leetcode cpp

目录

数据结构

STL

Sequence Containers

> 维持顺序的容器

- `vector`：动态数组
- `list`：双向链表
- `deque`：双端队列
- `array`：固定大小的数组
- `forward_list`：单向链表

Container Adaptors

> 基于其它容器实现的数据结构

- `stack`：栈：后入先出（LIFO）的数据结构
- `queue`：队列：先入先出（FIFO）的数据结构
- `priority_queue`：最大值先出的数据结构

Associative Containers

> 排好序的数据结构

- `set`：有序集合，不可重复
- `multiset`：可重复的 set
- `map`：有序表，在 set 的基础上加上映射关系，可以对每个元素 key 存一个值 value。
- `multimap`：可重复的 map

Unordered Associative Containers

> 对每个 Associative Containers 实现了哈希版本，即无序版本

- `unordered_set`：哈希集合
- `unordered_multiset`：可重复的哈希集合

- `unordered_map`：哈希表
- `unordered_multimap`：可重复的哈希表

关于 STL 的详细内容请参考：C++ STL总结 or https://zh.cppreference.com/

数组

找到所有数组中消失的数字

题目：

给你一个含 `n` 个整数的数组 `nums`，其中 `nums[i]` 在区间 `[1, n]` 内。请你找出所有在 `[1, n]` 范围内但没有出现在 `nums` 中的数字，并以数组的形式返回结果。

> “
> 输入：nums = [4,3,2,7,8,2,3,1]
> 输出：[5,6]

题解：

利用数组这种数据结构建立 n 个桶，把所有重复出现的位置进行标记，然后再遍历一遍数组，即可找到没有出现过的数字。

进一步地，我们可以直接对原数组进行标记：把出现的数字对应的位置设为负数，最后仍然为正数的位置即为没有出现过的数。

```
vector<int> findDisappearedNumbers(vector<int>& nums) {
    for (auto num : nums) {
        int pos = abs(num) - 1;
        if (nums[pos] > 0)
            nums[pos] = -nums[pos];
    }
    vector<int> ans;
    for (int i = 0; i < nums.size(); ++i) {
        if (nums[i] > 0)
            ans.push_back(i+1);
    }
    return ans;
}
```

旋转图像

题目：

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。

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

题解：

方法一：

旋转的关键等式为：

$$matrix[col][n-row-1] = matrix[row][col]$$

为了防止原 $matrix[col][n-row-1]$ 被覆盖，需要把它先旋转过去，带入公式得：

$$matrix[n-row-1][n-col-1] = matrix[col][n-row-1]$$

为了防止 $matrix[n-row-1][n-col-1]$ 被覆盖，需要：

$$matrix[n-col-1][row] = matrix[n-row-1][n-col-1]$$

为了防止 $matrix[n-col-1][row]$ 被覆盖，需要：

$$matrix[row][col] = matrix[n-col-1][row]$$

又回到了起点，所以一次旋转四个即可，遍历矩阵左四分之一部分即可，代码如下：

```cpp
void rotate(vector<vector<int>>& matrix) {
    int n = matrix.size();
    for (int i = 0; i < (n+1)/2; ++i) {
        for (int j = 0; j < n/2; ++j) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[n-j-1][i];
            matrix[n-j-1][i] = matrix[n-i-1][n-j-1];
            matrix[n-i-1][n-j-1] = matrix[j][n-i-1];
            matrix[j][n-i-1] = temp;
        }
    }
}
```

方法二：

两次翻转实现旋转，代码如下：

```cpp
void rotate(vector<vector<int>>& matrix) {
    int n = matrix.size();
    // 水平翻转
    for (int i = 0; i < n / 2; ++i) {
        for (int j = 0; j < n; ++j) {
            swap(matrix[i][j], matrix[n - i - 1][j]);
        }
    }
```

```
    // 主对角线翻转
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            swap(matrix[i][j], matrix[j][i]);
        }
    }
}
```

为什么?

> $$\begin{cases} matrix[n-row-1][col] = matrix[row][col] \\ matrix[col][row] = matrix[row][col] \end{cases}$$
>
> 等价于
>
> $$matrix[col][n-row-1] = matrix[row][col]$$

搜索二维矩阵 II

题目:

编写一个高效的算法来搜索 `m x 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。

> 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

题解:

类似二分查找，从左下角开始，如果当前值大于目标值，那么向上走，如果当前值小于目标值，向右走。

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int m = matrix.size(), n = matrix[0].size();
    if (m == 0)
        return false;
    int i = m - 1, j = 0;
    while(i >= 0 && j < n) {
        if (matrix[i][j] == target)
            return true;
        else if (matrix[i][j] > target)
            i--;
```

```
        else
            j++;
    }
    return false;
}
```

最多能完成排序的块

题目：

给定一个长度为 `n` 的整数数组 `arr` ，它表示在 `[0, n - 1]` 范围内的整数的排列。

我们将 `arr` 分割成若干 **块** (即分区)，并对每个块单独排序。将它们连接起来后，使得连接的结果和按升序排序后的原数组相同。

返回数组能分成的最多块数量。

> “
> 输入: arr = [1,0,2,3,4]
> 输出: 4
> 解释:
> 我们可以把它分成两块，例如 [1, 0], [2, 3, 4]。然而，分成 [1, 0], [2], [3], [4] 可以得到最多的块数。对每个块单独排序后，结果为 [0, 1], [2], [3], [4]

题解：

从左往右遍历，同时记录当前的最大值，每当当前最大值等于数组位置时，我们可以多一次分割。

为什么可以通过这个算法解决问题呢？如果当前最大值大于数组位置，则说明右边一定有小于数组位置的数字，需要把它也加入待排序的子数组；又因为数组只包含不重复的 0 到 n，所以当前最大值一定不会小于数组位置。

```
int maxChunksToSorted(vector<int>& arr) {
    int n = arr.size();
    int max = arr[0], ans = 0;
    for (int i = 0; i < n; ++i) {
        if (arr[i] > max)
            max = arr[i];
        if (max == i)
            ans++;
    }
    return ans;
}
```

栈和队列

用栈实现队列

题目：

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作：

实现 `MyQueue` 类：

- `void push(int x)` 将元素 x 推到队列的末尾
- `int pop()` 从队列的开头移除并返回元素
- `int peek()` 返回队列开头的元素
- `boolean empty()` 如果队列为空，返回 `true` ；否则，返回 `false`

题解：

我们可以用两个栈来实现一个队列：因为我们需要得到先入先出的结果，所以必定要通过一个额外栈翻转一次数组。这个翻转过程既可以在插入时完成，也可以在取值时完成。

```
class MyQueue {
    private:
    stack<int> in, out;
    void in2out() {
        if (out.empty()) {
            while (!in.empty()) {
                out.push(in.top());
                in.pop();
            }
        }
    }
    public:
    MyQueue() { }
    void push(int x) {
        in.push(x);
    }
    int pop() {
        in2out();
        int x = out.top();
        out.pop();
        return x;
    }
    int peek() {
        in2out();
        return out.top();
    }
    bool empty() {
        return in.empty() & out.empty();
    }
};
```

最小栈

题目：

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。

实现 `MinStack` 类:

- `MinStack()` 初始化堆栈对象。
- `void push(int val)` 将元素val推入堆栈。
- `void pop()` 删除堆栈顶部的元素。
- `int top()` 获取堆栈顶部的元素。
- `int getMin()` 获取堆栈中的最小元素。

题解：

新开一个栈 `min_s` ，栈顶是最小值，有序栈。`push` 时，若小于栈顶元素，同时压入 `min_s` 中； `pop` 时，若等于栈顶元素，同时弹出 `min_s` 栈顶。

