LEETCODE SOLUTIONS

Contents

List of Figures

List of Tables

This document summarizes solutions for LeetCode problems I have solved. Many of solution ideas come from LeetCode discussion forum. By coding and writing down the solution, it really help me understand the solution instead of memorizing it.

1
Problems by Category

1.1 Binary Search

1.1.1 Basics

- sqrt vs. square
  - LC69 - sqrt(x)
  - LC367 - Valid Perfect Square

1.1.2 Advanced

- Rotated sorted array
  - LC33 - Search in Rotated Sorted Array
  - LC81 - Search in Rotated Sorted Array II
  - LC153 - Find Minimum in Rotated Sorted Array
  - LC154 - Find Minimum in Rotated Sorted Array II
- Transform the problem
  - LC540 - Single Element in a Sorted Array
  - LC875 - Koko Eat Bananas

1.2 Array

1.2.1 Two Pointers

1.3 Linked List

2
Solutions

2.1 LC33 - Search in Rotated Sorted Array

2.1.1 Problem Description

LeetCode Problem 33: You are given an integer array nums sorted in ascending order (with **distinct** values), and an integer target.

Suppose that nums is rotated at some pivot unknown to you beforehand (i.e., $[0,1,2,4,5,6,7]$ might become $[4,5,6,7,0,1,2]$).

If target is found in the array return its index, otherwise, return -1.

2.1.2 Approach - Binary Search

The original array is sorted in ascending order and then rotated at some pivot. So the array can be always divided into two parts: one part is sorted and the other part is unsorted, containing the pivot. It is easy to check whether the target is in the sorted part and search inside that part. If the target is in the unsorted part, we can further divide the unsorted part into two parts (again one part will be sorted and the other will be unsorted) and continue to check the sorted part. This allow us using binary search to find the target.

The search space starts from the whole array and shrinks over each iteration, using [left, right] to indicate the search space. The mid is the separation point to divide the search space into two parts, one is sorted and the other is unsorted.

- If nums[left] $\leq$ nums[mid], the sub-array [left, mid] is sorted [1].
  It is easy to check whether the target is inside this part. If not, continue to divide and search the right half.

- If nums[left] $>$ nums[mid], the right sub-array [mid, right] is sorted.
  We can check whether the target is in the right part. If not, continue to divide and search the left half.

[1] This is true when an array contains distinct values, i.e., no duplicates.

```cpp
int search(vector<int>& nums, int target) {
    if (nums.empty()) return -1;

    int left = 0;
    int right = nums.size() - 1;
    int mid;

    while (left <= right) {
        mid = left + (right - left)/2;

        if (target == nums[mid]) {
                return mid;
            }
        else if (nums[left] <= nums[mid]) {
            // [left, mid] is sorted
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1;
            }
            else {
                left = mid + 1;
            }
        }
        else {
            // [mid, right] is sorted
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;
            }
            else {
                right = mid - 1;
            }
        }
    }

    return -1;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(\log n)$
  Since using binary search, the time complexity is $\mathcal{O}(\log n)$.

- **Space complexity**: $\mathcal{O}(1)$
  Only use limited variables for binary search.

2.2 *LC34 - Find First and Last Position of Element in Sorted Array*

2.3 *LC35 - Search Insert Position*

2.3.1 *Problem Description*

LeetCode Problem 35: 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. The array contains **distinct** values sorted in ascending order.

2.3.2 *Approach - Binary Search*

Since the array is sorted, the problem can be solved by using Binary Search. The key part is to understand what is searching for. Here is the insert position (not the target). [2] The Search range is the range of array, $[0, n-1]$.

[2] @Zhengguan Li gives some good explanations.

- If `nums[mid] == target`, return mid
- If `target > nums[mid]`, mid is not the potential insert position while `mid + 1` is. so `left = mid + 1`.
- If `target < nums[mid]`, mid is the potential insert position and `right = mid`;

To determine the while loop condition, we can use two-element case to test our left/right operations: 1) `left = mid + 1` and 2) `right = mid`. We can see that it can be safely reduced to one element but not zero. So we need end the while loop when there is only one element left, i.e., `while(left < right)`.

```
index:       [0,     1]
2 elements:  [5,     7]
              l/m    r
1 element:   l/m/r
or:                 l/m/r
```

After while loop, we need to check the remaining one element: `nums[left]` with `left == right`, which has not been checked in binary search loop.

