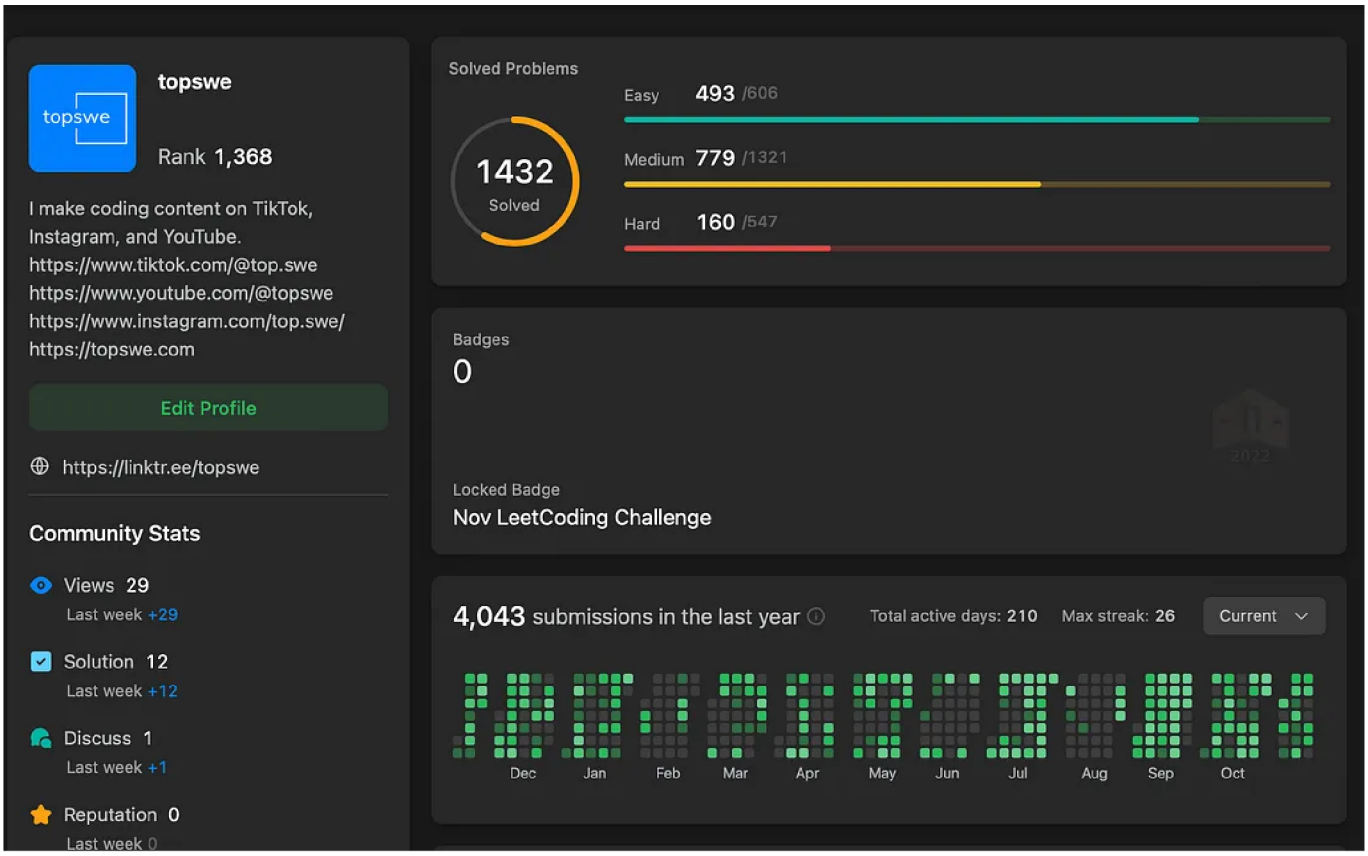
Leetcode Killer: Pattern 1 “Binary Search”

Top SWE · Follow
12 min read · Nov 19, 2022

Covers 10 Problems in 10 Pages in 10 minutes — learn the 1 correct pattern to solve all 200+ binary search problems once and for all.