```cpp
class MinStack {
    private:
    stack<int> s, min_s;
    public:
    MinStack() {}
    void push(int x) {
        s.push(x);
        if (min_s.empty() || min_s.top() >= x)
            min_s.push(x);
    }
    void pop() {
        if (!min_s.empty() && min_s.top() == s.top())
            min_s.pop();
        s.pop();
    }
    int top() {
        return s.top();
    }
    int getMin() {
        return min_s.top();
    }
};
```

> “
> 思考：为什么能保证 `min_s` 栈顶元素始终是最小值?

考虑 `min_s` 的变化， `push` 的是比栈顶元素更小的值，一定是新的最小值； `pop` 之后的栈顶值是原来的次小值，原来的最小值去掉后当然是新的最小值。

有效的括号

题目：

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。
- 每个右括号都有一个对应的相同类型的左括号。

> “
> 输入：s = "()[]{}"
> 输出：true

题解：

典型的栈应用，先进后出。

```
bool isValid(string s) {
    stack<char> str;
    for (auto e : s) {
        if (e == '(' || e == '{' || e == '[')
            str.push(e);
        else {
            if (str.empty())
                return false;
            else {
                char e1 = str.top();
                if ((e == ')' && e1 == '(') ||
                    (e == '}' && e1 == '{') ||
                    (e == ']' && e1 == '['))
                    str.pop();
                else
                    return false;
            }
        }
    }
    return str.empty();
}
```

单调栈

每日温度

题目：

给定一个整数数组 `temperatures` ，表示每天的温度，返回一个数组 `answer` ，其中 `answer[i]` 是指对于第 `i` 天，下一个更高温度出现在几天后。如果气温在这之后都不会升高，请在该位置用 `0` 来代替。

> “
> 输入: temperatures = [73,74,75,71,69,72,76,73]
> 输出: [1,1,4,2,1,1,0,0]

题解：

维持一个有序栈，栈顶是当前最低温度对应的日期，碰到比它更低的就 `push` ；碰到比他高的就计算天数之差，然后 `pop` ，继续和栈顶比较。

```
vector<int> dailyTemperatures(vector<int>& temperatures) {
    int n = temperatures.size();
    vector<int> ans(n);
    stack<int> day;
    for (int i = 0; i < n; ++i) {
        while (!day.empty()) {
            int j = day.top();
            if (temperatures[i] <= temperatures[j])
                break;
            else {
                ans[day.top()] = i - j;
                day.pop();
            }
        }
        day.push(i);
    }
    return ans;
}
```

优先队列

优先队列（priority queue）可以在 O(1) 时间内获得最大值，并且可以在 O(log n) 时间内取出最大值或插入任意值。优先队列常常用 `堆（heap）` 来实现。堆是一个完全二叉树，其每个节点的值总是大于等于子节点的值。实际实现堆时，我们通常用一个数组而不是用指针建立一个树。这是因为堆是完全二叉树，所以用数组表示时，位置 i 的节点的父节点位置一定为 `(i-1)/2` ，而它的两个子节点的位置又一定分别为 `2i+1` 和 `2i+2` 。

以下是堆的实现方法，其中最核心的两个操作是上浮和下沉：如果一个节点比父节点大，那么需要交换这个两个节点；交换后还可能比它新的父节点大，因此需要不断地进行比较和交换操作，我们称之为上浮；类似地，如果一个节点比父节小，也需要不断地向下进行比较和交换操作，我们称之为下沉。如果一个节点有两个子节点，我们总是交换最大的子节点。

```
vector<int> heap;
// 获得最大值
```

```
void top() {
    return heap[0];
}
// 插入任意值：把新的数字放在最后一位，然后上浮
void push(int k) {
    heap.push_back(k);
    swim(heap.size() - 1);
}
// 删除最大值：把最后一个数字挪到开头，然后下沉
void pop() {
    heap[0] = heap.back();
    heap.pop_back();
    sink(0);
}
// 上浮
void swim(int pos) {
    while (pos > 0 && heap[(pos-1)/2] < heap[pos])) {
        swap(heap[(pos-1)/2], heap[pos]);
        pos = (pos - 1) / 2;
    }
}
// 下沉
void sink(int pos) {
    while (2 * pos + 1 <= N) {
        int i = 2 * pos + 1;
        if (i < N && heap[i] < heap[i+1])
            ++i;
        if (heap[pos] >= heap[i])
            break;
        swap(heap[pos], heap[i]);
        pos = i;
    }
}
```

另外，正如我们在 STL 章节提到的那样，如果我们需要在维持大小关系的同时，还需要支持查找任意值、删除任意值、维护所有数字的大小关系等操作，可以考虑使用 set 或 map 来代替优先队列。

合并 k 个升序链表

题目：

给你一个链表数组，每个链表都已经按升序排列。请你将所有链表合并到一个升序链表中，返回合并后的链表。

> “
> 输入：lists = [[1,4,5],[1,3,4],[2,6]]
> 输出：[1,1,2,3,4,4,5,6]

题解：

把所有的链表存储在一个优先队列中，每次提取所有链表头部节点值最小的那个节点插入链表中，直到所有链表都被提取完为止。需要设置一个虚拟头结点，方便代码书写。

```
struct Comp {
    bool operator() (ListNode* l1, ListNode* l2) {
        return l1->val > l2->val;
    }
};
ListNode* mergeKLists(vector<ListNode*>& lists) {
    if (lists.empty())
        return nullptr;
    priority_queue<ListNode*, vector<ListNode*>, Comp> q;
    for (ListNode* list: lists) {
        if (list)
            q.push(list);
    }
    ListNode* dummy = new ListNode(0), *cur = dummy;
    while (!q.empty()) {
        cur->next = q.top();
        q.pop();
        cur = cur->next;
        if (cur->next)
            q.push(cur->next);
    }
    return dummy->next;
}
```

> “
> 一次插一个节点，剩下的再放回优先队列

*天际线问题

题目：

城市的 **天际线** 是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。给你所有建筑物的位置和高度，请返回由这些建筑物形成的天际线。

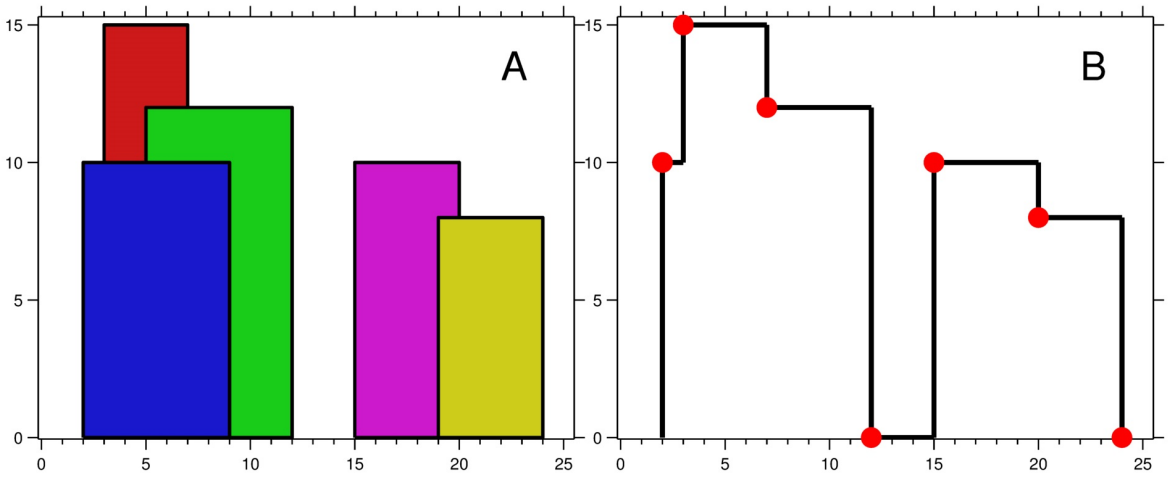
每个建筑物的几何信息由数组 `buildings` 表示，其中三元组 `buildings[i] = [lefti, righti, heighti]` 表示：

- `lefti` 是第 `i` 座建筑物左边缘的 `x` 坐标。
- `righti` 是第 `i` 座建筑物右边缘的 `x` 坐标。
- `heighti` 是第 `i` 座建筑物的高度。

你可以假设所有的建筑都是完美的长方形，在高度为 `0` 的绝对平坦的表面上。

> “

> 输入：buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
> 输出：[[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]



题解：

首先对关键点(每个建筑的左右边缘)升序排列，然后依次找包含(左边缘小于等于，右边缘大于)某关键点的建筑物的最大高度。暴力方法就是对每个关键点都遍历每个建筑，找到最大高度；可以用优先队列优化，`push` 进左边缘符合的建筑，`pop` 出右边缘不符合的建筑，队列头元素始终是最大高度，这样只需遍历建筑一次。

