

1. 两数之和

给定一个整数数组 `nums` 和一个整数目标值 `target`, 请你在该数组中找出 和为目标值 `target` 的那 两个 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        std::unordered_map<int, int> mp;
        for (int i = 0, j; i < nums.size(); ++i) {
            j = target - nums[i];
            if (mp.find(j) != mp.end()) {
                return {mp[j], i};
            } else {
                mp[nums[i]] = i;
            }
        }
        return {};
    }
};
```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: `数组`, `哈希表`
- 难度: 简单

2. 两数相加

给你两个 非空 的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储一位 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* res = new ListNode();
        ListNode* ptr = res;
        int isCarry = 0, sm = 0;
        // l1 and l2 step
        while (l1 != NULL && l2 != NULL) {
            sm = l1->val + l2->val + isCarry;
            isCarry = sm >= 10;
            ptr->next = new ListNode(sm % 10);
            ptr = ptr->next;
            l1 = l1->next;
            l2 = l2->next;
        }
        // l1 or l2 step
        l1 = l1 != NULL ? l1 : l2;
        while (l1 != NULL) {
            sm = l1->val + isCarry;
            isCarry = sm >= 10;
            ptr->next = new ListNode(sm % 10);
            ptr = ptr->next;
            l1 = l1->next;
        }
        // last isCarry = 1
        if (isCarry) {
            ptr->next = new ListNode(1);
        }
        return res->next;
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 递归, 链表, 数学
- 难度: 中等

3. 无重复字符的最长子串

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

```

class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int res = 0;
        std::unordered_map<char, int> mp;
        int start = 0;
        for (int end = 0; end < s.size(); ++end) {
            if (mp.find(s[end]) == mp.end()) {
                mp[s[end]] = end;
            } else {
                int tmp = mp[s[end]];
                for (int i = start; i <= tmp; ++i) {
                    mp.erase(s[i]);
                }
                start = tmp + 1;
                mp[s[end]] = end;
            }
            res = std::max(res, end - start + 1);
        }
        return res;
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(\min(m, n))$
- 解法:
- 标签: 哈希表, 字符串, 滑动窗口
- 难度: 中等

4. 寻找两个正序数组的中位数

给定两个大小分别为 m 和 n 的正序（从小到大）数组 nums1 和 nums2 。请你找出并返回这两个正序数组的 中位数。

进阶：你能设计一个时间复杂度为 $O(\log(m+n))$ 的算法解决此问题吗？

```

// https://zhuanlan.zhihu.com/p/70654378
/**
 *
 * 如图： 假设结果i, j分别在nums1, nums2中间
 * l1      i      r1    l2      j      r2
 * |---nums1-----| |-----nums2----|
 *
 * i->r1 + l2->j == (nums1 + nums2) / 2 == l1->i + j->r2
 * i在l1和r1之间二分搜索， 可以计算得到 l2->j = (nums1 + nums2) / 2 - l1->i
 */
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        if (nums1.size() > nums2.size()) std::swap(nums1, nums2);
        if (nums2.size() == 0) return -1; // ValueError
        int iMin = 0, iMax = nums1.size(), halfLen = (nums1.size() + nums2.size() + 1) / 2;
        while (iMin <= iMax) {
            int i = iMin + (iMax - iMin) / 2;
            int j = halfLen - i;
            if (i < nums1.size() && nums1[i] < nums2[j - 1]) {
                iMin = i + 1;
            } else if (i > 0 && nums1[i - 1] > nums2[j]) {
                iMax = i - 1;
            } else {
                int maxLeft;
                if (i == 0) {
                    maxLeft = nums2[j - 1];
                } else if (j == 0) {
                    maxLeft = nums1[i - 1];
                } else {
                    maxLeft = std::max(nums1[i - 1], nums2[j - 1]);
                }
                if ((nums1.size() + nums2.size()) % 2 == 1) { // 中位数是一个
                    return maxLeft;
                }
                int maxRight;
                if (i == nums1.size()) {
                    maxRight = nums2[j];
                } else if (j == nums2.size()) {
                    maxRight = nums1[i];
                } else {
                    maxRight = std::min(nums1[i], nums2[j]);
                }
                return (maxLeft + maxRight) / 2.0;
            }
        }
        return -1; // NonUse, for static check
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(log(min(nums1.size(), nums2.size())))
- 解法: 二分查找
- 标签: 数组, 二分查找, 分治
- 难度: 困难

5. 最长回文子串

给你一个字符串 s, 找到 s 中最长的回文子串。

```

class Solution {
public:
    string longestPalindrome(string s) {
        if (s.size() < 2) return s;
        for (int i = 0; i < s.size(); ++i) {
            expandAroundCenter(s, i, i);
            expandAroundCenter(s, i, i + 1);
        }
        return s.substr(start, end - start + 1);
    }

private:
    int start = 0;
    int end = 0;

    void expandAroundCenter(const std::string& s, int l, int r) {
        while (l >= 0 && r < s.size() && s[l] == s[r]) {
            l--;
            r++;
        }
        if (r - l - 2 > end - start) {
            start = l + 1;
            end = r - 1;
        }
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n^2)
- 解法: 中心扩展算法
- 标签: 字符串, 动态规划
- 难度: 中等

7. 整数反转

给你一个 32 位的有符号整数 x ，返回将 x 中的数字部分反转后的结果。

如果反转后整数超过 32 位的有符号整数的范围 $[-2^{31}, 2^{31} - 1]$ ，就返回 0。

假设环境不允许存储 64 位整数（有符号或无符号）。

```

class Solution {
public:
    int reverse(int x) {
        long long int res = 0;
        while (x != 0) {
            res = res * 10 + x % 10;
            x = x / 10;
        }
        return (res < INT_MIN || res > INT_MAX) ? 0 : res;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法:
- 标签: 数学
- 难度: 简单

8. 字符串转换整数 (atoi)

请你来实现一个 myAtoi(string s) 函数，使其能将字符串转换成一个 32 位有符号整数（类似 C/C++ 中的 atoi 函数）。

函数 myAtoi(string s) 的算法如下：

读入字符串并丢弃无用的前导空格

检查下一个字符（假设还未到字符末尾）为正还是负号，读取该字符（如果有）。确定最终结果是负数还是正数。如果两者都不存在，则假定结果为正。

读入下一个字符，直到到达下一个非数字字符或到达输入的结尾。字符串的其余部分将被忽略。

将前面步骤读入的这些数字转换为整数（即，“123” -> 123，“0032” -> 32）。如果没有读入数字，则整数为 0。必要时更改符号（从步骤 2 开始）。

如果整数数超过 32 位有符号整数范围 $[-2^{31}, 2^{31} - 1]$ ，需要截断这个整数，使其保持在这个范围内。具体来说，小于 -2^{31} 的整数应该被固定为 -2^{31} ，大于 $2^{31} - 1$ 的整数应该被固定为 $2^{31} - 1$ 。

返回整数作为最终结果。

注意：

本题中的空白字符只包括空格字符 ‘ ’。

除前导空格或数字后的其余字符串外，请勿忽略任何其他字符。

```
class Solution {
public:
    int myAtoi(string s) {
        long int result = 0;
        int indicator = 1;
        int i = 0;
        i = s.find_first_not_of(' ');
        if (i < s.size() && (s[i] == '-' || s[i] == '+')) {
            indicator = (s[i] == '-') ? -1 : 1;
            i++;
        }
        while (i < s.size() && ('0' <= s[i] && s[i] <= '9')) {
            result = result * 10 + (s[i] - '0');
            i++;
            if (result * indicator >= std::numeric_limits<int>::max()) return std::numeric_limits<int>::max();
            if (result * indicator <= std::numeric_limits<int>::min()) return std::numeric_limits<int>::min();
        }
        return result * indicator;
    }
};
```

- 空间复杂度：O(1)
- 时间复杂度：O(n)
- 解法：
- 标签：字符串
- 难度：中等

10. 正则表达式匹配

TODO:

11. 盛最多水的容器

给你 n 个非负整数 a_1, a_2, \dots, a_n ，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$ 。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器。

输入: [1,8,6,2,5,4,8,3,7]

输出: 49

解释: 图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下, 容器能够容纳水 (表示为蓝色部分) 的最大值为 49。

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int maxSpace = 0, l = 0, r = height.size() - 1;
        while (l < r) {
            maxSpace = std::max(maxSpace, std::min(height[l], height[r]) * (r - l));
            height[l] > height[r] ? r-- : l++;
        }
        return maxSpace;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: 双指针+贪心
- 标签: 贪心, 数组, 双指针
- 难度: 中等

13. 罗马数字转整数

罗马数字包含以下七种字符: I, V, X, L, C, D 和 M。

字符	数值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

例如, 罗马数字 2 写做 II , 即为两个并列的 1。12 写做 XII , 即为 X + II 。 27 写做 XXVII, 即为 XX + V + II 。

通常情况下, 罗马数字中小的数字在大的数字的右边。但也存在特例, 例如 4 不写做 IIII, 而是 IV。数字 1 在数字 5 的左边, 所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地, 数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况:

I 可以放在 V (5) 和 X (10) 的左边, 来表示 4 和 9。

X 可以放在 L (50) 和 C (100) 的左边, 来表示 40 和 90。

C 可以放在 D (500) 和 M (1000) 的左边, 来表示 400 和 900。

给定一个罗马数字, 将其转换成整数。输入确保在 1 到 3999 的范围内。

```

class Solution {
public:
    int romanToInt(string s) {
        std::map<char, int> mp = {
            {'I', 1},
            {'V', 5},
            {'X', 10},
            {'L', 50},
            {'C', 100},
            {'D', 500},
            {'M', 1000}
        };
        int res = mp[s.size() - 1];
        for (int i = s.size() - 2; i >= 0; --i) {
            res += mp[s[i]] < mp[s[i + 1]] ? -mp[s[i]] : mp[s[i]];
        }
        return res;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法:
- 标签: 哈希表, 数学, 字符串
- 难度: 简单

14. 最长公共前缀

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀, 返回空字符串 ""。

```

class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {
        if (strs.size() == 0) return "";
        if (strs.size() == 1) return strs[0];
        int iterLen = std::numeric_limits<int>::max();
        for (const std::string s : strs) iterLen = std::min(iterLen, (int)s.size());
        std::string res;
        for (int i = 0; i < iterLen; ++i) {
            int j = 1;
            while (j < strs.size() && strs[j - 1][i] == strs[j][i]) j++;
            if (j == strs.size()) {
                res += strs[0][i];
            } else {
                break;
            }
        }
        return res;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n^2)
- 解法: 双循环, 字典树
- 标签: 字符串
- 难度: 简单

FIXME: 字典树解法 <https://leetcode.com/problems/longest-common-prefix/solution/>

15. 三数之和

给你一个包含 n 个整数的数组 nums , 判断 nums 中是否存在三个元素 a, b, c , 使得 $a + b + c = 0$? 请你找出所有和为 0 且不重复的三元组。

注意: 答案中不可以包含重复的三元组。

示例 1:

输入: $\text{nums} = [-1,0,1,2,-1,-4]$

输出: $-1,-1,2],[-1,0,1]$

示例 2:

输入: $\text{nums} = []$

输出: $[]$

示例 3:

输入: $\text{nums} = [0]$

输出: $[]$

提示:

$0 \leq \text{nums.length} \leq 3000$

$-10^5 \leq \text{nums}[i] \leq 10^5$

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        std::vector<std::vector<int>> res;
        std::sort(nums.begin(), nums.end());
        for (int i = 0; i < (int) nums.size() - 2; ++i) {
            if (i > 0 && nums[i] == nums[i - 1]) continue; // result remove repeat
            int j = i + 1, k = nums.size() - 1;
            while (j < k) {
                int sm = nums[i] + nums[j] + nums[k];
                if (sm == 0) {
                    res.push_back(std::vector<int>{ nums[i], nums[j], nums[k] });
                    while (j + 1 < k && nums[j] == nums[j + 1]) j++;
                    while (j < k - 1 && nums[k] == nums[k - 1]) k--;
                    j++;
                    k--;
                } else if (sm > 0) {
                    k--;
                } else { // sm < 0
                    j++;
                }
            }
        }
        return res;
    }
};
```

- 空间复杂度:
- 时间复杂度: $O(n^2)$
- 解法: 双指针
- 标签: 数组, 双指针, 排序
- 难度: 中等

17. 电话号码的字母组合

给定一个仅包含数字 2-9 的字符串, 返回所有它能表示的字母组合。答案可以按任意顺序 返回。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

示例 1：

输入： digits = "23"
输出： ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]

示例 2：

输入： digits = ""
输出： []

示例 3：

输入： digits = "2"
输出： ["a", "b", "c"]

```
class Solution {
public:
    vector<string> letterCombinations(string digits) {
        if (digits.size() == 0) return {};
        std::vector<std::string> mp = {"0", "1", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
        std::vector<std::string> result;
        result.push_back("");
        for (const char& ch : digits) {
            std::vector<std::string> tmp;
            for (const char& t : mp[ch - '0']) {
                for (std::string& s : result) {
                    tmp.push_back(s + t);
                }
            }
            result.swap(tmp);
        }
        return result;
    }
};
```

- 空间复杂度： $O(result.size())$
- 时间复杂度： $O(digits.size())$
- 解法：
- 标签： 哈希表， 字符串， 回溯
- 难度： 中等

19. 删除链表的倒数第 N 个结点

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

进阶：你能尝试使用一趟扫描实现吗？

示例 1：

输入： head = [1,2,3,4,5], n = 2
输出： [1,2,3,5]

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummy = new ListNode();
        dummy->next = head;
        ListNode* slow = dummy;
        ListNode* fast = dummy;
        while (n >= 0) {
            fast = fast->next;
            n--;
        }
        while (fast != nullptr) {
            fast = fast->next;
            slow = slow->next;
        }
        slow->next = slow->next->next;
        return dummy->next;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: 快慢指针
- 标签: 链表, 双指针
- 难度: 中等

20. 有效的括号

给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串 s，判断字符串是否有效。

有效字符串需满足：

左括号必须用相同类型的右括号闭合。

左括号必须以正确的顺序闭合。

提示：

$1 \leq s.length \leq 10^4$
s 仅由括号 '()' 组成

```

class Solution {
public:
    bool isValid(string s) {
        std::stack<char> stk;
        for (const char& ch : s) {
            if (ch == '(') {
                stk.push(')');
            } else if (ch == '{') {
                stk.push('}');
            } else if (ch == '[') {
                stk.push(']');
            } else if (stk.empty() || stk.top() != ch) {
                return false;
            } else {
                stk.pop();
            }
        }
        return stk.empty();
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 栈, 字符串
- 难度: 简单

21. 合并两个有序链表

将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

输入: l1 = [1,2,4], l2 = [1,3,4]

输出: [1,1,2,3,4,4]

示例 2:

输入: l1 = [], l2 = []

输出: []

示例 3:

输入: l1 = [], l2 = [0]

输出: [0]

提示:

两个链表的节点数目范围是 [0, 50]

$-100 \leq \text{Node.val} \leq 100$

l1 和 l2 均按非递减顺序排列

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode* root = new ListNode();
        ListNode* cur = root;
        while (l1 != nullptr && l2 != nullptr) {
            if (l1->val < l2->val) {
                cur->next = l1;
                l1 = l1->next;
            } else { // l1->val >= l2->val
                cur->next = l2;
                l2 = l2->next;
            }
            cur = cur->next;
        }
        cur->next = l1 != nullptr ? l1 : l2;
        return root->next;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法:
- 标签: 递归, 链表
- 难度: 简单

22. 括号生成

数字 n 代表生成括号的对数, 请你设计一个函数, 用于能够生成所有可能的并且有效的 括号组合。

示例 1:

输入: n = 3

输出: ["((0))", "(00)", "(0)0", "0(0)", "000"]

示例 2:

输入: n = 1

输出: ["0"]

提示:

$1 \leq n \leq 8$

```

class Solution {
public:
    vector<string> generateParenthesis(int n) {
        std::vector<std::string> res;
        std::string cur = "";
        backtracking(res, n, n, cur);
        return res;
    }

private:
    void backtracking(std::vector<std::string>& res, int left, int right, std::string& cur) {
        if (left == 0 && right == 0) {
            res.push_back(cur);
            return ;
        }
        // add left
        cur += '(';
        if (left > 0) backtracking(res, left - 1, right, cur);
        cur.pop_back();
        // add right
        cur += ')';
        if (left < right && right > 0) backtracking(res, left, right - 1, cur);
        cur.pop_back();
    }
};

```

- 空间复杂度： 大约 $O(n)$
- 时间复杂度： $O((4^n) / \sqrt{n})$
- 解法： 回溯
- 标签： 字符串， 动态规划， 回溯
- 难度： 中等

23. 合并K个升序链表

TODO:

26. 删除有序数组中的重复项

给你一个有序数组 `nums`，请你原地删除重复出现的元素，使每个元素只出现一次，返回删除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用 $O(1)$ 额外空间的条件下完成。

提示：

$0 \leq \text{nums.length} \leq 3 * 10^4$
 $-10^4 \leq \text{nums}[i] \leq 10^4$
`nums` 已按升序排列

```

class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.size() == 0) return 0;
        int i = 0;
        for (int j = 1; j < nums.size(); ++j) {
            if (nums[j] != nums[j - 1]) {
                i++;
                nums[i] = nums[j];
            }
        }
        return i + 1;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: 双指针
- 标签: 数组, 双指针
- 难度: 简单

28. 实现 strStr()

实现 strStr() 函数。

给你两个字符串 haystack 和 needle，请你在 haystack 字符串中找出 needle 字符串出现的第一个位置（下标从 0 开始）。如果不存在，则返回 -1。

说明：

当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。

对于本题而言，当 needle 是空字符串时我们应当返回 0。这与 C 语言的 strstr() 以及 Java 的 indexOf() 定义相符。

```

class Solution {
public:
    int strStr(string haystack, string needle) {
        for (int i = 0; ; ++i) {
            for (int j = 0; ; ++j) {
                if (j == needle.size()) return i;
                if (i + j == haystack.size()) return -1;
                if (haystack[i + j] != needle[j]) break;
            }
        }
    }
};

```

FIXME: KMP 解法

- 空间复杂度: O(1)
- 时间复杂度:
- 解法: 双指针
- 标签: 双指针, 字符串, 字符匹配
- 难度: 简单

29. 两数相除

给定两个整数，被除数 dividend 和除数 divisor。将两数相除，要求不使用乘法、除法和 mod 运算符。

返回被除数 dividend 除以除数 divisor 得到的商。

整数除法的结果应当截去 (truncate) 其小数部分, 例如: $\text{truncate}(8.345) = 8$ 以及 $\text{truncate}(-2.7335) = -2$

```
class Solution {
public:
    int divide(int dividend, int divisor) {
        if (dividend == std::numeric_limits<int>::min() && divisor == -1) return std::numeric_limits<int>::max();
        long int dvd = std::labs(dividend), dvs = std::labs(divisor), res = 0;
        while (dvd >= dvs) {
            long int tmp = dvs, m = 1;
            while (tmp << 1 <= dvd) {
                tmp <<= 1;
                m <<= 1;
            }
            dvd -= tmp;
            res += m;
        }
        return (dividend > 0) == (divisor > 0) ? res : -res;
    }
};
```

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 位运算, 数学
- 难度: 中等

34. 在排序数组中查找元素的第一个和最后一个位置

给定一个按照升序排列的整数数组 nums , 和一个目标值 target 。找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值 target , 返回 $[-1, -1]$ 。

进阶:

你可以设计并实现时间复杂度为 $O(\log n)$ 的算法解决此问题吗?

