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
  - LC275 - H-Index II
- Unsorted array
  - LC162 - Find Peak Element

1.2 Array

1.3 Linked List

Some tips for solving linked list problems:

- Make sure it is not a NULL pointer when de-referencing the current node, next node, or even next of next node
- Never ever lose the control of the head pointer of the linked list
- Use dummy head to simplify conditions when head could be changed
- Use slow and fast pointers to find middle node or detect a cycle

1.3.1 *Slow and fast pointers (Tortoise and hare approach)*

- LC876 - Middle of the Linked List
- LC141 - Linked List Cycle
- LC142 - Linked List Cycle II

1.3.2 *Dummy head*

- LC203 - Remove Linked List Elements

2
Problems with Solutions

2.1 *LCxx - title*

2.1.1 *Problem Description*

LeetCode Problem XX:

2.1.2 *Solution*

Approach - Descriptions

```
code
```

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.

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

Approach	Time Complexity	Space Complexity
Approach 1 - Hash table	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Approach 2 - Floyd's cycle detection	$\mathcal{O}(n)$	$\mathcal{O}(1)$

2.2 *LC21 - Merge Two Sorted Lists*

2.2.1 *Problem Description*

LeetCode Problem 21: Merge two sorted linked lists and return it as a sorted list. The list should be made by splicing together the nodes of

the first two lists.

2.2.2 *Solution*

We can solve this problem by following the similar idea of merge function used in the merge sort, comparing elements between two linked list and move the smaller one each time. We can use either iteration or recursion method to implement the merge function.

Approach - Iteration Compare elements of two sorted list and move the smaller one to the new sorted linked list until one of linked list is empty. Then simply append the non-empty linked list to the end of the new sorted linked list. Remember to take care of the corner case where either of linked list is empty.

```
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    if (l1 == nullptr && l2 == nullptr) return nullptr;

    ListNode* dummyHead = new ListNode(-1);
    ListNode* pNew = dummyHead;
    ListNode* p1 = l1;
    ListNode* p2 = l2;

    while (p1 != nullptr || p2 != nullptr) {
        if (p1 == nullptr) {
            pNew->next = p2;
            break;
        }
        else if (p2 == nullptr) {
            pNew->next = p1;
            break;
        }
        else {
            if (p1->val <= p2->val) {
                pNew->next = p1;
                p1 = p1->next;
            }
            else {
                pNew->next = p2;
                p2 = p2->next;
            }
        }
        pNew = pNew->next;
    }

    return dummyHead->next;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  In the worst case, it will compare n times for all nodes of both linked lists (assume total number of nodes is n) and therefore the time complexity is $\mathcal{O}(n)$.

- **Space complexity**: $\mathcal{O}(1)$
  Only define a new dummy head and several pointers and modify the next node of existed linked lists.

Approach - Recursion We can recursively break down the merge of two linked list into the merge of linked list with smaller size.

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    // Base case:
    if (l1 == null) return l2;
    if (l2 == null) return l1;

    // Recursively break down problem into a smaller one
    if (l1.val <= l2.val) {
        l1.next = mergeTwoLists(l1.next, l2);
        return l1;
    }
    else {
        l2.next = mergeTwoLists(l1, l2.next);
        return l2;
    }
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  In the worst case, it will recursively call itself n times (total number of nodes, n) and therefore the time complexity is $\mathcal{O}(n)$.
- **Space complexity**: $\mathcal{O}(n)$
  Since the recursion could go up to n levels deep, the implicit stack space due to recursive function call is $\mathcal{O}(n)$.

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

Approach	Time Complexity	Space Complexity
Approach 1 - Iteration	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Approach 2 - Recursion	$\mathcal{O}(n)$	$\mathcal{O}(n)$

2.3 LC33 - Search in Rotated Sorted Array

2.3.1 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.

2.3.2 *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.

```
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.

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

2.5 *LC35 - Search Insert Position*

2.5.1 *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.

2.5.2 *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.

2.6 *LC69 - Sqrt(x)*

2.6.1 *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.

2.6.2 *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.