```
int searchInsert(vector<int>& nums, int target) {
    if (nums.empty()) {
        return -1;
    }

    int left = 0;
    int right = nums.size() - 1;
    int mid;

    while (left < right) {
        mid = left + (right - left)/2;

        if (target == nums[mid]) {
            return mid;
        }
        else if (target > nums[mid]) {
            left = mid + 1;  // mid is not the insert position
    but mid + 1 can be the potential insert position.
        }
        else {
            right = mid; // mid can be the insert position
```

```
        }
    }

    // 1 element left at the end
    // post-processing
    return nums[left] < target ? left + 1 : left;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(\log n)$
  Since using binary search, the time complexity is $\mathcal{O}(\log n)$.
- **Space complexity**: $\mathcal{O}(1)$
  Only use limited variables for binary search.

2.4 LC69 - Sqrt(x)

2.4.1 Problem Description

LeetCode Problem 69: Given a non-negative integer x, compute and return the square root of x. Since the return type is an integer, the decimal digits are **truncated**, and only **the integer part** of the result is returned.

2.4.2 Approach - Binary Search

Mathematically, $i = sqrt(x)$ can be viewed as $x = i * i$. The problem is to find an integer i in the search space $[1, x]$ [3] such that $i * i <= x$ and $(i+1) * (i+1) > x$. If $i * i > x$, we can exclude values larger than i. If $i * i < x$, we can exclude the value smaller than i. With this property, we can solve this problem using binary search. There are three potential issues:

[3] The search space could be $[1, x/2]$ for $x >= 4$. Yet, it just saves one iteration if using binary search.

- corner case: $x <= 1$;
- overflow caused by $i * i$ in the evaluation $i * i > x$;
- can not simply check $i == x/i$ to determine whether the answer is found since there may be multiple values satisfied this equations due to integer division.

```
int mySqrt(int x) {
    if (x <= 1) {
        return x;
    }

    int left = 1;
    int right = x;
    int mid;
```

```
    while (left <= right) {
        mid = left + (right - left)/2;

        if ((mid <= x/mid) && ((mid + 1) > x/(mid + 1))) {
            return mid;
        }
        else if (mid > x/mid) {
            right = mid - 1;
        }
        else {
            left = mid + 1;
        }
    }
    return -1;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(\log n)$
  Since using binary search, the time complexity is $\mathcal{O}(\log n)$.

- **Space complexity**: $\mathcal{O}(1)$
  Only use limited constant variables for binary search.

2.5 *LC70 - Climbing Stairs*

2.5.1 *Problem Description*

LeetCode Problem 70: You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

2.5.2 *Approach - Dynamic Programming*

Use dynamic programming to solve this problem. One can reach i-th step in one of two ways:

1. Taking a step of 1 from $(i-1)$th step.

2. Taking a step of 2 from $(i-2)$th step.

The total number of distinct ways to reach ith step is the sum of ways of reaching $(i-1)$th step and ways of reaching $(i-2)$th step. Let $f(i)$ denotes the number of distinct ways to reach ith step, then we have $f(i) = f(i-1) + f(i-2)$. This becomes a problem of finding ith number of the Fibonacci series with base cases $f(1) = 1$ and $f(2) = 2$. Check LC509 Fibonacci Number for detailed solutions.

2.6 LC81 - Search in Rotated Sorted Array II

2.6.1 Problem Description

LeetCode Problem 81: You are given an integer array nums sorted in ascending order (not necessarily distinct values), and an integer target.

Suppose that nums is rotated at some pivot unknown to you beforehand (i.e., $[0,1,2,4,4,4,5,6,6,7]$ might become $[4,5,6,6,7,0,1,2,4,4]$).

If target is found in the array return true, otherwise, return false.

2.6.2 Approach - Binary Search

This is a follow-up problem to LC33 - Search in Rotated Sorted Array. The difference is that it contains duplicates. Due to duplicates, when nums[left] == nums[mid], we don't know whether which part (left or right) is sorted. For example, $[3,1,3,3,3,3,3]$ with left part unsorted and $[3,3,3,3,3,1,3]$ with right part unsorted. For this case, if target != nums[mid], we can increase left by one, i.e., left++.