示例 1:

输入: $\text{nums} = [5,7,7,8,8,10]$, $\text{target} = 8$

输出: $[3,4]$

示例 2:

输入: $\text{nums} = [5,7,7,8,8,10]$, $\text{target} = 6$

输出: $[-1,-1]$

示例 3:

输入: $\text{nums} = []$, $\text{target} = 0$

输出: $[-1,-1]$

```

class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        if (nums.size() == 0) return { -1, -1 };
        int left = 0, right = nums.size() - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) {
                left = mid;
                break;
            } else if (nums[mid] > target) {
                right = mid - 1;
            } else { // nums[mid] < target
                left = mid + 1;
            }
        }
        if (nums[left] != target) return { -1, -1 }; // not found
        right = left;
        while (left > 0 && nums[left - 1] == target) left--;
        while (right < nums.size() - 1 && nums[right + 1] == target) right++;
        return { left, right };
    }
};

class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int idxLeft = lower_bound(nums, target),
            idxRight = lower_bound(nums, target + 1) - 1;
        if (idxLeft < nums.size() && nums[idxLeft] == target) {
            return { idxLeft, idxRight };
        } else {
            return { -1, -1 };
        }
    }

private:
    int lower_bound(const std::vector<int>& nums, int target) {
        int l = 0, r = nums.size() - 1;
        while (l <= r) {
            int mid = l + (r - l) / 2;
            nums[mid] < target ? l = mid + 1 : r = mid - 1;
        }
        return l;
    }
};

/** 
 * 解法:
 * 1. 搜索一次, 然后while左右循环找相同的值
 * 2. 搜索两次, 分别找最左idx和最右idx
 * 3. 先在范围 (0, nums.size() - 1) 二分搜索一次结果为idxLeft, 然后再在范围 (idxLeft, nums.size() - 1) 二分搜索一次结果为idxRight
 */

```

- 空间复杂度: O(1)
- 时间复杂度: O(log n)
- 解法: 二分查找
- 标签: 数组, 二分查找
- 难度: 中等

33. 搜索旋转排序数组

整数数组 nums 按升序排列，数组中的值互不相同。

在传递给函数之前，`nums` 在预先未知的某个下标 k ($0 \leq k < \text{nums.length}$) 上进行了旋转，使数组变为 $[\text{nums}[k], \text{nums}[k+1], \dots, \text{nums}[\text{n}-1], \text{nums}[0], \text{nums}[1], \dots, \text{nums}[\text{k}-1]]$ (下标从 0 开始计数)。例如，`[0,1,2,4,5,6,7]` 在下标 3 处经旋转后可能变为 `[4,5,6,7,0,1,2]`。

给你旋转后的数组 `nums` 和一个整数 `target`，如果 `nums` 中存在这个目标值 `target`，则返回它的下标，否则返回 -1。

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int low = 0, high = nums.size() - 1;
        while (low < high) {
            int mid = low + (high - low) / 2;
            if (nums[mid] == target) return mid;
            if (nums[low] <= nums[mid]) {
                if (nums[low] <= target && target < nums[mid]) {
                    high = mid - 1;
                } else {
                    low = mid + 1;
                }
            } else {
                if (nums[mid] < target && target <= nums[high]) {
                    low = mid + 1;
                } else {
                    high = mid - 1;
                }
            }
        }
        return nums[low] == target ? low : -1;
    }
};
```

- 空间复杂度: $O(1)$
- 时间复杂度: $O(\log n)$
- 解法:
- 标签: 数组, 二分查找
- 难度: 中等

36. 有效的数独

请你判断一个 9×9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。

数字 1-9 在每一行只能出现一次。

数字 1-9 在每一列只能出现一次。

数字 1-9 在每一个以粗实线分隔的 3×3 宫内只能出现一次。（请参考示例图）

数独部分空格内已填入了数字，空白格用 '!' 表示。

注意：

一个有效的数独（部分已被填充）不一定是可解的。

只需要根据以上规则，验证已经填入的数字是否有效即可。

```

class Solution {
public:
    bool isValidSudoku(vector<vector<char>>& board) {
        std::vector<std::vector<bool>> checkRow(9, std::vector<bool>(9, false));
        std::vector<std::vector<bool>> checkCol(9, std::vector<bool>(9, false));
        std::vector<std::vector<bool>> checkMat(9, std::vector<bool>(9, false));
        for (int i = 0; i < 9; ++i) {
            for (int j = 0; j < 9; ++j) {
                if (board[i][j] == '.') continue;
                int num = board[i][j] - '0' - 1, k = i / 3 * 3 + j / 3;
                if (checkRow[i][num] || checkCol[j][num] || checkMat[k][num]) return false;
                checkRow[i][num] = true;
                checkCol[j][num] = true;
                checkMat[k][num] = true;
            }
        }
        return true;
    }
};

```

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 数组, 哈希表, 矩阵
- 难度: 中等

38. 外观数列

给定一个正整数 n，输出外观数列的第 n 项。

「外观数列」是一个整数序列，从数字 1 开始，序列中的每一项都是对前一项的描述。

你可以将其视作是由递归公式定义的数字字符串序列：

```

countAndSay(1) = "1"
countAndSay(n) 是对 countAndSay(n-1) 的描述，然后转换成另一个数字字符串。

```

前五项如下：

1.	1
2.	11
3.	21
4.	1211
5.	111221

第一项是数字 1

描述前一项，这个数是 1 即“一个 1”，记作 "11"

描述前一项，这个数是 11 即“二个 1”，记作 "21"

描述前一项，这个数是 21 即“一个 2 + 一个 1”，记作 "1211"

描述前一项，这个数是 1211 即“一个 1 + 一个 2 + 二个 1”，记作 "111221"

要描述一个数字字符串，首先要将字符串分割为最小数量的组，每个组都由连续的最多相同字符组成。然后对于每个组，先描述字符的数量，然后描述字符，形成一个描述组。要将描述转换为数字字符串，先将每组中的字符数量用数字替换，再将所有描述组连接起来。

```

class Solution {
public:
    string countAndSay(int n) {
        std::string res = "1";
        while (--n) {
            std::string tmp = "";
            for (int i = 0; i < res.size(); ++i) {
                int cnt = 1;
                while ((i + 1 < res.size()) && (res[i] == res[i + 1])) {
                    cnt++;
                    i++;
                }
                tmp += std::to_string(cnt) + res[i];
            }
            res = tmp;
        }
        return res;
    }
};

```

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 字符串
- 难度: 中等

41. 缺失的第一个正数

TODO:

42. 接雨水

TODO:

44. 通配符匹配

TODO:

46. 全排列

给定一个不含重复数字的数组 `nums`，返回其所有可能的全排列。你可以按任意顺序返回答案。

```

// 基于原有的，有两个元素换位置都会产生新的排列
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        std::vector<std::vector<int>> res;
        permute(nums, res, 0);
        return res;
    }

private:
    void permute(std::vector<int>& nums,
                 std::vector<std::vector<int>>& res,
                 int begin) {
        if (begin == nums.size()) {
            res.push_back(nums);
            return ;
        }
        for (int i = begin; i < nums.size(); ++i) {
            std::swap(nums[begin], nums[i]);
            permute(nums, res, begin + 1);
            std::swap(nums[begin], nums[i]);
        }
    }
};


```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n^2)$
- 解法:
- 标签: 数组, 回溯
- 难度: 中等

48. 旋转图像

给定一个 $n \times n$ 的二维矩阵 matrix 表示一个图像。请你将图像顺时针旋转 90 度。

你必须在 原地 旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要 使用另一个矩阵来旋转图像。

```

class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        transpose(matrix);
        reverse(matrix);
        return;
    }

private:
    void reverse(std::vector<std::vector<int>>& matrix) { // 左右对称
        for (int i = 0; i < matrix.size(); ++i) {
            for (int j = 0; j < matrix[0].size() / 2; ++j) {
                std::swap(matrix[i][j], matrix[i][matrix[0].size() - j - 1]);
            }
        }
    }

    void transpose(std::vector<std::vector<int>>& matrix) { // 左上角到右下角为对称线条
        for (int i = 0; i < matrix.size(); ++i) {
            for (int j = i; j < matrix[0].size(); ++j) {
                std::swap(matrix[i][j], matrix[j][i]);
            }
        }
    }
};


```

- 空间复杂度: $O(1)$

- 时间复杂度: $O(n^2)$
- 解法:
- 标签: 数组, 数学, 矩阵
- 难度: 中等

49. 字母异位词分组

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

示例:

输入: ["eat", "tea", "tan", "ate", "nat", "bat"]

输出:

```
[  
    ["ate", "eat", "tea"],  
    ["nat", "tan"],  
    ["bat"]  
]
```

说明:

所有输入均为小写字母。

不考虑答案输出的顺序。

```
class Solution {  
public:  
    vector<vector<string>> groupAnagrams(vector<string>& strs) {  
        std::unordered_map<std::string, std::vector<std::string>> mp;  
        for (const std::string& str : strs) {  
            std::string tmp = str;  
            std::sort(tmp.begin(), tmp.end());  
            mp[tmp].push_back(str);  
        }  
        std::vector<std::vector<std::string>> res;  
        for (const auto& p : mp) res.push_back(p.second);  
        return res;  
    }  
};
```

- 空间复杂度:
- 时间复杂度:
- 解法: map + sort
- 标签: 哈希表, 字符串, 排序
- 难度: 中等

50. Pow(x, n)

实现 $\text{pow}(x, n)$ ，即计算 x 的 n 次幂函数（即, x^n ）。

示例 1:

输入: $x = 2.00000, n = 10$

输出: 1024.00000

示例 2:

输入: $x = 2.10000, n = 3$

输出: 9.26100

示例 3:

输入: $x = 2.00000, n = -2$
输出: 0.25000
解释: $2^{-2} = 1/(2^2) = 1/4 = 0.25$

```
class Solution {
public:
    double myPow(double x, int n) {
        if (n == 0) return 1.0f;
        if (n == std::numeric_limits<int>::min()) return myPow(x, std::numeric_limits<int>::min() + 1) / x;
        if (n < 0) {
            n = -n;
            x = 1 / x;
        }
        return myPow(x * x, n / 2) * (n % 2 ? x : 1.0);
    }
};
```

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 递归, 数学
- 难度: 中等

53. 最大子序和

给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例 1:

输入: $\text{nums} = [-2,1,-3,4,-1,2,1,-5,4]$
输出: 6
解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。

进阶: 如果你已经实现复杂度为 $O(n)$ 的解法，尝试使用更为精妙的分治法求解。

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        std::vector<int> dp(nums.size(), nums[0]);
        int mx = nums[0];
        for (int i = 1; i < nums.size(); ++i) {
            dp[i] = nums[i] + std::max(dp[i - 1], 0);
            mx = std::max(mx, dp[i]);
        }
        return mx;
    }
};

// 上边的dp空间优化到O(1)
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int i = nums[0], j = nums[0];
        for (int k = 1; k < nums.size(); ++k) {
            i = std::max(nums[k] + i, nums[k]);
            j = std::max(j, i);
        }
        return j;
    }
};
```

- 空间复杂度: $O(1)$

- 时间复杂度: $O(n)$
- 解法: dp
- 标签: 数组, 分治, 动态规划
- 难度: 简单

54. 螺旋矩阵

给你一个 m 行 n 列的矩阵 matrix，请按照 顺时针螺旋顺序，返回矩阵中的所有元素。

示例 1:

输入: matrix = `[1,2,3],[4,5,6],[7,8,9]`

输出: [1,2,3,6,9,8,7,4,5]

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        std::vector<int> res;
        if (matrix.size() == 0) return res;
        int rowUp = 0, rowDown = matrix.size() - 1;
        int colLeft = 0, colRight = matrix[0].size() - 1;
        while (rowUp <= rowDown && colLeft <= colRight) {
            // left -> right
            for (int col = colLeft; col <= colRight; ++col) res.push_back(matrix[rowUp][col]);
            // up -> down
            for (int row = rowUp + 1; row <= rowDown; ++row) res.push_back(matrix[row][colRight]);
            if (rowUp < rowDown && colLeft < colRight) {
                // right -> left
                for (int col = colRight - 1; col > colLeft; --col) res.push_back(matrix[rowDown][col]);
                // down -> up
                for (int row = rowDown; row > rowUp; --row) res.push_back(matrix[row][colLeft]);
            }
            rowUp++;
            rowDown--;
            colLeft++;
            colRight--;
        }
        return res;
    }
};
```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法: 模拟
- 标签: 数组, 矩阵, 模拟
- 难度: 中等

55. 跳跃游戏

给定一个非负整数数组 nums，你最初位于数组的第一个下标。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标。

示例 1:

输入: nums = [2,3,1,1,4]

输出: true

解释：可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。

示例 2：

输入： nums = [3,2,1,0,4]

输出： false

解释：无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0，所以永远不可能到达最后一个下标。

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int curMax = nums[0];
        for (int i = 1; i < nums.size(); ++i) {
            if (curMax < i) return false;
            curMax = std::max(curMax, i + nums[i]);
        }
        return true;
    }
};
```

- 空间复杂度： O(1)
- 时间复杂度： O(n)
- 解法： 贪心 + 动态规划
- 标签： 贪心，数组，动态规划
- 难度： 中等

56. 合并区间

以数组 intervals 表示若干个区间的集合，其中单个区间为 intervals[i] = [starti, endi]。请你合并所有重叠的区间，并返回一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。

示例 1：

输入： intervals = [1,3],[2,6],[8,10],[15,18]

输出： [1,6],[8,10],[15,18]

解释： 区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6]。

示例 2：

输入： intervals = [1,4],[4,5]

输出： [1,5]

解释： 区间 [1,4] 和 [4,5] 可被视为重叠区间。

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        std::sort(intervals.begin(), intervals.end(),
                  [] (const std::vector<int>& a, const std::vector<int>& b) { return a[0] < b[0]; });
        std::vector<std::vector<int>> result;
        for (const std::vector<int>& interval : intervals) {
            if (result.empty() || result.back()[1] < interval[0]) {
                result.push_back(interval);
            } else {
                result.back()[1] = std::max(result.back()[1], interval[1]);
            }
        }
        return result;
    }
};
```

- 空间复杂度： O(log n)
- 时间复杂度： O(n log n)

- 解法: 排序
- 标签: 数组, 排序
- 难度: 中等

62. 不同路径

一个机器人位于一个 $m \times n$ 网格的左上角 (起始点在下图中标记为“Start”)。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角 (在下图中标记为“Finish”)

问总共有多少条不同的路径?

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        std::vector<int> dp(n, 1);
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                dp[j] += dp[j - 1];
            }
        }
        return dp[n - 1];
    }
};
```

FIXME: 组合数学 解法

- 空间复杂度: $O(1)$
- 时间复杂度:
- 解法: dp, math
- 标签: 数学, 动态规划, 组合数学
- 难度: 中等

66. 加一

给定一个由整数组成的非空数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位，数组中每个元素只存储单个数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

```
class Solution {
public:
    vector<int> plusOne(vector<int>& digits) {
        bool isCarry = true;
        for (int i = digits.size() - 1; i >= 0; --i) {
            digits[i] += isCarry;
            isCarry = digits[i] > 9;
            digits[i] %= 10;
        }
        if (isCarry) {
            digits.insert(digits.begin(), 1);
        }
        return digits;
    }
};
```

- 空间复杂度: $O(1)$
- 时间复杂度: $O(n)$
- 解法:

- 标签: 数组, 数学
- 难度: 简单

69. x 的平方根

实现 int sqrt(int x) 函数。

计算并返回 x 的平方根, 其中 x 是非负整数。

由于返回类型是整数, 结果只保留整数的部分, 小数部分将被舍去。

示例 1:

输入: 4

输出: 2

示例 2:

输入: 8

输出: 2

说明: 8 的平方根是 2.82842...,

由于返回类型是整数, 小数部分将被舍去。

```
class Solution {
public:
    int mySqrt(int x) {
        if (x == 0) return 0;
        int l = 1, r = x / 2, mid;
        while (l < r) {
            mid = l + (r - l) / 2;
            if (mid <= x / mid && (mid + 1) > x / (mid + 1)) {
                return mid;
            } else if (mid > x / mid) {
                r = mid - 1;
            } else if (mid < x / mid) {
                l = mid + 1;
            }
        }
        return l;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(log n)
- 解法: 二分查找
- 标签: 数学, 二分查找
- 难度: 简单

70. 爬楼梯

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

注意: 给定 n 是一个正整数。

