L34.Find First and Last Position of Element in Sorted Array Solution 2 while(lo < hi) Right Biased template**

class Solution {

    // Convert problem into find the most recent value position smaller

    // or equal to target

    // 1. If find the value equal to target, return position

    // 2. If not find, return most recent smaller value's position plus 1

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

        int len = nums.length;

        int lo = 0;

        int hi = len - 1;

        while(lo < hi) {

            // Since insert always happen after element, use right biased

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

            if(nums[mid] == target) {

                return mid;

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

                hi = mid - 1;

            } else {

                lo = mid;

            }

        }

        if(nums[lo] < target) {

            return lo + 1;

        }

        return lo;

    }

}

Space Complexity: O(1)

Time Complexity: O(logn)

**Solution 3: 60min, too long to get the trick on setting hi = len rather than hi = len - 1, follow L34.Find First and Last Position of Element in Sorted Array Solution 2 while(lo < hi) template**

class Solution {

    // Convert problem into find the most recent value position smaller

    // or equal to target

    // 1. If find the value equal to target, return position

    // 2. If not find, return most recent smaller value's position plus 1

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

        int len = nums.length;

        int lo = 0;

        // No len(nums) - 1, as we might need to insert at the end of the array

        //int hi = len - 1;

        int hi = len;

        while(lo < hi) {

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

            if(nums[mid] >= target) {

hi = mid;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

}

Space Complexity: O(1)

Time Complexity: O(logn)

**Refer to**

<https://leetcode.com/problems/search-insert-position/discuss/15110/Very-concise-and-efficient-solution-in-Java>

public class Solution {

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

        int low = 0, high = nums.length;

        while(low < high) {

            int mid = low + (high - low) / 2;

            if(nums[mid] < target)

                low = mid + 1;

            else

                high = mid;

        }

        return low;

    }

}

<https://leetcode.wang/leetCode-35-Search-Insert-Position.html>

给定一个有序数组，依旧是二分查找，不同之处是如果没有找到指定数字，需要返回这个数字应该插入的位置。这道题比较简单，在二分查找的基础上，只要想清楚返回啥就够了。想的话，就考虑最简单的情况如果数组只剩下 2 5，target 是 1, 3, 6 的时候，此时我们应该返回什么就行。

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

    int start = 0;

    int end = nums.length - 1;

    if (nums.length == 0) {

        return 0;

    }

    while (start < end) {

        int mid = (start + end) / 2;

        if (target == nums[mid]) {

            return mid;

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

            end = mid;

        } else {

            start = mid + 1;

        }

    }

    //目标值在不在当前停的位置的前边还是后边

    if(target>nums[start]){

        return start + 1;

    }

    //如果小于的话，就返回当前位置，跑步超过第二名还是第二名，所以不用减 1。

    else{

        return start;

    }

}

这道题不难，但是对于二分查找又有了一些新认识。

首先，一定要注意，数组剩下偶数个元素的时候，中点取的是左端点。例如 1 2 3 4，中点取的是 2。正因为如此，我们更新 start 的时候不是直接取 mid ，而是 mid + 1。因为剩下两个元素的时候，mid 和 start 是相同的，如果不进行加 1 会陷入死循环。

然后上边的算法，返回最终值的时候，我们进行了一个 if 的判断，那么能不能避免呢。

我们开始更新 start 的时候，是 mid + 1，如果剩两个元素，例如 2 4，target = 6 的话，此时 mid = 0，start = mid + 1 = 1，我们返回 start + 1 = 2。如果 mid 是右端点，那么 mid = 1，start = mid + 1 = 2，这样就可以直接返回 start 了，不需要在返回的时候加 1 了

怎么做到呢？最最开始的时候我们取 end 的时候是 end = nums.length - 1。**如果我们改成 end = nums.length，这样每次取元素的时候，如果和之前对比，取到的就是右端点了。这样的话，最后返回的时候就不需要多加 1 了。选用 end = nums.length 的效果类似于 Solution 2 中的 Right Biased ==> int mid = lo + (hi - lo) / 2 + 1;**

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

    int start = 0;

    int end = nums.length;

    if (nums.length == 0) {

        return 0;

    }

    while (start < end) {

        int mid = (start + end) / 2;

        if (target == nums[mid]) {

            return mid;

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

            end = mid;

        } else {

            start = mid + 1;

        }

    }

    return start;