```
bool search(vector<int>& nums, int target) {
    if (nums.empty()) return false;

    int left = 0;
    int right = nums.size() - 1;
    int mid;

    while (left <= right) {
        mid = left + (right - left)/2;

        if (target == nums[mid]) {
            return true;
        }
        else if (nums[left] < nums[mid]) {
            // [left, mid] is sorted
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1;
            }
            else {
                left = mid + 1;
            }
        }
        else if (nums[left] == nums[mid]) { // handle corner
    case with duplicates
            left++;
        }
        else {
            // [mid, right] is sorted
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;
            }
            else {
                right = mid - 1;
            }
        }
    }
```

```
    return false;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(\log n)$ for the average case and $\mathcal{O}(n)$ for the worse case
  In the worst case, just move left pointer by one, which needs n steps. So the time complexity is $\mathcal{O}(n)$. For the average case, since using binary search, the time complexity is $\mathcal{O}(\log n)$.
- **Space complexity**: $\mathcal{O}(1)$
  Only use limited variables for indices.

2.7 *LC88 - Merge Sorted Array*

2.7.1 *Problem Description*

LeetCode Problem 88: Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

The number of elements initialized in nums1 and nums2 are m and n respectively. You may assume that nums1 has enough space (size that is equal to $m + n$) to hold additional elements from nums2.

2.7.2 *Approach - Two Pointers*

The problem is similar to the merge function of merge sort. The differences are: 1) no need to create a new auxiliary array to store the combined sorted array; 2) move pointers from right to left. For this type of problem, we can use two pointers for each array and move the pointers based on the comparison result between two arrays. Moreover, we need to consider the pointer out of bound cases since the array size may be different.

```
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n)
    {
    int i = m - 1; // pointer for nums 1, scan from right to
    left
    int j = n - 1; // pointer for nums 2, scan from right to
    left
    int k = m + n - 1; // poionter for temp array

    //    0  1  2  3  4  5
    //   [1, 2, 2, 3, 5, 6]
    //    i/k
    //   [2, 5, 6]
    //  j

    while (k >= 0) {
        if (i < 0) {
```

```
            nums1[k--] = nums2[j--];
        }
        else if (j < 0) {
            nums1[k--] = nums1[i--];
        }
        else if (nums1[i] >= nums2[j]) {
            nums1[k--] = nums1[i--];
        }
        else {
            nums1[k--] = nums2[j--];
        }
    }
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(m+n)$
  Since it will scan all elements in nums1 and nums2, the time complexity is the total size of two arrays, i.e., $\mathcal{O}(m+n)$.
- **Space complexity**: $\mathcal{O}(1)$
  Only use limited variables for indices.

2.8 *LC153 - Find Minimum in Rotated Sorted Array*

2.8.1 *Problem Description*

LeetCode Problem 153: Suppose an array of length n sorted in ascending order is rotated between 1 and n times. For example, the array nums $= [0,1,2,4,5,6,7]$ might become: $[4,5,6,7,0,1,2]$ if it was rotated 4 times, or $[0,1,2,4,5,6,7]$ if it was rotated 7 times. Notice that rotating an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]]. Given the sorted rotated array nums, return the minimum element of this array. All the integers of nums are **unique**.

2.8.2 *Approach - Binary Search*

Since the array is originally sorted and rotated between 1 and n times, there will be two potential forms after rotations:

- The whole array is sorted in the ascending order (i.e., the array rotates back to the original one);
- There is a pivot point in the array, which separate the array into two halves: one half is sorted and the other is not sorted.

So if we can detect which half is not sorted, we know that the minimum value should be in that unsorted half and ignore the sorted half. If both halves are sorted, the first left element is the minimum.

If we split the array into two halves, $[\text{left}, \text{mid}]$ and $[\text{mid}+1, \text{right}]$, there are 4 kings of relationship among nums[left], nums[mid], and nums[right]:

1. nums[left] $\leq$ nums[mid] $\leq$ nums[right]
   min is nums[left]

2. nums[left] $>$ nums[mid] $\leq$ nums[right]
   [left, mid] is not sorted and min is inside the left half

3. nums[left] <= nums[mid] $>$ nums[right]
   [mid+1, right] is not sorted and min is inside the right half

4. nums[left] $>$ nums[mid] $>$ nums[right]
   Impossible, since the original array is sorted in ascending order

So we can check nums[mid] and nums[right] [4]:

- If nums[mid] $>$ nums[right], search the right half
  This covers relationship 3.

- If nums[mid] $\leq$ nums[right], search the left half
  This covers relationship 1 and 2, since in both cases the minimum is on the left.

[4] If we just check nums[left] and nums[mid], the condition nums[left] <= nums[mid] can't distinguish relationship 1 and 3 which requires searching two different directions. This requires special consideration on 1) the whole array is sorted or 2) not falling into the subset subset of an array that is sorted when update left and right indices (e.g., $[0,1,2]$ of $[4,5,6,7,0,1,2]$).