```cpp
vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
    vector<int> boundaries;
    for (auto& building : buildings) {
        boundaries.emplace_back(building[0]);
        boundaries.emplace_back(building[1]);
    }
    // 对边缘排序
    sort(boundaries.begin(), boundaries.end());
    priority_queue<pair<int, int>> que;
    vector<vector<int>> ret;
    int n = buildings.size(), idx = 0;
    for (auto& boundary : boundaries) {
        // push
        while (idx < n && buildings[idx][0] <= boundary) {
            que.emplace(buildings[idx][2], buildings[idx][1]);
            idx++;
        }
        // pop
        while (!que.empty() && que.top().second <= boundary)
            que.pop();
        // 获取天际线高度
        int maxn = que.empty() ? 0 : que.top().first;
        if (ret.size() == 0 || maxn != ret.back()[1])
            ret.push_back({boundary, maxn});
    }
    return ret;
```

```
}
```

> 为什么能只遍历一次而不遗漏呢?

首先，`idx` 一直往前推，会不会漏掉包含后面的关键点的建筑?

不会。如果此建筑还没进 `que`，那么就还在后面，没漏掉；如果此建筑进了 `que`，那么就一定还没被 `pop`，没漏掉（假设被弹出了，因为被 `pop` 的条件是右边缘小于等于上一个关键点，又关键点是升序遍历的，所以右边缘肯定也小于等于此关键点，矛盾！）

其次，会不会有包含此关键点的建筑没被 `push`?

不会，因为 `buildings` 的左端是升序的，而 `push` 结束的条件是左端大于关键点，所以最后一个被 `push` 的建筑往后左端一定不符合。

双端队列

滑动窗口最大值

题目:

给你一个整数数组 `nums`，有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 `k` 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

> 输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
> 输出：[3,3,5,5,6,7]

题解:

每当向右移动时，如果窗口左端就是队列左端，那么 `pop_front` ；把队列右边小于窗口右端的值全部 `pop_back`，再 `push_back(i)`，这样双端队列的最左端永远是当前窗口内的最大值。从左到右递减。

```
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    deque<int> dq;
    vector<int> ans;
    for (int i = 0; i < nums.size(); ++i) {
        if (!dq.empty() && dq.front() == i - k)
            dq.pop_front();
        while (!dq.empty() && nums[dq.back()] < nums[i])
            dq.pop_back();
        dq.push_back(i);
        if (i >= k - 1)
            ans.push_back(nums[dq.front()]);
    }
```

```
    return ans;
}
```

> 总觉得这种东西妙不可言，可能是水平不够🤣

哈希表

两数之和

题目：

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。

> 输入：nums = [2,7,11,15], target = 9
> 输出：[0,1]

题解：

我们可以利用哈希表存储遍历过的值以及它们的位置，每次遍历到位置 i 的时候，查找哈希表里是否存在 target - nums[i]，若存在，则说明这两个值的和为 target。

```
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> hash;
    vector<int> ans;
    for (int i = 0; i < nums.size(); ++i) {
        int num = nums[i];
        auto pos = hash.find(target - num);
        if (pos != hash.end()) {
            ans.push_back(pos->second);
            ans.push_back(i);
            return ans;
        }
        else
            hash[num] = i;
    }
    return ans;
}
```

最长连续序列

题目：

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。

请你设计并实现时间复杂度为 `O(n)` 的算法解决此问题。

> 输入：nums = [100,4,200,1,3,2]
> 输出：4

题解：

我们可以把所有数字放到一个哈希表，然后不断地从哈希表中任意取一个值，并删除掉其之前之后的所有连续数字，然后更新目前的最长连续序列长度。重复这一过程，我们就可以找到所有的连续数字序列。

```
int longestConsecutive(vector<int>& nums) {
    unordered_set<int> hash;
    for (const int & num: nums)
        hash.insert(num);
    int ans = 0;
    while (!hash.empty()) {
        int cur = *(hash.begin());
        hash.erase(cur);
        int next = cur + 1, prev = cur - 1;
        while (hash.count(next))
            hash.erase(next++);
        while (hash.count(prev))
            hash.erase(prev--);
        ans = max(ans, next - prev - 1);
    }
    return ans;
}
```

直线上最多的点数

题目：

给你一个数组 `points` ，其中 `points[i] = [xi, yi]` 表示 **X-Y** 平面上的一个点。求最多有多少个点在同一条直线上。(点不重复)

> 输入：points = [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
> 输出：4

题解：

对每一个点，对不同斜率建立 `hash` 表。

```
int maxPoints(vector<vector<int>>& points) {
```

```
    unordered_map<double, int> hash;
    int max_count = 0, same_x, same_y;
    for (int i = 0; i < points.size(); ++i) {
        same_x = same_y = 1;
        for (int j = i+1; j < points.size(); ++j) {
            if (points[i][1] == points[j][1])
                ++same_y;
            else if (points[i][0] == points[j][0])
                ++same_x;
            else {
                double dx = points[i][0] - points[j][0], dy = points[i]
[1] - points[j][1];
                ++hash[dx/dy];
            }
        }
        max_count = max(max_count, max(same_x, same_y));
        for (auto item : hash)
            max_count = max(max_count, 1+item.second);
        hash.clear();
    }
    return max_count;
}
```

多重集合和映射

重新安排行程

题目：

给你一份航线列表 `tickets` ，其中 `tickets[i] = [fromi, toi]` 表示飞机出发和降落的机场地点。请你对该行程进行重新规划排序。

所有这些机票都属于一个从 `JFK` （肯尼迪国际机场）出发的先生，所以该行程必须从 `JFK` 开始。如果存在多种有效的行程，请你按字典排序返回最小的行程组合。

- 例如，行程 `["JFK", "LGA"]` 与 `["JFK", "LGB"]` 相比就更小，排序更靠前。

假定所有机票至少存在一种合理的行程。且所有的机票 必须都用一次 且 只能用一次。

> “
> 输入：tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]
> 输出：["JFK","ATL","JFK","SFO","ATL","SFO"]
> 解释：另一种有效的行程是 ["JFK","SFO","ATL","JFK","ATL","SFO"] ，但是它字典排序更大更靠后。

题解：

本题可以先用哈希表记录起止机场，其中键是起始机场，值是一个多重集合，表示对应的终止机场。因为一个人可能坐过重复的线路，所以我们需要使用多重集合储存重复值。储存完成之后，我们可以利用栈来恢复从终点到起点飞行的顺序，再将结果逆序得到从起点到终点的顺序。

```cpp
vector<string> findItinerary(vector<vector<string>>& tickets) {
    vector<string> ans;
    if (tickets.empty())
        return ans;
    unordered_map<string, multiset<string>> hash;
    for (const auto & ticket: tickets)
        hash[ticket[0]].insert(ticket[1]);
    stack<string> s;
    s.push("JFK");
    while (!s.empty()) {
        string next = s.top();
        if (hash[next].empty()) {
            ans.push_back(next);
            s.pop();
        }
        else {
            s.push(*hash[next].begin());
            hash[next].erase(hash[next].begin());
        }
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

> “
> 妙哉！

前缀和与积分图

一维的前缀和，二维的积分图，都是把每个位置之前的一维线段或二维矩形预先存储，方便加速计算。如果需要对前缀和或积分图的值做寻址，则要存在哈希表里；如果要对每个位置记录前缀和或积分图的值，则可以储存到一维或二维数组里，也常常伴随着动态规划。

Range Sum Query - Immutable

题目：

Given an integer array `nums`, handle multiple queries of the following type:

Calculate the **sum** of the elements of `nums` between indices `left` and `right` **inclusive** where `left <= right`.

