

# Contents

<b>1</b>	<b>int-summary</b>	<b>2</b>
1.1	概述	2
1.2	数组和字符串	2
1.2.1	三数之和	2
1.2.2	矩阵置零	3
1.2.3	字母异位词分组	3
1.2.4	无重复字符的最长子串	4
1.3	链表	4
1.3.1	两数相加	4
1.3.2	奇偶链表	5
1.3.3	最长回文子串	5
1.3.4	递增的三元子序列	6
1.3.5	相交链表	6
1.4	树	7
1.4.1	中序遍历	7
1.4.2	层序遍历	7
1.4.3	锯齿形层序遍历	7
1.4.4	前序 + 中序还原树	8
1.4.5	二叉搜索树第 k 小	9
1.5	图	9
1.5.1	岛屿数量	9
1.6	回溯法	10
1.6.1	回溯小结	10
1.6.2	电话号码的字母组合	10
1.6.3	括号生成	11
1.6.4	全排列	12
1.6.5	子集	12
1.6.6	单词搜索	12
1.7	排序与搜索	13
1.7.1	各排序算法总结	14
1.7.2	二分小结	14
1.7.3	颜色分类	15
1.7.4	前 k 个高频元素	15
1.7.5	数组中的第 k 个最大元素	16
1.7.6	寻找峰值	16
1.7.7	在排序数组中查找元素的第一个和最后一个位置	17
1.7.8	合并区间	18
1.7.9	搜索旋转排序数组	18
1.7.10	搜索二维矩阵 II	19
1.8	dp	19
1.8.1	跳跃游戏	19
1.8.2	不同路径	19
1.8.3	零钱兑换	20
1.8.4	最长递增子序列	20
1.9	设计	21
1.9.1	二叉树的序列化与反序列化	21
1.9.2	$O(1)$ 时间插入、删除和获取随机元素	22
1.10	数学	23
1.10.1	快乐数	23
1.10.2	阶乘后的零	24
1.10.3	Excel 表列序号	24
1.10.4	Pow(x, n)	24

1.10.5	x 的平方根	25
1.10.6	两数相除	25
1.10.7	分数到小数	29
1.11	其他	30
1.11.1	两整数之和	30
1.11.2	逆波兰表达式求值	31
1.11.3	多数元素	32
1.11.4	任务调度器	32

## 1 int-summary

### 1.1 概述

参考 1: <https://leetcode-cn.com/leetbook/detail/top-interview-questions-medium/>

### 1.2 数组和字符串

#### 1.2.1 三数之和

```
vector<vector<int>> threeSum(vector<int>& nums)
{
    int size = nums.size();
    vector<vector<int>> res;           // 保存结果（所有不重复的三元组）
    if (size < 3) {
        return res;                 // 特判
    }
    std::sort(nums.begin(), nums.end()); // 排序（默认递增）
    for (int i = 0; i < size; i++)      // 固定第一个数，转化为求两数之和
    {
        if (nums[i] > 0) { // !!! 容易漏掉
            return res; // 第一个数大于 0，后面都是递增正数，不可能相加为零了
        }
        // 去重：如果此数已经选取过，跳过
        if (i > 0 && nums[i] == nums[i - 1]) {
            continue;
        }
        // 双指针在 nums[i] 后面的区间中寻找和为 0-nums[i] 的另外两个数
        int left = i + 1;
        int right = size - 1;
        while (left < right)
        {
            if (nums[left] + nums[right] > -nums[i]) {
                right--; // 两数之和太大，右指针左移
            } else if (nums[left] + nums[right] < -nums[i]) {
                left++; // 两数之和太小，左指针右移
            } else {
                // 找到一个和为零的三元组，添加到结果中，左右指针内缩，继续寻找
                vector<int> tmp {nums[i], nums[left], nums[right]};
                res.push_back(tmp);
                left++;
                right--;
                // 去重：第二个数和第三个数也不重复选取 !!! 容易漏掉
                // 例如：[-4,1,1,1,2,3,3,3], i=0, left=1, right=5
                while (left < right && nums[left] == nums[left-1]) {

```

```

        left++;
    }
    while (left < right && nums[right] == nums[right+1]) {
        right--;
    }
}
}
}
return res;
}

```

### 1.2.2 矩阵置零

一个数是 0，那就把这行和这列都变成 0

```

void setZeroes(vector<vector<int>>& matrix) {
    int row = matrix.size();
    int col = matrix[0].size();
    vector<bool> rows(row, false);
    vector<bool> cols(col, false);
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            if (matrix[i][j] == 0) {
                rows[i] = true;
                cols[j] = true;
            }
        }
    }
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            if (rows[i] || cols[j]) {
                matrix[i][j] = 0;
            }
        }
    }
}

```

### 1.2.3 字母异位词分组

其实就是个倒排

```

vector<vector<string>> groupAnagrams(vector<string>& strs) {
    unordered_map<string, vector<string> > xmap;
    for (auto& it: strs) {
        string xit = it;
        sort(xit.begin(), xit.end());
        xmap[xit].emplace_back(it);
    }
    vector<vector<string>> res;
    for (auto& it: xmap) {
        res.emplace_back(it.second);
    }
    return res;
}

```

### 1.2.4 无重复字符的最长子串

双指针

```
int lengthOfLongestSubstring(string s) {
    set<char> set_char;
    int res = 0;
    // 双指针
    for (int i = 0, j = 0; i < s.size() && j < s.size(); ) {
        if (set_char.find(s[j]) != set_char.end()) {
            //找到重复了, 那就把起始的扔了
            set_char.erase(s[i]);
            ++i;
        } else {
            if (j - i + 1 > res) {
                res = j - i + 1;
            }
            set_char.insert(s[j]);
            //没重复的, 右指针继续往前找
            ++j;
        }
    }

    return res;
}
```

## 1.3 链表

### 1.3.1 两数相加

head->...->tail 是倒序的整数, 求两个整数的和, 并返回同样格式的链表

```
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    int carry = 0; // 进位
    ListNode* dummy_head = new ListNode(0); //需要有个 dummy head, 最后 return head->next
    ListNode* tmp = dummy_head;
    ListNode* ptr1 = l1;
    ListNode* ptr2 = l2;
    while (ptr1 != NULL || ptr2 != NULL) {
        int val1 = ptr1 != NULL? ptr1->val: 0;
        int val2 = ptr2 != NULL? ptr2->val: 0;
        int sum = val1 + val2 + carry;
        //cout << sum << " " << carry << " " << val1 << " " << val2 << endl;
        carry = sum / 10; // 很重要!!!! 新的 carry
        int remain = sum % 10;
        tmp->next = new ListNode(remain);
        ptr1 = (NULL == ptr1? NULL: ptr1->next); //判断的是 ptr1, 而不是 ptr1->next!!!!
        ptr2 = (NULL == ptr2? NULL: ptr2->next);
        tmp = tmp->next;
    }
    if (carry > 0) {
        tmp->next = new ListNode(carry);
    }
    return dummy_head->next;
}
```