```
int findMin(vector<int>& nums) {
    int left = 0;
    int right = nums.size() - 1;
    int mid;

    while (left < right) {
        mid = left + (right - left)/2;

        if (nums[mid] > nums[right]) {
            left = mid + 1;
        }
        else {
            right = mid;
        }
    }

    // Post-processing
    return nums[left];
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(\log n)$
  Since using binary search, the time complexity is $\mathcal{O}(\log n)$.

- **Space complexity**: $\mathcal{O}(1)$
  Only use limited constant variables for binary search.

2.9 LC154 - Find Minimum in Rotated Sorted Array II

2.9.1 Problem Description

LeetCode Problem 154: Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand (e.g., $[0,1,2,4,5,6,7]$ might become $[4,5,6,7,0,1,2]$). Find the minimum element. The array may **contain duplicates**.

This is a follow up problem to LC153 - Find Minimum in Rotated Sorted Array.

2.9.2 Approach - Binary Search

The big difference between this problem and LC153 is that **the array may contain duplicates**. So when nums[mid] == nums[right], the position of minimum could be in either left or right of mid (e.g., $[3,3,3,3,1,3]$, or $[3,1,3,3,3,3]$). So

- For relationship 1, nums[left] $\leq$ nums[mid] $\leq$ nums[right], the min is not necessariely on the left and could be anywhere when nums[left] == nums[mid] == nums[right].
- For relationship 2, nums[left] $>$ nums[mid] $\leq$ nums[right], when nums[mid] == nums[right], min could be in either left or right of mid.

```
int findMin(vector<int>& nums) {
    int left = 0;
    int right = nums.size() - 1;
    int mid;
    //  0  1  2  3  4
    // [2, 0, 1, 1, 1]
    //  l     m     r

    while (left < right) {
        mid = left + (right - left)/2;

        if (nums[mid] > nums[right]) {
            left = mid + 1;
        }
        else if (nums[mid] < nums[right]) {
            right = mid;
        }
        else { // nums[mid] == nums[right], main difference
            right--;
        }
    }

    return nums[left];
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$ (worst case) and $\mathcal{O}(\log n)$ (average case)
  In the worst case, it searches one element at a time and takes $\mathcal{O}(n)$ time. For the average case, the time complexity is $\mathcal{O}(\log n)$ with the binary search.

- **Space complexity**: $\mathcal{O}(1)$
  Only use limited variables.

2.10 LC240 - Search a 2D Matrix II

2.11 LC275 - H-Index II

2.11.1 Problem Description

LeetCode Problem 275: Given an array of citations **sorted in ascending order** (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.

According to the definition of h-index on Wikipedia: "A scientist has index h if h of his/her N papers have at least h citations each, and the other $N - h$ papers have no more than h citations each."

If there are several possible values for h, the maximum one is taken as the h-index.

2.11.2 Approach - Binary Search

Based on the definition on Wikipedia, for the citation sorted in **descending** order, H-Index is to find the **last** position where citation is greater than or equal to the position. In this problem, the citation array is sorted in **ascending** order. H-Index is to find the first position/index where

$$\text{citations[index]} >= n - \text{index} \tag{2.1}$$

Where n is the total number of papers (i.e., length of citation array). Note that we may not find the exact solution. So the above equation changes from $\geq$ to $>$. There are two important properties associated with H-Index definitions:

- If index i satisfies Eq. 2.1, then any index j that is larger than index i will also satisfy the equation.
  We know $n - i > n - j$ due to $j > i$. Since the citation is in ascending order, we have $\text{citations}[j] >= \text{citations}[i]$. Together with Eq. 2.1, we can obtain

  $$\text{citations}[j] >= \text{citations}[i] >= n - i > n - j$$

  which satisfies Eq. 2.1.

- If we find the exact solution (i.e., citations$[i] == n - i$), that's the only solution.

  - For $j > i$, based on the previous property, we know citations$[j] > n - j$, which is $>$ not $==$
  - For $j < i$, we can obtain citations$[j] <=$ citations$[i] == n - i < n - j$, which is $<$ not $==$

With above mentioned properties, we can use binary search to reach search space each iteration.