Implement the `NumArray` class:

- `NumArray(int[] nums)` Initializes the object with the integer array `nums`.
- `int sumRange(int left, int right)` Returns the **sum** of the elements of `nums` between indices `left` and `right` **inclusive** (i.e. `nums[left] + nums[left + 1] + ... + nums[right]`).

题解：

建立一个与数组 nums 长度相同的前缀和数组 psum，表示 nums 每个位置之前前所有数字的和。cpp 中可以用 `partial_sum` 函数实现。

```cpp
class NumArray {
    vector<int> psum;
    public:
    NumArray(vector<int>& nums) {
        psum.resize(nums.size() + 1);
        partial_sum(nums.begin(), nums.end(), psum.begin() + 1);
    }
    int sumRange(int left, int right) {
        return psum[right+1] - psum[left];
    }
};
```

Range Sum Query 2D - Immutable

题目：

Given a 2D matrix `matrix`, handle multiple queries of the following type:

Calculate the **sum** of the elements of `matrix` inside the rectangle defined by its **upper left corner** `(row1, col1)` and **lower right corner** `(row2, col2)`.

Implement the `NumMatrix` class:

- `NumMatrix(int[][] matrix)` Initializes the object with the integer matrix `matrix`.
- `int sumRegion(int row1, int col1, int row2, int col2)` Returns the **sum** of the elements of `matrix` inside the rectangle defined by its **upper left corner** `(row1, col1)` and **lower right corner** `(row2, col2)`.

You must design an algorithm where `sumRegion` works on `O(1)` time complexity.

题目：

类似于前缀和，我们可以把这种思想拓展到二维，即积分图（image integral）。我们可以先建立一个 intergral 矩阵，intergral[i][j] 表示以位置 (0, 0) 为左上角、位置 (i-1, j-1) 为右下角的长方形中所有数字的和。计算可以用 dp

```cpp
class NumMatrix {
    vector<vector<int>> integral;
```

```
    public:
    NumMatrix(vector<vector<int>> matrix) {
        int m = matrix.size(), n = m > 0 ? matrix[0].size() : 0;
        integral = vector<vector<int>>(m + 1, vector<int>(n + 1, 0));
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                integral[i][j] = matrix[i-1][j-1] + integral[i-1][j] + integral[i][j-1] - integral[i-1][j-1];
            }
        }
    }
    int sumRegion(int row1, int col1, int row2, int col2) {
        return integral[row2+1][col2+1] - integral[row2+1][col1] - integral[row1][col2+1] + integral[row1][col1];
    }
};
```

Subarray Sum Equals K

题目：

Given an array of integers `nums` and an integer `k`, return *the total number of subarrays whose sum equals to* `k`.

A subarray is a contiguous **non-empty** sequence of elements within an array.

> “
> Input: nums = [1,2,3], k = 3
> Output: 2

题解：

本题同样是利用前缀和，不同的是这里我们使用一个哈希表 hashmap，其键是前缀和，而值是该前缀和出现的次数。在我们遍历到位置 i 时，假设当前的前缀和是 `psum`，那么 `hashmap[psum-k]` 即为以当前位置结尾、满足条件的区间个数。

```
int subarraySum(vector<int>& nums, int k) {
    int count = 0, psum = 0;
    unordered_map<int, int> hashmap;
    hashmap[0] = 1; // 初始化很重要
    for (int i: nums) {
        psum += i;
        count += hashmap[psum-k];
        ++hashmap[psum];
    }
    return count;
}
```

> 妙哉！😘

练习

Reshape the Matrix

题目：

In MATLAB, there is a handy function called `reshape` which can reshape an `m x n` matrix into a new one with a different size `r x c` keeping its original data.

You are given an `m x n` matrix `mat` and two integers `r` and `c` representing the number of rows and the number of columns of the wanted reshaped matrix.

The reshaped matrix should be filled with all the elements of the original matrix **in the same row-traversing order** as they were.

If the `reshape` operation with given parameters is possible and legal, output the new reshaped matrix; Otherwise, output the original matrix.

> **in the same row-traversing order**：相同的行遍历顺序

题解：

行遍历顺序映射到一维数组，再映射回来即可。

```cpp
vector<vector<int>> matrixReshape(vector<vector<int>>& mat, int r, int c)
{
    int m = mat.size(), n = mat[0].size();
    if (m * n != r * c)
        return mat;
    vector<int> tmp(m * n);
    int pos = 0;
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
            tmp[pos++] = mat[i][j];
    pos = 0;
    vector<vector<int>> ans(r, vector<int>(c));
    for (int i = 0; i < r; ++i)
        for (int j = 0; j < c; ++j)
            ans[i][j] = tmp[pos++];
    return ans;
}
```

Implement Stack using Queues

题目：

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (`push` , `top` , `pop` , and `empty`).

Implement the `MyStack` class:

- `void push(int x)` Pushes element x to the top of the stack.
- `int pop()` Removes the element on the top of the stack and returns it.
- `int top()` Returns the element on the top of the stack.
- `boolean empty()` Returns `true` if the stack is empty, `false` otherwise.

题解：

借助辅助队列 `in` 实现栈存储顺序。

```
class MyStack {
    queue<int> in, out;
    public:
    MyStack() { }
    void push(int x) {
        in.push(x);
        while(!out.empty()) {
            in.push(out.front());
            out.pop();
        }
        out = in;
        in = queue<int>();
    }
    int pop() {
        int x = out.front();
        out.pop();
        return x;
    }
    int top() {
        return out.front();
    }
    bool empty() {
        return out.empty();
    }
};
```

Next Greater Element II

题目：

Given a circular integer array `nums` (i.e., the next element of `nums[nums.length - 1]` is `nums[0]`), return *the* ***next greater number*** *for every element in* `nums` .

The **next greater number** of a number `x` is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, return `-1` for this number.

> “
> Input: nums = [1,2,3,4,3]
> Output: [2,3,4,-1,4]

题解：

单调栈加循环数组，一个朴素的思想是，我们可以把这个循环数组「拉直」，即复制该序列的前 n−1 个元素拼接在原序列的后面。这样我们就可以将这个新序列当作普通序列，用单调栈来处理。

其实我们不需要显性地将该循环数组「拉直」，而只需要在处理时对下标取模即可。

```
vector<int> nextGreaterElements(vector<int>& nums) {
    int n = nums.size();
    stack<int> day;
    vector<int> ans(n, -1);
    for (int i = 0; i < n*2 - 1; ++i) {
        while(!day.empty()) {
            int x = day.top();
            if (nums[i%n] > nums[x]) {
                ans[x] = nums[i%n];
                day.pop();
            }
            else
                break;
        }
        if (i < n)  // 避免重复计算
            day.push(i);
    }
    return ans;
}
```

Contains Duplicate

题目：

Given an integer array `nums`, return `true` if any value appears **at least twice** in the array, and return `false` if every element is distinct.

> “
> Input: nums = [1,2,3,1]
> Output: true

题解：

hash表

```cpp
bool containsDuplicate(vector<int>& nums) {
    unordered_set<int> s;
    for (int x: nums) {
        if (s.find(x) != s.end())
            return true;
        s.insert(x);
    }
    return false;
}
```

Degree of an Array

题目：

Given a non-empty array of non-negative integers `nums`, the **degree** of this array is defined as the maximum frequency of any one of its elements.

Your task is to find the smallest possible length of a (contiguous) subarray of `nums`, that has the same degree as `nums`.