```

// dp
// 空间O(n), 时间O(n)
class Solution {
public:
    int climbStairs(int n) {
        if (n <= 2) return n;
        std::vector<int> arr(n);
        arr[0] = 1; // 爬一个台阶有一种方法
        arr[1] = 2; // 爬两个台阶有两种方法
        for (int i = 2; i < n; ++i) arr[i] = arr[i - 1] + arr[i - 2];
        return arr[n - 1];
    }
};

// dp空间优化
// 空间O(1), 时间O(n)
class Solution {
public:
    int climbStairs(int n) {
        if (n <= 2) return n;
        int pre = 1, cur = 2, tmp = 0;
        for (int i = 2; i < n; ++i) {
            tmp = pre + cur;
            pre = cur;
            cur = tmp;
        }
        return cur;
    }
};

// 递归+mem
class Solution {
public:
    int climbStairs(int n) {
        if (n <= 2) return n;
        std::vector<int> mem(n + 1, -1);
        mem[1] = 1;
        mem[2] = 2;
        return recursion(n, mem);
    }
};

private:
    int recursion(int n, std::vector<int>& mem) {
        if (mem[n] != -1) return mem[n];
        mem[n] = recursion(n - 1, mem) + recursion(n - 2, mem);
        return mem[n];
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n); 最优是O(log n), 用的数学fib函数
- 解法: dp, 递归+mem
- 标签: 记忆化搜索, 数学, 动态规划
- 难度: 简单

73. 矩阵置零

给定一个 $m \times n$ 的矩阵，如果一个元素为 0，则将其所在行和列的所有元素都设为 0。请使用 原地 算法。

进阶:

一个直观的解决方案是使用 $O(mn)$ 的额外空间，但这并不是一个好的解决方案。

一个简单的改进方案是使用 $O(m + n)$ 的额外空间，但这仍然不是最好的解决方案。

你能想出一个仅使用常量空间的解决方案吗？

```

class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        std::vector<bool> flagRowZeroIdx(matrix.size(), false);
        std::vector<bool> flagColZeroIdx(matrix[0].size(), false);
        for (int row = 0; row < matrix.size(); ++row) {
            for (int col = 0; col < matrix[0].size(); ++col) {
                if (matrix[row][col] == 0) {
                    flagRowZeroIdx[row] = true;
                    flagColZeroIdx[col] = true;
                }
            }
        }
        for (int row = 0; row < matrix.size(); ++row) {
            if (flagRowZeroIdx[row]) {
                for (int col = 0; col < matrix[0].size(); ++col) {
                    matrix[row][col] = 0;
                }
            }
        }
        for (int col = 0; col < matrix[0].size(); ++col) {
            if (flagColZeroIdx[col]) {
                for (int row = 0; row < matrix.size(); ++row) {
                    matrix[row][col] = 0;
                }
            }
        }
    }
    return ;
}
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(row * col)
- 解法:
- 标签: 数组, 哈希表, 矩阵
- 难度: 中等

75. 颜色分类

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。

```

class Solution {
public:
    void sortColors(vector<int>& nums) {
        int second = nums.size() - 1, zero = 0;
        for (int i = 0; i <= second; ++i) {
            while (nums[i] == 2 && i < second) {
                std::swap(nums[i], nums[second]);
                second--;
            }
            while (nums[i] == 0 && i > zero) {
                std::swap(nums[i], nums[zero]);
                zero++;
            }
        }
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: 双指针

- 标签: 数组, 双指针, 排序
- 难度: 中等

76. 最小覆盖子串

TODO:

78. 子集

给你一个整数数组 `nums`，数组中的元素互不相同。返回该数组所有可能的子集（幂集）。

解集不能包含重复的子集。你可以按任意顺序返回解集。

```
// 回溯
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        std::vector<std::vector<int>> res;
        std::vector<int> cur;
        backtracking(nums, res, 0, cur);
        return res;
    }

private:
    void backtracking(std::vector<int>& nums,
                      std::vector<std::vector<int>>& res,
                      int idx,
                      std::vector<int>& cur) {
        if (idx >= nums.size()) {
            res.push_back(cur);
            return;
        }
        backtracking(nums, res, idx + 1, cur);
        cur.push_back(nums[idx]);
        backtracking(nums, res, idx + 1, cur);
        cur.pop_back();
    }
};

// 位运算
// 集合长度为n, 子集数量为2^n个
// [], [ ], [ ], [   ], [   ], [     ], [       ]
// [], [1], [ ], [1   ], [1   ], [1     ], [1       ]
// [], [1], [2], [1, 2], [ ], [1   ], [2   ], [1, 2   ]
// [], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]
// 以[1, 2, 3]为例, 1在每两个连续子集中出现一次, 2在每四个连续子集中出现两次, 3在每八个子集中出现四次 (最初所有子集都是空的)
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        int p = 1 << nums.size(); // 初始化子集数量
        std::vector<std::vector<int>> res(p);
        for (int i = 0; i < p; ++i) { // 每个子集迭代
            for (int j = 0; j < nums.size(); ++j) { // 每个元素迭代
                if ((i >> j) & 1) {
                    res[i].push_back(nums[j]);
                }
            }
        }
        return res;
    }
};
```

- 空间复杂度: $O(n)$

- 时间复杂度: $O(n^2)$
- 解法: 回溯, 位运算
- 标签: 位运算, 数组, 回溯
- 难度: 中等

79. 单词搜索

给定一个 $m \times n$ 二维字符网格 board 和一个字符串单词 word。如果 word 存在于网格中，返回 true；否则，返回 false。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

```
class Solution {
public:
    bool exist(vector<vector<char>>& board, string word) {
        for (int i = 0; i < board.size(); ++i) {
            for (int j = 0; j < board[0].size(); ++j) {
                if (dfs(board, word, 0, i, j)) return true;
            }
        }
        return false;
    }

private:
    // dfs + backtracking
    bool dfs(std::vector<std::vector<char>>& board, const std::string& word,
             int idx, int i, int j) {
        if (i < 0 || j < 0 || i >= board.size() || j >= board[0].size() || board[i][j] != word[idx]) return false;
        if (idx == word.size() - 1) return true;
        char cur = board[i][j];
        board[i][j] = '*'; // used
        bool found = dfs(board, word, idx + 1, i + 1, j)
                    || dfs(board, word, idx + 1, i - 1, j)
                    || dfs(board, word, idx + 1, i, j + 1)
                    || dfs(board, word, idx + 1, i, j - 1);
        board[i][j] = cur; // reset
        return found;
    }
};
```

- 空间复杂度:
- 时间复杂度:
- 解法: dfs+回溯
- 标签: 数组, 回溯, 矩阵
- 难度: 中等

84. 柱状图中最大的矩形

TODO:

88. 合并两个有序数组

给你两个有序整数数组 nums1 和 nums2，请你将 nums2 合并到 nums1 中，使 nums1 成为一个有序数组。

初始化 nums1 和 nums2 的元素数量分别为 m 和 n。你可以假设 nums1 的空间大小等于 $m + n$ ，这样它就有足够的空间保存来自 nums2 的元素。

示例 1:

输入: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3

输出: [1,2,2,3,5,6]

示例 2:

输入: nums1 = [1], m = 1, nums2 = [], n = 0

输出: [1]

```
class Solution {
public:
    void merge(vector<int>& A, int m, vector<int>& B, int n) {
        int a = m - 1, // nums1的数组最后一位索引
            b = n - 1, // nums2的数组最后一位索引
            i = m + n - 1; // 合并后数组的最后一一位
        while (a >= 0 && b >= 0) {
            if (A[a] > B[b]) A[i--] = A[a--];
            else A[i--] = B[b--];
        }
        while (b >= 0) A[i--] = B[b--];
    }
};
```

- 空间复杂度:
- 时间复杂度:
- 解法: 归并排序
- 标签: 数组, 双指针, 排序
- 难度: 简单

91. 解码方法

一条包含字母 A-Z 的消息通过以下映射进行了 编码 :

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

要解码 已编码的消息, 所有数字必须基于上述映射的方法, 反向映射回字母 (可能有多种方法)。例如, "11106" 可以映射为:

"AAJF" , 将消息分组为 (1 1 10 6)

"KJF" , 将消息分组为 (11 10 6)

注意, 消息不能分组为 (1 11 06) , 因为 "06" 不能映射为 "F" , 这是由于 "6" 和 "06" 在映射中并不等价。

给你一个只含数字的 非空 字符串 s , 请计算并返回 解码 方法的 总数 。

题目数据保证答案肯定是一个 32 位 的整数。

```
class Solution {
public:
    int numDecodings(string s) {
        if (s.empty() || s[0] == '0') return 0;
        int pre1 = 1, pre2 = 1, cur = 1;
        for (int i = 1; i < s.size(); ++i) {
            cur = 0;
            int first = s[i] - '0', second = std::stoi(s.substr(i - 1, 2));
            if (1 <= first && first <= 9) cur += pre1;
            if (10 <= second && second <= 26) cur += pre2;
            pre2 = pre1;
            pre1 = cur;
        }
        return cur;
    }
};
```

- 空间复杂度: $O(1)$
- 时间复杂度: $O(n)$
- 解法: dp
- 标签: 字符串, 动态规划
- 难度: 中等

94. 二叉树的中序遍历

给定一个二叉树的根节点 root，返回它的 中序 遍历。

```

// 递归
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        std::vector<int> res;
        recursion(root, res);
        return res;
    }

private:
    void recursion(TreeNode* node, std::vector<int>& vec) {
        if (node == NULL) return;
        if (node->left != NULL) recursion(node->left, vec);
        vec.push_back(node->val);
        if (node->right != NULL) recursion(node->right, vec);
    }
};

// 迭代
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        std::stack<TreeNode*> stk;
        std::vector<int> res;
        TreeNode* cur = root;
        while (cur != NULL || !stk.empty()) {
            while (cur != NULL) {
                stk.push(cur);
                cur = cur->left;
            }
            cur = stk.top();
            stk.pop();
            res.push_back(cur->val);
            cur = cur->right;
        }
        return res;
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法: 递归, 迭代
- 标签: 栈, 树, 深度优先搜索, 二叉树
- 难度: 简单

98. 验证二叉搜索树

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

节点的左子树只包含小于当前节点的数。

节点的右子树只包含大于当前节点的数。

所有左子树和右子树自身必须也是二叉搜索树。

```

// 递归
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        return isValidBST(root, std::numeric_limits<long int>::min(), std::numeric_limits<long int>::max());
    }

private:
    bool isValidBST(TreeNode* root, long int minVal, long int maxVal) {
        if (root == nullptr) return true;
        if (root->val >= maxVal || root->val <= minVal) return false;
        return isValidBST(root->left, minVal, root->val) && isValidBST(root->right, root->val, maxVal);
    }
};

// 迭代
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        if (root == nullptr) return true;
        std::stack<TreeNode*> stk;
        TreeNode* pre = nullptr;
        while (root != nullptr || !stk.empty()) {
            while (root != nullptr) {
                stk.push(root);
                root = root->left;
            }
            root = stk.top();
            stk.pop();
            if (pre != nullptr && pre->val >= root->val) return false;
            pre = root;
            root = root->right;
        }
        return true;
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法: 递归, 迭代
- 标签: 树, 深度优先搜索, 二叉搜索树, 二叉树
- 难度: 中等

101. 对称二叉树

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。



但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的：



进阶：你可以运用递归和迭代两种方法解决这个问题吗？

```

// 递归
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        return recursion(root->left, root->right);
    }

private:
    bool recursion(TreeNode* left, TreeNode* right) {
        if (left == NULL && right == NULL) return true;
        if (left == NULL || right == NULL) return false;
        return (left->val == right->val) && recursion(left->left, right->right) && recursion(left->right, right->left);
    }
};

// 迭代
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        std::stack<TreeNode*> stk;
        stk.push(root);
        stk.push(root);
        TreeNode* node1;
        TreeNode* node2;
        while (!stk.empty()) {
            node1 = stk.top();
            stk.pop();
            node2 = stk.top();
            stk.pop();
            if (node1 == NULL && node2 == NULL) continue;
            if (node1 == NULL || node2 == NULL) return false;
            if (node1->val != node2->val) return false;
            stk.push(node1->left);
            stk.push(node2->right);
            stk.push(node1->right);
            stk.push(node2->left);
        }
        return true;
    }
};

```

- 空间复杂度:
- 时间复杂度: $O(n)$
- 解法:
- 标签: 树, 深度优先搜索, 广度优先搜索, 二叉树

- 难度： 简单

102. 二叉树的层序遍历

给你一个二叉树，请你返回其按 层序遍历 得到的节点值。 （即逐层地，从左到右访问所有节点）。

```

// 迭代，按层遍历
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        std::vector<std::vector<int>> res;
        if (root == NULL) return res;
        std::vector<TreeNode*> lyr = {root};
        while (!lyr.empty()) {
            std::vector<int> lyrVal;
            std::vector<TreeNode*> lyrNxt;
            for (TreeNode* node : lyr) {
                lyrVal.push_back(node->val);
                if (node->left != NULL) lyrNxt.push_back(node->left);
                if (node->right != NULL) lyrNxt.push_back(node->right);
            }
            lyr = lyrNxt;
            res.push_back(lyrVal);
        }
        return res;
    }
};

// 递归
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        std::vector<std::vector<int>> res;
        buildLyr(root, 0, res);
        return res;
    }

private:
    void buildLyr(TreeNode* node, int depth, std::vector<std::vector<int>>& vec) {
        if (node == NULL) return;
        if (vec.size() == depth) vec.push_back(std::vector<int>());
        vec[depth].push_back(node->val);
        buildLyr(node->left, depth + 1, vec);
        buildLyr(node->right, depth + 1, vec);
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法: 递归, 迭代
- 标签: 树, 广度优先搜索, 二叉树
- 难度: 中等

103. 二叉树的锯齿形层序遍历

给定一个二叉树，返回其节点值的锯齿形层序遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

例如：

给定二叉树 [3,9,20,null,null,15,7]，



返回锯齿形层序遍历如下：

```
[  
    [3],  
    [20, 9],  
    [15, 7]  
]
```

```
/**  
 * Definition for a binary tree node.  
 * struct TreeNode {  
 *     int val;  
 *     TreeNode *left;  
 *     TreeNode *right;  
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}  
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}  
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}  
 * };  
 */  
class Solution {  
public:  
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {  
        std::vector<std::vector<int>> res;  
        if (root == nullptr) return res;  
        std::vector<TreeNode*> layer = {root};  
        while (!layer.empty()) {  
            std::vector<TreeNode*> nextLayer;  
            std::vector<int> layerValues;  
            for (TreeNode* node : layer) {  
                layerValues.push_back(node->val);  
                if (node->left != nullptr) nextLayer.push_back(node->left);  
                if (node->right != nullptr) nextLayer.push_back(node->right);  
            }  
            layer = nextLayer;  
            res.push_back(layerValues);  
        }  
        for (int i = 1; i < res.size(); i += 2) {  
            for (int j = 0; j < res[i].size() / 2; ++j) {  
                std::swap(res[i][j], res[i][res[i].size() - j - 1]);  
            }  
        }  
        return res;  
    }  
};
```

- 空间复杂度：
- 时间复杂度：
- 解法：广度优先搜索-迭代
- 标签：树，广度优先搜索，二叉树
- 难度：中等

104. 二叉树的最大深度

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

示例:

给定二叉树 [3,9,20,null,null,15,7],



返回它的最大深度 3。

```
/*
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == NULL) return 0;
        return std::max(maxDepth(root->left), maxDepth(root->right)) + 1;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度:
- 解法: 递归
- 标签: 树, 深度优先搜索, 广度优先搜索, 二叉树
- 难度: 简单

105. 从前序与中序遍历序列构造二叉树

根据一棵树的前序遍历与中序遍历构造二叉树。

注意: 你可以假设树中没有重复的元素。

例如, 给出

前序遍历 preorder = [3,9,20,15,7]

中序遍历 inorder = [9,3,15,20,7]

返回如下的二叉树:

```
3
 / \
9  20
 /   \
15   7
```

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
        if inorder:
            idx = inorder.index(preorder.pop(0))
            root = TreeNode(inorder[idx])
            root.left = self.buildTree(preorder, inorder[0:idx])
            root.right = self.buildTree(preorder, inorder[idx + 1:])
        return root
```

FIXME: cpp

- 空间复杂度:
- 时间复杂度:
- 解法: 递归
- 标签: 树, 数组, 哈希表, 分治, 二叉树
- 难度: 中等

108. 将有序数组转换为二叉搜索树

给你一个整数数组 nums，其中元素已经按 升序 排列，请你将其转换为一棵 高度平衡 二叉搜索树。

高度平衡 二叉树是一棵满足「每个节点的左右两个子树的高度差的绝对值不超过 1」的二叉树。

输入: nums = [-10,-3,0,5,9]

输出: [0,-3,9,-10,null,5]

解释: [0,-10,5,null,-3,null,9] 也将被视为正确答案

输入: nums = [1,3]

输出: [3,1]

解释: [1,3] 和 [3,1] 都是高度平衡二叉搜索树。

提示:

$1 \leq \text{nums.length} \leq 10^4$

$-10^4 \leq \text{nums}[i] \leq 10^4$

nums 按 严格递增 顺序排列

```

// 递归+分治
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        if (nums.size() == 0) return NULL;
        return helper(nums, 0, nums.size() - 1);
    }

private:
    TreeNode* helper(const std::vector<int>& nums, int left, int right) {
        if (left > right) return NULL;
        int mid = left + (right - left) / 2;
        TreeNode* node = new TreeNode(nums[mid]);
        node->left = helper(nums, left, mid - 1);
        node->right = helper(nums, mid + 1, right);
        return node;
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法: 递归+分治
- 标签: 树, 二叉搜索树, 数组, 分治, 二叉树
- 难度: 简单

116. 填充每个节点的下一个右侧节点指针

给定一个 完美二叉树，其所有叶子节点都在同一层，每个父节点都有两个子节点。二叉树定义如下：

```

struct Node {
int val;
Node *left;
Node *right;
Node *next;
}

```

填充它的每个 `next` 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 `next` 指针设置为 `NULL`。