I have solved over 1400 problems on Leetcode in a year! I hope this helps you finalize your understanding of binary search.

According to “Programming Pearls” by Jon Bentley, only 10% of professional software engineers can write a correct binary search — one of the fundamental algorithms you first learn. Common mistakes include integer overflow, off by 1 errors, and infinite loops. (time limit exceeded) Now is the time to finally master this method once and for all.

Context:

Consider the problem of finding a target number in a sorted array.

A linear scan takes O(n), however this is wasteful. Instead consider initializing a range of [left, right] = [0, len(arr)-1]. The middle number is simply (left+right)//2. If it is >= our target we can recurse on the left half (including this middle number, so right = mid, hence range is now [left,

mid]) Otherwise, it is less than our target so we know the target cannot be in the left half of the array, so we can recurse solely on the right half. (excluding this middle number, so left = mid+1, hence range is now [mid+1, right]) We recurse until left == right, meaning left is the index of target. This takes O(logn) time — a significant improvement.

Intuitively you want to imagine your range [left,right] shrinking by a factor of two each iteration, until left == right meaning your range has size exactly 1, and so this last value must be what you are looking for.

Let’s go through a sample walkthrough:

[1,2,3,4,5], target = 4.

1. Left = 0, right = 4. Mid = 2. Arr[mid] < target, so we recurse on the right half by setting left = mid+1.
2. Left = 3, right = 4. Mid = 3. Arr[mid] >= target, so we recurse on the left half by setting right = mid
3. Left = 3, right = 3. Left == right so we stop the iteration. Index 3 is our answer, and indeed arr[3] = 4.

Although most commonly binary search is used to search over arrays, note that as long as our boolean “check” function (how we are deciding whether to recurse left or recurse right. Ex. above was arr[mid] >= target) is MONOTONIC on our range[left,right], binary search can be applied. (Monotonic boolean function means of the form of all Falses then all Trues or vice versa)

2 templates for binary search:

1. Searching for the minimum in a range [0,n].

Visualization:

FFF…FFFTTTT…TTTT [function range] (monotonically increasing)

0123… k … n [function domain]

We want the first T after all the F’s. (index k)

2. Searching for the maximum in a range [0,n].

Visualization:

TTT…TTTFFFF…FFFF [function range] (monotonically decreasing)

0123… k … n [function domain]

We want the last T before all the F’s. (index k)

Code template for 1 and 2

```
def binarySearchMin(...):
   left, right = x, y
   while left < right:
       mid = left + (right-left)//2
       if check(mid):
           right = mid
       else:
           left = mid+1
   return left

def binarySearchMax(...):
```

```
    left, right = x, y
    while left < right:
        mid = ceil(left + (right-left)/2)
        if check(mid):
            left = mid
        else:
            right = mid-1
    return left
```

Time: O(log(right-left)) [this is O(log(range_size))]

Space: O(1)

Note that binarySearchMin is roughly equivalent to bisect_left, and binarySearchMax to bisect_right… (library functions in python)

Let’s go over a few notes.

1. You will need to initialize left, right correctly to the exact range of values you want to search over. [left,right] = the SOLUTION space. For arrays, often left = 0, right = len(arr)-1. (the min/max indices)
2. Why mid = left + (right-left)//2 instead of mid = (left+right)//2 ? This handles overflow, as it is possible for left+right to be > INT_MAX.
3. Why are the mid values slightly different in Max vs Min? (solves INFINITE LOOP) Note that when we set left = mid, it is possible for the while loop to never terminate. Consider [1,2], we are using the max template with mid = (left+right)//2, and left = 0, right = 1. If check(0) is true, we will keep setting left = mid = 0, and so our range [0,1] will never decrease to size 1. Hence we need to take the ceiling (of the float interpreted mid) to round up so we can set left = mid = 1, and terminate.