### 1.3.2 奇偶链表

12345 变成 13524

```
ListNode* oddEvenList(ListNode* head) {
    // 先把第一个偶数保存下来,
    // 跳着指 (2->4, 3->5),
    // 最后再把奇数的指向第一个偶数,
    // return 的应该还是 head
    if (head == nullptr) {
        return nullptr;
    }
    ListNode* even_head = head->next; //第一个偶数, 存下来
    ListNode* odd = head;
    ListNode* even = even_head;
    while (even != nullptr && even->next != nullptr) {
        odd->next = even->next;
        odd = odd->next;
        even->next = odd->next;
        even = even->next;
    }
    odd->next = even_head;
    return head;
}
```

### 1.3.3 最长回文子串

dp

```
string longestPalindrome(string s) {
    //  $p(i, j)$  表示  $i:j$  是回文串
    // 转移:
    // if  $s_i == s_j$  then  $p(i, j) = p(i+1, j-1)$ 
    // 边界:  $len=1$  是,  $len=2$ , 如果  $s_i == s_j$  那是
    // 结果就是所有  $p(i, j)=1$  的  $j-i+1$  的  $max$ 
    int n = s.size();
    if (n < 2) {
        return s;
    }
    int max_len = 1;
    int begin = 0;
    //  $n * n$  的矩阵
    vector<vector<bool>> dp(n, vector<bool>(n));
    for (int i = 0; i < n; ++i) {
        dp[i][i] = true; //1 个字符的肯定是
    }
    // L 是子串长度
    for (int L = 2; L <= n; ++L) {
        for (int i = 0; i < n; ++i) {
            // 根据 L 找 j 的位置,  $L = j-i+1$ 
            int j = L + i - 1;
            if (j >= n) {
                break; // 到头了
            }
            if (s[i] != s[j]) {
```

```

        dp[i][j] = false;
    } else {
        if (j - i < 3) {
            dp[i][j] = true;
        } else {
            dp[i][j] = dp[i + 1][j - 1];
        }
    }

    if (dp[i][j] && L > max_len) {
        max_len = L;
        begin = i;
    }

}

return s.substr(begin, max_len);
}

```

#### 1.3.4 递增的三元子序列

```

bool increasingTriplet(vector<int>& nums) {
    // first < second, 且 second 肯定大于 first, 那么如果 second 右边的比 second 大, 就是找到了
    int n = nums.size();
    //if (n < 3) {
    //    return false;
    //}
    int first = INT_MAX, second = INT_MAX;
    for (int i = 0; i < n; ++i) {
        int num = nums[i];
        if (num <= first) {
            first = num; // 更新第一个数
        } else if (num <= second) {
            second = num; // 这个数比 first 大, 那就是 second
        } else {
            // 如果这个数比两个数都大, 那 return
            return true;
        }
    }
    return false;
}

```

#### 1.3.5 相交链表

返回交点

```

ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    // a b 一直走, 判断是否相等, 假设 b 比 a 长
    // a 到 null 的时候, a 从 b 的头开始, 这样和 b 一起走 b-a 的长度;
    // b 到 null 的时候, 二者都走了 b-a, b 从 a 的头开始, 就能和 a 相遇了
    // 假设没交点, 那最后两个都会指向 null
    if (headA == nullptr || headB == nullptr) {
        return nullptr;
    }
}

```

```

ListNode* p1 = headA;
ListNode* p2 = headB;
while (p1 != p2) {
    p1 = (p1 == nullptr? headB: p1->next);
    p2 = (p2 == nullptr? headA: p2->next);
}
return p1;
}

```

## 1.4 树

### 1.4.1 中序遍历

栈一直塞左子树，取出栈顶，扔到 res 里去，pop 出来，开始遍历原栈顶的右子树

```

vector<int> inorderTraversal(TreeNode* root) {
    stack<TreeNode*> stk;
    vector<int> res;
    while (root != nullptr || !stk.empty()) { // 两个条件 或!!!!
        while (root != nullptr) { // 一直把 root 的左子树丢进去
            stk.push(root);
            root = root->left;
        }
        root = stk.top();
        stk.pop(); // 栈顶扔出来
        res.emplace_back(root->val); // 值搞进去
        root = root->right; // 开始原栈顶的右子树
    }
    return res;
}

```

### 1.4.2 层序遍历

队列 (bfs) queue

### 1.4.3 锯齿形层序遍历

队列 + 优先队列 deque

```

vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    // 层序遍历，加个参数，奇数左到右，偶数右到左
    // dequeue, 双端队列
    vector<vector<int>> res;
    if (root == nullptr) {
        return res;
    }
    queue<TreeNode*> q;
    q.push(root);
    bool left_order = true;
    while (!q.empty()) {
        deque<int> level_lst;
        int size = q.size();
        for (int i = 0; i < size; ++i) { // 这里写 size, 而不是 q.size, 因为 q 一直在变!!!
            TreeNode* node = q.front();
            q.pop();
            if (left_order) {

```

```

        level_lst.push_back(node->val);
    } else {
        level_lst.push_front(node->val);
    }
    if (node->left) {
        q.push(node->left);
    }
    if (node->right) {
        q.push(node->right);
    }
}
res.emplace_back(vector<int>(level_lst.begin(), level_lst.end()));
left_order = !left_order;
}
return res;
}

```

#### 1.4.4 前序 + 中序还原树

前序: 根 [左] [右] 中序: [左] 根 [右] 找到根在中序里的位置 (先用 map 存好值-位置关系, ol 查), 然后递归

```

TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    // 递归:
    // 通过前序找到根, 再在中序里找到根的位置, 左边是左子树, 右边是右子树, 这样就知道在前序里走几步是左, 后面的就是右
    // 因此, 区间的端点就是递归的参数
    // 把中序的值和 index 存到一个 map 里, 这样就能知道在前序中的区间位置了
    int len_pre = preorder.size();
    int len_in = inorder.size();
    if (len_pre != len_in) {
        return nullptr;
    }
    unordered_map<int, int> xmap;
    for (int i = 0; i < len_in; ++i) {
        xmap.emplace(inorder[i], i);
    }
    // 前序, 左右区间; 中序 map, 左右区间
    return buildTreeSub(preorder, 0, len_pre - 1, xmap, 0, len_in - 1);
}

// 这里 xmap 要传引用, 不然会超时..
TreeNode* buildTreeSub(vector<int>& preorder, int pre_start, int pre_end, unordered_map<int, int>& xmap, int in_start, int in_end) {
    if (pre_start > pre_end || in_start > in_end) { // 终止条件
        return nullptr;
    }
    int root_val = preorder[pre_start];
    TreeNode* root = new TreeNode(root_val);
    int in_index = xmap[root_val]; // 肯定会有
    root->left = buildTreeSub(preorder, pre_start + 1, pre_start + in_index - in_start, xmap, in_start, in_index - 1);
    root->right = buildTreeSub(preorder, pre_start + in_index - in_start + 1, pre_end, xmap, in_index + 1, in_end);
    return root;
}

// 迭代法 (看不懂):
// 前序中的任意连续两个节点 u, v 而言, 要么 v 是 u 的左儿子,

```



// 要么  $u$  没有左儿子的话, 那么  $v$  就是  $u$  或者  $u$  的祖先的右儿子 ( $u$  向上回溯, 到第一个有右儿子的就是他的右儿子)