}

**Solution 4: 60min, intuitive solution No Template Based, not follow the template from L34.Find First and Last Position of Element in Sorted Array Solution 2**

class Solution {

    // Convert problem into find the most recent value position smaller

    // or equal to target

    // 1. If find the value equal to target, return position

    // 2. If not find, return most recent smaller value's position plus 1

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

        int len = nums.length;

        int lo = 0;

        int hi = len - 1;

        while(lo < hi) {

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

            if(nums[mid] == target) {

                return mid;

            // If A[m] < target, then the insertion position should be at least at m + 1, because it can be even larger, i.e. to the right of m + 1(including m + 1), we set l = m + 1;

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

                lo = mid + 1;

            // If A[m] > target, the insertion position should be at most m, because it can even smaller. So we would set r = m

            } else {

                hi = mid;

            }

        }

        // Note: numw[lo] != target not applicable, must nums[lo] < target

        // When nums[lo] < target we have lo + 1, same as when nums[mid] < target

        // we have lo = mid + 1

        if(nums[lo] < target) {

            return lo + 1;

        }

        return lo;

    }

}

**Refer to**

<https://leetcode.com/problems/search-insert-position/discuss/249092/Come-on-forget-the-binary-search-patterntemplate!-Try-understand-it>!

l means left index, r means right index, m means middle index

**Comments**

I have seen many different styles of binary search solution, I saw some people tend to use some pattern whenever solving binary search problems, i.e.

while(l <= r){...}. Same while condition

r = m - 1;, right is always m - 1

l = m + 1: , left is always m + 1

**It is a good idea to use pattern/template for binary search?**I can only speak for myself. At least for me, I do not think so. I prefer to choose the binary search rules based on the context of the problem to decide how to play with l and r.If you are using pattern, you have to remember it. You are not understanding the core of binary search! This is a big issue I think.

**Algorithms**

I want to give my explanation below:

First of all, we assume [l, r] is the possible answer range(inclusive) for this question. So initially l = 0; and r = n - 1;

we calculate int m = l + (r - l)/2; rather than int m = (l + r)/2; to avoid overflow.

Clearly, if A[m] = target, m is exactly the position we should insert the target

if A[m] < target, then the insertion position should be **at least** at m + 1, because it can be even larger, i.e. to the right of m + 1(including m + 1), we set l = m + 1;

if A[m] > target, the insertion position should be **at most** m, because it can even smaller. So we would set r = m.

An example:

target: 6 and m = 7

1 5 7 9

        m

      result

Then how to determine the while loop condition, l < r, l <= r or l < r - 1?

The answer is that we analyze and conclude with the one **which will not cause deal while loop** in our code while we are moving our left and right pointers

a. l = m + 1

b. r = m

We will analyze which would happen when we have 3, 2 and 1 elements at the end below

First let's assume there are **3 elements** left at last as shown below:

x  x  x  5 7  9  x  x  x

            lm  r

we can see that l = m + 1 can help shrink the search space by 2:

x  x  x  5 7  9  x  x  x

            lm  r

and r = m can help shrink the size by 1:

x  x  x  5 7  9  x  x  x

            lr

so 3 elements will not cause dead while loop. we should be fine.

Next we reduce to **2 elements**:

x  x  x  5  7  x  x  x

          l/m  r

Same way, we can see that l = m + 1 and r = m can both shrink the size by 1, no dead loop as well.

Then we can safely go to the cases when we have only **1 element** at the end:

x  x  x  5  x  x  x

          l/m/r

We can see that l = m + 1 will not cause dead loop, but with r = m we **cannot shrink the size anymore**, so it might lead to a dead while loop. So we need to jump out of the while loop when we have 1 element. The condition of 1 element is i = j, so we choose our while condition: while (l < r)

At the end, we need to check the last element: nums[l/r] which has not been checked in binary search loop with target to determine the index. We call it the **post processing**.

**Summary**

Actually I always do my analysis start from 2 elements instead of 3 elements, because for most of binary search problems, we always can shrink the search space to 2 or 1 element at the end. With 2 elements, it is pretty quick to identify the condition based on your rule.

**Final Code**

class Solution {

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

if(nums == null || nums.length == 0) return 0;

int n = nums.length;

int l = 0;

int r = n - 1;

while(l < r){

int m = l + (r - l)/2;

if(nums[m] == target) return m;

else if(nums[m] > target) r = m; // right could be the result

else l = m + 1; // m + 1 could be the result

}

// 1 element left at the end

// post-processing

return nums[l] < target ? l + 1: l;

}

}

**Refer to**

[L34.Find First and Last Position of Element in Sorted Array (Ref.L278,L704)](note://D037E718AB784CAE8BECDC53F4451102)

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