```
int hIndex(vector<int>& citations) {
    if (citations.empty()) return 0;

    int n = citations.size();
    int left = 0;
    int right = n - 1;
    int mid;
    int value;

    // [x, y]
    // l/m r
    while (left < right) {
        mid = left + (right - left)/2;
        value = citations[mid];

        if (value == n - mid) {
            // find the solution
            return n - mid;
        }
        else if (value > n - mid) {
            right = mid; // mid qualified as h-index and
    continue to search left for a higher one; no mid -1 since
    exact solution may not exist
        }
        else {
            left = mid + 1;
        }
    }

    // post-processing: left == right
    return (citations[right] >= n - right) ? (n - right) : 0;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(\log n)$
  The time complexity is $\mathcal{O}(\log n)$ with the binary search.

- **Space complexity**: $\mathcal{O}(1)$
  Only use limited variables.

2.12 LC278 - First Bad Version

2.12.1 Problem Description

LeetCode Problem 278: 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 returns whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

2.12.2 Approach - Binary Search

Each version has its corresponding status:

$$\begin{aligned} \text{versions} &: \begin{bmatrix} 1 & 2 & \cdots & i & i+1 & \cdots & n \end{bmatrix} \\ \text{status} &: \begin{bmatrix} G & G & \cdots & G & B & \cdots & B \end{bmatrix} \end{aligned}$$

The problem can be solved using binary search to find the first bad status. Similar to problem LC34 Find First and Last Position of Element in Sorted Array.

2.13 LC287 - Find the Duplicate Number

2.13.1 Problem Description

LeetCode Problem 287: Given an array of integers nums containing $n+1$ integers where each integer is in the range $[1, n]$ inclusive. There is only **one duplicate number** in nums, return this duplicate number.
Constraints:

- $2 \leq n \leq 3 * 10^4$
- nums.length $== n + 1$
- $1 \leq nums[i] \leq n$
- All the integers in nums appear only **once** except for **precisely one integer** which appears two or more times.

Follow-ups:

- How can we prove that at least one duplicate number must exist in nums?

Based on pigeonhole principle [5], at least one duplicate number must exist in nums. We can also use contradiction method.

- Can you solve the problem **without** modifying the array nums?
  In approach 2, the sort method requires modifying the array nums. Simply fix is to copy array nums, which will require $\mathcal{O}(n)$ space.
- Can you solve the problem using only constant, $\mathcal{O}(1)$ extra space?
- Can you solve the problem with runtime complexity less than $\mathcal{O}(n^2)$?
- **My follow-up:** Can you solve the problem with more than one duplicate number?
  Only approach 3 using set can work.
- **My follow-up:** what if there is a missing number?
  Approach 2 using sort and binary search won't work.

[5] Pigeonhole principle: if n items are put into m containers, with $n > m$, then at least one container must contain more than one item. Note that it **doesn't** tell you there **isn't** a duplicate if there aren't too many items, $n <= m$.

2.13.2 *Approach 1 - Binary Search*

We know that the search space of the duplicate number is between 1 to n (inclusive, $[1, n]$). For a given number x in the search space $[1, n]$, go through array nums and count all the numbers in the range $[1, x]$ which is denoted as count. If $count > x$, then there are more than x elements in the range $[1, x]$ and thus that range contains a duplicate based on pigeonhole principle. If $count <= x$, then there are $n + 1 - count$ elements (array size is $n + 1$) in the range $[x + 1, n]$. That is, **at least** $n + 1 - x$ elements in a range of size $n - x$, where $n + 1 - count \geq n + 1 - x > n - x$. Thus this range must contains a duplicate. [6] With this property, we can use binary search idea to reduce search space by half after each iteration until search space is only one number.