2.7 LC70 - Climbing Stairs

2.7.1 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?

2.7.2 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.

2.8 LC81 - Search in Rotated Sorted Array II

2.8.1 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.

2.8.2 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.

2.9 *LC88 - Merge Sorted Array*

2.9.1 *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.

2.9.2 *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.

2.10 *LC92 - Reverse Linked List II*

2.10.1 *Problem Description*

LeetCode Problem 92: Reverse a linked list from position m to n. Do it

in one-pass. Note: $1 \leq m \leq n \leq$ length of list.

2.10.2 *Solution*

This problem is the extension of the basis reverse problem in LC206 - Reverse Linked List.

Approach - Iteration Iterate through the list and reverse sub-list from m-th node to n-th node. After reversing, make $(m-1)$-th node point to n-th node and make m-th node point to $(n+1)$-th node. In order to do it one-pass, we need to store $(m-1)$-th and m-th nodes.

```
ListNode* reverseBetween(ListNode* head, int m, int n) {
    if (head == nullptr || head->next == nullptr || m >= n) {
        return head;
    }


    // Create a dummay head
    // better handling reversing from the first node
    ListNode* dummyHead = new ListNode(-1);
    dummyHead->next = head;

    ListNode* curr = dummyHead;
    // 0 - dummy head, 1 - head,
    for (int i = 0; i < m-1; i++) {
            curr = curr->next;
    }

    // After for loop, i == m-1, curr is the m-1 th node
    ListNode* nodeMPre = curr;
    ListNode* nodeM = curr->next;

    // Reversing the sub-list
    ListNode* prev = nodeM;
    curr = nodeM->next; // reversing starting from (m+1)th node
    ListNode* next;
    for (int i = m + 1; i <= n; i++) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }

    // After for loop, prev is the n-th node, curr is the (n+1)
    th node
    nodeMPre->next = prev;
    nodeM->next = curr;

    return dummyHead->next;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  We check up to $n+1$-th node. For the worst case, we need to iterate

all nodes.

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

Approach - Recursion Break down the large problem into smaller problem each iteration, which consists of two steps:

1. Recursively solving the smaller problem until reaching the node needs to be reversed
   reverseBetween(head->next, m-1, n-1)

2. Recursively reverse the node connection, return the new head of reversed list where the tail of the reversed list point to the $n+1$th node

Using an example $1 \rightarrow \boxed{2 \rightarrow 3 \rightarrow 4} \rightarrow 5$ to illustrate the idea:

- 1st function call: reverse(①, 2, 4)
  After return from reverse(②, 1, 3), the list becomes

$$\boxed{2 \leftarrow 3 \leftarrow 4} \leftarrow 1$$
$$\downarrow$$
$$5(tail)$$

  Simply return head ① of the list with sublist reversed.

- 2nd function call: reverse(②, 1, 3)
  After returning from reverse(③, 1, 2), the sub-list becomes

$$\boxed{2 \rightarrow 3 \leftarrow 4}$$
$$\downarrow$$
$$5(tail)$$

  We can repeat the step: save tail pointer, reverse connection, and connect tail. The sub-list becomes

$$\boxed{2 \leftarrow 3 \leftarrow 4}$$
$$\downarrow$$
$$5(tail)$$

  In the end, return head ④ of the reversed sub-list.

- 3rd function call: reverse(③, 1, 2)
  After return from reverse(④, 1, 1), the sub-list becomes $\boxed{3 \rightarrow 4} \rightarrow 5$.

We can do the following steps to reverse node connections:

$\boxed{3 \to 4} \to 5(\text{tail})$, store tail pointer
$\boxed{3 \leftarrow 4}$ $5(\text{tail})$, reverse connection
$\boxed{3 \leftarrow 4}$, connect tail
$\downarrow$
$5(tail)$

In the end, return head ④ of the reversed sub-list.

- 4th function call: reverse(④, 1, 1)
  Since just one node to reverse, no need to reverse and return head ④.