## 1.4.5 二叉搜索树第 $k$ 小

左边比根小, 右边比根大, 那就中序遍历, 遍历完成左, 然后根, 然后右, 然后  $k-$ , 减到 0 就是了中序就是栈

```
int kthSmallest(TreeNode* root, int k) {
    // 栈, 中序遍历, 左子树都比它小, 所以找  $topk$  小, 就先遍历完左的, 再遍历它, 再右
    stack<TreeNode*> stk;
    while (root != nullptr || stk.size() > 0) {
        while (root != nullptr) {
            stk.push(root);
            root = root->left;
        }
        root = stk.top();
        stk.pop();
        --k;
        if (k == 0) {
            break;
        }
        root = root->right;
    }
    return root->val;
}
```

## 1.5 图

### 1.5.1 岛屿数量

以 1 开始, dfs, visited 置 0, dfs 就是上下左右地递归:

```
int numIslands(vector<vector<char>>& grid) {
    // dfs, 看成一个无向图, 垂直或者水平相邻的 1 之间是一条边
    // 遇到 1, 就以它为起点, dfs, 每个走到的 1 重新记为 0!!!
    // 这样, 走了多少次 dfs, 就有多少个岛屿
    // dfs 中 就是先置 0, 然后上下左右分别递归找
    int rows = grid.size();
    if (rows == 0) {
        return 0;
    }
    int cols = grid[0].size();
    int num_islands = 0;
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < cols; ++c) {
            if (grid[r][c] == '1') {
                ++num_islands;
                dfs(grid, r, c);
            }
        }
    }
    return num_islands;
}

void dfs(vector<vector<char>>& grid, int r, int c) {
    int rows = grid.size();
    int cols = grid[0].size();
```

```

grid[r][c] = '0';
if (r - 1 >= 0 && grid[r - 1][c] == '1') {
    dfs(grid, r - 1, c); // 上
}
if (r + 1 < rows && grid[r + 1][c] == '1') {
    dfs(grid, r + 1, c); // 下
}
if (c - 1 >= 0 && grid[r][c - 1] == '1') {
    dfs(grid, r, c - 1); // 左
}
if (c + 1 < cols && grid[r][c + 1] == '1') {
    dfs(grid, r, c + 1); // 右
}
}

```

## 1.6 回溯法

### 1.6.1 回溯小结

回溯法：一种通过探索所有可能的候选解来找出所有的解的算法。如果候选解被确认不是一个解（或者至少不是最后一个解），回溯算法会通过在上一步进行一些变化抛弃该解，即回溯并且再次尝试。

套路：调用：

```

vector<string> res; // 也可能是 vec 的 vec
string cur; // 也可能是 vec, 看题目
backtrace(res, cur, xxx);
return res;

```

回溯函数：

```

void backtrace(vector<string>& res, string& cur, xxx) { // xxx 一般有两个参数, 当前值 a, 上限 len
    if (aaaa) { // a+1 之类的 加到上限了如
        res.push_back(cur);
        return;
    }
    if (bbbb) {
        cur.push_back('aaa'); // 扔进去
        backtrace(res, cur, xxxx); // a+1 之类的操作, 把 len 也传进去
        cur.pop_back(); // 放出来
    }
}

```

模板：

回溯(子集, 全集):

```

if 满足条件:
    加入答案
for 元素 in 全集:
    元素加入子集
    回溯(子集, 全集)
    元素退出子集

```

### 1.6.2 电话号码的字母组合

```

vector<string> letterCombinations(string digits) {
    // 回溯 +dfs
    unordered_map<char, string> phone_map {

```

```

        {'2', "abc"},
        {'3', "def"},
        {'4', "ghi"},
        {'5', "jkl"},
        {'6', "mno"},
        {'7', "pqrs"},
        {'8', "tuv"},
        {'9', "wxyz"}
    };
    vector<string> res;
    if (digits.empty()) {
        return res;
    }
    string comb;
    backtrack(res, phone_map, digits, 0, comb);
    return res;
}

void backtrack(vector<string>& res, const unordered_map<char, string>& phone_map,
    const string& digits, int index, string& comb_str) {
    // index: 输入的 digits 的第 index 个字母
    if (index == digits.length()) {
        res.push_back(comb_str);
    } else {
        char digit = digits[index];
        const string& letters = phone_map.at(digit);
        for (const char& letter: letters) {
            comb_str.push_back(letter); // 先搞一个
            backtrack(res, phone_map, digits, index + 1, comb_str);
            comb_str.pop_back(); // 扔掉, 换一个新的
        }
    }
}
}

```

### 1.6.3 括号生成

```

vector<string> generateParenthesis(int n) {
    vector<string> res;
    string cur;
    backtrack(res, cur, 0, 0, n);
    return res;
}

// open 左括号个数, close 右括号个数
void backtrack(vector<string>& res, string& cur, int open, int close, int n) {
    if (cur.size() == n * 2) { // 一共 2n 个左右括号
        res.push_back(cur);
        return;
    }
    if (open < n) { // 还可以继续加左括号 (最多可以加 n 个)
        cur.push_back('(');
        backtrack(res, cur, open + 1, close, n);
        cur.pop_back();
    }
    if (close < open) { // 准备加新的右括号了

```

```

        cur.push_back(' ');
        backtrack(res, cur, open, close + 1, n);
        cur.pop_back();
    }
}

```

#### 1.6.4 全排列

```

vector<vector<int>> permute(vector<int>& nums) {
    vector<vector<int>> res;
    backtrack(res, nums, 0, nums.size());
    return res;
}

void backtrack(vector<vector<int>> &res, vector<int>& output, int first, int len) {
    if (first == len) {
        res.push_back(output);
    }
    for (int i = first; i < len; ++i) {
        swap(output[i], output[first]); // 交换
        backtrack(res, output, first + 1, len);
        swap(output[i], output[first]); // 换回去
    }
}

```

#### 1.6.5 子集

调用两次 dfs，因为对于子集来说，每个数字可以选也可以不选。

```

void dfs(vector<vector<int>> &res, const vector<int>& nums, vector<int>& cur_res, int cur) {
    if (cur == nums.size()) {
        res.push_back(cur_res);
        return;
    }
    // 调用两次 dfs，因为对于子集来说，每个数字可以选也可以不选。
    cur_res.push_back(nums[cur]);
    dfs(res, nums, cur_res, cur + 1);
    cur_res.pop_back();
    dfs(res, nums, cur_res, cur + 1);
}

vector<vector<int>> subsets(vector<int>& nums) {
    vector<vector<int>> res;
    vector<int> cur_res;
    dfs(res, nums, cur_res, 0);
    return res;
}

```

#### 1.6.6 单词搜索

```

bool check(vector<vector<char>> &board, vector<vector<int>> &visited,
    int i, int j, string word, int k) {
    if (board[i][j] != word[k]) { // 不匹配，不行
        return false;
    } else if (k == word.length() - 1) { // 到最后一个词了且相等，ok
        return true;
    }
}

```

```

    }
    visited[i][j] = true;
    // 上下左右
    vector<pair<int, int> > directions{{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
    bool res = false;
    for (const auto& dir: directions) {
        int new_i = i + dir.first;
        int new_j = j + dir.second;
        if (new_i >= 0 && new_i < board.size() && new_j >= 0 && new_j < board[0].size()) {
            if(!visited[new_i][new_j]) {
                bool flag = check(board, visited, new_i, new_j, word, k + 1);
                if (flag) {
                    res = true;
                    break;
                }
            }
        }
    }
    visited[i][j] = false; //还原
    return res;
}

bool exist(vector<vector<char>>& board, string word) {
    int h = board.size(), w = board[0].size();
    vector<vector<int> > visited(h, vector<int>(w));
    for (int i = 0; i < h; ++i) {
        for (int j = 0; j < w; ++j) {
            bool flag = check(board, visited, i, j, word, 0);
            if (flag) {
                return true;
            }
        }
    }
    return false;
}
}