A quick sanity check is if you have : mid = left + (right-left)//2, AND left = mid => you will certainly loop forever.

4. Why return left? After the while loop, left == right so we can return either left or right.
5. Why are we setting left and right to different values in min vs max?
6. Think about min case: …FFFTTT… If check(mid) is true, and we want to find the minimum true, so mid might be the answer but we should still check to the left in case there are smaller trues. Hence right = mid. Otherwise check(mid) is false, so we should definitely search on the right side for the min true. Hence left = mid + 1.
7. Max case: …TTTFFF… If check(mid) is true, and we want to find the max true, mid may be the answer but we should still check to the right in case there are larger trues. Hence left = mid. Otherwise check(mid) is false, so we should definitely search to the left side for the max true. Hence right = mid-1.

One key is that now when solving binary search problems, you should be able to characterize any binary search problem in the **point form** description of:

1. Min or max template
2. Initialize range: Left = ?, right = ?
3. Check function: check(mid) = ?. Once we figure out what these 2 values are for the problem at hand, we can apply the template for a correct solution.
4. (perhaps some pre/post processing)

As well, given such a template you should immediately know what the code looks like.

Now let’s go over some questions that utilize this framework.

Question 0: (the array example as above) 704. Binary Search

Problem: Given a non-empty array of integers sorted non-decreasing and a target, return the index of the target in the array. If target is not present in the array, return -1. Note that target will occur at most once in the array.

Ex 1. [1,2,3], target = 2, return 1.

Ex 2. [1,2,3], target = 4, return -1.

In the array example we could use either the Min or Max templates.

```
def searchMin(self, nums: List[int], target: int) -> int: #MIN
    left, right = 0, len(nums)-1
    while left < right:
        mid = left + (right-left)//2
        if nums[mid] >= target:
            right = mid
        else:
            left = mid+1
    return left if nums[left] == target else -1

def searchMax(self, nums: List[int], target: int) -> int: #MAX
    left, right = 0, len(nums)-1
    while left < right:
        mid = ceil(left + (right-left)/2)
        if nums[mid] <= target:
            left = mid
        else:
```

```
            right = mid-1
    return left if nums[left] == target else -1
```

Time: O(logn), n = len(nums). Space: O(1)

The only difference is the last line. It is possible for target to not exist in nums, hence we need to check at the end if our final value is indeed target. (Note: if target does not exist in nums, left will be the index that the value SHOULD be at if inserted into the sorted array)

Also note that in the MIN template, we have to update our check function. (from <= to >=) Always take a second to verify that the logic makes sense, using the template as your starting point..

Note that if there are multiple occurrences of target, using the Max template will give us the RIGHTMOST index of target. And using the Min template will give us the LEFTMOST index of target. This is important for the next question.

Note that now with the templates, we can easily describe the solution in point form.

Key: with this point form you should be able to immediately tell what the code is!

Ex. SearchMin:

1. Min template
2. Left = 0, right = len(nums)-1

3. Check() = nums[mid] >= target
4. Existence checking after the binary search

Ex. SearchMax:

1. Max template
2. Left = 0, right = len(nums)-1
3. Check() = nums[mid] <= target
4. Existence checking after the binary search

Question 1: 34. Find First and Last position of element in sorted array

Problem: Given an array of integers sorted non-decreasing and a target, return the first and last index of the target in the array. If target is not present in the array, return [-1, -1]

Ex 1. [1,2,3], target = 2. Return [1,1]

Ex 2. [1,2,2,3], target = 2. Return [1,2]

Ex. 3. [1,2,3] target = 4. Return [-1,-1]

```
def searchRange(self, nums: List[int], target: int) -> List[int]:
   if not nums:
       return [-1,-1]
   return [searchMin(), searchMax()]
```

Time: O(logn), n = len(nums). Space: O(1)

Notice that by re-using the same code as in Question 0, we can easily solve this problem. We just have to be careful that in this question nums can be empty, so we have an additional check. Edge cases are important to clarify and consider!

