Binary Search

Li Yin[1]

January 26, 2019

[1]www.liyinscience.com

0.1 Binary Search

To search in an sorted array or string, using brute force with a for loop, it takes $O(n)$ time. Binary search is designed to speed up the searching if the array or string is already sorted. It has used divide and conquer method, each time we compare our target with the middle element of the array and with the comparison result to decide the next search region: either the left half or the right half. Therefore, each step we filter out half of the array which gives the time complexity function $T(n) = T(n/2) + O(1)$, which decrease the time complexity to $O(\log n)$.

Binary Search can be applied to different tasks:

1. Find Exact target, find the first position that value>= target, find the last position that value<= target.(this is called lower_bound, and upper_bound.

0.1.1 Standard Binary Search and Python Module bisect

The binary search is usually carried out on Static sorted array or 2D matrix. There are three basic cases: (1) find the exact target that value = target; If there are duplicates, we are more likely to be asked to (2) find the first position that has value >= target; (3) find the first position that has value <= target. Here, we use two example array: one without duplicates and the other has duplicates.

```
a = [2, 4, 5, 9]
b = [0, 1, 1, 1, 1, 1]
```

Find the Exact Target This is the most basic application of binary search. We can set two pointers, l and r. Each time we compute the middle position, and check if it is equal to the target. If, it is, return the position; if it is smaller than the target, move to the left half, otherwise, move to the right half. The Python code is given:

```
def standard_binary_search(lst, target):
    l, r = 0, len(lst) - 1
    while l <= r:
        mid = l + (r - l) // 2
        if lst[mid] == target:
            return mid
        elif lst[mid] < target:
            l = mid + 1
        else:
            r = mid - 1
    return -1 # target is not found
```

Now, run the example:

```
print("standard_binary_search: ", standard_binary_search(a,3),
    standard_binary_search(a,4), standard_binary_search(b, 1))
```

The print out is:

```
standard_binary_search:  -1 1 2
```

From the example, we can see there is multiple **duplicates** of the target exist, it can possible return any one of them. And for the case when the target is not exist, it simply returns -1. In reality, we might need to find a position where we can potentially insert the target to keep the sorted array sorted. There are two cases: (1) the first position that we can insert, which is the first position that has value>= target (2) and the last position we can insert, which is the first position that has value > target. For example, if we try to insert 3 in a, and 1 in b, the first position should be 1 and 1 in each array, and the last position is 1 and 6 instead. For this two cases, we have a Python built-in Module **bisect** which offers two methods: bisect_left() and bisect_right() for this two cases respectively.

Find the First Position that value >= target This way the target position separate the array into two halves: value < target, target_position, value>= target. In order to meet the purpose, we make sure if value < target, we move to the right side, else, move to the left side.

```
# bisect_left, no longer need to check the mid element,
# it separate the list in to two halfs: value < target, mid,
    value >= target
def bisect_left_raw(lst, target):
    l, r = 0, len(lst)-1
    while l <= r:
        mid = l + (r-l)//2
        if lst[mid] < target: # move to the right half if the
    value < target, till
            l = mid + 1 #[mid+1, right]
        else:# move to the left half is value >= target
            r = mid - 1  #[left, mid-1]
    return l # the final position is where
```

Find the First Position that value > target This way the target position separate the array into two halves: value <= target, target_position, value> target. Therefore, we simply change the condition of if value < target to if value <= target, then we move to the right side.

```
#bisect_right: separate the list into two halfs: value<= target,
     mid, value > target
def bisect_right_raw(lst, target):
    l, r = 0, len(lst)-1
    while l <= r:
        mid = l + (r-l)//2
        if lst[mid] <= target:
            l = mid + 1
        else:
            r = mid -1
    return l
```

Now, run an example:

```
print("bisect left raw: find 3 in a :", bisect_left_raw(a,3), '
    find 1 in b: ', bisect_left_raw(b, 1))
print("bisect right raw: find 3 in a :", bisect_right_raw(a, 3),
     'find 1 in b: ', bisect_right_raw(b, 1))
```

The print out is:

```
bisect left raw: find 3 in a : 1 find 1 in b:  1
bisect right raw: find 3 in a : 1 find 1 in b:  6
```

Bonus For the last two cases, if we return the position as l-1, then we get the last position that value < target, and the last position value <= target.