```

## 1.7 排序与搜索

### 1.7.1 各排序算法总结

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

大小顶堆参考：

```
//小顶堆（是大于。。不是小于），这也是默认
priority_queue<int,vector<int>,greater<int>> > q;
//大顶堆
priority_queue<int,vector<int>,less<int>> > q;
//默认大顶堆
priority_queue<int> a;

// 自定义比较函数：（小顶堆，实现大于操作）
struct MyCmp {
    bool operator()(pair<int, int>& a, pair<int, int>& b) {
        return a.second > b.second;
    }
};
// 小顶堆
priority_queue<pair<int, int>, vector<pair<int, int>>, MyCmp> q;
```

### 1.7.2 二分小结

```
int search(vector<int>& nums, int target) {
    int low = 0, high = nums.size() - 1;
    while (low <= high) { // 小于等于
        int mid = low + (high - low) / 2; // 标准写法，背下来
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] > target) {
            high = mid - 1;
        } else {
```

```

        low = mid + 1;
    }
}
return -1;
}

```

### 1.7.3 颜色分类

即荷兰国旗问题数组里有 0 1 2，要求相同颜色的相邻单指针，还可以用双指针，没太懂

```

void sortColors(vector<int>& nums) {
    int n = nums.size();
    int ptr = 0;
    // 遍历两次，第一遍把 0 交换到前面去，第二遍把 1 交换到 0 之后
    // 用指针 ptr 标记最后一个 0 的下一位，第二遍从 ptr 开始
    for (int i = 0; i < n; ++i) {
        if (nums[i] == 0) {
            swap(nums[i], nums[ptr]);
            ++ptr;
        }
    }
    for (int i = ptr; i < n; ++i) {
        if (nums[i] == 1) {
            swap(nums[i], nums[ptr]);
            ++ptr;
        }
    }
}

```

### 1.7.4 前 k 个高频元素

多存个 map，堆里存的是个 pair

```

struct MyCmp {
    bool operator()(pair<int, int>& a, pair<int, int>& b) {
        return a.second > b.second;
    }
};

vector<int> topKFrequent(vector<int>& nums, int k) {
    // 先遍历一遍，map 存 <k, cnt>，然后遍历 map，用个小顶堆
    // 如果堆的元素个数小于 k，就可以直接插入堆中。
    // 如果堆的元素个数等于 k，则检查堆顶与当前出现次数的大小。
    // 如果堆顶更大，说明至少有 k 个数字的出现次数比当前值大，故舍弃当前值；
    // 否则，就弹出堆顶，并将当前值插入堆中。

    // c++ 的堆是 priority_queue
    unordered_map<int, int> word_count;
    for (auto& v: nums) {
        word_count[v]++;
    }
    // pop 的是优先级最高的元素，top 也是优先级最高的
    // priority_queue<int, vector<int>, cmp> 这是定义方式，一定要有个 vec
    priority_queue<pair<int, int>, vector<pair<int, int> >, MyCmp> q;
    for (auto& [num, cnt]: word_count) {
        if (q.size() < k) {
            q.emplace(num, cnt);
        }
    }
}

```

```

        } else {
            if (q.top().second < cnt) {
                q.pop();
                q.emplace(num, cnt);
            }
        }
    }
    vector<int> res;
    while (!q.empty()){
        res.emplace_back(q.top().first);
        q.pop();
    }
    return res;
}

```

### 1.7.5 数组中的第 k 个最大元素

堆顶就是了

```

int findKthLargest(vector<int>& nums, int k) {
    //小顶堆，堆顶就是要的
    struct MyCmp {
        bool operator()(int a, int b) {
            return a > b;
        }
    };
    priority_queue<int, vector<int>, MyCmp> q;
    for (auto& i: nums) {
        if (q.size() < k) {
            q.emplace(i);
        } else {
            if (i > q.top()) {
                q.pop();
                q.emplace(i);
            }
        }
    }
    return q.top();
}

```

### 1.7.6 寻找峰值

二分，类似旋转数组，如果 mid 不是符合条件的，那看看是在上升还是在下降，如果是在上升，那就看右边区间，如果是下降，那看左边。

```

// 可以搞成匿名函数
// pair<int, int> get(int i, int n, vector<int> & nums) {
//     // 方便处理 nums[-1] 和 nums[n] 的边界情况
//     if (i == -1 || i == n) {
//         return {0, 0};
//     }
//     return {1, nums[i]};
//     // 保证能取到的比越界的大，都能取到的时候，用实际的数比较
// }
int findPeakElement(vector<int>& nums) {
    // 二分，类似旋转数组，如果 mid 不是符合条件的，那看看是在上升还是在下降，
    // 如果是在上升，那就看右边区间，如果是下降，那看左边。

```



```

int left = 0, right = nums.size() - 1;
int n = nums.size();
auto get = [&](int i) -> pair<int, int> {
    // 方便处理 nums[-1] 和 nums[n] 的边界情况
    if (i == -1 || i == n) {
        return {0, 0};
    }
    return {1, nums[i]};
    // 保证能取到的比越界的大, 都能取到的时候, 用实际的数比较
};
while (left <= right) {
    int mid = left + (right - left) / 2; // 标准 mid 写法
    if (get(mid - 1) < get(mid) && get(mid) > get(mid + 1)) {
        return mid;
    }
    if (get(mid) < get(mid + 1)) {
        left = mid + 1;
    } else {
        right = mid - 1;
    }
}
return -1;
}

```

#### 1.7.7 在排序数组中查找元素的第一个和最后一个位置

```

public:
int binary_search(vector<int>& nums, int target, bool lower) {
    // ans 初始化为 n!!!, 因为外面要-1, 对于 [1] 且 target=1 的 case, 会有问题
    int left = 0, right = nums.size() - 1, ans = nums.size();
    // 不要急着 return, 要找到边界
    while (left <= right) {
        int mid = left + (right - left) / 2;
        // lower = true, 想找左边界, 只要 nums[mid] >= target 就可能可以, 只有 <target 的时候才停
        // lower = false, 想找右边第一个 >target 的
        // 都是找左区间
        if (nums[mid] > target || (lower && nums[mid] >= target)) {
            ans = mid;
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return ans;
}

vector<int> searchRange(vector<int>& nums, int target) {
    // 其实要找的就是第一个 =target 的位置, 和第一个 >target 的位置-1
    int left_idx = binary_search(nums, target, true);
    int right_idx = binary_search(nums, target, false) - 1;
    if (left_idx <= right_idx && right_idx < nums.size()
        && left_idx >= 0 && nums[left_idx] == target && nums[right_idx] == target) {
        return vector<int>{left_idx, right_idx};
    }
    return vector<int>{-1, -1};
}