[6] Simple explanation: the whole range is "too crowded" and so either the first or the second half of the range is too crowded. If the first half is too crowded, then it contains a duplicate. Otherwise, the second half contains a duplicate.

```
int findDuplicate(vector<int>& nums) {
    int left = 1;
    int right = nums.size() - 1;
    int mid;
    int count = 0;

    while (left < right) {
        mid = left + (right - left)/2;
        count = 0;
        for(int num:nums) {
            count += num <= mid;
        }

        count > mid ? right = mid : left = mid + 1;
    }

    return left;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n \log n)$
  Since using binary search, it takes $\mathcal{O}(\log n)$ steps. For each iteration, it will go through the whole array to count the number, which takes $\mathcal{O}(n)$ time. So the time complexity is $\mathcal{O}(n \log n)$.
- **Space complexity**: $\mathcal{O}(1)$
  Only use limited variables for binary search and count.

2.13.3 *Approach 2 - Sort & Search*

If the array is sorted, we can do the search on the sorted array. The sorted array has the following properties:

- Any duplicate numbers will be adjacent in the sorted array
  Simply compare each element to its previous element. If they are equal, return the element.
- **If no missing number** in the range $[1, n]$, the difference between the element value and its corresponding index will be reduced by 1 when encountering the duplicate.
  The difference is $[x, x, \cdots, x, x-1, \cdots, x-1]$. We can use the binary search to find the last element of x. **Note** the substraction is only conducted on the mid indices not all elements in the array.

```
int findDuplicate(vector<int>& nums) {
    // sort array
    sort(nums.begin(), nums.end());

    // option 1: compare previous element
    // for (vector<int>::size_type i = 1; i != nums.size(); i++)
     {
    //     if (nums[i] == nums[i-1]) {
    //         return nums[i];
    //     }
    // }

    // option 2: binary search
    int left = 0;
    int right = nums.size() - 1;
    int mid;
    int diff;
    int target = 0 - nums[0]; // target is the difference
    bewteen first value and its index

    // Use binary search to find the last target in a ascending
    order
    // [e, e, ..., e, s, s, ...]
    while (left < right - 1) {
        mid = left + (right - left)/2;
        diff = mid - nums[mid];

        if (target == diff) {
            left = mid;
        }
        else if (target > diff) {
```

```
            left = mid + 1;
        }
        else {
            right = mid - 1; // not mid - 1, since it may
    accidently exclude the final solution
        }
    }

    // two elements left after while loop left == right - 1
    return nums[right];
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n \log n)$
  There are two steps:the first step using sort algorithm takes $\mathcal{O}(n \log n)$; The second step takes $\mathcal{O}(n)$ if comparing with previous element or $\mathcal{O}(\log n)$ if using binary search. So the total time complexity is $\mathcal{O}(n \log n)$, $\mathcal{O}(n \log n) = \mathcal{O}(n \log n) + \mathcal{O}(n)$ or $\mathcal{O}(n \log n) = \mathcal{O}(n \log n) + \mathcal{O}(\log n)$.

- **Space complexity**: $\mathcal{O}(1)$ (or $\mathcal{O}(n)$)
  The sort is in-place and only use limited variables for search after sorting. So the space complexity is $\mathcal{O}(1)$. If we cannot modify the input array, then we must allocate linear space $\mathcal{O}(n)$ for a copy of array and sort that instead.