```
ListNode* reverseBetween(ListNode* head, int m, int n) {
    if (m == n) {
        return head;
    }

    if (m > 1) {
        head->next = reverseBetween(head->next, m-1, n-1);
        return head;
    }

    // when m == 1, reach the start node for reversing
    // Head is at position 1 based on the definition of the
    problem (one-based in index)
    ListNode* newHead = reverseBetween(head->next, 1, n-1); //
    return head of reversed sublist
    ListNode* tail = head->next->next;
    head->next->next = head; // reverse
    head->next = tail;
    return newHead;

}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  We check up to $n+1$-th node. For the worst case, we need to iterate all nodes.

- **Space complexity**: $\mathcal{O}(n)$
  The extra space comes from implicit stack space due to recursive function call. The recursion could go up to n levels deep.

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

Approach	Time Complexity	Space Complexity
Approach 1 - Iteration	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Approach 2 - Recursion	$\mathcal{O}(n)$	$\mathcal{O}(n)$

2.11 LC141 - Linked List Cycle

2.11.1 Problem Description

LeetCode Problem 141: Given head, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Return true if there is a cycle in the linked list. Otherwise, return false.

2.11.2 Solution

Approach 1 - Hash Table To detect if a linked list containts a cycle, we can store each node's address and check whether a node has been visited before.

```
bool hasCycle(ListNode *head) {
    unordered_set<ListNode*> seen;

    for (ListNode *p = head; p != nullptr; p = p->next) {
        if (seen.count(p)) {
            return true;
        }
        seen.insert(p);
    }

    return false;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  Need to visit each of the n nodes in the list at once. Adding a element or checking the existence in the hash table costs $\mathcal{O}(1)$ time.
- **Space complexity**: $\mathcal{O}(n)$
  The space depends on the number of elements added to the hash table, which contains at most n elements.

Approach 2 - Floyd's Cycle Finding Algorithm Imagine two runners running on a track at different speed. If the track is circle, they will eventually meet. Consider two pointers at different speed – a **slow pointer** moves 1 step at a time while a **fast pointer** moves 2 steps at a time [4]. So the number of iteration for the fast runner to catch up the

[4] Can the fast pointer move 3 time, 4 time, or n times faster? Can they eventually meet?

slow runner can be obtained as

$$\text{iteration to catch up} = \frac{\text{distance between the two runners}}{\text{difference of speed}} \tag{2.1}$$

where difference of speed is 1 and the max value of distance between the two runners is the length of the cycle.

```
bool hasCycle(ListNode *head) {
    if (head == nullptr || head->next == nullptr)
        return false;

    ListNode* slow = head;
    ListNode* fast = head;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;

        if (slow == fast) return true;
    }

    return false;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  Denotes n as the total number of nodes in the linked list. For time complexity, analyze the following two cases separately:
  - List has no cycle
    The fast pointer reaches the end first. Since it moves two steps a time, the time complexity is $\mathcal{O}(n/2)$.
  - List has a cycle
    Time complexity can be calculated based on the movement of the slow pointer. We can break down the movement into two steps, the non-cyclic part and the cyclic part:
    * The slow pointer takes "non-cyclic length" steps to enter the cycle. At this point the fast pointer has already reached the cycle. Number of iterations = non-cyclic length = l_1.
    * Both pointers are now in the cycle. Based on Eq. 2.1, it takes at most "cyclic length" for the fast runner to catch up with the slow runner. So number of iterations in the worst case = cyclic length = c.

    Combine above mentioned two steps, the time complexity is $\mathcal{O}(l_1 + c)$, which is $\mathcal{O}(n)$ since $n = l_1 + c$.
- **Space complexity**: $\mathcal{O}(1)$
  Only use two pointers (slow and fast). So the space complexity is $\mathcal{O}(1)$.

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

Approach	Time Complexity	Space Complexity
Approach 1 - Hash table	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Approach 2 - Floyd's cycle detection	$\mathcal{O}(n)$	$\mathcal{O}(1)$

2.12 LC142 - Linked List Cycle II

2.12.1 Problem Description

LeetCode Problem 142: Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

Notice that you **should not modify** the linked list.