初始状态下，所有 `next` 指针都被设置为 `NULL`。

进阶：

你只能使用常量级额外空间。

使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

示例：

输入: `root = [1,2,3,4,5,6,7]`

输出: `[1,#,2,3,#,4,5,6,7,#]`

解释：给定二叉树如图 A 所示，你的函数应该填充它的每个 `next` 指针，以指向其下一个右侧节点，如图 B 所示。序列化的输出按层序遍历排列，同一层节点由 `next` 指针连接，'#' 标志着每一层的结束。

```

/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* left;
    Node* right;
    Node* next;

    Node() : val(0), left(NULL), right(NULL), next(NULL) {}

    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}

    Node(int _val, Node* _left, Node* _right, Node* _next)
        : val(_val), left(_left), right(_right), next(_next) {}
};

*/
// dfs
class Solution {
public:
    Node* connect(Node* root) {
        dfs(root, nullptr);
        return root;
    }

private:
    void dfs(Node* cur, Node* nxt) {
        if (cur == nullptr) return;
        cur->next = nxt;
        dfs(cur->left, cur->right); // 示例中的节点2和 (节点4, 5, 6) 的情况
        dfs(cur->right, cur->next == nullptr ? nullptr : cur->next->left); // 示例中的节点5的情况
    }
};

// bfs
class Solution {
public:
    Node* connect(Node* root) {
        if (root == nullptr) return root;
        Node* pre = root;
        Node* cur = nullptr;
        while (pre->left) {
            cur = pre; // 分层遍历, cur是行中哪一个的指针
            while (cur) {
                cur->left->next = cur->right;
                if (cur->next) cur->right->next = cur->next->left; // node2指向node3的话, node2->node5->next = node5->next
                cur = cur->next;
            }
            pre = pre->left; // pre是指向那一行的指针
        }
        return root;
    }
};

```

- 空间复杂度:
- 时间复杂度:
- 解法: dfs, bfs
- 标签: 树, 深度优先搜索, 广度优先搜索, 二叉树
- 难度: 中等

118. 杨辉三角

在杨辉三角中, 每个数是它左上方和右上方的数的和。

示例:

输入: 5

输出:

```
[  
    [1],  
    [1,1],  
    [1,2,1],  
    [1,3,3,1],  
    [1,4,6,4,1]  
]
```

```
// 模拟  
class Solution {  
public:  
    vector<vector<int>> generate(int numRows) {  
        std::vector<std::vector<int>> res;  
        for (int i = 1; i <= numRows; ++i) {  
            std::vector<int> lyr;  
            if (i == 1) {  
                lyr.push_back(1);  
            } else if (i == 2) {  
                lyr.push_back(1);  
                lyr.push_back(1);  
            } else {  
                lyr.push_back(1);  
                for (int j = 1; j <= i - 2; ++j) lyr.push_back(res[i - 2][j - 1] + res[i - 2][j]);  
                lyr.push_back(1);  
            }  
            res.push_back(lyr);  
        }  
        return res;  
    }  
};
```

- 空间复杂度: $O(n^2)$
- 时间复杂度: $O(n^2)$
- 解法: dp, 模拟
- 标签: 数组, 动态规划
- 难度: 简单

121. 买卖股票的最佳时机

给定一个数组 prices，它的第 i 个元素 $prices[i]$ 表示一支给定股票第 i 天的价格。

你只能选择某一天买入这只股票，并选择在未来的某一个不同的日子卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0。

示例 1:

输入: [7,1,5,3,6,4]

输出: 5

解释: 在第 2 天 (股票价格 = 1) 的时候买入，在第 5 天 (股票价格 = 6) 的时候卖出，最大利润 = $6 - 1 = 5$ 。

注意利润不能是 $7 - 1 = 6$ ，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

示例 2:

输入: $prices = [7,6,4,3,1]$

输出: 0

解释: 在这种情况下，没有交易完成，所以最大利润为 0。

提示：

$1 \leq \text{prices.length} \leq 10^5$
 $0 \leq \text{prices}[i] \leq 10^4$

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int minPrice = std::numeric_limits<int>::max(), res = 0;
        for (const int& price : prices) {
            if (price < minPrice) {
                minPrice = price;
            } else if (price - minPrice > res) {
                res = price - minPrice;
            }
        }
        return res;
    }
};
```

- 空间复杂度： $O(1)$
- 时间复杂度： $O(n)$
- 解法： dp
- 标签： 数组， 动态规划
- 难度： 简单

122. 买卖股票的最佳时机 II

给定一个数组 prices ，其中 $\text{prices}[i]$ 是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1：

输入: $\text{prices} = [7,1,5,3,6,4]$

输出: 7

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5-1 = 4$ 。

随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = $6-3 = 3$ 。

示例 2：

输入: $\text{prices} = [1,2,3,4,5]$

输出: 4

解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5-1 = 4$ 。

注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

示例 3：

输入: $\text{prices} = [7,6,4,3,1]$

输出: 0

解释: 在这种情况下，没有交易完成，所以最大利润为 0。

提示：

$1 \leq \text{prices.length} \leq 3 * 10^4$
 $0 \leq \text{prices}[i] \leq 10^4$

```

// 只计算股票增加的价格
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int res = 0;
        for (int i = 1; i < prices.size(); ++i) {
            if (prices[i] > prices[i - 1]) res += prices[i] - prices[i - 1];
        }
        return res;
    }
};

// 计算峰顶和峰谷的差值
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int i = 0, peak = prices[0], valley = prices[0], res = 0;
        while (i < prices.size() - 1) {
            // find peak
            while (i < prices.size() - 1 && prices[i] >= prices[i + 1]) i++;
            peak = prices[i];
            // find valley
            while (i < prices.size() - 1 && prices[i] <= prices[i + 1]) i++;
            valley = prices[i];
            res += valley - peak;
        }
        return res;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: dp
- 标签: 贪心, 数组, 动态规划
- 难度: 简单

124. 二叉树中的最大路径和

TODO:

125. 验证回文串

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。

说明：本题中，我们将空字符串定义为有效的回文串。

示例 1：

输入: "A man, a plan, a canal: Panama"

输出: true

示例 2：

输入: "race a car"

输出: false

```

class Solution {
public:
    bool isPalindrome(string s) {
        if (s.empty()) return true;
        int l = 0, r = s.size() - 1;
        while (l <= r) {
            if (!std::isalnum(s[l])) {
                l++;
            } else if (!std::isalnum(s[r])) {
                r--;
            } else {
                if (std::tolower(s[l]) != std::tolower(s[r])) return false;
                l++;
                r--;
            }
        }
        return true;
    }
};

```

- 空间复杂度: $O(1)$
- 时间复杂度: $O(n)$
- 解法: 双指针
- 标签: 双指针, 字符串
- 难度: 简单

127. 单词接龙

TODO:

128. 最长连续序列

给定一个未排序的整数数组 nums ，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

进阶：你可以设计并实现时间复杂度为 $O(n)$ 的解决方案吗？

示例 1：

输入: $\text{nums} = [100,4,200,1,3,2]$

输出: 4

解释: 最长数字连续序列是 $[1, 2, 3, 4]$ 。它的长度为 4。

示例 2：

输入: $\text{nums} = [0,3,7,2,5,8,4,6,0,1]$

输出: 9

```

class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        int res = 0;
        std::unordered_set<int> st;
        for (const int& num : nums) st.insert(num);
        for (const int& num : st) {
            if (st.find(num - 1) == st.end()) {
                int i = num + 1;
                while (st.find(i) != st.end()) i++;
                res = std::max(res, i - num);
            }
        }
        return res;
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法: 哈希表, 空间换时间
- 标签: 并查集, 数组, 哈希表
- 难度: 中等

130. 被围绕的区域

给你一个 $m \times n$ 的矩阵 board，由若干字符 'X' 和 'O'，找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

示例 1:

输入: board = "X","X","X","X","X","O","O","X","X","O","X","X","O","X","X"
输出: "X","X","X","X","X","X","X","X","X","X","X","X","O","X","X"

解释: 被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。

示例 2:

输入: board = "X"
输出: "X"

```

// X X X X      X X X X      X X X X
// X X 0 X  -> X X 0 X  -> X X X X
// X 0 X X      X 1 X X      X 0 X X
// X 0 X X      X 1 X X      X 0 X X

class Solution {
public:
    void solve(vector<vector<char>>& board) {
        if (board.empty()) return;
        for (int i = 0; i < board.size(); ++i) {
            check(board, i, 0);
            if (board[0].size() > 1) check(board, i, board[0].size() - 1); // 至少两列才会去判断right列
        }
        for (int j = 0; j < board[0].size(); ++j) {
            check(board, 0, j);
            if (board.size() > 1) check(board, board.size() - 1, j); // 至少两行才会去判断down行
        }
        for (int i = 0; i < board.size(); ++i) {
            for (int j = 0; j < board[0].size(); ++j) {
                if (board[i][j] == '0') board[i][j] = 'X';
                if (board[i][j] == '1') board[i][j] = '0';
            }
        }
    }

private:
    void check(std::vector<std::vector<char>>& board, int i, int j) {
        if (board[i][j] == '0') {
            board[i][j] = '1';
            if (i > 1) check(board, i - 1, j);
            if (j > 1) check(board, i, j - 1);
            if (i + 1 < board.size()) check(board, i + 1, j);
            if (j + 1 < board[0].size()) check(board, i, j + 1);
        }
    }
};


```

- 空间复杂度: $O(1)$
 - 时间复杂度: $O(m * n)$
 - 解法:
 - 标签: 深度优先搜索, 广度优先搜索, 并查集, 数组, 矩阵
 - 难度: 中等

131. 分割回文串

给你一个字符串 s，请你将 s 分割成一些子串，使每个子串都是回文串。返回 s 所有可能的分割方案。

回文串 是正着读和反着读都一样的字符串。

示例 1：

输入： s = "aab"

输出： "a","a","b"],["aa","b"

示例 2：

输入： s = "a"

输出： "a"

```

class Solution {
public:
    vector<vector<string>> partition(string s) {
        std::vector<std::vector<std::string>> result;
        std::vector<std::string> curList;
        dfs(result, s, 0, curList);
        return result;
    }

private:
    void dfs(std::vector<std::vector<std::string>>& result,
              std::string& s, int start,
              std::vector<std::string>& curList) { // backtracking
        if (start >= s.size()) result.push_back(curList);
        for (int end = start; end < s.size(); ++end) {
            if (isPalindrome(s, start, end)) {
                curList.push_back(s.substr(start, end - start + 1));
                dfs(result, s, end + 1, curList);
                curList.pop_back();
            }
        }
    }
};

bool isPalindrome(std::string& s, int l, int r) {
    while (l < r) {
        if (s[l] != s[r]) return false;
        l++;
        r--;
    }
    return true;
}
};

```

- 空间复杂度:
- 时间复杂度: $O(n * 2^n)$
- 解法:
- 标签: 字符串, 动态规划, 回溯
- 难度: 中等

134. 加油站

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 $gas[i]$ 升。

你有一辆油箱容量无限的汽车，从第 i 个加油站开往第 $i+1$ 个加油站需要消耗汽油 $cost[i]$ 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

说明:

如果题目有解，该答案即为唯一答案。

输入数组均为非空数组，且长度相同。

输入数组中的元素均为非负数。

示例 1:

输入:

$gas = [1,2,3,4,5]$

$cost = [3,4,5,1,2]$

输出: 3

解释:

从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 $= 0 + 4 = 4$ 升汽油

开往 4 号加油站，此时油箱有 $4 - 1 + 5 = 8$ 升汽油
开往 0 号加油站，此时油箱有 $8 - 2 + 1 = 7$ 升汽油
开往 1 号加油站，此时油箱有 $7 - 3 + 2 = 6$ 升汽油
开往 2 号加油站，此时油箱有 $6 - 4 + 3 = 5$ 升汽油
开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。
因此，3 可为起始索引。

示例 2：

输入：

gas = [2,3,4]

cost = [3,4,3]

输出：-1

解释：

你不能从 0 号或 1 号加油站出发，因为没有足够的汽油可以让你行驶到下一个加油站。

我们从 2 号加油站出发，可以获得 4 升汽油。此时油箱有 $= 0 + 4 = 4$ 升汽油

开往 0 号加油站，此时油箱有 $4 - 3 + 2 = 3$ 升汽油

开往 1 号加油站，此时油箱有 $3 - 3 + 3 = 3$ 升汽油

你无法返回 2 号加油站，因为返程需要消耗 4 升汽油，但是你的油箱只有 3 升汽油。

因此，无论怎样，你都不可能绕环路行驶一周。

```
class Solution {  
public:  
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {  
        int start = 0, // 起始点  
            total = 0, // 总计汽油  
            tank = 0; // 邮箱剩余  
        for (int i = 0; i < gas.size(); ++i) {  
            tank = tank + gas[i] - cost[i];  
            if (tank < 0) {  
                start = i + 1;  
                total += tank;  
                tank = 0;  
            }  
        }  
        return (total + tank < 0) ? -1 : start;  
    }  
};  
  
/**  
 * sum(gas) - sum(cost) >= 0 一定能找到一个起始点到达  
 * /
```

- 空间复杂度：O(1)
- 时间复杂度：O(n)
- 解法：
- 标签：贪心，数组
- 难度：中等

136. 只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

示例 1：

输入: [2,2,1]

输出: 1

示例 2:

输入: [4,1,2,1,2]

输出: 4

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int res = 0;
        for (int num : nums) res ^= num;
        return res;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: a xor a = 0
- 标签: 位运算, 数组
- 难度: 简单

138. 复制带随机指针的链表

给你一个长度为 n 的链表，每个节点包含一个额外增加的随机指针 random，该指针可以指向链表中的任何节点或空节点。

构造这个链表的深拷贝。深拷贝应该正好由 n 个全新节点组成，其中每个新节点的值都设为其对应的原节点的值。新节点的 next 指针和 random 指针也都应指向复制链表中的新节点，并使原链表和复制链表中的这些指针能够表示相同的链表状态。复制链表中的指针都不应指向原链表中的节点。

例如，如果原链表中有 X 和 Y 两个节点，其中 X.random --> Y。那么在复制链表中对应的两个节点 x 和 y，同样有 x.random --> y。

返回复制链表的头节点。

用一个由 n 个节点组成的链表来表示输入/输出中的链表。每个节点用一个 [val, random_index] 表示：

val: 一个表示 Node.val 的整数。

random_index: 随机指针指向的节点索引（范围从 0 到 n-1）；如果不指向任何节点，则为 null。

你的代码只接受原链表的头节点 head 作为传入参数。

```

/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* next;
    Node* random;

    Node(int _val) {
        val = _val;
        next = NULL;
        random = NULL;
    }
};

class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (head == nullptr) return nullptr;
        std::unordered_map<Node*, Node*> mp;
        Node* p = head;
        while (p != nullptr) {
            mp[p] = new Node(p->val);
            p = p->next;
        }
        p = head;
        while (p != nullptr) {
            mp[p]->next = mp[p->next];
            mp[p]->random = mp[p->random];
            p = p->next;
        }
        return mp[head];
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 哈希表, 链表
- 难度: 中等

139. 单词拆分

给定一个非空字符串 s 和一个包含非空单词的列表 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

拆分时可以重复使用字典中的单词。

你可以假设字典中没有重复的单词。

示例 1：

输入: s = "leetcode", wordDict = ["leet", "code"]

输出: true

解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

示例 2：

输入: s = "applepenapple", wordDict = ["apple", "pen"]

输出: true

解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。

注意你可以重复使用字典中的单词。

示例 3：

输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        std::vector<int> dp(s.size() + 1, false);
        dp[0] = true;
        for (int i = 1; i <= s.size(); ++i) { // end
            for (int j = 0; j < i; ++j) { // start
                if (dp[j] && std::find(wordDict.begin(), wordDict.end(), s.substr(j, i - j)) != wordDict.end())
                    dp[i] = true;
                break; // 减少j不必要的循环
            }
        }
        return dp.back();
    }
};
```

- 空间复杂度:
- 时间复杂度:
- 解法: dp
- 标签: 字典树, 记忆化搜索, 哈希表, 字符串, 动态规划
- 难度: 中等

140. 单词拆分 II

TODO:

141. 环形链表

给定一个链表，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 pos 是 -1，则在该链表中没有环。注意：pos 不作为参数进行传递，仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 true 。否则，返回 false 。

进阶：你能用 O(1)（即，常量）内存解决此问题吗？

```

/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if (head == NULL) return false;
        ListNode* slow = head;
        ListNode* fast = head;
        while (fast->next != NULL && fast->next->next != NULL) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) return true;
        }
        return false;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: 双指针
- 标签: 哈希表, 链表, 双指针
- 难度: 简单

146. LRU 缓存机制

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。

实现 LRUCache 类:

LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存

int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1。

void put(int key, int value) 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

进阶：你是否可以在 O(1) 时间复杂度内完成这两种操作？

示例：

输入

["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]

[2, [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]

输出

[null, null, null, 1, null, -1, null, -1, 3, 4]

解释

```

LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1); // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2); // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1); // 返回 -1 (未找到)
lRUCache.get(3); // 返回 3
lRUCache.get(4); // 返回 4