Python Built-in Module bisect This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. It offers six methods as shown in Table 1. However, only two are most commonly used: bisect_left and bisect_right. Let's see come examplary

Table 1: Methods of **bisect**

Method	Description
bisect_left(a, x, lo=0, hi=len(a)	The parameters lo and hi may be used to specify a subset of the list; the function is the same as bisect_left_raw
bisect_right(a, x, lo=0, hi=len(a)	The parameters lo and hi may be used to specify a subset of the list; the function is the same as bisect_right_raw
bisect(a, x, lo=0, hi=len(a))	Similar to bisect_left(), but returns an insertion point which comes after (to the right of) any existing entries of x in a.
insort_left(a, x, lo=0, hi=len(a))	This is equivalent to a.insert(bisect.bisect_left(a, x, lo, hi), x).
insort_right(a, x, lo=0, hi=len(a))	This is equivalent to a.insert(bisect.bisect_right(a, x, lo, hi), x).
insort(a, x, lo=0, hi=len(a))	Similar to insort_left(), but inserting x in a after any existing entries of x.

code:

```
from bisect import bisect_left,bisect_right, bisect
print("bisect left: find 3 in a :", bisect_left(a,3), 'find 1 in
    b: ', bisect_left(b, 1)) # lower_bound, the first position
   that value>= target
print("bisect right: find 3 in a :", bisect_right(a, 3), 'find 1
    in b: ', bisect_right(b, 1)) # upper_bound, the last
   position that value <= target
```

The print out is:

```
bisect left: find 3 in a : 1 find 1 in b:  1
bisect right: find 3 in a : 1 find 1 in b:  6
```

0.1.2 Binary Search in Rotated Sorted Array

The extension of the standard binary search is on array that the array is ordered in its own way like rotated array.

Binary Search in Rotated Sorted Array (See LeetCode probelm, 33. Search in Rotated Sorted Array (medium). Suppose an array sorted (without duplicates) in ascending order is rotated at some pivot unknown to you beforehand. (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2). You are given a target value to search. If found in the array return its index, otherwise return -1. You may assume no duplicate exists in the array.

```
Example 1:

Input: nums = [3, 4,5,6,7,0,1,2], target = 0
Output: 5

Example 2:

Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

In the rotated sorted array, the array is not purely monotonically. Instead, there is one drop in the array because of the rotation, where cuts the array into two parts. Suppose we are starting with a standard binary search with example 1, at first, we will check index 3, then we need to move to the right side? Assuming we compare our middle item with the left item,

```
if nums[mid] > nums[l]: # the left half is sorted
elif nums[mid] < nums[l]: # the right half is sorted
else: # for case like [1,3], move to the right half
```

For a standard binary search, we simply need to compare target with middle item to decide which way to go. In this case, we can use objection. Check which side is sorted, because no matter where the left, right and the middle index is, there is always one side that is sorted. So if the left side is sorted, and the value is in the range of the [left, mid], then we move to the left part, else we object the left side, and move to the right side instead.

The code is shown:

```
'''implemente the rotated binary search'''
def RotatedBinarySearch(nums, target):
    if not nums:
        return -1

    l,r = 0,len(nums)-1
    while l<=r:
        mid = l+ (r-l)//2
        if nums[mid] == target:
            return mid
         if nums[l] < nums[mid]: # if the left part is sorted
                if nums[l] <= target <= nums[mid]:
```

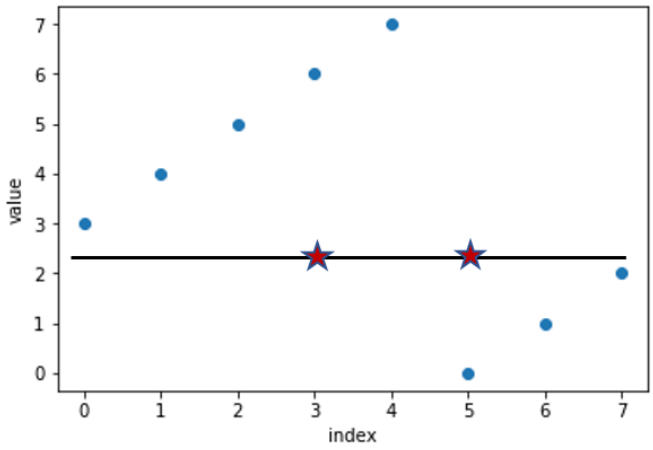

Figure 1: Example of Rotated Sorted Array

```
                    r = mid-1
                else:
                    l = mid+1
            elif nums[l] > nums[mid]: # if the right side is
    sorted
                if nums[mid] <= target <= nums[r]:
                    l = mid+1
                else:
                    r = mid-1
            else:
                l = mid + 1
    return -1
```

What happens if there is duplicates in the rotated sorted array?

In fact, similar comparing rule applies:

```
if nums[mid] > nums[l]: # the left half is sorted
elif nums[mid] < nums[l]: # the right half is sorted
else: # for case like [1,3], or [1, 3, 1, 1, 1] or [3, 1, 2,
    3, 3, 3]
   only l++
```

0.1.3 Binary Search on Result Space

If the question gives us the context: the target is in the range [left, right], we need to search the first or last position that satisfy a condition function. We can apply the concept of standard binary search and bisect_left and bisect_right and its mutant. Where we use the condition function to replace

the value comparison between target and element at middle position. There steps we need:

1. get the result search range [l, r] which is the initial value for l and r pointers.
2. decide the valid function to replace such as if lst[mid] < target
3. decide which binary search we use: standard, bisect_left/ bisect_right or its mutant.

For example:

0.1 **441. Arranging Coins (easy)**. You have a total of n coins that you want to form in a staircase shape, where every k-th row must have exactly k coins. Given n, find the total number of full staircase rows that can be formed. n is a non-negative integer and fits within the range of a 32-bit signed integer.

```
Example 1:

n = 5

The coins can form the following rows:
*
* *
* *

Because the 3rd row is incomplete, we return 2.
```

Analysis: Given a number n>=1, the minimum row is 1, and the maximum is n. Therefore, our possible result range is [1, n]. These can be treated as index of the sorted array. For a given row, we write a function to check if it is possible. We need a function $r*(r+1)//2 <= n$. For this problem, we need to search in the range of [1, n] to find the last position that is valid. This is bisect_left or bisect_right, where we use the function replace the condition check:

```
def arrangeCoins(self, n):
    def isValid(row):
        return (row*(row+1))//2 <= n
    # we need to find the last position that is valid (<=)
    def bisect_right():
        l, r = 1, n
        while l <= r:
            mid = l + (r-l) // 2
            if isValid(mid): # replaced compared with the
    standard binary search
                l = mid + 1
            else:
                r = mid - 1
        return l-1
    return bisect_right()
```

0.2 **278. First Bad Version.** You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which will return whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Solution: we keep doing binary search till we searched all possible area.

```
class Solution(object):
    def firstBadVersion(self, n):
        """
        :type n: int
        :rtype: int
        """
        l,r=0,n-1
        last = -1
        while l<=r:
            mid = l+(r-l)//2
            if isBadVersion(mid+1): #move to the left, mid is index, s
                r=mid-1
                last = mid+1 #to track the last bad one
            else:
                l=mid-1
        return last
```

0.1.4 LeetCode Problems

0.1 **35. Search Insert Position (easy).** Given a sorted array 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 may assume no duplicates in the array.