2.12.2 Solution

Approach - Floyd's Cycle Detection Use the Floyd's cycle detection method to find the meet point, which consists of two phases:

- Phase I: Detect a cycle and find the meeting point if there is a cycle
  Use a slow pointer that moves 1 step a time and a fast pointer that moves 2 steps a time to detect a cycle and find the meeting point if there is a cycle. Follow approach 2 in LC141 - Linked List Cycle
- Phase II: If there is a cycle, return the entry location of the cycle
  For analysis, define the following notations:
  - l_1: the distance (i.e., number of nodes) between the head and the entry point of the cycle
  - l_2: the distance (i.e., number of nodes) between the entry point and the meeting point in the cycle
  - c: length of the cycle
  - n: number of cycles the fast pointer moves ($n >= 1$)

  We can obtain:
  - the travel distance of the slow pointer is $l_1 + l_2$
  - the travel distance of the slow pointer is $l_1 + nc + l_2$

– Since the fast pointer moves twice fast as the slow pointer, the travel distance of the fast pointer is twice as the slow pointer,

$$\begin{aligned} 2(l_1 + l_2) &= l_1 + nc + l_2 \\ l_1 + l_2 &= nc \\ l_1 &= (n-1)c + (c - l_2) \end{aligned} \tag{2.2}$$

Eq. 2.2 implies that the distance between the head and entry point of the cycle is equal to the distance between the meeting point and entry point along the direction of the forward movement.

So we can have two pointers with the same speed (1 step a time). One starts from the head and the other starts from the meeting points. When these two pointers meet, the meeting node is where the cycle begins.

```
ListNode *detectCycle(ListNode *head) {
    if (head == nullptr || head->next == nullptr)
        return nullptr;

    ListNode* slow = head;
    ListNode* fast = head;
    ListNode* entry = head;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;

        // Detect a cycle and find the entry point
        if (slow == fast) {
            while (entry != slow) {
                entry = entry->next;
                slow = slow->next;
            }
            return entry;
        }
    }
    return nullptr;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  Denotes n as the total number of nodes in the linked list. For time complexity, analyze the following two cases separately:

  – List has no cycle
  The fast pointer reaches the end first. Since it moves two steps a time, the time complexity is $\mathcal{O}(n/2)$.

  – List has a cycle
  For Phase I, it takes $l_1 + l_2$ iterations to find the meet point. For Phase II, it takes $c - l_2$ iterations to find the entry point of the cycle. So the total number of iterations is $l_1 + l_2 + c - l_2 = n$.

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

2.13 LC153 - Find Minimum in Rotated Sorted Array

2.13.1 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**.

2.13.2 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] [5]:

[5] 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 $[4,5,6,7,0,1,2]$).

- 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.

```
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.

2.14 *LC154 - Find Minimum in Rotated Sorted Array II*

2.14.1 *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.

2.14.2 *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.

2.15 LC203 - Remove Linked List Elements

2.15.1 Problem Description

LeetCode Problem 203: Remove all elements from a linked list of integers that have value val.

2.15.2 *Solution*

Approach - Iteration This problem uses the basic list operation, remove operation. Remember to take care of the corner case where the head will be deleted [6] and release memory [7] after removing the node.

[6] Create a **dummy head** to simplify the handling on special case, deleting head.

[7] Specifically delete the node in C++ or Garbage collection in java will reclaim memory when there are no more references to an object

```
public ListNode removeElements(ListNode head, int val) {
    ListNode dummyHead = new ListNode(0, head);

    ListNode prev = dummyHead;
    ListNode curr = dummyHead.next;

    while (curr != null) {
        // Delete node
        if (curr.val == val) {
            prev.next = curr.next;
            curr.next = null; // set the next pointer of the
    deleted node to null, may no be necessary
            curr = prev.next;
        }
        else {
            prev = curr;
            curr = curr.next;
        }
    }

    return dummyHead.next;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  Since go through all the nodes in the linked list, the time complexity is $\mathcal{O}(n)$.

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

Approach - Recursion The problem can also be solved by using recursion, recursively removing elements from the sub-list starting from the next node. Also remember to properly set the next pointer of the deleted node to null.