Question 2: 35. Search Insert Position

Problem: Given a non-empty array of integers sorted non-decreasing and a target, return the index of the target in the array. If target is not present in the array, return the index where it would be if inserted

Ex 1. [1,2,3], target = 2, return 1.

Ex 2. [1,2,3], target = 4, return 3.

In the array example we could use either the Min or Max templates. We choose the min.

```
def searchInsert(self, nums: List[int], target: int) -> int:
    left, right = 0, len(nums)
    while left < right:
        mid = left + (right-left)//2
        if nums[mid] >= target:
            right = mid
        else:
            left = mid + 1
    return left
```

Time: O(logn), n = len(nums). Space: O(1)

1. Min template
2. Left = 0, right = len(nums)
3. Check() = nums[mid] >= target

Note that we do not need the if post-processing check compared to Question 0.

Peak Elements (Q3–4)

Question 3 and 4: 162. Find Peak Element & 852. Peak Index in a Mountain Array

Problem: Given an array of integers nums, return the index of the element that is strictly greater than its neighbors. Assume nums[-1] = nums[len(nums)] = float(‘-inf’)

Ex. 1 [3,2,1] return 0

Ex. 2 [1,2,1] return 1

```
def findPeakElement(self, nums: List[int]) -> int:
    left, right = 0, len(nums)-1
    while left < right:
        mid = ceil(left + (right-left)/2)
        if (nums[mid-1] if mid-1 >= 0 else float('-inf')) <= nums[mid]:
            left = mid
        else:
            right = mid-1
    return left
```

Time: O(logn), n = len(nums). Space: O(1)

1. Max template
2. Left = 0, right = len(nums)-1
3. Check(mid) = **(nums[mid-1] if mid-1 >= 0 else float(‘-inf’)) <= nums[mid]**

Exercise: Solve using the Min template!

General Binary Search

We now consider applying the binary search templates to problems that are not on arrays. Often, we can binary search over the ‘**solution space**’! This may be a bit unfamiliar, but remember the binary search template applies to any monotone function!

Question 5: 278. First Bad Version

Problem: You have n versions of a product [1,2,…,n] and you want to find the first bad version, which causes all subsequent versions to be bad. You are given a function isBadVersion(int version_number) that returns True if and only if the version is bad.

Notice that this is exactly the Min template. (the function is monotone of the form: …FFFTTT…)

```
def firstBadVersion(self, n: int) -> int:
    left, right = 1, n
    while left < right:
        mid = left + (right-left)//2
```

```
            if isBadVersion(mid):
                right = mid
            else:
                left = mid + 1
        return left
```

Time: O(logn). Space: O(1)

1. Min template
2. Left = 1, right = n
3. Check(mid) = isBadVersion(mid)

Question 6: 69. Sqrt(x)

Problem: Given a non-negative integer x, return the square root of x rounded down to the nearest integer.

Ex. 1. 4 return 2

Ex. 2. 8 return 2

```
    def mySqrt(self, x: int) -> int:
        left, right = 0, x
        while left < right:
            mid = ceil(left + (right-left)/2)
            if mid * mid <= x:
                left = mid
            else:
                right = mid-1
        return left
```

Time: O(logx). Space: O(1)

We know we can apply binary search because the function Check(mid) = mid * mid <= x is MONOTONE.

1. Max template
2. Left = 0, right = x
3. Check(mid) = mid * mid <= x.

Note: Applying the min template (check(mid) = mid * mid >= x) will give the square root of x rounded up to the nearest integer.

Question 7: 875. Koko Eating Bananas

Problem: Given an array of piles of bananas (piles) and a number of hours (h), return the **minimum** k bananas per hour that Koko the Gorilla can eat at such that she finishes eating all of the piles before h hours. Koko can only eat bananas one pile at a time, and If there are less than k bananas in a pile she will eat exactly those bananas in that hour.

(Assume the answer always exists. Ex. h >= len(piles))

Ex. [1,2], h = 2, return 2.