```

```
}
```

### 1.7.8 合并区间

```
vector<vector<int>> merge(vector<vector<int>>& intervals) {  
    // 先排序，然后第一个区间扔进去，遍历下一个的时候，  
    // 看看和前面的最后一个区间有没有交集，如果无交集（当前左 > 已有右），那就扔到最后  
    // 如果有交集，那就取这两个区间 max 的右端点  
    if (intervals.size() == 0) {  
        return {};  
    }  
    sort(intervals.begin(), intervals.end());  
    vector<vector<int>> merged;  
    for (int i = 0; i < intervals.size(); ++i) {  
        int left = intervals[i][0], right = intervals[i][1];  
        if (merged.size() == 0 || left > merged.back()[1]) {  
            merged.push_back({left, right});  
        } else {  
            merged.back()[1] = max(merged.back()[1], right);  
        }  
    }  
    return merged;  
}
```

### 1.7.9 搜索旋转排序数组

```
int search(vector<int>& nums, int target) {  
    // 局部有序，二分  
    int n = nums.size();  
    if (n == 0) {  
        return -1;  
    }  
    if (n == 1) {  
        return nums[0] == target ? 0 : -1;  
    }  
    int left = 0, right = n - 1;  
    while (left <= right) {  
        int mid = left + (right - left) / 2;  
        if (nums[mid] == target) {  
            return mid;  
        }  
        // 看下 mid 在哪个区间里，因为有两个上升的区间，和 nums[0] 比就行  
        if (nums[0] <= nums[mid]) {  
            // mid 在第一个上升区间里  
            if (nums[0] <= target && target < nums[mid]) {  
                // target 也在这个区间里  
                right = mid - 1;  
            } else {  
                left = mid + 1;  
            }  
        } else {  
            left = mid + 1;  
        }  
    }  
    if (nums[mid] < target && target <= nums[n - 1]) {  
        left = mid + 1;  
    } else {  
        return -1;  
    }  
}
```

```

        right = mid - 1;
    }
}
}
return -1;
}

```

### 1.7.10 搜索二维矩阵 II

```

bool searchMatrix(vector<vector<int>>& matrix, int target) {
    // 右上角开始, 保证只有一个搜索方向, 要么变大要么变小, z 字形
    int m = matrix.size(), n = matrix[0].size();
    int x = 0, y = n - 1;
    while (x < m && y >= 0) {
        if (matrix[x][y] == target) {
            return true;
        }
        if (matrix[x][y] > target) {
            --y;
        } else {
            ++x;
        }
    }
    return false;
}

```

## 1.8 dp

### 1.8.1 跳跃游戏

```

bool canJump(vector<int>& nums) {
    // 贪心
    // 对于每个位置 x, 实时维护最远可到达的位置 x+nums[x],
    // 如果这个位置 x 在最远可到达位置内, 那么可以从起点经过若干次跳跃到达
    // 在遍历的过程中, 如果最远可到达位置 >= 数组最后一个位置, 就可以 return True
    int n = nums.size();
    int most_right = 0;
    for (int i = 0; i < n; ++i) {
        if (i <= most_right) {
            most_right = max(most_right, i + nums[i]);
            if (most_right >= n - 1) {
                return true;
            }
        }
    }
    return false;
}

```

### 1.8.2 不同路径

简单二维 dp, 注意边界条件

```

int uniquePaths(int m, int n) {
    // f(i, j) 表示从左上角走到 (i, j) 的路径数量,
    // 这个点只可能是从左边或者上面走过来的, 所以
    // f(i, j) = f(i-1, j) + f(i, j-1)
}

```

```

// 对于第 0 行和第 0 列,  $f(i,0)=1$ ,  $f(0,j)=1$ , 因为只有直着能走到
//  $f(0,0) = 1$ 
vector<vector<int>> f(m, vector<int>(n));
for (int i = 0; i < m; ++i) {
    f[i][0] = 1;
}
for (int j = 0; j < n; ++j) {
    f[0][j] = 1;
}
for (int i = 1; i < m; ++i) {
    for (int j = 1; j < n; ++j) {
        f[i][j] = f[i - 1][j] + f[i][j - 1];
    }
}
return f[m - 1][n - 1];
}

```

### 1.8.3 零钱兑换

```

int coinChange(vector<int>& coins, int amount) {
    // dp[i]: 组成金额 i 需要的最少硬币数
    //  $dp[i] = \min(dp[i - c[j]] + 1, j = 0, \dots, n-1)$ 
    // !!! 注意, 是两项,  $dp[i]$  和  $dp[i - coins[j]] + 1$ 
    //  $dp[i] = \min(dp[i], dp[i - coins[j]] + 1)$ ;
    //  $c[j]$  是第 j 个面额, +1 表示选择这个面额, 那  $i - c[j]$  就是剩下的面额了
    // !! 需要判断凑不出的情况: 把 dp 初始化为 amount + 1, 如果凑不出就不更新,
    // 如果最后还是 amount + 1 那就是凑不出, 当然也可以是 amount + 999
    int xmax = amount + 1;
    // 因为最后一个下标要是 amount, 所以大小是 amount + 1
    vector<int> dp(amount + 1, xmax);
    dp[0] = 0;
    for (int i = 1; i <= amount; ++i) {
        for (int j = 0; j < coins.size(); ++j) {
            // 遍历每种面额
            if (coins[j] <= i) {
                dp[i] = min(dp[i], dp[i - coins[j]] + 1);
            }
        }
    }
    return dp[amount] > amount? -1: dp[amount];
}

```

### 1.8.4 最长递增子序列

```

int lengthOfLIS(vector<int>& nums) {
    // 不要求连续, 比如 [3,6,2,7] 是 [0,3,1,6,2,2,7] 的子序列
    // dp[i]: 以第 i 个数字结尾 (选了 nums[i]) 的最长递增子序列的长度
    //  $dp[i] = \max(dp[j]) + 1, 0 \leq j < i, nums[j] < nums[i]$ , 这样才能递增
    // 最终的结果是  $\max(dp[i])$ 
    int n = nums.size();
    if (n == 0) {
        return 0;
    }
    vector<int> dp(n, 0);
    int res = 0;
}

```

```

    for (int i = 0; i < n; ++i) {
        dp[i] = 1;
        for (int j = 0; j < i; ++j) {
            if (nums[j] < nums[i]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        res = max(res, dp[i]);
    }
    return res;
}

```

## 1.9 设计

### 1.9.1 二叉树的序列化与反序列化

```

class Codec {
public:

    // 前序遍历，到叶子的时候，左右儿子均搞成 None
    void rser(TreeNode* root, string& str) {
        if (root == nullptr) {
            str += "None,";
        } else {
            str += to_string(root->val) + ",";
            rser(root->left, str);
            rser(root->right, str);
            //cout << str << endl;
        }
    }

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        string res;
        rser(root, res);
        return res;
    }

    TreeNode* rde(list<string>& data_vec) {
        // 如果当前的元素为 None，则当前为空树
        // 否则先解析这棵树的左子树，再解析它的右子树
        // list 的 front 是第一个元素的值，begin 是迭代器
        if (data_vec.front() == "None") {
            data_vec.erase(data_vec.begin());
            return nullptr;
        }
        TreeNode* root = new TreeNode(stoi(data_vec.front()));
        data_vec.erase(data_vec.begin());
        root->left = rde(data_vec);
        root->right = rde(data_vec);
        return root;
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        list<string> data_vec;