```
public ListNode removeElements(ListNode head, int val) {
    // Base case
    if (head == null) return null;

    head.next = removeElements(head.next, val);

    return head.val == val ? head.next : head;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  Since go through all the nodes in the linked list, the time complexity is $\mathcal{O}(n)$.
- **Space complexity**: $\mathcal{O}(n)$
  The extra space comes from implicit stack space due to recursive function call. The recursion could go up to n levels deep.

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

Approach	Time Complexity	Space Complexity
Approach - Iteration	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Approach - Recursion	$\mathcal{O}(n)$	$\mathcal{O}(n)$

2.16 LC206 - Reverse Linked List

2.16.1 Problem Description

LeetCode Problem 206: Reverse a singly linked list.

```
Input: 1 -> 2 -> 3 -> 4 -> 5 -> NULL
Output: 5 -> 4 -> 3 -> 2 -> 1 -> NULL
```

2.16.2 Solution

Approach - Iteration Reversing a linked list can be viewed as changing direction of each node from pointing to the next node to pointing to the previous node

```
      1 -> 2 -> 3 -> 4 -> 5 -> NULL
NULL <- 1 <- 2 <- 3 <- 4 <- 5
```

```
ListNode* reverseList(ListNode* head) {
    // If list is empty or only have one elment, no need to
    reverse
    if (head == nullptr || head->next == nullptr)
        return head;

    ListNode* prev = nullptr;
    ListNode* curr = head;
    ListNode* next = nullptr;

    while (curr != nullptr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
```

```
    }

    // After while loop, curr is nullptr and prev is the valid
    head
    return prev;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  Iteratively go through all nodes
- **Space complexity**: $\mathcal{O}(1)$
  Only use limited variables.

Approach - Recursion Break down the large problem into small problems by recursion. For a given node, assume the rest of the list has already been reversed, just reverse the direction of the current node. For example, at node n_k, assume the list from node n_{k+1} to n_m has been reversed.

$$n_1 \to \cdots \to n_{k-1} \to n_k \to n_{k+1} \leftarrow \cdots \leftarrow n_m$$

To change the direction of node n_k,

- Make n_{k+1}'s next node points to n_k (i.e., $n_k \to \text{next} \to \text{next} = n_k$)
- Make n_k points to NULL

$$n_1 \to \cdots \to n_{k-1} \to \quad \begin{array}{c} n_k \\ \downarrow \\ \text{null} \end{array} \quad \leftarrow n_{k+1} \leftarrow \cdots \leftarrow n_m$$

Note that

- If n_k is not pointing to NULL, we will have a cycle here (i.e., $n_k \rightleftharpoons n_{k+1}$)
- If n_k is the original head, it will point to NULL.
- If n_k is not the original head, n_k's next pointer will be reset from NULL to n_{k-1} in the parent function call during recursion.

Doing above steps recursively until only one node or empty node left.

```
ListNode* reverseList(ListNode* head) {
    // Base case
    if (head == nullptr || head->next == nullptr)
        return head;

    ListNode* p = reverseList(head->next);
```

```
    head->next->next = head;
    head->next = nullptr;
    return p;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  Recursively go through all nodes of the list.
- **Space complexity**: $\mathcal{O}(n)$
  The extra space comes from implicit stack space due to recursive function call. The recursion could go up to n levels deep.