> “
> Input: nums = [1,2,2,3,1,4,2]
> Output: 6

题解：

一眼滑动窗口，复习一下啦

```cpp
int findShortestSubArray(vector<int> &nums) {
    int n = nums.size();
    int freq[50000];//记录频数
    memset(freq, 0, sizeof(freq));
    int degree = 0;//记录数组度
    for (int i = 0; i < n; i++)
        degree = max(++freq[nums[i]], degree);
    memset(freq, 0, sizeof(freq));
    int left = 0, right = 0, minSpan = INT_MAX;//窗口边界和最小跨度
    while (right < n) {
        freq[nums[right]]++;//右窗口划进一个数，其频数加一
        while (left <= right && degree == freq[nums[right]]) {
            minSpan = min(minSpan, right - left + 1);//记录最小窗口大小
            freq[nums[left++]]--;//收缩左窗口
        }
        ++right;//扩大右窗口
    }
    return minSpan;
}
```

当然可以用 hash，每一个数映射到一个长度为 3 的数组，数组中的三个元素分别代表这个数出现的次数、这个数在原数组中第一次出现的位置和这个数在原数组中最后一次出现的位置。当我们记录完所有信息后，我们需要遍历该哈希表，找到元素出现次数最多，且前后位置差最小的数。

```cpp
int findShortestSubArray(vector<int>& nums) {
    unordered_map<int, vector<int>> mp;
    int n = nums.size();
    for (int i = 0; i < n; i++) {
        if (mp.count(nums[i])) {
            mp[nums[i]][0]++;
            mp[nums[i]][2] = i;
        }
        else
            mp[nums[i]] = {1, i, i};
    }
    int maxNum = 0, minLen = 0;
    for (auto& [_, vec] : mp) { //c++17 [结构化绑定]，分解初始化
        if (maxNum < vec[0]) {
            maxNum = vec[0];
            minLen = vec[2] - vec[1] + 1;
        }
        else if (maxNum == vec[0]) {
            if (minLen > vec[2] - vec[1] + 1)
                minLen = vec[2] - vec[1] + 1;
        }
    }
    return minLen;
}
```

Longest Harmonious Subsequence

题目：

We define a harmonious array as an array where the difference between its maximum value and its minimum value is **exactly** `1`.

Given an integer array `nums`, return *the length of its longest harmonious subsequence among all its possible subsequences*.

A **subsequence** of array is a sequence that can be derived from the array by deleting some or no elements without changing the order of the remaining elements.

> “
> Input: nums = [1,3,2,2,5,2,3,7]
> Output: 5
> Explanation: The longest harmonious subsequence is [3,2,2,2,3].

题解：

可以先排序，然后滑动窗口。

```
int findLHS(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    int l = 0, r = 0, ans = 0;
    while(r < nums.size()) {
        while(l < r && nums[r] - nums[l] > 1)
            ++l;
        if (nums[r] == nums[l] + 1)
            ans = max(ans, r-l+1);
        ++r;
    }
    return ans;
}
```

用一个哈希映射来存储每个数出现的次数，和谐子序列的长度等于 x 和 x+1 出现的次数和

```
int findLHS(vector<int>& nums) {
    unordered_map<int, int> cnt;
    int res = 0;
    for (int num : nums)
        cnt[num]++;
    for (auto [key, val] : cnt)
        if (cnt.count(key + 1))
            res = max(res, val + cnt[key + 1]);
    return res;
}
```

Find the Duplicate Number

题目：

Given an array of integers `nums` containing `n + 1` integers where each integer is in the range `[1, n]` inclusive.

There is only **one repeated number** in `nums`, return *this repeated number*.

You must solve the problem **without** modifying the array `nums` and uses only constant extra space.

> “
> Input: nums = [1,3,4,2,2]
> Output: 2

题解：

找重复的数和找消失的数差不多，标记负数法。

```cpp
int findDuplicate(vector<int>& nums) {
    for (auto num : nums) {
        int pos = abs(num) - 1;
        if (nums[pos] > 0)
            nums[pos] = -nums[pos];
        else
            return pos + 1;
    }
    return 0;
}
```

Super Ugly Number

题目：

A **super ugly number** is a positive integer whose **prime factors** are in the array `primes`.

Given an integer `n` and an array of integers `primes`, return *the* `nth` ***super ugly number***.

The `nth` **super ugly number** is **guaranteed** to fit in a **32-bit** signed integer.

> “
> **prime factors**：质数因子

> “
> Input: n = 12, primes = [2,7,13,19]
> Output: 32
> Explanation: [1,2,4,7,8,13,14,16,19,26,28,32] is the sequence of the first 12 super ugly numbers given primes = [2,7,13,19].

题解：

优先队列，起始先将最小丑数 1 放入队列，每次从队列取出最小值 x，然后将 x 所对应的丑数 x*primes[i] 进行入队。循环多次，第 n 次出队的值即是答案。

需要防止重复入队。

```cpp
int nthSuperUglyNumber(int n, vector<int>& primes) {
    priority_queue<int, vector<int>, greater<int>> q;   //最小堆
    q.push(1);
    while (1) {
        n--;
        int x = q.top();
        q.pop();
        if (n == 0)
            return x;
        for (int k : primes) {
            if (k <= INT_MAX / x)
                q.push(k * x);
```

```
            if (x % k == 0) //去重
                break;
        }
    }
    return 0;
}
```

> 假设 prime = {2, 3, 5}，那么 $x = 2^i \times 3^j \times 5^k$ ，定义如下：
>
> - 只要 i 不为 0，都由 $2^{(i-1)} \times 3^j \times 5^k$ 生成 x
> - 当 i 为 0 时，只要 j 不为 0， 都由 $3^{(j-1)} \times 5^k$ 生成 x
> - 当 i, j 都为 0 时，由 $5^{(k-1)}$ 生成 x
>
> 就是总是用较大的部分乘以 prime 来生成。

> 数学真头疼😨

Advantage Shuffle

题目：

You are given two integer arrays `nums1` and `nums2` both of the same length. The **advantage** of `nums1` with respect to `nums2` is the number of indices `i` for which `nums1[i] > nums2[i]`.

Return *any permutation of* `nums1` *that maximizes its* ***advantage*** *with respect to* `nums2`.

> Input: nums1 = [2,7,11,15], nums2 = [1,10,4,11]
> Output: [2,11,7,15]

题解：

有重复的数，需要有序，用 `multiset`；田忌赛马原理，要么略赢一筹，要么输得彻底。

```
vector<int> advantageCount(vector<int>& nums1, vector<int>& nums2) {
    int n = nums1.size();
    vector<int> ans(n);
    multiset<int> h;
    for (int i = 0; i < n; ++i)
        h.insert(nums1[i]);
    for (int i = 0; i < n; ++i) {
        auto it = h.upper_bound(nums2[i]);
        if (it == h.end())
            it = h.begin();
        ans[i] = *it;
        h.erase(it);
    }
```

```
14      return ans;
15  }
```

字符串

字符串比较

Valid Anagram

题目：

Given two strings `s` and `t`, return `true` *if* `t` *is an* ***anagram*** *of* `s`, *and* `false` *otherwise.*

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

> **anagram**：相同字母异序词

> Input: s = "anagram", t = "nagaram"
> Output: true

题解：

直接 hash 数组即可

```
bool isAnagram(string s, string t) {
    if (s.length() != t.length())
        return false;
    int hash[26];
    memset(hash, 0, sizeof(hash));
    for (int i = 0; i < s.length(); ++i) {
        ++hash[s[i] - 'a'];
        --hash[t[i] - 'a'];
    }
    for (int i = 0; i < 26; ++i) {
        if (hash[i])
            return false;
    }
    return true;
}
```

Isomorphic Strings

题目：

Given two strings `s` and `t`, *determine if they are isomorphic.*

Two strings `s` and `t` are isomorphic if the characters in `s` can be replaced to get `t`.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

> Input: s = "egg", t = "add"
> Output: true

题解：

用数组记录字符上次出现的位置

```
bool isIsomorphic(string s, string t) {
    if (s.length() != t.length())
        return false;
    int hash1[128], hash2[128];
    memset(hash1, 0, sizeof(hash1));
    memset(hash2, 0, sizeof(hash2));
    for (int i = 0; i < s.length(); ++i) {
        if (hash1[s[i]] != hash2[t[i]])
            return false;
        hash1[s[i]] = hash2[t[i]] = i + 1;
    }
    return true;
}
```

Palindromic Substrings

题目：

Given a string `s`, return *the number of **palindromic substrings** in it*.