```

```

    string str;
    // 人肉实现下 split
    for (auto &ch: data) {
        if (ch == ',') {
            data_vec.push_back(str);
            str.clear();
        } else {
            str.push_back(ch);
        }
    }
    if (!str.empty()) {
        data_vec.push_back(str);
        str.clear();
    }
    return rde(data_vec);
}
};

// Your Codec object will be instantiated and called as such:
// Codec ser, deser;
// TreeNode* ans = deser.deserialize(ser.serialize(root));

```

### 1.9.2 O(1) 时间插入、删除和获取随机元素

```

class RandomizedSet {
public:
    // 数组可以 o(1) 地获取元素，哈希可以 o(1) 插入删除，
    // 二者结合起来就是 vec+hashmap
    RandomizedSet() {
        // 初始化随机种子
        srand((unsigned)time(NULL));
    }

    bool insert(int val) {
        // 塞进 vec 里，同时记录下标到 map 中
        if (indices.count(val)) {
            return false;
        }
        int index = nums.size();
        nums.emplace_back(val);
        indices[val] = index;
        return true;
    }

    bool remove(int val) {
        // 为了 o(1)，先把这个数找出来，
        // 然后在 vec 把这个元素换成最后一个元素，pop_back 就行
        // hashmap 里也删掉，同时更新 last 的下标
        if (!indices.count(val)) {
            return false;
        }
        int index = indices[val];
        int last = nums.back();
        nums[index] = last;
    }
};

```

```

        nums.pop_back();
        indices[last] = index;
        indices.erase(val);
        return true;
    }

    int getRandom() {
        int rand_idx = rand() % nums.size();
        return nums[rand_idx];
    }
    vector<int> nums;
    unordered_map<int, int> indices;
};

/**
 * 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();
 */

```

## 1.10 数学

### 1.10.1 快乐数

```

int square_sum(int n) {
    int sum = 0;
    while (n > 0) {
        int bit = n % 10;
        sum += bit * bit;
        n /= 10;
    }
    return sum;
}

bool isHappy(int n) {
    // 只有两种情况，
    // 一直走，最后是 1，相当于无环链表，是快乐数，可以快慢指针
    // 一个有环链表，可以快慢指针，相遇点不是 1
    // 没有第三种情况，因为数字再大，也会归结到一个小很多的数开始的链表，
    // 具体证明可以参考 https://leetcode-cn.com/problems/happy-number/solution/kuai-le-shu-by-leetcode-s
    int fast = n, slow = n;
    // !! 用 do while, 因为一开始 fast=slow, 但仍想走一次循环
    do {
        fast = square_sum(square_sum(fast));
        slow = square_sum(slow);
    } while (fast != slow);
    if (fast == 1) {
        return true;
    }
    return false;
}

```

### 1.10.2 阶乘后的零

$$\sum_{k=1}^{\infty} \left\lfloor \frac{n}{5^k} \right\rfloor <$$

其实这个就是 n 一直除以 5，然后加起来

```
int trailingZeroes(int n) {
    // 因为 5*2=10，其实就是看质因子中 5 和 2 的个数，5 的个数肯定没有 2 多，
    // 如上图，其实就是一直除以 5，再加起来
    int res = 0;
    while (n) {
        n /= 5;
        res += n;
    }
    return res;
}
```

### 1.10.3 Excel 表列序号

```
int titleToNumber(string columnTitle) {
    // 其实是 26 进制转 10 进制
    int num = 0;
    long multiple = 1; // 从 1 开始，需要是 long，因为 int 会爆!!!!
    //倒着来，其实是从最低位开始
    for (int i = columnTitle.size() - 1; i >= 0; --i) {
        int k = columnTitle[i] - 'A' + 1; // 记得 +1...
        num += k * multiple;
        multiple *= 26;
    }
    return num;
}
```

### 1.10.4 Pow(x, n)

```
double pow_sub(double x, long long N) {
    double res = 1.0;
    double x_contribute = x;
    while (N > 0) {
        if (N % 2 == 1) {
            res *= x_contribute;
        }
        x_contribute *= x_contribute;
        N /= 2;
    }
    return res;
}

double myPow(double x, int n) {
    // 其实就是把幂指数 n 进行二进制拆分，如 n=9，那就是
    // 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 2^3+1
    // ==> x^9 = x^8 * x^1
    // 这么变成二进制：
    // n=9, n%2=1, 要!
    // n/=2==> n=4, n%2=0, 不要!
```



```

// n/=2==>n=2, n%2=0, 不要!
// n/=2 ==>n=1, n%2=1, 要!
// 因为除了 1 外, 2^n 全是偶数, 所以如果 n%2=1, 那就需要这个 1
// 还需要考虑如果 n 是负的, 那就是 1/xx
long long N = n;
return N >= 0? pow_sub(x, N): 1 / pow_sub(x, -N);
}

```

### 1.10.5 x 的平方根

```

int mySqrt(int x) {
    // 二分
    // 只返回整数部分, 那就是  $k^2 \leq x$  的最大  $k$ , 可以从 0 到 x 开始二分
    int left = 0, right = x, res = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if ((long long)mid * mid <= x) {
            res = mid; // 一直更新 不 break
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return res;
}

```

### 1.10.6 两数相除

```

// 求 dividend / divisor
int divide(int dividend, int divisor) {
    // // 参考 https://leetcode-cn.com/problems/divide-two-integers/solution/jian-dan-yi-dong-javac-p
    // // 先处理边界
    // // INT_MIN: -2147483648=2^31, INT_MAX: 2147483648=-2^31
    // // int 属于 [-2^31 +1, 2^31 - 1]
    // 假设 x < 0, y < 0, 求 x/y 相当于找个最大的正数 z, 使得
    // yz >= x, 注意应该是最大!, 举例: 算 -10/-2,
    // 如果 z=5, -10=-10, z=4, 那 -8 > -10, z=6, 那 -12 < -10
    // 因此, 是最大的 z, 使得 yz >= x, 因为 z=6 就 yz < x 了

    if (dividend == INT_MIN) {
        if (divisor == 1) {
            return INT_MIN;
        }
        if (divisor == -1) {
            return INT_MAX;
        }
    }
    if (divisor == INT_MIN) {
        // a / (-2^31) = 0, 因为除数绝对值最大
        return dividend == INT_MIN? 1: 0;
    }
    if (dividend == 0) {
        return 0;
    }
    bool rev = false; // 看最后要不要变号