Ex. [3, 6, 7, 11], h = 8, return 4

```
def minEatingSpeed(self, piles: List[int], h: int) -> int:
    def check(mid):
        return sum(ceil(p / mid) for p in piles) <= h
```

```
        left, right = 1, max(piles)
        while left < right:
            mid = left + (right-left)//2
            if check(mid):
                right = mid
            else:
                left = mid + 1
        return left
```

Time: O(n*log(max(piles))), where n = len(piles). There are O(log(max(piles))) iterations of the while loop, and each check() call takes O(n) time. Space: O(1)

How do we know to apply binary search? 1. Answer being Minimum k or maximum k may suggest to try applying the min or max binary search templates. Check if the function is monotone. 2. The VERIFICATION problem is easy. Meaning, if we fix our answer k, verifying/’check’ing if it works or not is straightforward.

Both 1 and 2 apply for this question, since say we fix k = 4. We can then simulate the total number of hours it will take for Koko to eat all of the bananas. If this is <= h, then this speed k works! So we can recurse by setting right = k. (this k could be the minimum, but lets try smaller in case it’s not) Otherwise this speed k does not work and we set left = k+1.

1. Min template
2. Left = 1, right = max(piles) [think about why!]
3. Check(mid) = **sum(ceil(p / mid) for p in piles) <= h**

Question 8: 2226. Maximum Candies Allocated to K Children

Problem: Given an array of piles of candies and an integer k, you want to allocate k equally sized piles of candies to k children. You can divide any pile into any number of subpiles of candies. Return the **maximum** number of candies each child can receive. (max equal pile size)

Ex 1. [5,8,6], k = 3. Return 5. Divide 8 into (5,3) and divide 6 into (5,1). We have 3 piles of size 5. We don’t have to use the remaining pile of 3 candies and 1 candy.

Ex 2. [2,5], k = 11. Return 0.

```
def maximumCandies(self, candies: List[int], k: int) -> int:
     def check(mid):
         return sum(c // mid for c in candies) >= k
     left, right = 0, max(candies)
     while left < right:
         mid = ceil(left + (right-left)/2)
         if check(mid):
             left = mid
         else:
             right = mid-1
     return left
```

Time: O(n*log(max(candies))), where n = len(candies). Space: O(1)

Basically the same as above question 7, however note we use a max template since we want the maximum number of candies.

1. Max template
2. Left = 0, right = max(candies)
3. check(mid) = **sum(c // mid for c in candies) >= k**

Let’s finish this binary search series with a similar question

Question 9: 2187. Minimum Time to Complete Trips

Problem: Given an array ‘times’ where times[i] consists of the time taken by bus i to complete a trip, and an integer ‘totalTrips’, we want to return the **minimum** time for all buses to complete at least ‘totalTrips’ number of trips. Each bus makes trips successively (once finish a trip can immediately start the next trip), and operates independently of the other buses.

Input: time = [1,2,3], totalTrips = 5, Output: 3

Input: time = [2], totalTrips = 1, Output: 2

```
def minimumTime(self, time: List[int], totalTrips: int) -> int:
    def check(mid):
        return sum(mid // t for t in time) >= totalTrips
    left, right = 1, min(time) * totalTrips
    while left < right:
        mid = left + (right-left)//2
        if check(mid):
            right = mid
        else:
            left = mid + 1
    return left
```

O(nlog(min(time)*totalTrips)) time : O(1) space

1. Min template
2. Left = 1, right = min(time) * totalTrips (think about this!)
3. check(mid) = sum(mid // t for t in time) >= totalTrips

Thanks for reading to the end! Stay tuned for more Leetcode Killer ‘Pattern’ blogs!

 # Follow for more

Check out my TikTok/Youtube for more coding content and daily leetcode solutions!

https://www.tiktok.com/@top.swe

https://www.youtube.com/@topswe

https://www.instagram.com/top.swe/

https://topswe.com

https://linktr.ee/topswe