A string is a **palindrome** when it reads the same backward as forward.

A **substring** is a contiguous sequence of characters within the string.

> Input: s = "aaa"
> Output: 6
> Explanation: Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".

题解：

从每个位置开始从向左右延伸为回文字符串，注意奇偶长度，可以设置辅助函数

```
int countSubstrings(string s) {
    int count = 0;
    for (int i = 0; i < s.length(); ++i) {
```

```
        count += extendSubstrings(s, i, i); // 奇数长度
        count += extendSubstrings(s, i, i + 1); // 偶数长度
    }
    return count;
}
int extendSubstrings(string s, int l, int r) {
    int count = 0;
    while (l >= 0 && r < s.length() && s[l] == s[r]) {
        --l;
        ++r;
        ++count;
    }
    return count;
}
```

Count Binary Substrings

题目：

Given a binary string `s`, return the number of non-empty substrings that have the same number of `0`'s and `1`'s, and all the `0`'s and all the `1`'s in these substrings are grouped consecutively.

Substrings that occur multiple times are counted the number of times they occur.

> “
> Input: s = "00110011"
> Output: 6
> Explanation: There are 6 substrings that have equal number of consecutive 1's and 0's: "0011", "01", "1100", "10", "0011", and "01".

题解：

从左往右遍历数组，记录和当前位置数字相同且连续的长度，以及其之前连续的不同数字的长度。举例来说，对于 00110 的最后一位，我们记录的相同数字长度是 1，因为只有一个连续 0；我们记录的不同数字长度是 2，因为在 0 之前有两个连续的 1。若不同数字的连续长度大于等于当前数字的连续长度，则说明存在一个且只存在一个以当前数字结尾的满足条件的子字符串。

```
int countBinarySubstrings(string s) {
    int pre = 0, cur = 1, count = 0;
    for (int i = 1; i < s.length(); ++i) {
        if (s[i] == s[i-1])
            ++cur;
        else {
            pre = cur;
            cur = 1;
        }
        if (pre >= cur)
```

```
            ++count;
        }
        return count;
    }
```

字符串理解

Basic Calculator II

题目：

Given a string `s` which represents an expression, *evaluate this expression and return its value*.

The integer division should truncate toward zero.

You may assume that the given expression is always valid. All intermediate results will be in the range of `[-2^31, 2^31 - 1]`.

> “
> Input: s = "3+2*2"
> Output: 7

题解：

用栈实现，碰到加减号就入栈，碰到乘除号就出栈相乘除，结果入栈。

```
int calculate(string s) {
    vector<int> stk;
    char preSign = '+';
    int num = 0;
    for (int i = 0; i < s.length(); ++i) {
        if (isdigit(s[i]))
            num = num * 10 + (s[i] - '0');  //注意加括号防止 int 溢出
        if (!isdigit(s[i]) && s[i] != ' ' || i == s.length()-1) {
            switch (preSign) {
                case '+':
                    stk.push_back(num);
                    break;
                case '-':
                    stk.push_back(-num);
                    break;
                case '*':
                    stk.back() *= num;
                    break;
                default:
                    stk.back() /= num;
            }
            preSign = s[i];
            num = 0;
```

```
        }
    }
    return accumulate(stk.begin(), stk.end(), 0);
}
```

字符串匹配

Find the Index of the First Occurrence in a String

题目：

Given two strings `needle` and `haystack`, return the index of the first occurrence of `needle` in `haystack`, or `-1` if `needle` is not part of `haystack`.

> “
> Input: haystack = "sadbutsad", needle = "sad"
> Output: 0
> Explanation: "sad" occurs at index 0 and 6.

题解：

KMP 算法来喽

```
void Set_Next(vector<int> &next, string t) {
    int j = 0, k = -1;
    next[0] = -1;
    while(j < t.length()) {
        if (k == -1 || t[j] == t[k]) {
            ++j;
            ++k;
            if (t[j] != t[k]) next[j] = k;
            else next[j] = next[k];
        }
        else k = next[k];
    }
}
int strStr(string s, string t) {
    int m = s.length(), n = t.length();
    vector<int> next(n+1);
    Set_Next(next, t);
    int i = 0, j = 0;
    while (i < m && j < n) {
        if (j == -1 || s[i] == t[j]) {
            ++i;
            ++j;
        }
        else j = next[j];
    }
```

```
    if (j >= n) return (i - t.length());
    return -1;
}
```

> 原理可参考：模式匹配 KMP 算法 | Liano-Blog

练习

Longest Palindrome

题目：

Given a string `s` which consists of lowercase or uppercase letters, return *the length of the **longest palindrome*** that can be built with those letters.

Letters are **case sensitive**, for example, `"Aa"` is not considered a palindrome here.

> Input: s = "abccccdd"
> Output: 7
> Explanation: One longest palindrome that can be built is "dccaccd", whose length is 7.

题解：

用 `hash` 表记录字符出现个数，每两个就可以令回文字符串长度增加 2，如果有奇数个，那么最后回文字符串长度可以再加 1

```
int longestPalindrome(string s) {
    unordered_map<char, int> count;
    int ans = 0;
    for (char c : s)
        ++count[c];
    for (auto p : count) {
        int v = p.second;
        ans += v / 2 * 2;
        if (v % 2 == 1 && ans % 2 == 0)
            ++ans;
    }
    return ans;
}
```

Longest Substring Without Repeating Characters

题目：

Given a string `s`, find the length of the **longest substring** without repeating characters.

> “
> Input: s = "abcabcbb"
> Output: 3

题解：

滑窗，滑呀滑

```
int lengthOfLongestSubstring(string s) {
    if(s.length() == 0)
        return 0;
    unordered_set<char> hash;
    int ans = 0, l = 0;
    for(int i = 0; i < s.length(); ++i){
        while (hash.find(s[i]) != hash.end()){
            hash.erase(s[l]);
            l++;
        }
        ans = max(ans, i - l + 1);
        hash.insert(s[i]);
    }
    return ans;
}
```

Basic Calculator III

题目：

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open `(` and closing parentheses `)`, the plus `+` or minus sign `-`, non-negative integers and empty spaces ``.

The expression string contains only non-negative integers, `+`, `-`, `*`, `/` operators , open `(` and closing parentheses `)` and empty spaces ``. The integer division should truncate toward zero.

> “
> input: (2+6 * 3+5- (3 * 15 / 7+2) * 5)+3
> output: -12

题解：

表达式计算 plus 版，其实没区别，遇到括号就递归计算，然后跳到右括号处。

```
int findClosing(string s) {
    int level = 0, i = 0;
    for (i = 0; i < s.length(); ++i) {
        if (s[i] == '(')
            level++;
```

```
        else if (s[i] == ')') {
            level--;
            if (level == 0) break;
        }
        else continue;
    }
    return i;
}
int calculate(string s) {
    vector<int> stk;
    char preSign = '+';
    int num = 0;
    for (int i = 0; i < s.length(); ++i) {
        if (isdigit(s[i]))
            num = num * 10 + (s[i] - '0');
        if (s[i] == '(') {
            int j = findClosing(s.substr(i));
            num = calculate(s.substr(i+1, j-1));
            i += j;
        }
        if (!isdigit(s[i]) && s[i] != ' ' || i == s.length()-1) {
            switch (preSign) {
                case '+':
                    stk.push_back(num);
                    break;
                case '-':
                    stk.push_back(-num);
                    break;
                case '*':
                    stk.back() *= num;
                    break;
                case '/':
                    stk.back() /= num;
                    break;
            }
            preSign = s[i];
            num = 0;
        }
    }
    return accumulate(stk.begin(), stk.end(), 0);
}
```

> “
> 没会员，摆烂

Longest Palindromic Substring

题目：

Given a string `s`, return *the longest palindromic substring* in `s`.

