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

2
Solutions

2.1 LC69 - Sqrt(x)

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). [1] The Search range is the range of array, $[0, n-1]$.

[1] @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 *LC70 - Climbing Stairs*

2.4.1 *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?

2.4.2 *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.

2.5 *LC240 - Search a 2D Matrix II*

2.6 *LC278 - First Bad Version*

2.6.1 *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.

2.6.2 *Approach*

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.

2.7 LC287 - Find the Duplicate Number

2.7.1 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 [2], 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.

[2] 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, $n <= m$.

2.7.2 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. [3] 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.

[3] 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.

2.7.3 *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.

2.7.4 *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.

2.7.5 *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 $\mathcal{O}(n^2)$	$\mathcal{O}(n)$

2.8 *LC875 - Koko Eating Bananas*

2.8.1 *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:

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

2.8.2 *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, $[1, M]$. [4] 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,

- 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. [5].

[4] 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).

[5] 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.

2.9 LC509 - Fibonacci Number

2.9.1 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 $F(n)$.