```
Example 1:

Input: [1,3,5,6], 5
Output: 2

Example 2:
Input: [1,3,5,6], 2
Output: 1

Example 3:
Input: [1,3,5,6], 7
Output: 4
```

```
Example 4:
Input: [1,3,5,6], 0
Output: 0
```

Solution: Standard Binary Search Implementation. For this problem, we just standardalize the python coding of binary search, which takes $O(logn)$ time complexity and O(1) space complexity without using recursion function. In the following code, we use exclusive right index with len(nums), therefore it stops if l == r; it can be as small as 0 or as large as n of the array length for numbers that are either smaller or equal to the nums[0] or larger or equal to nums[-1]. We can also make the right index inclusive.

```
# exclusive version
def searchInsert(self, nums, target):
    l, r = 0, len(nums) #start from 0, end to the len (exclusive)
    while l < r:
        mid = (l+r)//2
        if nums[mid] < target: #move to the right side
            l = mid+1
        elif nums[mid] > target: #move to the left side, not mid-1
            r= mid
        else: #found the traget
            return mid
    #where the position should go
    return l
```

```
# inclusive version
def searchInsert(self, nums, target):
   l = 0
    r = len(nums)-1
    while l <= r:
        m = (l+r)//2
        if target > nums[m]: #search the right half
            l = m+1
        elif target < nums[m]: # search for the left half
            r = m-1
        else:
            return m
    return l
```

Standard binary search

1. 611. Valid Triangle Number (medium)
2. 704. Binary Search (easy)
3. 74. Search a 2D Matrix) Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

(a) Integers in each row are sorted from left to right.

(b) The first integer of each row is greater than the last integer of the previous row.

```
For example,
Consider the following matrix:

[
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]

Given target = 3, return true.
```

Solution: 2D matrix search, time complexity from $O(n^2)$ to $O(lgm + lgn)$.

```
def searchMatrix(self, matrix, target):
        """
        :type matrix: List[List[int]]
        :type target: int
        :rtype: bool
        """

        if not matrix:
            return False
        row, col = len(matrix), len(matrix[0])
        if row==0 or col==0: #for [[]]
            return False
        sr, er = 0, row-1
        #fisrst search the mid row
        while sr<=er:
            mid = sr+(er-sr)//2
            if target>matrix[mid][-1]: #go to the right
    side
                sr=mid+1
            elif target < matrix[mid][0]: #go the the left
    side
                er = mid-1
            else: #value might be in this row
                #search in this row
                lc, rc = 0, col-1
                while lc<=rc:
                    midc = lc+(rc-lc)//2
                    if matrix[mid][midc]==target:
                        return True
                    elif target<matrix[mid][midc]: #go to
    left
                        rc=midc-1
                    else:
                        lc=midc+1
                return False
        return False
```

Also, we can treat is as one dimensional, and the time complexity is $O(lg(m*n))$, which is the same as $O(log(m)+log(n))$.

```
class Solution:
    def searchMatrix(self, matrix, target):
        if not matrix or target is None:
            return False

        rows, cols = len(matrix), len(matrix[0])
        low, high = 0, rows * cols - 1

        while low <= high:
            mid = (low + high) / 2
            num = matrix[mid / cols][mid % cols]

            if num == target:
                return True
            elif num < target:
                low = mid + 1
            else:
                high = mid - 1

        return False
```

Check `http://www.cnblogs.com/grandyang/p/6854825.html` to get more examples.

Search on rotated and 2d matrix:

1. 81. Search in Rotated Sorted Array II (medium)
2. 153. Find Minimum in Rotated Sorted Array (medium) The key here is to compare the mid with left side, if mid-1 has a larger value, then that is the minimum
3. 154. Find Minimum in Rotated Sorted Array II (hard)

Search on Result Space:

1. 367. Valid Perfect Square (easy) (standard search)
2. 363. Max Sum of Rectangle No Larger Than K (hard)
3. 354. Russian Doll Envelopes (hard)
4. 69. Sqrt(x) (easy)