> “
> Input: s = "babad"
> Output: "bab"

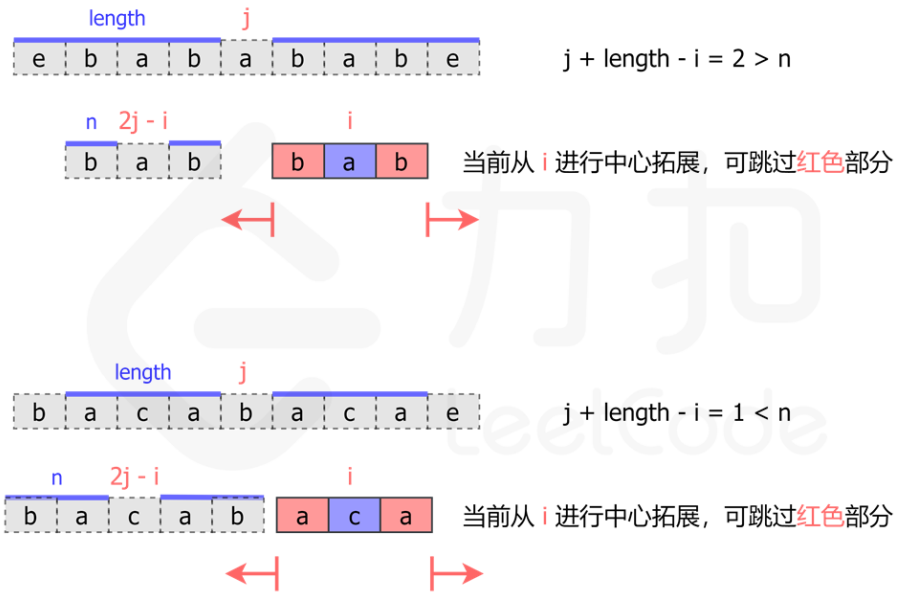
题解：

滑窗法上面题目说过了，二维动态规划也可以做，这里介绍 `Manacher` 算法，时间复杂度 `O(n)`，嘎嘎快。

> “
> 直接搬力扣官方题解了。越来越懒了😳

在中心扩展算法的过程中，我们能够得出每个位置的臂长。那么当我们要得出以下一个位置 i 的臂长时，能不能利用之前得到的信息呢?

答案是肯定的。具体来说，如果位置 j 的臂长为 length，并且有 j + length > i，如下图所示:



当在位置 i 开始进行中心拓展时，我们可以先找到 i 关于 j 的对称点 2 * j - i。那么如果点 2 * j - i 的臂长等于 n，我们就可以知道，点 i 的臂长至少为 min(j + length - i, n)。那么我们就可以直接跳过 i 到 i + min(j + length - i, n) 这部分，从 i + min(j + length - i, n) + 1 开始拓展。

我们只需要在中心扩展法的过程中记录右臂在最右边的回文字符串，将其中心作为 j，在计算过程中就能最大限度地避免重复计算。

那么现在还有一个问题：如何处理长度为偶数的回文字符串呢?

我们可以通过一个特别的操作将奇偶数的情况统一起来：我们向字符串的头尾以及每两个字符中间添加一个特殊字符 #，比如字符串 aaba 处理后会变成 #a#a#b#a#。那么原先长度为偶数的回文字符串 aa 会变成长度为奇数的回文字符串 #a#a#，而长度为奇数的回文字符串 aba 会变成长度仍然为奇数的回文字符串 #a#b#a#，我们就不需要再考虑长度为偶数的回文字符串了。

注意这里的特殊字符不需要是没有出现过的字母，我们可以使用任何一个字符来作为这个特殊字符。这是因为，当我们只考虑长度为奇数的回文字符串时，每次我们比较的两个字符奇偶性一定是相同的，所以原来字符串中的字符不会与插入的特殊字符互相比较，不会因此产生问题。

```
int expand(const string& s, int left, int right) {
    while (left >= 0 && right < s.size() && s[left] == s[right]) {
        --left;
        ++right;
    }
    return (right - left - 2) / 2;
}
string longestPalindrome(string s) {
    int start = 0, end = -1;
    string t = "#";
    for (char c: s) {
        t += c;
        t += '#';
    }
    t += '#';
    s = t;
    vector<int> arm_len;
    int right = -1, j = -1;
    for (int i = 0; i < s.size(); ++i) {
        int cur_arm_len;
        if (right >= i) {
            int i_sym = j * 2 - i;
            int min_arm_len = min(arm_len[i_sym], right - i);
            cur_arm_len = expand(s, i - min_arm_len, i + min_arm_len);
        }
        else
            cur_arm_len = expand(s, i, i);
        arm_len.push_back(cur_arm_len);
        if (i + cur_arm_len > right) {
            j = i;
            right = i + cur_arm_len;
        }
        if (cur_arm_len * 2 + 1 > end - start) {
            start = i - cur_arm_len;
            end = i + cur_arm_len;
        }
    }
```

```
    string ans;
    for (int i = start; i <= end; ++i) {
        if (s[i] != '#')
            ans += s[i];
    }
    return ans;
}
```

链表

链表是由节点和指针构成的数据结构，每个节点存有一个值，和一个指向下一个节点的指针，因此很多链表问题可以用递归来处理。LeetCode 默认的链表表示方法如下：

```cpp
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) {}
};
```

由于在进行链表操作时，尤其是删除节点时，经常会因为对当前节点进行操作而导致内存或指针出现问题。有两个小技巧可以解决这个问题：一是尽量处理当前节点的下一个节点而非当前节点本身，二是建立一个虚拟节点 (`dummy node`)，使其指向当前链表的头节点，这样即使原链表所有节点全被删除，也会有一个 `dummy` 存在，返回 `dummy->next` 即可。

基本操作

Reverse Linked List

题目：

Given the `head` of a singly linked list, reverse the list, and return *the reversed list*.

题解：

递归写法：

```cpp
ListNode* reverseList(ListNode* head, ListNode* prev=nullptr) {
    if (!head) {
        return prev;
    }
    ListNode* next = head->next;
    head->next = prev;
    return reverseList(next, head);
}
```

非递归写法：

```cpp
ListNode* reverseList(ListNode* head) {
    ListNode *prev = nullptr, *next;
    while (head) {
        next = head->next;
        head->next = prev;
        prev = head;
        head = next;
    }
    return prev;
}
```

Merge Two Sorted Lists

题目：

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists in a one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list.*

> “
> Input: list1 = [1,2,4], list2 = [1,3,4]
> Output: [1,1,2,3,4,4]

题解：

递归写法：

```cpp
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    if (!l2)
        return l1;
    if (!l1)
        return l2;
    if (l1->val > l2->val) {
        l2->next = mergeTwoLists(l1, l2->next);
        return l2;
    }
    l1->next = mergeTwoLists(l1->next, l2);
    return l1;
}
```

非递归写法：

```cpp
ListNode* mergeTwoLists(ListNode *l1, ListNode *l2) {
    ListNode *dummy = new ListNode(0), *node = dummy;
    while (l1 && l2) {
        if (l1->val <= l2->val) {
            node->next = l1;
```

```
            l1 = l1->next;
        }
        else {
            node->next = l2;
            l2 = l2->next;
        }
        node = node->next;
    }
    node->next = l1 ? l1 : l2;
    return dummy->next;
}
```

Swap Nodes in Pairs

题目：

Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

> “
> Input: head = [1,2,3,4]
> Output: [2,1,4,3]