```

```

class LRUCache {
public:
    LRUCache(int capacity) : capacity(capacity) {}

    int get(int key) {
        auto it = cached.find(key);
        if (it == cached.end()) return -1;
        updateToRecentUsed(it);
        return it->second.first;
    }

    void put(int key, int value) {
        auto it = cached.find(key);
        if (it != cached.end()) {
            updateToRecentUsed(it);
        } else {
            if (cached.size() == capacity) {
                cached.erase(recentUsed.back());
                recentUsed.pop_back();
            }
            recentUsed.push_front(key);
        }
        cached[key] = {value, recentUsed.begin()};
    }

private:
    std::unordered_map<int, std::pair<int, std::list<int>::iterator>> cached;
    std::list<int> recentUsed;
    int capacity;

    void updateToRecentUsed(std::unordered_map<int, std::pair<int, std::list<int>::iterator>>::iterator it) {
        int key = it->first;
        recentUsed.erase(it->second.second);
        recentUsed.push_front(key);
        it->second.second = recentUsed.begin();
    }
};

/** 
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache* obj = new LRUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */

/** 
 * DoubleList中，key是LRUCache.key，value是LRUCache.value
 * HashMap中，key是LRUCache.key，value是指向DoubleListNode的指针
 * 如上代码不是这样的
 */

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(1)$
- 解法: 双向链表+哈希表
- 标签: 设计, 哈希表, 链表, 双链表
- 难度: 中等

148. 排序链表

给你链表的头结点 head，请将其按升序排列并返回排序后的链表。

进阶：你可以在 $O(n \log n)$ 时间复杂度和常数级空间复杂度下，对链表进行排序吗？

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (head == nullptr || head->next == nullptr) return head;
        ListNode* slow = head;
        ListNode* fast = head->next;
        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next;
        }
        fast = slow->next;
        slow->next = nullptr;
        return merge(sortList(head), sortList(fast));
    }

private:
    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode* l = new ListNode(-1);
        ListNode* p = l;
        while (l1 != nullptr && l2 != nullptr) {
            if (l1->val < l2->val) {
                p->next = l1;
                l1 = l1->next;
            } else {
                p->next = l2;
                l2 = l2->next;
            }
            p = p->next;
        }
        p->next = (l1 != nullptr) ? l1 : l2;
        return l->next;
    }
};

```

- 空间复杂度:
- 时间复杂度: $O(n \log n)$
- 解法: 快慢指针 + 归并排序
- 标签: 链表, 双指针, 分治, 排序, 归并排序
- 难度: 中等

149. 直线上最多的点数

TODO:

150. 逆波兰表达式求值

根据逆波兰表示法，求表达式的值。

有效的算符包括 +、-、*、/。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明:

整数除法只保留整数部分。

给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        std::stack<int> stk;
        int t1, t2;
        for (const std::string& s : tokens) {
            if (s == "+") {
                t2 = stk.top(); stk.pop();
                t1 = stk.top(); stk.pop();
                stk.push(t1 + t2);
            } else if (s == "-") {
                t2 = stk.top(); stk.pop();
                t1 = stk.top(); stk.pop();
                stk.push(t1 - t2);
            } else if (s == "*") {
                t2 = stk.top(); stk.pop();
                t1 = stk.top(); stk.pop();
                stk.push(t1 * t2);
            } else if (s == "/") {
                t2 = stk.top(); stk.pop();
                t1 = stk.top(); stk.pop();
                stk.push(t1 / t2);
            } else {
                stk.push(std::stoi(s));
            }
        }
        return stk.top();
    }
};
```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法: 栈
- 标签: 栈, 数组, 数学
- 难度: 中等

152. 乘积最大子数组

给你一个整数数组 nums ，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

示例 1:

输入: [2,3,-2,4]

输出: 6

解释: 子数组 [2,3] 有最大乘积 6。

示例 2:

输入: [-2,0,-1]

输出: 0

解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。

```

class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int res = nums[0];
        for (int i = 1, iMax = res, iMin = res; i < nums.size(); ++i) {
            if (nums[i] < 0) std::swap(iMax, iMin);
            iMax = std::max(nums[i], iMax * nums[i]);
            iMin = std::min(nums[i], iMin * nums[i]);
            res = std::max(res, iMax);
        }
        return res;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法:
- 标签: 数组, 动态规划
- 难度: 中等

155. 最小栈

设计一个支持 push , pop , top 操作，并能在常数时间内检索到最小元素的栈。

push(x) —— 将元素 x 推入栈中。

pop() —— 删除栈顶的元素。

top() —— 获取栈顶元素。

getMin() —— 检索栈中的最小元素。

提示:

pop、top 和 getMin 操作总是在 非空栈 上调用。

```

class MinStack {
public:
    /** initialize your data structure here. */
    MinStack() {}

    void push(int val) {
        stk.push(val);
        minStk.empty() ? minStk.push(val) : minStk.push(std::min(val, minStk.top()));
    }

    void pop() {
        stk.pop();
        minStk.pop();
    }

    int top() {
        return stk.top();
    }

    int getMin() {
        return minStk.top();
    }

private:
    std::stack<int> minStk;
    std::stack<int> stk;
};

/** 
 * Your MinStack object will be instantiated and called as such:
 * MinStack* obj = new MinStack();
 * obj->push(val);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->getMin();
 */

```

- 空间复杂度: $O(2n)$
- 时间复杂度: $O(1)$
- 解法: 双栈
- 标签: 栈, 设计
- 难度: 简单

160. 相交链表

给你两个单链表的头节点 headA 和 headB，请你找出并返回两个单链表相交的起始节点。如果两个链表没有交点，返回 null。

```

// 哈希表
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        std::unordered_set<ListNode*> st;
        while (headA) {
            st.insert(headA);
            headA = headA->next;
        }
        while (headB) {
            if (st.find(headB) != st.end()) return headB;
            headB = headB->next;
        }
        return nullptr;
    }
};

// 双指针
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        if (headA == nullptr || headB == nullptr) return nullptr;
        ListNode* p1 = headA, * p2 = headB;
        while (p1 && p2 && p1 != p2) {
            p1 = p1->next;
            p2 = p2->next;
            if (p1 == p2) return p1;
            if (p1 == nullptr) p1 = headB;
            if (p2 == nullptr) p2 = headA;
        }
        return p1;
    }
};

```

- 空间复杂度: $O(1)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 哈希表, 链表, 双指针
- 难度: 简单

162. 寻找峰值

峰值元素是指其值大于左右相邻值的元素。

给你一个输入数组 nums , 找到峰值元素并返回其索引。数组可能包含多个峰值, 在这种情况下, 返回任何一个峰值所在位置即可。

你可以假设 $\text{nums}[-1] = \text{nums}[n] = -\infty$ 。

```

class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int l = 0, r = nums.size() - 1;
        while (l < r) {
            int mid = l + (r - l) / 2;
            if (nums[mid] > nums[mid + 1]) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return l;
    }
};

```

- 空间复杂度: $O(1)$
- 时间复杂度: $O(\log N)$
- 解法:
- 标签: 数组, 二分查找
- 难度: 中等

163. 缺失的区间

给定一个排序的整数数组 `nums`，其中元素的范围在闭区间 `[lower, upper]` 当中，返回不包含在数组中的缺失区间。

示例：

输入: `nums = [0, 1, 3, 50, 75]`, `lower = 0` 和 `upper = 99`,

输出: `["2", "4->49", "51->74", "76->99"]`

```

class Solution {
public:
    vector<string> findMissingRanges(vector<int>& nums, int lower, int upper) {
        std::vector<std::string> res;
        for (const int& num : nums) {
            if (num > lower) {
                res.push_back(std::to_string(lower) + (num - 1 > lower ? ("->" + std::to_string(num - 1)) : ""));
            }
            if (num == upper) {
                return res;
            }
            lower = num + 1;
        }
        if (lower <= upper) {
            res.push_back(std::to_string(lower) + (upper > lower ? ("->" + std::to_string(upper)) : ""));
        }
        return res;
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 哈希表, 链表, 双指针
- 难度: 简单

166. 分数到小数

给定两个整数，分别表示分数的分子 numerator 和分母 denominator，以字符串形式返回小数。

如果小数部分为循环小数，则将循环的部分括在括号内。

如果存在多个答案，只需返回任意一个。

对于所有给定的输入，保证答案字符串的长度小于 104。

```
class Solution {
public:
    string fractionToDecimal(int numerator, int denominator) {
        if (numerator == 0) return "0"; // 除数为0的情况
        std::string res;
        if (numerator > 0 != denominator > 0) res += '-';
        long long int n = std::abs(numerator);
        long long int d = std::abs(denominator);
        res += std::to_string(n / d); // 添加整数部分
        if (n % d == 0) return res;
        res += '.';
        std::unordered_map<int, int> mp;
        for (unsigned long long int r = n % d; r; r %= d) { // 模拟除法过程
            if (mp.count(r) > 0) { // 遇见一个已知的余数，所以我们到了循环部分的末尾
                res.insert(mp[r], 1, '(');
                res += ')';
                break;
            }
            mp[r] = res.size(); // 首先看到剩余的部分，记住它的当前位置
            r *= 10;
            res += std::to_string(r / d);
        }
        return res;
    }
};
```

- 空间复杂度:
- 时间复杂度:
- 解法: 模拟
- 标签: 哈希表, 数学, 字符串
- 难度: 中等

169. 多数元素

给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数大于 $\lfloor n/2 \rfloor$ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1:

输入: [3,2,3]

输出: 3

示例 2:

输入: [2,2,1,1,1,2,2]

输出: 2

```

class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int cnt = 0, res;
        for (int n : nums) {
            if (cnt == 0) res = n;
            cnt += (res == n) ? 1 : -1;
        }
        return res;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: Boyer-Moore投票算法
- 标签: 数组, 哈希表, 分治, 计数, 排序
- 难度: 简单

171. Excel表列序号

给定一个Excel表格中的列名称，返回其相应的列序号。

例如，

```

A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28
...

```

示例 1:

输入: "A"

输出: 1

示例 2:

输入: "AB"

输出: 28

示例 3:

输入: "ZY"

输出: 701

```

class Solution {
public:
    int titleToNumber(string columnTitle) {
        int res = 0;
        for (int i = 0; i < columnTitle.size(); ++i) res = res * 26 + (columnTitle[i] - 'A' + 1);
        return res;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法:
- 标签: 数学, 字符串
- 难度: 简单

172. 阶乘后的零

给定一个整数 n，返回 n! 结果尾数中零的数量。

```
class Solution {
public:
    int trailingZeroes(int n) {
        return n == 0 ? 0 : n / 5 + trailingZeroes(n / 5);
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(log n)
- 解法: 组成0的数的因子为 $2 * 5$, 因子2是充足的, 只需要计算5的因子数量就好了
- 标签: 数学
- 难度: 简单

179. 最大数

给定一组非负整数 nums，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数。

注意：输出结果可能非常大，所以你需要返回一个字符串而不是整数。

```
class Solution {
public:
    string largestNumber(vector<int>& nums) {
        std::vector<std::string> arr(nums.size());
        for (int i = 0; i < nums.size(); ++i) arr[i] = std::to_string(nums[i]);
        std::sort(arr.begin(), arr.end(), [] (const std::string& s1, const std::string& s2) {
            return s1 + s2 > s2 + s1;
        });
        std::string res;
        for (const std::string& s : arr) res += s;
        while (res[0] == '0' && res.size() > 1) res.erase(0, 1); // 去除0-9之前的0
        return res;
    }
};
```

- 空间复杂度: O(n)
- 时间复杂度: O(n log n)
- 解法:
- 标签: 贪心, 字符串, 排序
- 难度: 中等

189. 旋转数组

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

进阶：

尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。

你可以使用空间复杂度为 O(1) 的原地 算法解决这个问题吗？

示例 1:

输入: nums = [1,2,3,4,5,6,7], k = 3

输出: [5,6,7,1,2,3,4]

解释:

向右旋转 1 步: [7,1,2,3,4,5,6]

向右旋转 2 步: [6,7,1,2,3,4,5]

向右旋转 3 步: [5,6,7,1,2,3,4]

示例 2:

输入: nums = [-1,-100,3,99], k = 2

输出: [3,99,-1,-100]

解释:

向右旋转 1 步: [99,-1,-100,3]

向右旋转 2 步: [3,99,-1,-100]

```
class Solution {
public:
    void rotate(vector<int>& nums, int k) {
        k %= nums.size();
        reverse(nums, 0, nums.size() - 1);
        reverse(nums, 0, k - 1);
        reverse(nums, k, nums.size() - 1);
        return;
    }

private:
    void reverse(std::vector<int>& nums, int start, int end) {
        while (start < end) {
            std::swap(nums[start], nums[end]);
            start++;
            end--;
        }
        return ;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: trick
- 标签: 数组, 数学, 双指针
- 难度: 中等

190. 颠倒二进制位

颠倒给定的 32 位无符号整数的二进制位。

提示:

请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。

在 Java 中，编译器使用二进制补码记法来表示有符号整数。因此，在上面的示例 2 中，输入表示有符号整数 -3，输出表示有符号整数 -1073741825。

进阶: 如果多次调用这个函数，你将如何优化你的算法？

```
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        uint32_t res = 0, power = 31;
        while (n != 0) {
            res += (n & 1) << power;
            n >>= 1;
            power--;
        }
        return res;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: 位运算
- 标签: 位运算, 分治
- 难度: 简单

191. 位1的个数

编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为 '1' 的个数（也被称为汉明重量）。

提示：

请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。

在 Java 中，编译器使用二进制补码记法来表示有符号整数。因此，在上面的示例 3 中，输入表示有符号整数 -3。

示例 1：

输入：000000000000000000000000000000001011

输出：3

解释：输入的二进制串 000000000000000000000000000000001011 中，共有三位为 '1'。

示例 2：

输入：0000000000000000000000000000000010000000

输出：1

解释：输入的二进制串 0000000000000000000000000000000010000000 中，共有一位为 '1'。

示例 3：

输入：1111111111111111111111111111111101

输出：31

解释：输入的二进制串 11111111111111111111111111111101 中，共有 31 位为 '1'。

提示：输入必须是长度为 32 的二进制串。

进阶：如果多次调用这个函数，你将如何优化你的算法？

```
class Solution {
public:
    int hammingWeight(uint32_t n) {
        int bits = 0;
        unsigned int mask = 1;
        for (int i = 0; i < 32; ++i) {
            if ((n & mask) != 0) bits++;
            mask <= 1;
        }
        return bits;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(1)
- 解法: 位运算
- 标签: 位运算
- 难度: 简单

198. 打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() == 0) return 0;
        if (nums.size() == 1) return nums[0];
        std::vector<int> dp(nums.size());
        dp[0] = nums[0];
        dp[1] = std::max(nums[0], nums[1]);
        for (int i = 2; i < nums.size(); ++i) {
            dp[i] = std::max(dp[i - 1], nums[i] + dp[i - 2]);
        }
        return dp[nums.size() - 1];
    }
};

class Solution {
public:
    int rob(vector<int>& nums) {
        int pre = 0, cur = 0, tmp;
        for (const int& num : nums) {
            tmp = std::max(pre + num, cur);
            pre = cur;
            cur = tmp;
        }
        return cur;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: dp
- 标签: 数组, 动态规划
- 难度: 中等

200. 岛屿数量

给你一个由 '1' (陆地) 和 '0' (水) 组成的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1:

输入: grid = [
["1","1","1","1","0"],
["1","1","0","1","0"],
["1","1","0","0","0"],
["0","0","0","0","0"]
]
输出: 1

```

class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        if (grid.size() == 0 || grid[0].size() == 0) return 0;
        int islands = 0;
        for (int i = 0; i < grid.size(); ++i) {
            for (int j = 0; j < grid[0].size(); ++j) {
                if (grid[i][j] == '1') {
                    islands++;
                    eraseIslands(grid, i, j);
                }
            }
        }
        return islands;
    }

private:
    void eraseIslands(std::vector<std::vector<char>>& grid, int i, int j) {
        if (i < 0 || i >= grid.size() || j < 0 || j >= grid[0].size() || grid[i][j] == '0') return ;
        grid[i][j] = '0';
        eraseIslands(grid, i - 1, j);
        eraseIslands(grid, i + 1, j);
        eraseIslands(grid, i, j - 1);
        eraseIslands(grid, i, j + 1);
    }
};

/** 
 * 找到一个是陆地的就算是小岛，然后将岛上的陆地全部标记为海
 */

```

- 空间复杂度: O(1)
- 时间复杂度: O(n^2)
- 解法: dfs
- 标签: 深度优先搜索, 广度优先搜索, 并查集, 数组, 矩阵
- 难度: 中等

202. 快乐数

编写一个算法来判断一个数 n 是不是快乐数。

「快乐数」定义为：

对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和。
 然后重复这个过程直到这个数变为 1，也可能是 无限循环 但始终变不到 1。
 如果 可以变为 1，那么这个数就是快乐数。
 如果 n 是快乐数就返回 true； 不是，则返回 false。

示例 1：

输入: 19

输出: true

解释:

$1^2 + 9^2 = 82$

$8^2 + 2^2 = 68$

$6^2 + 2^2 = 100$

$1^2 + 2^2 + 0^2 + 0^2 = 1$

示例 2：

输入: n = 2

输出: false

提示: $1 \leq n \leq 2^{31} - 1$

```

class Solution {
public:
    bool isHappy(int n) {
        int slow = n, fast = n;
        do {
            slow = digitSquareSum(slow);
            fast = digitSquareSum(fast);
            fast = digitSquareSum(fast);
        } while (slow != fast);
        return slow == 1;
    }

private:
    int digitSquareSum(int n) {
        int sm = 0, tmp;
        while (n) {
            tmp = n % 10;
            sm += tmp * tmp;
            n /= 10;
        }
        return sm;
    }
};

```

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 哈希表, 数字, 双指针
- 难度: 简单

204. 计数质数

统计所有小于非负整数 n 的质数的数量。

```

class Solution {
public:
    int countPrimes(int n) {
        std::vector<int> isPrimes(n, true);
        int cnt = 0;
        for (int i = 2; i < n; ++i) {
            if (isPrimes[i]) {
                cnt++;
                for (long int j = i; i * j < n; ++j) {
                    isPrimes[i * j] = false;
                }
            }
        }
        return cnt;
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 数组, 数字, 枚举, 数论
- 难度: 简单

206. 反转链表

给你单链表的头节点 $head$ ，请你反转链表，并返回反转后的链表。

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* cur = head;
        ListNode* pre = NULL;
        while (cur != NULL) {
            ListNode* tmp = cur->next;
            cur->next = pre;
            pre = cur;
            cur = tmp;
        }
        return pre;
    }
};

```

- 空间复杂度: $O(1)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 递归, 链表
- 难度: 简单

207. 课程表

你这个学期必须选修 `numCourses` 门课程，记为 0 到 `numCourses - 1`。

在选修某些课程之前需要一些先修课程。先修课程按数组 `prerequisites` 给出，其中 `prerequisites[i] = [ai, bi]`，表示如果要学习课程 `ai` 则必须先学习课程 `bi`。

例如，先修课程对 `[0, 1]` 表示：想要学习课程 0，你需要先完成课程 1。

请你判断是否可能完成所有课程的学习？如果可以，返回 `true`；否则，返回 `false`。