```

```

if (dividend > 0) {
    dividend = -dividend;
    rev = !rev;
}
if (divisor > 0) {
    divisor = -divisor;
    rev = !rev;
}
auto quick_add = [](int y, int z, int x) {
    // 判断  $zy$  是否  $\geq x$ 
    //  $y$  负数,  $x$  负数,  $z$  正数!
    // 对于三个负数  $a b c$ , 要比较  $a+b$  与  $c$ , 因为  $a+b$  可能溢出
    // 所以要改成  $a$  与  $c-b$  比较, 因为两个负数的差不会越界

    // 计算  $y*z$  类似  $y^z$ 
    //  $3^5 = 3^{(1*2^2 + 0*2^1 + 1)}$ 
    //  $3 * 5 = 3 * (1*2^2 + 0*2^1 + 1) = 3*1*2^2 + 3*0 + 3*1$ 
    // 都是相当于对  $z=5$  不断除以 2
    //  $y^z$ : 如果是 1, 那就  $res*=y$ , 然后  $y*=y$ 
    //  $y*z$ : 如果是 1, 那就  $res+=y$ , 然后  $y+=y$ 

    int result = 0, add = y;
    while(z) {
        if (z & 1) {
            //  $z$  的二进制的最后一位是 1,  $z \% 2 == 1$ ,
            // 要保证  $result + add \geq x$ 
            if (result < x - add) {
                return false; //注意这里是直接 return false
            }
            result += add;
        }
        if (z != 1) {
            // 要保  $add + add \geq x$ 
            if (add < x - add) {
                return false; //注意这里是直接 return false
            }
            add += add;
        }
        z >>= 1; //  $z/2$ 
    }
    return true;
};

int left = 0, right = INT_MAX, res = 0;
while (left <= right) {
    int mid = left + ((right - left) >> 1);
    bool check = quick_add(divisor, mid, dividend);
    if (check) {
        res = mid;
        if (mid == INT_MAX) {
            break;
        }
        left = mid + 1; // 想找更大的 直到找到最大的
    } else {

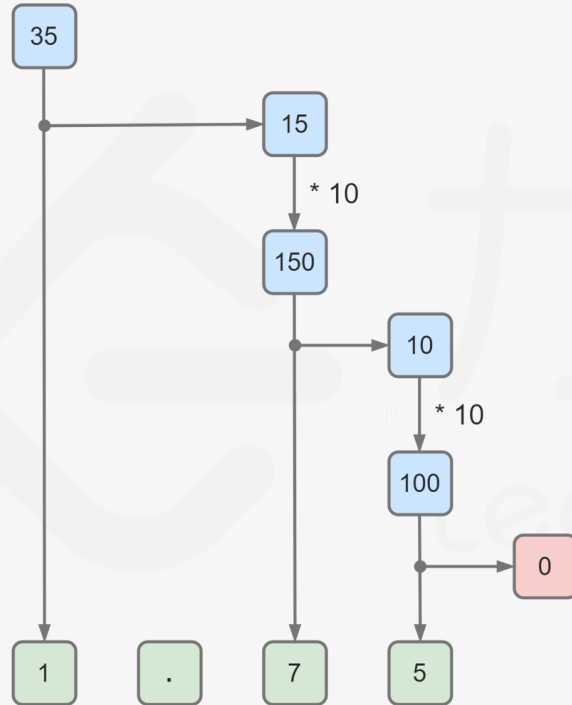
```

```
        right = mid - 1;
    }
}
return rev? -res: res;
}
```

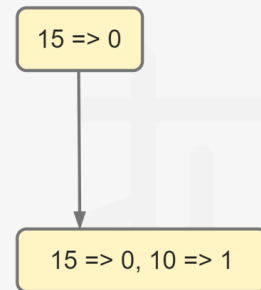


### 1.10.7 分数到小数

35 / 20

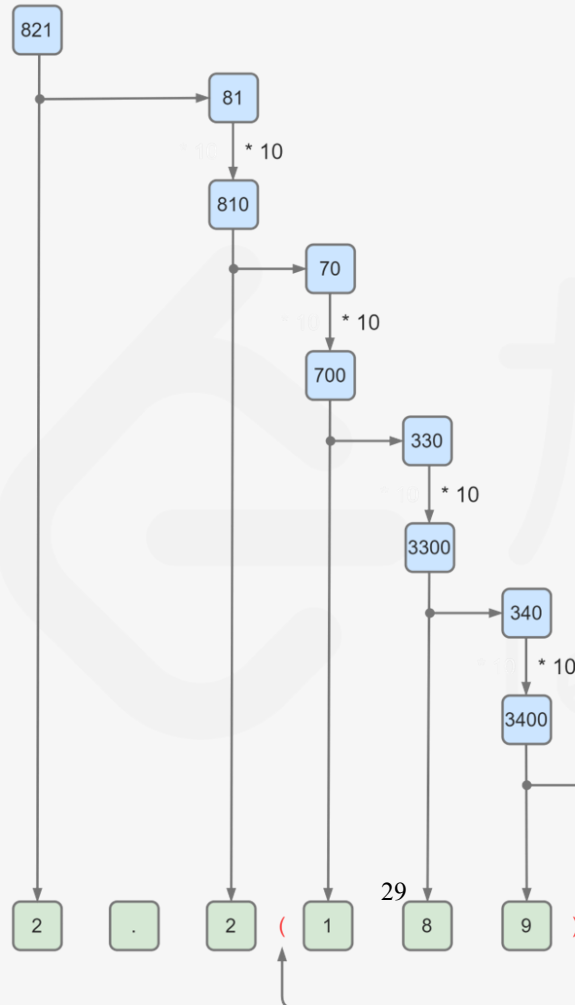


remainderIndexMap

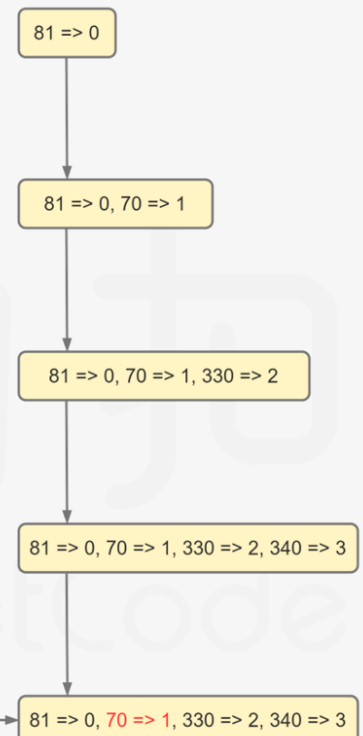


有限小数:

821 / 370



remainderIndexMap



```

string fractionToDecimal(int numerator, int denominator) {
    // !!!!! 无限不循环小数属于实数中的无理数,
    // !!!!! 并且任何一个无限不循环小数都找不到一个与之相等的分数来对应。
    // 所以如果是无限小数, 肯定是无限循环小数
    // 每一位小数 = 余数 *10 再除以除数得到的商。
    // 循环小数: 通过判断被除数 (*10 之前) 有没有出现过,
    // 出现的位置就是循环节的开始, 到结尾就是循环节的结束 (哈希记录)
    // 如果余数 ==0, 那就是可以除尽了, 不是循环小数
    long numerator_long = numerator; //防止溢出, 转成 int64!!!
    long denominator_long = denominator;
    if (numerator_long % denominator_long == 0) { //整除
        return to_string(numerator_long / denominator_long);
    }
    string res;
    if ((numerator_long < 0) ^ (denominator_long < 0)) {
        //异或, 为 true 说明二者异号
        res.push_back('-');
    }
    numerator_long = abs(numerator_long);
    denominator_long = abs(denominator_long);
    long integer = numerator_long / denominator_long;
    res += to_string(integer);
    res.push_back('.');
    string fraction;
    unordered_map<long, int> remainder_index_map;
    long remainder = numerator_long % denominator_long;
    int idx = 0;
    while (remainder != 0 && !remainder_index_map.count(remainder)) {
        remainder_index_map[remainder] = idx; // 记录 *10 之前的 remainder 的位置
        remainder *= 10;
        fraction += to_string(remainder / denominator_long);
        remainder %= denominator;
        ++idx;
    }
    if (remainder != 0) {
        int first_idx = remainder_index_map[remainder];
        // 把循环节部分用括号包起来
        fraction = fraction.substr(0, first_idx) + '(' + \
            fraction.substr(first_idx) + ')';
    }
    res += fraction;
    return res;
}