Approach - Stack Another idea is to use a stack to store references to all nodes. Since elements in stack are last-in-first-out, we can change the connection moving from the end of list to the head by popping node one by one.

```
ListNode* reverseList(ListNode* head) {
    if (head == nullptr || head->next == nullptr)
        return head;

    stack<ListNode*> s; // Define empty stack

    // Push all nodes to stack
    ListNode* p;
    for (p = head; p != nullptr; p = p->next) {
        s.push(p);
    }

    // Pop nodes - last in first out
    ListNode* newHead = s.top();
    p = newHead;
    s.pop();
    while (!s.empty()) {
        p->next = s.top();  // note run top first and then pop.
    If reversed, after pop, the stack may become empty and top
    will cause undefined behavior
        s.pop();
        p = p->next;
    }
    p->next = nullptr;

    return newHead;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  It takes n iterations to push all nodes and another n iterations to pop the node and change the direction. So the time complexity is $\mathcal{O}(n) = \mathcal{O}(n) + \mathcal{O}(n)$.

- **Space complexity**: $\mathcal{O}(n)$
  The extra space comes from stack storage.

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

Approach	Time Complexity	Space Complexity
Approach - Iteration	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Approach - Recursion	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Approach - Stack	$\mathcal{O}(n)$	$\mathcal{O}(n)$

2.17 LC234 - Palindrome Linked List

2.17.1 Problem Description

LeetCode Problem 234: Given a singly linked list, determine if it is a palindrome [8].

[8] palindrome: a word, phrase, or sequence that reads the same backward as forward, e.g., madam.

2.17.2 Solution

Approach - Array We can use an array to store node addresses or node values. Then have two pointers moving from two sides to center and check whether it is palindrome.

```
public boolean isPalindrome(ListNode head) {
    if (head == null || head.next == null) return true;

    ArrayList<ListNode> arr = new ArrayList<ListNode>();

    for (ListNode p = head; p != null; p = p.next) {
        arr.add(p);
    }

    int i = 0;
    int j = arr.size() - 1;
    while (i < j) {
        if (arr.get(i).val != arr.get(j).val)
            return false;

        i++;
        j--;
    }

    return true;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  It takes $\mathcal{O}(n)$ time to go through all nodes and store values. Then

it takes at most $\mathcal{O}(n/2)$ time to compare values. So the time complexity is $\mathcal{O}(n)$.

- **Space complexity**: $\mathcal{O}(n)$
  Since saving all node values, the space complexity is $\mathcal{O}(n)$.

Approach - Stack Instead of using array, we can use a stack to store list nodes and use the stack's property, last in first out [9].

[9] Remember to clean stack if end early

```
public boolean isPalindrome(ListNode head) {
    if (head == null || head.next == null) return true;


    Stack<ListNode> s = new Stack<ListNode>();

    ListNode p;
    for (p = head; p != null; p = p.next) {
        s.push(p);
    }

    ListNode h = head;
    boolean isPal = true;
    while (!s.empty()) {
        p = s.pop();
        if (h == p)
            break;

        if (h.val != p.val) {
            isPal = false;
            break;
        }
        h = h.next;
    }

    s.clear();
    return isPal;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  It takes $\mathcal{O}(n)$ time to go through all nodes and store values into stack. Then it takes at most $\mathcal{O}(n/2)$ time to compare values. So the time complexity is $\mathcal{O}(n)$.

- **Space complexity**: $\mathcal{O}(n)$
  Since saving all node values in a stack, the space complexity is $\mathcal{O}(n)$.

Approach - Reverse Half We can use fast and slow pointers to find the 2nd half of the list. Then reverse the 2nd half and compare the two halves.

```
bool isPalindrome(ListNode* head) {
    if (head == nullptr) return true;
```

```
    ListNode* slow = head;
    ListNode* fast = head;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }

    // Reverse the right half
    ListNode* p2ndHalf = reverse(slow);
    ListNode* p = head;

    while (p2ndHalf != nullptr && p != nullptr) {
        if (p2ndHalf->val != p->val) {
            return false;
        }
        p2ndHalf = p2ndHalf->next;
        p = p->next;
    }

    return true;
}

ListNode* reverse(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* curr = head;
    ListNode* next = nullptr;

    while (curr != nullptr) {
        next= curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  Finding the start of 2nd half takes $\mathcal{O}(n/2)$ time. Reversing the 2nd half takes $\mathcal{O}(n/2)$ time. Comparing values takes at most $\mathcal{O}(n/2)$ time. So the time complexity is $\mathcal{O}(n)$.

- **Space complexity**: $\mathcal{O}(1)$
  Since directly modify the linked list and use the limited variables, the space complexity is $\mathcal{O}(1)$.

Approach - Recursion This problem can also be solved by using recursion. During the recursion, when returning from the base cases, it returns from tail node to head. We just need to find a way to store the node pointer moving from head to tail [10] and compare with node values during recursion.