```

class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        std::vector<std::vector<int>> graph(numCourses); // 邻接表
        std::vector<int> degree(numCourses, 0); // 入度数组
        std::vector<int> bfs; // bfs queue
        for (auto& node : prerequisites) {
            graph[node[1]].push_back(node[0]); // 构建邻接表
            degree[node[0]]++; // 计算入度
        }
        for (int i = 0; i < numCourses; ++i) { // 找出入度为0的，也就是不需要先决条件直接可以上课的
            if (degree[i] == 0) {
                bfs.push_back(i);
            }
        }
        for (int i = 0; i < bfs.size(); ++i) {
            for (int j : graph[bfs[i]]) {
                degree[j]--; // 依赖它的后续课的入度-1
                if (degree[j] == 0) { // 如果因此减为0，入列； 也就是上它课程的先决条件的课程可以上了，它就可以上了
                    bfs.push_back(j);
                }
            }
        }
        return bfs.size() == numCourses; // 如果可以上的课程等于总课程数，则满足期望
    }
};

```

FIXME: 深度优先搜索 解法

- 空间复杂度:
- 时间复杂度:
- 解法: 拓扑排序+bfs
- 标签: 深度优先搜索, 广度优先搜索, 图, 拓扑排序
- 难度: 中等

208. 实现 Trie (前缀树)

Trie (发音类似 "try") 或者说 前缀树 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，例如自动补完和拼写检查。

请你实现 Trie 类：

Trie() 初始化前缀树对象。

void insert(String word) 向前缀树中插入字符串 word。

boolean search(String word) 如果字符串 word 在前缀树中，返回 true (即，在检索之前已经插入)；否则，返回 false。

boolean startsWith(String prefix) 如果之前已经插入的字符串 word 的前缀之一为 prefix，返回 true；否则，返回 false。

```

class Trie {
public:
    /** Initialize your data structure here. */
    Trie() {}

    /** Inserts a word into the trie. */
    void insert(string word) {
        Trie* node = this;
        for (char ch : word) {
            ch -= 'a';
            if (node->next[ch] == nullptr) {
                node->next[ch] = new Trie();
            }
            node = node->next[ch];
        }
        node->isWord = true;
    }

    /** Returns if the word is in the trie. */
    bool search(string word) {
        Trie* node = this;
        for (char ch : word) {
            ch -= 'a';
            if (node->next[ch] == nullptr) {
                return false;
            }
            node = node->next[ch];
        }
        return node->isWord;
    }

    /** Returns if there is any word in the trie that starts with the given prefix. */
    bool startsWith(string prefix) {
        Trie* node = this;
        for (char ch : prefix) {
            ch -= 'a';
            if (node->next[ch] == nullptr) {
                return false;
            }
            node = node->next[ch];
        }
        return true;
    }
};

private:
    Trie* next[26] = {};
    bool isWord = false;
};

/** 
 * Your Trie object will be instantiated and called as such:
 * Trie* obj = new Trie();
 * obj->insert(word);
 * bool param_2 = obj->search(word);
 * bool param_3 = obj->startsWith(prefix);
 */

```

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 设计, 字典树, 哈希表, 字符串
- 难度: 中等

210. 课程表 II

现在你总共有 n 门课需要选, 记为 0 到 n-1。

在选修某些课程之前需要一些先修课程。例如，想要学习课程 0，你需要先完成课程 1，我们用一个匹配来表示他们: [0,1]

给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。

可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

示例 1:

输入: 2, [1,0]

输出: [0,1]

解释: 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 [0,1]。

示例 2:

输入: 4, [1,0],[2,0],[3,1],[3,2]

输出: [0,1,2,3] or [0,2,1,3]

解释: 总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。

因此，一个正确的课程顺序是 [0,1,2,3]。另一个正确的排序是 [0,2,1,3]。

```
# py3
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:
        edges = collections.defaultdict(list) # 存储有向图
        visited = [0] * numCourses # 标记每个节点的状态: 0=未搜索, 1=搜索中, 2=已完成
        result = [] # 用数组来模拟栈, 下标 0 为栈底, n-1 为栈顶
        valid = True # 判断有向图中是否有环

        for info in prerequisites:
            edges[info[1]].append(info[0])

        def dfs(u: int): # 节点
            nonlocal valid
            visited[u] = 1 # 将节点标记为「搜索中」
            for v in edges[u]: # 搜索其相邻节点    只要发现有环，立刻停止搜索
                if visited[v] == 0: # 如果「未搜索」那么搜索相邻节点
                    dfs(v)
                    if valid == False:
                        return
                elif visited[v] == 1: # 如果「搜索中」说明找到了环
                    valid = False
                    return
            visited[u] = 2 # 将节点标记为「已完成」
            result.append(u)

        for i in range(numCourses): # 每次挑选一个「未搜索」的节点，开始进行深度优先搜索
            if valid and not visited[i]:
                dfs(i)

        if not valid:
            return []

        return result[::-1] # 如果没有环，那么就有拓扑排序    注意下标 0 为栈底，因此需要将数组反序输出
```

FIXME: cpp 版本

- 空间复杂度: $O(n + m)$, 其中 n 为课程数, m 为先修课程的要求数
- 时间复杂度: $O(n + m)$
- 解法:
- 标签: 深度优先搜索, 广度优先搜索, 图, 拓扑排序
- 难度: 中等

212. 单词搜索 II

TODO:

215. 数组中的第K个最大元素

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

```
// 空间复杂度: O(n)
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        std::priority_queue<int> pq(nums.begin(), nums.end()); // 大根堆
        for (int i = 0; i < k - 1; ++i) pq.pop();
        return pq.top();
    }
};

// 空间复杂度: O(k)
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        std::priority_queue<int, std::vector<int>, std::greater<int>> pq; // 小根堆
        for (int num : nums) {
            pq.push(num);
            if (pq.size() > k) pq.pop();
        }
        return pq.top();
    }
};
```

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 数组, 分治, 快速选择, 排序, 堆 (优先队列)
- 难度: 中等

217. 存在重复元素

给定一个整数数组，判断是否存在重复元素。

如果存在一值在数组中出现至少两次，函数返回 `true`。如果数组中每个元素都不相同，则返回 `false`。

示例 1:

输入: [1,2,3,1]

输出: true

示例 2:

输入: [1,2,3,4]

输出: false

示例 3:

输入: [1,1,1,3,3,4,3,2,4,2]

输出: true

```

// sort
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        std::sort(nums.begin(), nums.end());
        for (int i = 1; i < nums.size(); ++i) if (nums[i] == nums[i - 1]) return true;
        return false;
    }
};

// hashTable
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        std::unordered_set<int> st;
        for (int num : nums) {
            if (st.find(num) != st.end()) return true;
            st.insert(num);
        }
        return false;
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 递归, 链表
- 难度: 简单

218. 天际线问题

TODO:

227. 基本计算器 II

给你一个字符串表达式 s ，请你实现一个基本计算器来计算并返回它的值。

整数除法仅保留整数部分。

示例 1:

输入: $s = "3+2*2"$

输出: 7

示例 2:

输入: $s = "3/2"$

输出: 1

示例 3:

输入: $s = "3+5 / 2"$

输出: 5

```

class Solution {
public:
    int calculate(string s) {
        if (s.size() == 0) return 0;
        std::stack<int> stk;
        int curNum = 0;
        char opt = '+';
        for (int i = 0; i < s.size(); ++i) {
            char curCh = s[i];
            if (std::isdigit(curCh)) curNum = (curNum * 10) + (curCh - '0');
            if ((std::isdigit(curCh) == false && std::iswspace(curCh) == false) || i == s.size() - 1) {
                if (opt == '-') {
                    stk.push(-curNum);
                } else if (opt == '+') {
                    stk.push(curNum);
                } else if (opt == '*') {
                    int stkTop = stk.top();
                    stk.pop();
                    stk.push(stkTop * curNum);
                } else if (opt == '/') {
                    int stkTop = stk.top();
                    stk.pop();
                    stk.push(stkTop / curNum);
                }
                opt = curCh;
                curNum = 0;
            }
        }
        int result = 0;
        while (stk.size() != 0) {
            result += stk.top();
            stk.pop();
        }
        return result;
    }
};

```

FIXME: O(1)空间解法

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 栈, 数学, 字符串
- 难度: 中等

230. 二叉搜索树中第K小的元素

给定一个二叉搜索树的根节点 root，和一个整数 k，请你设计一个算法查找其中第 k 个最小元素（从 1 开始计数）。

提示：

树中的节点数为 n。

$1 \leq k \leq n \leq 104$

$0 \leq \text{Node.val} \leq 104$

进阶：如果二叉搜索树经常被修改（插入/删除操作）并且你需要频繁地查找第 k 小的值，你将如何优化算法？

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        std::stack<TreeNode*> stk;
        while (true) {
            while (root != nullptr) {
                stk.push(root);
                root = root->left;
            }
            root = stk.top();
            stk.pop();
            k--;
            if (k == 0) return root->val;
            root = root->right;
        }
        return -1;
    }
};

```

- 空间复杂度: $O(h + k)$, h 为树高
- 时间复杂度: $O(k)$
- 解法: 迭代+前序遍历
- 标签: 树, 深度优先搜索, 二叉搜索树, 二叉树
- 难度: 中等

234. 回文链表

请判断一个链表是否为回文链表。

示例 1:

输入: 1->2

输出: false

示例 2:

输入: 1->2->2->1

输出: true

进阶:

你能否用 $O(n)$ 时间复杂度和 $O(1)$ 空间复杂度解决此题?

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        // find mid ptr
        ListNode* fastPtr = head;
        ListNode* slowPtr = head;
        while (fastPtr->next != NULL && fastPtr->next->next != NULL) {
            fastPtr = fastPtr->next->next;
            slowPtr = slowPtr->next;
        }
        if (fastPtr->next != NULL) slowPtr = slowPtr->next; // 如果节点数为奇数，中间的那个直接忽略，left链表会比right链
        // reverse mid right
        ListNode* pre = NULL;
        ListNode* cur = slowPtr;
        while (cur != NULL) {
            ListNode* tmp = cur->next;
            cur->next = pre;
            pre = cur;
            cur = tmp;
        }
        // compare left and right
        while (pre != NULL) {
            if (pre->val != head->val) return false;
            pre = pre->next;
            head = head->next;
        }
        return true;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法:
- 标签: 栈, 递归, 链表, 双指针
- 难度: 简单

236. 二叉树的最近公共祖先

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为: “对于有根树 T 的两个节点 p、q, 最近公共祖先表示为一个节点 x, 满足 x 是 p、q 的祖先且 x 的深度尽可能大 (一个节点也可以是它自己的祖先)。”

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root == nullptr || root == p || root == q) return root; // 如果root已经空了，返回root，也就是null；如果root是p或者q，返回root
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);
        if (left == nullptr) { // 说明都在right下找到了
            return right;
        } else if (right == nullptr) { // 说明都在left下找到了
            return left;
        } else { // 说明分别在left和right下，所以要返回root
            return root;
        }
    }
};

// FIXME: 迭代写法

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法: 递归/迭代, 深度优先搜索
- 标签: 树, 深度优先搜索, 二叉树
- 难度: 中等

237. 删除链表中的节点

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点。传入函数的唯一参数为 要被删除的节点。

现有一个链表 -- head = [4,5,1,9]，它可以表示为：

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    void deleteNode(ListNode* node) {
        node->val = node->next->val;
        node->next = node->next->next;
    }
};

```

- 空间复杂度: $O(1)$
- 时间复杂度: $O(1)$
- 解法:
- 标签: 链表
- 难度: 简单

238. 除自身以外数组的乘积

给你一个长度为 n 的整数数组 nums , 其中 $n > 1$, 返回输出数组 output , 其中 $\text{output}[i]$ 等于 nums 中除 $\text{nums}[i]$ 之外其余各元素的乘积。

示例:

输入: [1,2,3,4]

输出: [24,12,8,6]

提示: 题目数据保证数组之中任意元素的全部前缀元素和后缀 (甚至是整个数组) 的乘积都在 32 位整数范围内。

说明: 请不要使用除法, 且在 $O(n)$ 时间复杂度内完成此题。

进阶:

你可以在常数空间复杂度内完成这个题目吗? (出于对空间复杂度分析的目的, 输出数组不被视为额外空间。)

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        int prod = 1;
        std::vector<int> res(nums.size());
        for (int i = 0; i < nums.size(); ++i) {
            res[i] = prod;
            prod = prod * nums[i];
        }
        prod = 1;
        for (int i = nums.size() - 1; i >= 0; --i) {
            res[i] *= prod;
            prod = prod * nums[i];
        }
        return res;
    }
};
```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法: trick
- 标签: 数组, 前缀和
- 难度: 中等

239. 滑动窗口最大值

TODO:

```

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        window = MonotonicQueue()
        result = []
        for i, _ in enumerate(nums):
            if i < k - 1:
                window.push(nums[i])
            else:
                window.push(nums[i])
                result.append(window.max())
                window.pop(nums[i - k + 1])
        return result

# 单调队列
class MonotonicQueue:
    def __init__(self):
        self.deque = []

    def push(self, num):
        while len(self.deque) != 0 and self.deque[-1] < num:
            self.deque.pop()
        self.deque.append(num)

    def pop(self, num):
        if len(self.deque) != 0 and self.deque[0] == num:
            self.deque.pop(0)

    def max(self):
        return self.deque[0]

```

240. 搜索二维矩阵 II

编写一个高效的算法来搜索 $m \times n$ 矩阵 matrix 中的一个目标值 target。该矩阵具有以下特性：

每行的元素从左到右升序排列。

每列的元素从上到下升序排列。

示例 1：

输入： matrix = `[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]`, target = 5

输出： true

```

class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        if (matrix.empty() || matrix[0].empty()) return false;
        int row = 0, col = matrix[0].size() - 1;
        while (col >= 0 && row <= matrix.size() - 1) {
            if (target == matrix[row][col]) {
                return true;
            } else if (target < matrix[row][col]) {
                col--;
            } else if (target > matrix[row][col]) {
                row++;
            }
        }
        return false;
    }
};

```

- 空间复杂度：
- 时间复杂度： $O(m + n)$
- 解法：
- 标签： 数组，二分查找，分治，矩阵

- 难度： 中等

242. 有效的字母异位词

给定两个字符串 s 和 t，编写一个函数来判断 t 是否是 s 的字母异位词。

示例 1:

输入: s = "anagram", t = "nagaram"

输出: true

示例 2:

输入: s = "rat", t = "car"

输出: false

说明:

你可以假设字符串只包含小写字母。

进阶: 如果输入字符串包含 unicode 字符怎么办？你能否调整你的解法来应对这种情况？

```
// 排序
// 时间 O(nlogn)
// 空间 O(1)
class Solution {
public:
    bool isAnagram(string s, string t) {
        std::sort(s.begin(), s.end());
        std::sort(t.begin(), t.end());
        return s == t;
    }
};

// 哈希表
// 时间 O(n)
// 空间 O(1)
class Solution {
public:
    bool isAnagram(string s, string t) {
        std::vector<int> mpS(26, 0), mpT(26, 0);
        for (const char& ch : s) mpS[ch - 'a']++;
        for (const char& ch : t) mpT[ch - 'a']++;
        return mpS == mpT;
    }
};
```

- 空间复杂度：
- 时间复杂度：
- 解法：
- 标签： 哈希表， 字符串， 排序
- 难度： 简单

251. 展开二维向量

请设计并实现一个能够展开二维向量的迭代器。该迭代器需要支持 next 和 hasNext 两种操作。

示例:

```
Vector2D iterator = new Vector2D(1,2],[3],[4);
```

```
iterator.next(); // 返回 1
```

```
iterator.next(); // 返回 2
```

```
iterator.next(); // 返回 3  
iterator.hasNext(); // 返回 true  
iterator.hasNext(); // 返回 true  
iterator.next(); // 返回 4  
iterator.hasNext(); // 返回 false
```

```
class Vector2D {  
public:  
    Vector2D(vector<vector<int>>& vec2d) {  
        for (const std::vector<int>& line : vec2d) {  
            vec1d.insert(vec1d.end(), line.begin(), line.end());  
        }  
    }  
  
    int next() {  
        return vec1d[i++];  
    }  
  
    bool hasNext() {  
        return i < v.size();  
    }  
  
private:  
    vector<int> vec1d;  
    int i = 0;  
};
```

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签:
- 难度: 中等

253. 会议室 II

给定一个会议时间安排的数组，每个会议时间都会包括开始和结束的时间 $[[s_1, e_1], [s_2, e_2], \dots]$ ($s_i < e_i$)，为避免会议冲突，同时要考虑充分利用会议室资源，请你计算至少需要多少间会议室，才能满足这些会议安排。

示例 1:

输入: `0, 30],[5, 10],[15, 20`

输出: 2

示例 2:

输入: `7,10],[2,4`

输出: 1

```

class Solution {
public:
    int minMeetingRooms(vector<vector<int>>& intervals) {
        std::map<int, int> mp;
        for (const std::vector<int>& interval : intervals) {
            mp[interval[0]]++;
            mp[interval[1]]--;
        }
        int rooms = 0, res = 0;
        for (const std::map<int, int>::iterator& it : mp) {
            rooms += it.second;
            res = max(res, rooms);
        }
        return res;
    }
};