2.13.4 *Approach 3 - Set*

Store unique element when iterating over the array and check whether element is already stored. We can use unordered_set container in C++ standard library, which on average has constant time complexity on inserting and looking up an element.

```
int findDuplicate(vector<int>& nums) {
    unordered_set<int> seen;

    for (int num:nums) {
        if (seen.count(num)) {
            return num;
        }
        else
        {
            seen.insert(num);
        }
    }

    return -1;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$ for average case, $\mathcal{O}(n^2)$ for worst case
  It takes $\mathcal{O}(n)$ to go through each element in the array. For inserting or looking up a element in a unordered set, it takes $\mathcal{O}(1)$ for average case or takes $\mathcal{O}(n)$ for the worst case. So the total time complexity is $\mathcal{O}(n)$ for average case and $\mathcal{O}(n^2)$ for worst case,
- **Space complexity**: $\mathcal{O}(n)$
  In the worst case, it needs $\mathcal{O}(n)$ space to store n unique elements.

2.13.5 *Comparison of Different Approaches*

The table below summarizes the time complexity and space complexity of different approaches:

Approach	Time Complexity	Space Complexity
Approach 1 - Binary Search	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$
Approach 2 - Sort & Search	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$ or $\mathcal{O}(n)$
Approach 2 - Set	$\mathcal{O}(n)$ or $\mathcal{O}(n^2)$	$\mathcal{O}(n)$

2.14 *LC367 - Valid Perfect Square*

2.14.1 *Problem Description*

LeetCode Problem 367: Given a positive integer num, write a function which returns True if num is a perfect square else False.

Follow up: Do not use any built-in library function such as sqrt.

2.14.2 *Approach - Binary*

The problem can be transformed to find an integer i in the search space $[1, num]$ such that $i * i ==$ num [7].

- If $i * i >$ num, no need to search right (i.e., $i + 1$, $i + 2$, $\cdots$), since square of these values will also be larger than num.
- If $i * i <$ num, no need to search left, since square of these values will also be smaller than num.
- If $i * i ==$ num, find the answer

With these properties, we can use binary search. Due to potential overflow of using $i * i == num$, we convert it into integer division $i == num/i$. In order to check the equality of integer division, we need to check both division ($i == num/i$) and reminder ($num\%i == 0$) since multiple integers may satisfy the equality condition $i == num/i$ (e.g., $3 == 9/3, 3 == 10/3$).

[7] Corner case: integer overflow when doing $i * i$. Two fixes: 1) use long int; 2) or change multiplication to division.

```
bool isPerfectSquare(int num) {
    if (num <= 0) return false;

    int left = 1;
    int right = num;
    int mid;
    int res;
    int remain;

    while (left <= right) {
        mid = left + (right - left)/2;
        res = num / mid; // avoid overflow mid*mid
        remain = num % mid;

        if (res == mid && remain == 0) {
            return true;
        }
        else if (res < mid) {
            right = mid - 1;
        }
        else {
            left = mid + 1;
        }
    }

    return false;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(\log n)$
  The time complexity is $\mathcal{O}(\log n)$ with the binary search.

- **Space complexity**: $\mathcal{O}(1)$
  Only use limited variables.

2.15 *LC374 - Guess Number Higher or Lower*

2.15.1 *Problem Description*

LeetCode Problem 374: We are playing the Guess Game. The game is as follows: I pick a number from 1 to n. You have to guess which number I picked. Every time you guess wrong, I will tell you whether the number I picked is higher or lower than your guess.

2.15.2 *Approach - Binary*

The problem is to find the picked number in the search space $[1, n]$, which can be solved efficient by classic binary search.

2.16 LC509 - Fibonacci Number

2.16.1 Problem Description

LeetCode Problem 509: The Fibonacci numbers, commonly denoted $F(n)$ form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

$$\begin{aligned} F(0) &= 0,\ F(1) = 1 \\ F(n) &= F(n-1) + F(n-2),\ \text{for } n > 1 \end{aligned}$$

Given n, calculate $F(n)$.

2.17 LC540 - Single Element in a Sorted Array

2.17.1 Problem Description

LeetCode Problem 540:You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once. Find this single element that appears only once.

Follow up: Your solution should run in O(log n) time and O(1) space.

2.17.2 Approach - Binary Search

The array is sorted with most elements appear **exactly twice** except for one element [8]. So we can search array in pairs (two elements), i.e., $[2i, 2i-1]$. For any given pair, if elements in a pair are the same, it means the single element is on the right. Otherwise, the single element is either in this pair or on the left. With this property, we can use binary search to speed up the search. The search space is the pair index, i.e., $[0, n/2]$. We want to find the first even-index number not followed by the same number [9].

```
int singleNonDuplicate(vector<int>& nums) {
    if (nums.empty()) return -1;

    int left = 0;
    int right = nums.size()/2;  // pair index
    int mid;

    while (left < right) {
        mid = left + (right - left)/2;

        if (nums[2*mid] == nums[2*mid + 1]) {
            left = mid + 1;
        }
        else {
```

[8] The total number is odd and the last index is even with zero-based indexing. The target element can only appear in even index of nums, i.e., nums[0], nums[2], ...