[10] We can use private class member to store the pointer or pass by reference in C++

```
class Solution {
    private ListNode h;
```

```
    private boolean check(ListNode p) {
        if (p == null) return true; // base case

        boolean isPal = check(p.next);

        if (!isPal || h.val != p.val)
            return false;

        h = h.next; // update class member h
        return isPal;
    }
    public boolean isPalindrome(ListNode head) {
        h = head;
        ListNode p = head;
        return check(p);
    }
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(n)$
  Since go through all nodes, so the time complexity is $\mathcal{O}(n)$.

- **Space complexity**: $\mathcal{O}(n)$
  The extra space comes from implicit stack space due to recursive function call. The recursion could go up to n levels deep.

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

Approach	Time Complexity	Space Complexity
Approach - Array	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Approach - Stack	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Approach - Reverse Half	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Approach - Recursion	$\mathcal{O}(n)$	$\mathcal{O}(n)$

2.18 LC237 - Delete Node in a Linked List

2.18.1 Problem Description

LeetCode Problem 237: Write a function to delete a node in a singly-linked list. You will not be given access to the head of the list, instead you will be given access to the node to be deleted directly.

It is **guaranteed** that the node to be deleted is **not a tail node** in the list.

2.18.2 *Approach*

The usual way of deleting a node node from a linked list is to modify the next pointer of the node before it, to point to the node after it. Since we do not have access to the node before the one we want to delete, we cannot modify the next pointer of that node in any way. Instead, we have to replace the value of the node we want to delete with the value in the node after it, and then delete the node after it. Because we know that the node we want to delete is not the tail of the list, we can guarantee that this approach is possible.

```
void deleteNode(ListNode* node) {
    ListNode* next = node->next;

    // Update the current node to duplicate the next node
    node->val = next->val;
    node->next = next->next;

    // Delete next node
    delete next;
}
```

Complexity Analysis:

- **Time complexity**: $\mathcal{O}(1)$
  Directly modify the node.
- **Space complexity**: $\mathcal{O}(1)$
  Only use limited number of variables.

2.19 *LC240 - Search a 2D Matrix II*

2.20 *LC275 - H-Index II*

2.20.1 *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.

2.20.2 *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

$$\text{citations[index]} >= n - \text{index} \tag{2.3}$$

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. 2.3, 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 citations$[j]$ >= citations$[i]$. Together with Eq. 2.3, we can obtain

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

  which satisfies Eq. 2.3.

- 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.

2.21 *LC278 - First Bad Version*

2.21.1 *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.

2.21.2 *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.

2.22 LC287 - Find the Duplicate Number

2.22.1 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 [11], 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.

[11] 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, $n <= m$.

2.22.2 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. [12] 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.

[12] 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.

2.22.3 *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.

2.22.4 *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.

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

2.23 *LC367 - Valid Perfect Square*

2.23.1 *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.

2.23.2 *Approach - Binary*

The problem can be transformed to find an integer i in the search space $[1, num]$ such that $i * i == \text{num}$ [13].

- 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., $3 == 9/3, 3 == 10/3$) [14].

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

[14] The perfect squared number is the smallest value due to round down from integer 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) { // mid is small, search right
            left = mid + 1;
        }
        else { // mid is large, search left
            right = 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.

2.24 LC374 - Guess Number Higher or Lower

2.24.1 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.

2.24.2 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.

2.25 LC509 - Fibonacci Number

2.25.1 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 $F(n)$.

2.26 LC540 - Single Element in a Sorted Array

2.26.1 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.

2.26.2 Approach - Binary Search

The array is sorted with most elements appear **exactly twice** except for one element [15]. So we can search array in pairs (two elements),

[15] 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], ...

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 [16].

[16] 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.

```
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 {
            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.

2.27 LC875 - Koko Eating Bananas

2.27.1 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:

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

2.27.2 *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, $[1, M]$. [17] 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. [18].

[17] 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).

[18] 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; // minimum 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);
            if (pile > maxPile) maxPile = pile;
        }
        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.