/*
 * 使用 TreeMap 来做的，遍历时间区间，对于起始时间，映射值自增1，对于结束时间，映射值自减1，然后定义结果变量 res，和房间数 rooms,
 */

```

- 空间复杂度：
- 时间复杂度：
- 解法：
- 标签：
- 难度： 中等

268. 丢失的数字

给定一个包含 $[0, n]$ 中 n 个数的数组 nums ，找出 $[0, n]$ 这个范围内没有出现在数组中的那个数。

进阶：你能否实现线性时间复杂度、仅使用额外常数空间的算法解决此问题？

示例 1：

输入： $\text{nums} = [3,0,1]$

输出： 2

解释： $n = 3$ ，因为有 3 个数字，所以所有的数字都在范围 $[0,3]$ 内。2 是丢失的数字，因为它没有出现在 nums 中。

示例 2：

输入： $\text{nums} = [0,1]$

输出： 2

解释： $n = 2$ ，因为有 2 个数字，所以所有的数字都在范围 $[0,2]$ 内。2 是丢失的数字，因为它没有出现在 nums 中。

示例 3：

输入： $\text{nums} = [9,6,4,2,3,5,7,0,1]$

输出： 8

解释： $n = 9$ ，因为有 9 个数字，所以所有的数字都在范围 $[0,9]$ 内。8 是丢失的数字，因为它没有出现在 nums 中。

示例 4：

输入： $\text{nums} = [0]$

输出： 1

解释： $n = 1$ ，因为有 1 个数字，所以所有的数字都在范围 $[0,1]$ 内。1 是丢失的数字，因为它没有出现在 nums 中。

提示：

$n == \text{nums.length}$

$1 \leq n \leq 10^4$

$0 \leq \text{nums}[i] \leq n$

nums 中的所有数字都独一无二

```
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int expectSum = nums.size() * (nums.size() + 1) / 2;
        int actualSum = 0;
        for (const int& num : nums) actualSum += num;
        return expectSum - actualSum;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: 数学
- 标签: 位运算, 数组, 哈希表, 数学, 排序
- 难度: 简单

269. 火星词典

TODO:

277. 搜寻名人

给你一个字符串表达式 s，请你实现一个基本计算器来计算并返回它的值。

整数除法仅保留整数部分。

示例 1:

输入: s = "3+2*2"

输出: 7

示例 2:

输入: s = " 3/2 "

输出: 1

示例 3:

输入: s = " 3+5 / 2 "

输出: 5

```

class Solution {
public:
    int calculate(string s) {
        if (s.size() == 0) return 0;
        std::stack<int> stk;
        int curNum = 0;
        char opt = '+';
        for (int i = 0; i < s.size(); ++i) {
            char curCh = s[i];
            if (std::isdigit(curCh)) curNum = (curNum * 10) + (curCh - '0');
            if ((std::isdigit(curCh) == false && std::iswspace(curCh) == false) || i == s.size() - 1) {
                if (opt == '-') {
                    stk.push(-curNum);
                } else if (opt == '+') {
                    stk.push(curNum);
                } else if (opt == '*') {
                    int stkTop = stk.top();
                    stk.pop();
                    stk.push(stkTop * curNum);
                } else if (opt == '/') {
                    int stkTop = stk.top();
                    stk.pop();
                    stk.push(stkTop / curNum);
                }
                opt = curCh;
                curNum = 0;
            }
        }
        int result = 0;
        while (stk.size() != 0) {
            result += stk.top();
            stk.pop();
        }
        return result;
    }
};

/** 
 * 先计算 * / , + - 转换后压入栈就好，最后计算
 */

```

FIXME: O(1)的空间 解法

- 空间复杂度: O(n)
- 时间复杂度: O(n)
- 解法: 栈
- 标签: 栈, 数学, 字符串
- 难度: 中等

279. 完全平方数

给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

给你一个整数 n，返回和为 n 的完全平方数的 最少数量。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

示例 1:

输入: n = 12

输出: 3

解释: $12 = 4 + 4 + 4$

示例 2:

输入: n = 13

输出: 2

解释: $13 = 4 + 9$

```
class Solution {
public:
    int numSquares(int n) {
        if (n <= 0) return 0;
        std::vector<int> cntPerfectSquares(n + 1, std::numeric_limits<int>::max());
        cntPerfectSquares[0] = 0;
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j * j <= i; ++j) {
                cntPerfectSquares[i] = std::min(cntPerfectSquares[i], cntPerfectSquares[i - j * j] + 1);
            }
        }
        return cntPerfectSquares.back();
    }
};
```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n^2)$
- 解法: dp
- 标签: 广度优先搜索, 数学, 动态规划
- 难度: 中等

283. 移动零

给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:

输入: [0,1,0,3,12]

输出: [1,3,12,0,0]

说明:

必须在原数组上操作，不能拷贝额外的数组。

尽量减少操作次数。

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        for (int i = 0, lastZeroIdx = 0; i < nums.size(); ++i) {
            if (nums[i] != 0) {
                std::swap(nums[lastZeroIdx], nums[i]);
                lastZeroIdx++;
            }
        }
    }
};
```

- 空间复杂度: $O(1)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 数组, 双指针
- 难度: 简单

285. 二叉搜索树中的中序后继

给你一个二叉搜索树和其中的某一个结点，请你找出该结点在树中顺序后继的节点。

结点 p 的后继是值比 p.val 大的结点中键值最小的结点。

示例 1:

输入: root = [2,1,3], p = 1

输出: 2

解析: 这里 1 的顺序后继是 2。

请注意 p 和返回值都应是 TreeNode 类型。

示例 2:

输入: root = [5,3,6,2,4,null,null,1], p = 6

输出: null

解析: 因为给出的结点没有顺序后继，所以答案就返回 null 了。

注意:

假如给出的结点在该树中没有顺序后继的话，请返回 null

我们保证树中每个结点的值是唯一的

```
// 迭代
class Solution {
public:
    TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
        TreeNode *res = nullptr;
        while (root) {
            if (root->val > p->val) {
                res = root;
                root = root->left;
            } else {
                root = root->right;
            }
        }
        return res;
    }
};

// 递归
class Solution {
public:
    TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
        if (root == nullptr) return nullptr;
        if (root->val <= p->val) {
            return inorderSuccessor(root->right, p);
        } else {
            TreeNode *left = inorderSuccessor(root->left, p);
            return left ? left : root;
        }
    }
};
```

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 
- 难度: 中等

287. 寻找重复数

给定一个包含 $n + 1$ 个整数的数组 nums，其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。

假设 `nums` 只有一个重复的整数，找出这个重复的数。

你设计的解决方案必须不修改数组 `nums` 且只用常量级 O(1) 的额外空间。

```
class Solution {
public:
    int findDuplicate(vector<int>& nums) {
        for (int i = 0; i < nums.size(); ++i) {
            int idx = std::abs(nums[i]) - 1;
            nums[idx] *= -1;
            if (nums[idx] > 0) return std::abs(nums[i]);
        }
        return -1;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法:
- 标签: 位运算, 数组, 双指针, 二分查找
- 难度: 中等

289. 生命游戏

根据 百度百科，生命游戏，简称为生命，是英国数学家约翰·何顿·康威在 1970 年发明的细胞自动机。

给定一个包含 $m \times n$ 个格子的面板，每一个格子都可以看成是一个细胞。每个细胞都具有一个初始状态: 1 即为活细胞 (live)，或 0 即为死细胞 (dead)。每个细胞与其八个相邻位置 (水平, 垂直, 对角线) 的细胞都遵循以下四条生存定律:

如果活细胞周围八个位置的活细胞数少于两个，则该位置活细胞死亡；

如果活细胞周围八个位置有两个或三个活细胞，则该位置活细胞仍然存活；

如果活细胞周围八个位置有超过三个活细胞，则该位置活细胞死亡；

如果死细胞周围正好有三个活细胞，则该位置死细胞复活；

下一个状态是通过将上述规则同时应用于当前状态下的每个细胞所形成的，其中细胞的出生和死亡是同时发生的。给你 $m \times n$ 网格面板 board 的当前状态，返回下一个状态。

```

// State transitions
// 0 : dead to dead
// 1 : live to live
// 2 : live to dead
// 3 : dead to live
//
// live 1, dead 0
class Solution {
public:
    void gameOfLife(vector<vector<int>>& board) {

        for (int i = 0; i < board.size(); ++i) {
            for (int j = 0; j < board[0].size(); ++j) {
                int nextSurviveStatus = canSurvive(board, i, j);
                if (board[i][j] == 1) {
                    board[i][j] = nextSurviveStatus ? 1 : 2;
                } else { // board[i][j] == 0
                    board[i][j] = nextSurviveStatus ? 3 : 0;
                }
            }
        }

        for (int i = 0; i < board.size(); ++i) {
            for (int j = 0; j < board[0].size(); ++j) {
                if (board[i][j] == 2) {
                    board[i][j] = 0;
                } else if (board[i][j] == 3) {
                    board[i][j] = 1;
                }
            }
        }
    }

    return;
}

private:
    bool canSurvive(const std::vector<std::vector<int>>& board, int i, int j) {
        int aroundSurviveNum = 0;
        // from left up, clockwise
        if (i - 1 >= 0 && j - 1 >= 0)
            aroundSurviveNum += board[i - 1][j - 1] == 1 ||

        if (i - 1 >= 0)
            aroundSurviveNum += board[i - 1][j] == 1 ||

        if (i - 1 >= 0 && j + 1 < board[0].size())
            aroundSurviveNum += board[i - 1][j + 1] == 1 ||

        if (j + 1 < board[0].size())
            aroundSurviveNum += board[i][j + 1] == 1 ||

        if (i + 1 < board.size() && j + 1 < board[0].size())
            aroundSurviveNum += board[i + 1][j + 1] == 1 ||

        if (i + 1 < board.size())
            aroundSurviveNum += board[i + 1][j] == 1 ||

        if (i + 1 < board.size() && j - 1 >= 0)
            aroundSurviveNum += board[i + 1][j - 1] == 1 ||

        if (j - 1 >= 0)
            aroundSurviveNum += board[i][j - 1] == 1 ||

        // judge result
        bool result = board[i][j] == 1 || board[i][j] == 2;
        if (result) {
            if (aroundSurviveNum < 2) {
                result = false;
            } else if (aroundSurviveNum == 2 || aroundSurviveNum == 3) {
                result = true;
            } else if (aroundSurviveNum > 3) {
                result = false;
            }
        } else { // board[i][j] == 0 || board[i][j] == 3
            if (aroundSurviveNum == 3) {
                result = true;
            }
        }
    }

    return result;
}
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(m * n)
- 解法: 模拟+状态压缩

- 标签: 数组, 矩阵, 模拟
- 难度: 中等

295. 数据流的中位数

TODO:

297. 二叉树的序列化与反序列化

TODO:

300. 最长递增子序列

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，`[3,6,2,7]` 是数组 `[0,3,1,6,2,2,7]` 的子序列。

示例 1:

输入: `nums = [10,9,2,5,3,7,101,18]`

输出: 4

解释: 最长递增子序列是 `[2,3,7,101]`，因此长度为 4。

示例 2:

输入: `nums = [0,1,0,3,2,3]`

输出: 4

示例 3:

输入: `nums = [7,7,7,7,7,7]`

输出: 1

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        std::vector<int> tails(nums.size());
        int size = 0;
        for (int x : nums) {
            int i = 0, j = size;
            while (i != j) {
                int mid = i + (j - i) / 2;
                if (tails[mid] < x) {
                    i = mid + 1;
                } else {
                    j = mid;
                }
            }
            tails[i] = x;
            if (i == size) size++;
        }
        return size;
    }
};

/**
 * tails: 子序列长度为index时候，最后一个num的最小值
 * tails肯定是一个递增序列
 *
 * (1) 如果 x 大于所有tails，则追加它，将大小增加 1
 * (2) 如果tails[i-1] < x <= tails[i]，更新tails[i]
 */
```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n \log n)$
- 解法:
- 标签: 数组, 二分查找, 动态规划
- 难度: 中等

308. 二维区域和检索 - 可变 (前缀和)

TODO:

315. 计算右侧小于当前元素的个数

TODO:

322. 零钱兑换

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

你可以认为每种硬币的数量是无限的。

示例 1:

输入: coins = [1, 2, 5], amount = 11

输出: 3

解释: $11 = 5 + 5 + 1$

示例 2:

输入: coins = [2], amount = 3

输出: -1

示例 3:

输入: coins = [1], amount = 0

输出: 0

示例 4:

输入: coins = [1], amount = 1

输出: 1

示例 5:

输入: coins = [1], amount = 2

输出: 2

```

class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        std::vector<int> dp(amount + 1, amount + 1);
        dp[0] = 0;
        for (int i = 1; i <= amount; ++i) {
            for (int j = 0; j < coins.size(); ++j) {
                if (coins[j] <= i) {
                    dp[i] = std::min(dp[i], dp[i - coins[j]] + 1);
                }
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
};

```

- 空间复杂度: $O(\text{amount} + 1)$
- 时间复杂度: $O(\text{coins.size()} * \text{amount})$
- 解法: dp
- 标签: 广度优先搜索, 数组, 动态规划
- 难度: 中等

324. 摆动排序 II

给你一个整数数组 nums，将它重新排列成 $\text{nums}[0] < \text{nums}[1] > \text{nums}[2] < \text{nums}[3] \dots$ 的顺序。

你可以假设所有输入数组都可以得到满足题目要求的结果。

示例 1:

输入: $\text{nums} = [1,5,1,1,6,4]$
 输出: $[1,6,1,5,1,4]$
 解释: $[1,4,1,5,1,6]$ 同样是符合题目要求的结果，可以被判题程序接受。

示例 2:

输入: $\text{nums} = [1,3,2,2,3,1]$
 输出: $[2,3,1,3,1,2]$

```

// Small half:    4 . 3 . 2 . 1 . 0 .
// Large half:   . 9 . 8 . 7 . 6 . 5
// -----
// Together:     4 9 3 8 2 7 1 6 0 5
class Solution {
public:
    void wiggleSort(vector<int>& nums) {
        std::vector<int> sorted(nums);
        std::sort(sorted.begin(), sorted.end());
        for (int i = nums.size() - 1, j = 0, k = i / 2 + 1; i >= 0; i--) {
            if (i % 2 == 1) {
                nums[i] = sorted[k];
                k++;
            } else {
                nums[i] = sorted[j];
                j++;
            }
        }
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n \log n)$
- 解法:

- 标签: 数组, 分治, 快速选择, 排序
- 难度: 中等

326. 3的幂

给定一个整数，写一个函数来判断它是否是 3 的幂次方。如果是，返回 true；否则，返回 false。

整数 n 是 3 的幂次方需满足：存在整数 x 使得 $n = 3^x$

```
class Solution {
public:
    bool isPowerOfThree(int n) {
        long int pw = 1;
        while (pw < n) pw *= 3;
        return pw == n;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法:
- 标签: 递归, 数学
- 难度: 简单

328. 奇偶链表

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nodes 为节点总数。

示例 1:

输入: 1->2->3->4->5->NULL

输出: 1->3->5->2->4->NULL

示例 2:

输入: 2->1->3->5->6->4->7->NULL

输出: 2->3->6->7->1->5->4->NULL

说明:

应当保持奇数节点和偶数节点的相对顺序。

链表的第一个节点视为奇数节点，第二个节点视为偶数节点，以此类推。

```

/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (head == nullptr) return nullptr;
        ListNode* odd = head; // 奇数
        ListNode* even = head->next; // 偶数
        ListNode* evenHead = even;
        while (even != nullptr && even->next != nullptr) {
            odd->next = even->next;
            odd = odd->next;
            even->next = odd->next;
            even = even->next;
        }
        odd->next = evenHead;
        return head;
    }
};

```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法: 双指针
- 标签: 链表
- 难度: 中等

329. 矩阵中的最长递增路径

TODO:

334. 递增的三元子序列

给你一个整数数组 nums，判断这个数组中是否存在长度为 3 的递增子序列。

如果存在这样的三元组下标 (i, j, k) 且满足 $i < j < k$ ，使得 $\text{nums}[i] < \text{nums}[j] < \text{nums}[k]$ ，返回 true；否则，返回 false。

```

class Solution {
public:
    bool increasingTriplet(vector<int>& nums) {
        int left = std::numeric_limits<int>::max(),
            mid = std::numeric_limits<int>::max();
        for (const int& n : nums) {
            if (n <= left) { // 等于号是为了输入可能是 { 1, 1, 1, 1 } 这种情况
                left = n;
            } else if (n <= mid) {
                mid = n;
            } else {
                return true;
            }
        }
        return false;
    }
};

```

- 空间复杂度: $O(1)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 贪心, 数组
- 难度: 中等

340. 至多包含K个不同字符的最长子串

给定一个字符串 s ，找出至多包含 k 个不同字符的最长子串 T 。

示例 1:

输入: $s = "eceba"$, $k = 2$

输出: 3

解释: 则 T 为 "ece"，所以长度为 3。

示例 2:

输入: $s = "aa"$, $k = 1$

输出: 2

解释: 则 T 为 "aa"，所以长度为 2。

```
class Solution {
public:
    int lengthOfLongestSubstringKDistinct(string s, int k) {
        std::unordered_map<char, int> mp;
        int maxLen = 0;
        for (int i = 0, j = 0; i < s.size(); ++i) {
            if (mp.size() <= k) {
                mp[s[i]]++;
            }
            while (mp.size() > k) {
                if (--mp[s[j]] == 0) {
                    mp.erase(s[j]);
                }
                j++;
            }
            maxLen = std::max(maxLen, i - j + 1);
        }
        return maxLen;
    }
};
```

- 空间复杂度:
- 时间复杂度:
- 解法: 滑动窗口
- 标签: ,
- 难度: 中等

341. 扁平化嵌套列表迭代器

给你一个嵌套的整型列表。请你设计一个迭代器，使其能够遍历这个整型列表中的所有整数。

列表中的每一项或者为一个整数，或者是另一个列表。其中列表的元素也可能是整数或是其他列表。

示例 1:

输入: [1,1],[2,[1,1]

输出: [1,1,2,1,1]

解释: 通过重复调用 `next` 直到 `hasNext` 返回 `false`，`next` 返回的元素的顺序应该是: [1,1,2,1,1]。

示例 2:

输入: [1,[4,[6]]]

输出: [1,4,6]