[9] Note that the last pair only has one element. In the code, the while condition is left < right, which prevents out-of-bound accesse using nums[2*mid + 1]. When it reaches the last pair, *left == right* and it will jump out of the while loop.

```
            right = mid;
        }
    }

    return nums[2*left];
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(\log n)$
  The time complexity is $\mathcal{O}(\log n)$ with the binary search.
- **Space complexity**: $\mathcal{O}(1)$
  Only use limited variables.

2.18 LC875 - Koko Eating Bananas

2.18.1 Problem Description

LeetCode Problem 875: Koko loves to eat bananas. There are N piles of bananas, the i-th pile has piles[i] bananas. The guards have gone and will come back in H hours.

Koko can decide her bananas-per-hour eating speed of K. Each hour, she chooses some pile of bananas, and eats K bananas from that pile. If the pile has less than K bananas, she eats all of them instead, and won't eat any more bananas during this hour.

Koko likes to eat slowly, but still wants to finish eating all the bananas before the guards come back.

Return the minimum integer K such that she can eat all the bananas within H hours.

Constraints:

- $1 \leq$ piles.length $\leq 10^4$
- piles.lenght $\leq H \leq 10^9$
- $1 \leq$ piles$[i] \leq 10^9$

2.18.2 Approach - Binary Search

The difficult part of this problem is how to convert it into classic binary search problem. Based on the problem description, we can define the search space of K is between 1 and the max number of bananas M on a pile, $[1, M]$. [10] Each value in the search space has a status associated with it (let N denotes not finish eating and Y denotes finish eating). Then we have $[N, N, \cdots, N, Y, \cdots, Y]$. For given K,

[10] Since Koko likes to eat slowly, we need to start from 1 (not zero, since Koko still wants to eat; not minimum number of bananas, since Koko could eat as slow as 1 banana if $H >$ total number of bananas). If $K >= M$, Koko can finish eating all the bananas in N hours (each hour eat one pile with total N piles).

- If Koko can finish eating, no need to check $k+1, k+2, \ldots$ since Koko can finish eating with **bigger** numbers.
- If Koko can't finish eating, no need to check $k-1, k-2, \ldots$ since koko can **NOT** finish eating with **smaller** numbers.

Based on this property, we can use binary search to reduce search space after each iteration until only one element left. Assume there always existing K such that Koko can finish eating within H hours. We also need to create help functions on checking whether Koko can finish eating. [11].

[11] A nice way to round up integer values: $x/y + (x\%y == 0\,?\,0 : 1)$.

```
class Solution {
public:
    int minEatingSpeed(vector<int>& piles, int H) {
        if (piles.empty() || (H < piles.size())) {
            return -1;
        }

        int lo = 1; // minmum is 1 not zero since eat at least 1
     banana
        // int hi = *max_element(piles.begin(), piles.end()); //
    max element of the piles. Require * since max_element
    returns an iterator; time complexity O(N)
        // int hi = BIG_FIXED_VALUE
        int hi = getMaxPile(piles);

        int mid;

        // time complexity: O(logM)
        while (lo < hi) {
            mid = lo + (hi - lo)/2;

            if (canEatAll(piles, H, mid)) {
                hi = mid;
            }
            else {
                lo = mid + 1;
            }
        }

        // lo == hi after while loop
        return lo;
    }
private:
    // time complexity: O(N)
    bool canEatAll(vector<int>& piles, int H, int k) {
        int actualHour = 0;

        // for (vector<int>::iterator it = piles.begin(); it !=
    piles.end(); ++it) {*it}
        // for(vector<int>::size_type i = 0; i != piles.size();
    ++i) {piles[i]}
        for(int pile:piles) {
            actualHour += pile/k + (pile%k == 0 ? 0 : 1); //
    round to the larger integer value
        }

        return actualHour <= H;
```

```
    }

    int getMaxPile(vector<int>& piles) {
        int maxPile = piles[0];
        for (int pile : piles) {
            maxPile = max(pile, maxPile);
        }
        return maxPile;
    }

    int max(int a, int b) {
        return (a < b) ? b : a;
    }
};
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(N \log M)$
  It takes $\mathcal{O}(\log M)$ (M is the maximum number of bananas in a pile) steps to find K in the search space $[1, M]$. For each iteration, it takes $\mathcal{O}(N)$ to check all piles to see whether Koko can finishe eating. Additionally, it takes $\mathcal{O}(N)$ to find the maximum number of bananas in a pile. So the total time complexity is $\mathcal{O}(N \log M) = \mathcal{O}(N \log M) + \mathcal{O}(N)$.
- **Space complexity**: $\mathcal{O}(1)$
  Only use several variables for binary search and computation.