题解：

递归写法：

```
ListNode* swapPairs(ListNode* head) {
    if (!head || !head->next)
        return head;
    ListNode* newHead = head->next;
    head->next = swapPairs(newHead->next);
    newHead->next = head;
    return newHead;
}
```

非递归写法：

创建虚拟头结点 `dummyHead`，令 `temp` 表示当前到达的节点，初始时 `temp = dummyHead`。每次需要交换 `temp` 后面的两个节点。

如果 `temp` 的后面没有节点或者只有一个节点，则没有更多的节点需要交换，因此结束交换。否则，获得 `temp` 后面的两个节点 `node1` 和 `node2`，通过更新节点的指针关系实现两两交换节点。

具体而言，交换之前的节点关系是 `temp -> node1 -> node2`，交换之后的节点关系要变成 `temp -> node2 -> node1`，因此需要进行如下操作。

```
temp.next = node2
node1.next = node2.next
node2.next = node1
```

完成上述操作之后，节点关系即变成 `temp -> node2 -> node1`。再令 `temp = node1`，对链表中的其余节点进行两两交换，直到全部节点都被两两交换。

```
ListNode* swapPairs(ListNode* head) {
    ListNode* dummyHead = new ListNode(0);
    dummyHead->next = head;
    ListNode* temp = dummyHead;
    while (temp->next != nullptr && temp->next->next != nullptr) {
        ListNode* node1 = temp->next;
        ListNode* node2 = temp->next->next;
        temp->next = node2;
        node1->next = node2->next;
        node2->next = node1;
        temp = node1;
    }
    return dummyHead->next;
}
```

> 原则上不对当前节点进行操作，所以会有 `dummyHead` 和 `temp`

其他技巧

Intersection of Two Linked Lists

题目：

Given the heads of two singly linked-lists `headA` and `headB`, return *the node at which the two lists intersect*. If the two linked lists have no intersection at all, return `null`.

Note that the linked lists must **retain their original structure** after the function returns.

> **intersect**: 交集

题解：

双指针法，我们使用两个指针，分别指向两个链表的头节点，并以相同的速度前进，若到达链表结尾，则移动到另一条链表的头节点继续前进。按照这种前进方法，两个指针会在 `a + b + c` 次前进后同时到达相交节点。

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    ListNode *l1 = headA, *l2 = headB;
    while (l1 != l2) {
        l1 = l1 ? l1->next : headB;
        l2 = l2 ? l2->next : headA;
    }
    return l1;
}
```

Palindrome Linked List

题目：

Given the `head` of a singly linked list, return `true` *if it is a palindrome or* `false` *otherwise*.

题解：

可以直接把值复制进数组，对数组操作，但这样就失去了链表的乐趣🤣

这里先使用快慢指针找到链表中点，快指针一次走两格，慢指针一次走一格，快指针走到终点时慢指针指向中点；再把链表切成两半，然后把后半段翻转；最后比较两半是否相等。

```
// 主函数
bool isPalindrome(ListNode* head) {
    if (!head || !head->next)
        return true;
    ListNode *slow = head, *fast = head;
    while (fast->next && fast->next->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    slow->next = reverseList(slow->next);
    slow = slow->next;
    while (slow) {
        if (head->val != slow->val)
            return false;
        head = head->next;
        slow = slow->next;
    }
    return true;
}
// 辅函数
ListNode* reverseList(ListNode* head) {
    ListNode *prev = nullptr, *next;
    while (head) {
        next = head->next;
        head->next = prev;
        prev = head;
```

```
        head = next;
    }
    return prev;
}
```

练习

Remove Duplicates from Sorted List

题目：

Given the `head` of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list* ***sorted*** *as well*.

题解：

直接一次遍历即可

```
ListNode* deleteDuplicates(ListNode* head) {
    if (!head)
        return head;
    ListNode* cur = head;
    while (cur->next) {
        if (cur->val == cur->next->val)
            cur->next = cur->next->next;
        else
            cur = cur->next;
    }
    return head;
}
```

> “
> 正常工程代码一定要记得 delete

Odd Even Linked List

题目：

Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in `O(1)` extra space complexity and `O(n)` time complexity.

> “

> Input: head = [2,1,3,5,6,4,7]
> Output: [2,3,6,7,1,5,4]

题解：

以第一个节点，第二个节点分别作为奇偶链表的头结点，然后隔一个添加一个节点。最后链接两个链表。

```
ListNode* oddEvenList(ListNode* head) {
    if (!head)
        return head;
    ListNode *odd = head, *evenhead = head->next, *even = evenhead;
    while (even && even->next) {
        odd->next = odd->next->next;
        even->next = even->next->next;
        odd = odd->next;
        even = even->next;
    }
    odd->next = evenhead;
    return head;
}
```

Remove Nth Node From End of List

题目：

Given the `head` of a linked list, remove the `nth` node from the end of the list and return its head.

题解：

使用双指针，快指针比慢指针提前 n 个节点，那么快指针到终点时，慢指针指向位置即倒数第 n 个节点。为了便于删除，我们需要慢指针再慢一个节点，最后慢指针指向目标删除节点的前驱。

```
ListNode* removeNthFromEnd(ListNode* head, int n) {
    ListNode* dummy = new ListNode(0, head);
    ListNode* first = head;
    ListNode* second = dummy;
    for (int i = 0; i < n; ++i)
        first = first->next;
    while (first) {
        first = first->next;
        second = second->next;
    }
    second->next = second->next->next;
    return dummy->next;
}
```

Sort List

题目：

Given the `head` of a linked list, return *the list after sorting it in* ***ascending order***.

> “
> Input: head = [-1,5,3,4,0]
> Output: [-1,0,3,4,5]

题解：

快慢指针找中点，归并排序。递归实现

```
ListNode* sortList(ListNode* head) {
    return sortList(head, nullptr);
}
ListNode* sortList(ListNode* head, ListNode* tail) {
    if (head == nullptr)
        return head;
    if (head->next == tail) {
        head->next = nullptr;
        return head;
    }
    ListNode* slow = head, *fast = head;
    while (fast != tail && fast->next != tail) {
        slow = slow->next;
        fast = fast->next->next;
    }
    ListNode* mid = slow;
    return merge(sortList(head, mid), sortList(mid, tail));
}
ListNode* merge(ListNode* head1, ListNode* head2) {
    ListNode* dummyHead = new ListNode(0);
    ListNode* temp = dummyHead, *temp1 = head1, *temp2 = head2;
    while (temp1 != nullptr && temp2 != nullptr) {
        if (temp1->val <= temp2->val) {
            temp->next = temp1;
            temp1 = temp1->next;
        }
        else {
            temp->next = temp2;
            temp2 = temp2->next;
        }
        temp = temp->next;
    }
    if (temp1 != nullptr)
        temp->next = temp1;
```

```
    else if (temp2 != nullptr)
        temp->next = temp2;
    return dummyHead->next;
}
```

树

作为链表的升级版，我们通常接触的树都是二叉树（`binary tree`），即每个节点最多有两个子节点；LeetCode 默认的树表示方法如下：

```
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) {}
};
```

树的递归

Maximum Depth of Binary Tree

题目：

Given the `root` of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

题解：

`one-line code`

```
int maxDepth(TreeNode* root) {
    return root? 1 + max(maxDepth(root->left), maxDepth(root->right)): 0;
}
```

> “
> cool! 😘

Balanced Binary Tree

题目：

Given a binary tree, determine if it is **height-balanced**.

题解：

```cpp
// 主函数
bool isBalanced(TreeNode* root) {
    return helper(root) != -1;
}
// 辅函数
int helper(TreeNode* root) {
    if (!root)
        return 0;
    int left = helper(root->left), right = helper(root->right);
    if (left == -1 || right == -1 || abs(left - right) > 1)
        return -1;  // 避免重复计算
    return 1 + max(left, right);
}
```