解释: 通过重复调用 next 直到 hasNext 返回 false, next 返回的元素的顺序应该是: [1,4,6]。

```
/*
 * // This is the interface that allows for creating nested lists.
 * // You should not implement it, or speculate about its implementation
 * class NestedInteger {
 *     public:
 *         // Return true if this NestedInteger holds a single integer, rather than a nested list.
 *         bool isInteger() const;
 *
 *         // Return the single integer that this NestedInteger holds, if it holds a single integer
 *         // The result is undefined if this NestedInteger holds a nested list
 *         int getInteger() const;
 *
 *         // Return the nested list that this NestedInteger holds, if it holds a nested list
 *         // The result is undefined if this NestedInteger holds a single integer
 *         const vector<NestedInteger> &getList() const;
 *     };
 */

class NestedIterator {
public:
    NestedIterator(vector<NestedInteger> &nestedList) {
        for (int i = nestedList.size() - 1; i >= 0; --i) {
            stk.push(&nestedList[i]);
        }
    }

    int next() {
        int nxt = stk.top()->getInteger();
        stk.pop();
        return nxt;
    }

    bool hasNext() {
        while (!stk.empty()) {
            NestedInteger* p = stk.top();
            if (p->isInteger()) return true;
            std::vector<NestedInteger> & vec = p->getList();
            stk.pop();
            for (int i = vec.size() - 1; i >= 0; --i) {
                stk.push(&vec[i]);
            }
        }
        return false;
    }
private:
    std::stack<NestedInteger*> stk;
};

/*
 * Your NestedIterator object will be instantiated and called as such:
 * NestedIterator i(nestedList);
 * while (i.hasNext()) cout << i.next();
 */
```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法:
- 标签: 栈, 树, 深度优先搜索, 设计, 队列, 迭代器
- 难度: 中等

344. 反转字符串

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 char[] 的形式给出。

不要给另外的数组分配额外的空间，你必须原地修改输入数组、使用 O(1) 的额外空间解决这一问题。

你可以假设数组中的所有字符都是 ASCII 码表中的可打印字符。

示例 1:

输入: ["h", "e", "l", "l", "o"]

输出: ["o", "l", "l", "e", "h"]

示例 2:

输入: ["H", "a", "n", "n", "a", "h"]

输出: ["h", "a", "n", "n", "a", "H"]

```
class Solution {  
public:  
    void reverseString(vector<char>& s) {  
        int left = 0, right = s.size() - 1, tmp;  
        while (left < right) {  
            tmp = s[left];  
            s[left] = s[right];  
            s[right] = tmp;  
            left++;  
            right--;  
        }  
    }  
};
```

- 空间复杂度: O(1)
- 时间复杂度:
- 解法: 双指针
- 标签: 递归, 双指针, 字符串
- 难度: 简单

347. 前 K 个高频元素

给你一个整数数组 nums 和一个整数 k，请你返回其中出现频率前 k 高的元素。你可以按任意顺序返回答案。

示例 1:

输入: nums = [1,1,1,2,2,3], k = 2

输出: [1,2]

示例 2:

输入: nums = [1], k = 1

输出: [1]

```

class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        std::unordered_map<int, int> mp;
        for (const int& num : nums) mp[num]++;
        std::vector<int> res;
        std::priority_queue<std::pair<int, int>> pq; // 
        for (std::unordered_map<int, int>::iterator it = mp.begin(); it != mp.end(); ++it) {
            pq.push(std::make_pair(it->second, it->first));
            if (pq.size() > (int) mp.size() - k) {
                res.push_back(pq.top().second);
                pq.pop();
            }
        }
        return res;
    }
};

```

FIXME: 快速选择 解法

FIXME: 桶排序 解法

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 数组, 哈希表, 分治, 桶排序, 计数, 快速选择, 排序, 堆 (优先队列)
- 难度: 中等

348. 设计井字棋

Design a Tic-tac-toe game that is played between two players on a $n \times n$ grid.

You may assume the following rules:

A move is guaranteed to be valid and is placed on an empty block.

Once a winning condition is reached, no more moves is allowed.

A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.

Example:

Given $n = 3$, assume that player 1 is "X" and player 2 is "O" in the board.

Follow up:

Could you do better than $O(n^2)$ per move() operation?

Hint:

Could you trade extra space such that move() operation can be done in O(1)?

You need two arrays: int rows[n], int cols[n], plus two variables: diagonal, anti_diagonal.

FIXME: 中文题目

```

class TicTacToe {
public:
    /** Initialize your data structure here. */
    TicTacToe(int n): rows(n), cols(n), N(n), diag(0), rev_diag(0) {}

    int move(int row, int col, int player) {
        int add = player == 1 ? 1 : -1;
        rows[row] += add;
        cols[col] += add;
        diag += (row == col ? add : 0);
        rev_diag += (row == N - col - 1 ? add : 0);
        return (std::abs(rows[row]) == N
            || std::abs(cols[col]) == N
            || std::abs(diag) == N
            || std::abs(rev_diag) == N) ? player : 0;
    }

private:
    std::vector<int> rows;
    std::vector<int> cols;
    int diag; // 对角线
    int rev_diag; // 反对角线
    int N; // 棋盘大小
};

/*
 * 那么根据提示中的，我们建立一个大小为n的一维数组rows和cols，还有变量对角线diag和逆对角线rev_diag，这种方法的思路是，如果玩家1在第
 */

```

- 空间复杂度：
- 时间复杂度：
- 解法：
- 标签：
- 难度： 中等

350. 两个数组的交集 II

给定两个数组，编写一个函数来计算它们的交集。

示例 1：

输入： nums1 = [1,2,2,1], nums2 = [2,2]

输出： [2,2]

示例 2：

输入： nums1 = [4,9,5], nums2 = [9,4,9,8,4]

输出： [4,9]

```

// 双指针
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        std::sort(nums1.begin(), nums1.end());
        std::sort(nums2.begin(), nums2.end());
        int s1 = 0, s2 = 0;
        std::vector<int> res;
        while (s1 < nums1.size() && s2 < nums2.size()) {
            if (nums1[s1] == nums2[s2]) {
                res.push_back(nums1[s1]);
                s1++;
                s2++;
            } else if (nums1[s1] < nums2[s2]) {
                s1++;
            } else { // nums1[s1] > nums2[s2]
                s2++;
            }
        }
        return res;
    }
};

// 哈希表
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        if (nums1.size() > nums2.size()) std::swap(nums1, nums2); // make num1 length min
        std::map<int, int> cntNums2;
        std::vector<int> res;
        for (int num : nums2) cntNums2.find(num) != cntNums2.end() ? cntNums2[num]++;
        for (int num : nums1) {
            if (cntNums2.find(num) != cntNums2.end() && cntNums2[num] > 0) {
                res.push_back(num);
                cntNums2[num]--;
            }
        }
        return res;
    }
};

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法: 双指针, 哈希表
- 标签: 数组, 哈希表, 双指针, 二分查找, 排序
- 难度: 简单

371. 两整数之和

不使用运算符 + 和 - , 计算两整数 a 、 b 之和。

```

class Solution {
public:
    int getSum(int a, int b) {
        long long int carry; // 64-bit
        while (b != 0) {
            carry = a & b; // 进位
            a = a ^ b; // 求和, 忽略进位
            b = ((carry & 0xffffffff) << 1); // limited to 32 bits
        }
        return a;
    }
};

```

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 位运算, 数学
- 难度: 中等

378. 有序矩阵中第 K 小的元素

给你一个 $n \times n$ 矩阵 matrix，其中每行和每列元素均按升序排序，找到矩阵中第 k 小的元素。
请注意，它是排序后的第 k 小元素，而不是第 k 个不同的元素。

示例 1:

输入: matrix = [1,5,9],[10,11,13],[12,13,15], k = 8

输出: 13

解释: 矩阵中的元素为 [1,5,9,10,11,12,13,13,15]，第 8 小元素是 13

示例 2:

输入: matrix = [-5], k = 1

输出: -5

```
class Solution {
public:
    int kthSmallest(vector<vector<int>>& matrix, int k) {
        int low = matrix[0][0],
            high = matrix.back().back();
        while (low < high) {
            int mid = low + (high - low) / 2;
            int cnt = 0, j = matrix[0].size() - 1;
            for (int i = 0; i < matrix.size(); ++i) { // 从上到下
                while (j >= 0 && matrix[i][j] > mid) j--; // 从右往左
                cnt += (j + 1);
            }
            if (cnt < k) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return low;
    }
};

/***
 * 1. 二分搜索
 * 2. 堆
 */
```

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 数组, 二分查找, 矩阵, 排序, 堆 (优先队列)
- 难度: 中等

380. O(1) 时间插入、删除和获取随机元素

设计一个支持在平均时间复杂度 $O(1)$ 下，执行以下操作的数据结构。

`insert(val)`: 当元素 val 不存在时，向集合中插入该项。

`remove(val)`: 元素 val 存在时，从集合中移除该项。

getRandom: 随机返回现有集合中的一项。每个元素应该有相同的概率被返回。

示例：

```
// 初始化一个空的集合。
RandomizedSet randomSet = new RandomizedSet();

// 向集合中插入 1 。返回 true 表示 1 被成功地插入。
randomSet.insert(1);

// 返回 false , 表示集合中不存在 2 。
randomSet.remove(2);

// 向集合中插入 2 。返回 true 。集合现在包含 [1,2] 。
randomSet.insert(2);

// getRandom 应随机返回 1 或 2 。
randomSet.getRandom();

// 从集合中移除 1 , 返回 true 。集合现在包含 [2] 。
randomSet.remove(1);

// 2 已在集合中, 所以返回 false 。
randomSet.insert(2);

// 由于 2 是集合中唯一的数字, getRandom 总是返回 2 。
randomSet.getRandom();
```

```

class RandomizedSet {
public:
    /** Initialize your data structure here. */
    RandomizedSet() {}

    /** Inserts a value to the set. Returns true if the set did not already contain the specified element. */
    bool insert(int val) {
        if (mp.find(val) != mp.end()) return false;
        nums.emplace_back(val);
        mp[val] = nums.size() - 1;
        return true;
    }

    /** Removes a value from the set. Returns true if the set contained the specified element. */
    bool remove(int val) {
        if (mp.find(val) == mp.end()) return false;
        int last = nums.back();
        mp[last] = mp[val];
        nums[mp[val]] = last;
        nums.pop_back();
        mp.erase(val);
        return true;
    }

    /** Get a random element from the set. */
    int getRandom() {
        return nums[rand() % nums.size()];
    }

private:
    std::vector<int> nums;
    std::unordered_map<int, int> mp; // key是Set的item, value是nums的index
};

/** 
 * Your RandomizedSet object will be instantiated and called as such:
 * RandomizedSet* obj = new RandomizedSet();
 * bool param_1 = obj->insert(val);
 * bool param_2 = obj->remove(val);
 * int param_3 = obj->getRandom();
 */

```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(1)$
- 解法: hashmap + array
- 标签: 设计, 数组, 哈希表, 数学, 随机化
- 难度: 中等

384. 打乱数组

给你一个整数数组 `nums`，设计算法来打乱一个没有重复元素的数组。

实现 Solution class:

`Solution(int[] nums)` 使用整数数组 `nums` 初始化对象
`int[] reset()` 重设数组到它的初始状态并返回
`int[] shuffle()` 返回数组随机打乱后的结果

示例:

输入

`["Solution", "shuffle", "reset", "shuffle"]`

`[1, 2, 3, [], [], []]`

输出

[null, [3, 1, 2], [1, 2, 3], [1, 3, 2]]

解释

```
Solution solution = new Solution([1, 2, 3]);
solution.shuffle(); // 打乱数组 [1,2,3] 并返回结果。任何 [1,2,3] 的排列返回的概率应该相同。例如，返回 [3, 1, 2]
solution.reset(); // 重设数组到它的初始状态 [1, 2, 3]。返回 [1, 2, 3]
solution.shuffle(); // 随机返回数组 [1, 2, 3] 打乱后的结果。例如，返回 [1, 3, 2]
```

```
class Solution {
public:
    Solution(vector<int>& nums) {
        this->origin = nums;
    }

    /** Resets the array to its original configuration and return it. */
    vector<int> reset() {
        return this->origin;
    }

    /** Returns a random shuffling of the array. */
    vector<int> shuffle() {
        std::vector<int> result = this->origin;
        for (int i = 0; i < result.size(); ++i) {
            int j = (std::rand() % (result.size() - i));
            std::swap(result[i], result[i + j]);
        }
        return result;
    }

private:
    std::vector<int> origin;
};

/**
 * Your Solution object will be instantiated and called as such:
 * Solution* obj = new Solution(nums);
 * vector<int> param_1 = obj->reset();
 * vector<int> param_2 = obj->shuffle();
 */

/**
 * 蓄水池抽样算法
 * 蓄水池抽样算法中蓄水池的容量是一定的，但是在此题中，我们让蓄水池容量递增，每次在蓄水池之外取一个元素映射到当前访问的位置(i.e., 与当
 */
```

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$
- 解法: 蓄水池抽样算法
- 标签: 数组, 数学, 随机化
- 难度: 中等

387. 字符串中的第一个唯一字符

给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。

示例：

s = "leetcode"

返回 0

s = "loveleetcode"

返回 2

提示：你可以假定该字符串只包含小写字母。

```
class Solution {
public:
    int firstUniqChar(string s) {
        std::unordered_map<char, int> cnt;
        for (const char& ch : s) cnt.find(ch) != cnt.end() ? cnt[ch]++ : cnt[ch] = 1;
        for (int i = 0; i < s.size(); ++i) if (cnt[s[i]] == 1) return i;
        return -1;
    }
};

class Solution {
public:
    int firstUniqChar(string s) {
        std::vector<int> minIdx(26, s.size());
        std::vector<int> existsTimes(26, 0);
        for (int i = 0; i < s.size(); ++i) {
            existsTimes[s[i] - 'a']++;
            minIdx[s[i] - 'a'] = std::min(minIdx[s[i] - 'a'], i);
        }
        int res = s.size();
        for (int i = 0; i < 26; ++i) if (existsTimes[i] == 1) res = std::min(res, minIdx[i]);
        return res == s.size() ? -1 : res;
    }
};
```

- 空间复杂度: O(1)
- 时间复杂度: O(n)
- 解法:
- 标签: 队列, 哈希表, 字符串, 计数
- 难度: 简单

395. 至少有 K 个重复字符的最长子串

给你一个字符串 s 和一个整数 k，请你找出 s 中的最长子串，要求该子串中的每一字符出现次数都不少于 k。返回这一子串的长度。

示例 1:

输入: s = "aaabb", k = 3

输出: 3

解释: 最长子串为 "aaa"，其中 'a' 重复了 3 次。

示例 2:

输入: s = "ababbc", k = 2

输出: 5

解释: 最长子串为 "ababb"，其中 'a' 重复了 2 次，'b' 重复了 3 次。

```
# py3
class Solution:
    def longestSubstring(self, s: str, k: int) -> int:
        for c in set(s):
            if s.count(c) < k:
                return max(self.longestSubstring(t, k) for t in s.split(c))
        return len(s)
```

FIXME: cpp 版本

- 空间复杂度:
- 时间复杂度:
- 解法:
- 标签: 哈希表, 字符串, 分治, 滑动窗口

- 难度： 中等

412. Fizz Buzz

写一个程序，输出从 1 到 n 数字的字符串表示。

- 1.如果 n 是3的倍数，输出“Fizz”；
- 2.如果 n 是5的倍数，输出“Buzz”；
- 3.如果 n 同时是3和5的倍数，输出 “FizzBuzz”。

示例：n = 15,

返回：

```
[  
    "1",  
    "2",  
    "Fizz",  
    "4",  
    "Buzz",  
    "Fizz",  
    "7",  
    "8",  
    "Fizz",  
    "Buzz",  
    "11",  
    "Fizz",  
    "13",  
    "14",  
    "FizzBuzz"  
]
```

```
class Solution {  
public:  
    vector<string> fizzBuzz(int n) {  
        std::vector<std::string> res;  
        for (int i = 1; i <= n; ++i) {  
            if (i % 3 == 0 && i % 5 == 0) {  
                res.push_back("FizzBuzz");  
            } else if (i % 3 == 0) {  
                res.push_back("Fizz");  
            } else if (i % 5 == 0) {  
                res.push_back("Buzz");  
            } else {  
                res.push_back(std::to_string(i));  
            }  
        }  
        return res;  
    }  
};
```

- 空间复杂度： O(n)
- 时间复杂度： O(n)
- 解法：
- 标签： 数学， 字符串， 模拟
- 难度： 简单

454. 四数相加 II

给定四个包含整数的数组列表 A , B , C , D ,计算有多少个元组 (i, j, k, l) ，使得 $A[i] + B[j] + C[k] + D[l] = 0$ 。

为了使问题简单化，所有的 A, B, C, D 具有相同的长度 N，且 $0 \leq N \leq 500$ 。所有整数的范围在 -2^{28} 到 $2^{28} - 1$ 之间，最终结果不会超过 $2^{31} - 1$ 。

例如：

输入：

```
A = [ 1, 2]  
B = [-2,-1]  
C = [-1, 2]  
D = [ 0, 2]
```

输出：

```
2
```

解释：

两个元组如下：

1. $(0, 0, 0, 1) \rightarrow A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0$
2. $(1, 1, 0, 0) \rightarrow A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0$

```
class Solution {  
public:  
    int fourSumCount(vector<int>& nums1, vector<int>& nums2, vector<int>& nums3, vector<int>& nums4) {  
        std::unordered_map<int, int> mp;  
        int res = 0;  
        for (const int& n1 : nums1) {  
            for (const int& n2 : nums2) {  
                mp.find(n1 + n2) != mp.end() ? mp[n1 + n2]++ : mp[n1 + n2] = 1;  
            }  
        }  
        for (const int& n3 : nums3) {  
            for (const int& n4 : nums4) {  
                if (mp.find(0 - n3 - n4) != mp.end()) res += mp[0 - n3 - n4];  
            }  
        }  
        return res;  
    }  
};
```

- 空间复杂度：
- 时间复杂度： $O(n^4)$
- 解法： 哈希表
- 标签： 数组，哈希表
- 难度： 中等