```

## 1.11 其他

### 1.11.1 两整数之和

```

int getSum(int a, int b) {
    // 不能用 ++, 那就位运算
    // 正整数的补码与原码相同;
    // 负整数的补码为其原码除符号位外的所有位取反后加 1。
    // 可以将减法运算转化为补码的加法运算来实现。
    // 0 + 0 = 0
    // 0 + 1 = 1

```

```

// 1 + 0 = 1
// 1 + 1 = 0 (进位)
// 相当于不考虑进位, 就是  $a \oplus b$  (异或),
// 而进位的值是  $a \& b$ , 进位完就是左移一位  $(a \& b) \ll 1$ 
// 注意, 实际的  $a \oplus b$  是很多位的, 所以进位也是很多位的,
// 所以要有个 while, 一直加进位直到没有进位为止!!!
while (b != 0) {
    // 当我们赋给 signed 类型一个超出它表示范围的值时, 结果是 undefined;
    // 而当我们赋给 unsigned 类型一个超出它表示范围的值时, 结果是
    // 初始值对无符号类型表示数值总数取模的余数!!
    // 因此, 我们可以使用无符号类型来防止溢出。
    unsigned int carry = (unsigned int)(a & b) << 1;
    // 另外, 这里得是  $(unsigned\ int)(a \& b)$  再<<1, 而不是  $(a \& b) \ll 1$  再 unsigned int!!!
    a = a ^ b;
    b = carry;
}
return a;
}

```

### 1.11.2 逆波兰表达式求值

```

bool is_num(string& token) {
    return !(token == "+" || token == "-" || token == "*" || token == "/");
}

int evalRPN(vector<string>& tokens) {
    // 栈: 从左到右遍历表达式
    // 遇到数字, 入栈
    // 遇到运算符 op, pop 栈顶 b 出来作为右, 再 pop 栈顶 a 出来作为左,
    // 计算  $a\ op\ b$  的结果再入栈
    // 遍历完后, 栈内只有一个数, 就是结果
    // 注意 题目要求除法只保留整除的结果, 所以 stk 用 int 就行
    stack<int> stk;
    int n = tokens.size();
    for (int i = 0; i < n; ++i) {
        string& token = tokens[i];
        if (is_num(token)) {
            stk.push(atoi(token.c_str())); // string 转 int
        } else {
            int right = stk.top();
            stk.pop();
            int left = stk.top();
            stk.pop();
            switch (token[0]) {
                case '+':
                    stk.push(left + right);
                    break;
                case '-':
                    stk.push(left - right);
                    break;
                case '*':
                    stk.push(left * right);
                    break;
                case '/':

```

```

        stk.push(left / right);
        break;
    }
}
}
return stk.top();
}

```

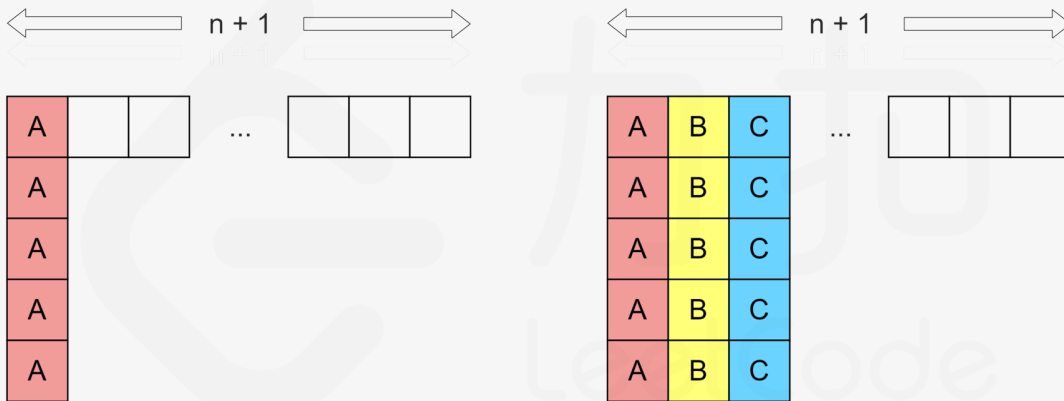
### 1.11.3 多数元素

```

int majorityElement(vector<int>& nums) {
    // 因为题目说了大于半数，所以排序后肯定会占据一半以上的区间，
    // 所以中间的数肯定是它
    sort(nums.begin(), nums.end());
    return nums[nums.size() / 2];
}

```

### 1.11.4 任务调度器



```

int leastInterval(vector<char>& tasks, int n) {
    unordered_map<char, int> freq; // 记录每一种任务的个数
    for (auto& i: tasks) {
        ++freq[i];
    }
    // stl 的 max_element
    // 获取所有任务中最多的次数
    int max_exec = max_element(freq.begin(), freq.end(),
        [](const auto& a, const auto& b) {
            return a.second < b.second;
        })->second;
    // <1> [var] 表示值传递方式捕捉变量 var
    // <2> [=] 表示值传递方式捕捉所有父作用域的变量 (包括 this 指针)
    // <3> [&var] 表示引用传递捕捉变量 var
    // <4> [&] 表示引用传递捕捉所有父作用域的变量 (包括 this 指针)
    // <5> [this] 表示值传递方式捕捉当前的 this 指针
    // <6> [=, &a, &b] 表示以引用传递的方式捕捉变量 a 和 b, 而以值传递方式捕捉其他所有的变量
    // <7> [&, a, this] 表示以值传递的方式捕捉 a 和 this, 而以引用传递方式捕捉其他所有变量
}

```



```

// 计算总共有多少个任务出现了 max_exec 次
int max_cnt = accumulate(freq.begin(), freq.end(), 0,
    [=](int acc, const auto& u){
        return acc + (u.second == max_exec);
    });

// 对于 max_exec 个任务 a 来讲，每执行一次后面要 n 个空位，
// 所以要  $(max\_exec - 1) * (n + 1) + 1$ ，最后这个  $+1$  是最后一个 a 任务，因为它执行完就行了
// 而总共有 max_cnt 个 a 任务，如果  $max\_cnt \leq n + 1$ ，那么可以塞进去
// 就有  $max\_exec - 1) * (n + 1) + max\_cnt$  了
// 填后面任务按这个方法：
// 我们从倒数第二行开始，按照反向列优先的顺序（即先放入靠左侧的列，同一列中先放入下方的行），
// 依次放入每一种任务，并且同一种任务需要连续地填入。
// 如果  $max\_cnt > n + 1$ ，那排完  $n + 1$  后还要再排 k 列，然后才是其他任务，
// 这个时候就不需要再这么按顺序了，因为任意两个任务间肯定大于 n，所以总时间就是 tasks
// xx.size() 需要强转成 int，因为原来是 size_t
return max((max_exec - 1) * (n + 1) + max_cnt, static_cast<int>(tasks.size()));
}

```