

要点

只关心返回值

有多个重复片段可以利用函数或lambda减少代码重复

各种排序!!!!!!!!!!!!!!!!!!!!!!

拓扑排序

每一题用最优解思考但也要会非最优解

48 65 97 ASCLL 0 A a

字符转整数 加减法

注意边界条件 如空数组 全不满足 等等

复习每题时间复杂度 适当背题

不要脑袋空想 打草稿更快

acm模式 listnode都要自己写

写复杂的题写注释才好写 复杂题先想想题目可以用什么方法 想好在写 一般题目不会直接暴力求

$a+b=c \rightarrow a=c-b$

寻找是否存在 用哈希表

ip地址

写完代码检查一下是否能缩减

字符串操作大全

字符转整数

字符串分割

c++和c方法

手撕 池组件 智能指针 stl容器等 网络编程 锁 各种锁乐观读写等 运算符重载 设计模式 lambda 大小根堆 排序

注意lambda两种递归形式：

```
function<void(int,int)> dfs = [&](int i,int j){} //使用function定义的lambda表达式  
在{}内可以调用自己实现递归  
//function将lambda表达式装进去
```

```
auto dfs = [&](auto&& dfs, int i, int j) -> void {} //使用auto定义必须在形参中加入自  
身才能在{}调用自己实现递归
```

std::function 和 auto 的区别

- `std::function`：是一种通用的、类型安全的函数包装器，允许将各种可调用对象（如普通函数、lambda、函数对象等）赋值给它。`std::function` 可以持有自身的引用，这使得它支持递归定义。
- `auto`：在声明 lambda 表达式时，用 `auto` 推导 lambda 的具体类型。然而，lambda 表达式本身不是一个真正的类型（它的类型是编译器生成的匿名类型），所以 `auto` 仅能推导它的类型而无法在声明中递归调用自己。

面试前必看

写复杂的题写注释才好写 复杂题先想想题目可以用什么方法 想好在写 拿笔画一画不要偷懒 一般题目不会直接暴力求

1.string各种操作 substr 拼接

2.map各种操作

3.priority_queue 各种操作 自定义比较

4.字符大小写转换的加减法

5.algorithm一些常用函数

6.*max_element

一、哈希

两数之和

字母异位词分组

二、双指针

相向双指针

1.两数之和 II - 输入有序数组

给你一个下标从 1 开始的整数数组 `numbers`，该数组已按 **非递减顺序排列**，请你从数组中找出满足相加之和等于目标数 `target` 的两个数。如果设这两个数分别是 `numbers[index1]` 和 `numbers[index2]`，则 $1 \leq \text{index1} < \text{index2} \leq \text{numbers.length}$ 。

以长度为 2 的整数数组 `[index1, index2]` 的形式返回这两个整数的下标 `index1` 和 `index2`。

你可以假设每个输入 **只对应唯一的答案**，而且你 **不可以** 重复使用相同的元素。

你所设计的解决方案必须只使用常量级的额外空间。

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        ranges::sort(nums);
        vector<vector<int>> ans;
        int n = nums.size();
        for (int i = 0; i < n - 2; i++) {
            int x = nums[i];
            if (i && x == nums[i - 1]) continue; // 跳过重复数字
            if (x + nums[i + 1] + nums[i + 2] > 0) break; // 优化一
            if (x + nums[n - 2] + nums[n - 1] < 0) continue; // 优化二
            int j = i + 1, k = n - 1;
            while (j < k) {
                int s = x + nums[j] + nums[k];
                if (s > 0) {
                    k--;
                } else if (s < 0) {
                    j++;
                } else {
                    ans.push_back({x, nums[j], nums[k]});
                    for (j++; j < k && nums[j] == nums[j - 1]; j++); // 跳过重复数字
                    for (k--; k > j && nums[k] == nums[k + 1]; k--); // 跳过重复数字
                }
            }
        }
        return ans;
    }
};
```

```

    }
};
/*
先排序 再双向指针
*/

```

2.移动零

3.三数之和

```

class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        ranges::sort(nums);
        vector<vector<int>> ans;
        int n = nums.size();
        for (int i = 0; i < n - 2; i++) {
            int x = nums[i];
            if (i && x == nums[i - 1]) continue; // 跳过重复数字
            if (x + nums[i + 1] + nums[i + 2] > 0) break; // 优化一
            if (x + nums[n - 2] + nums[n - 1] < 0) continue; // 优化二
            int j = i + 1, k = n - 1;
            while (j < k) {
                int s = x + nums[j] + nums[k];
                if (s > 0) {
                    k--;
                } else if (s < 0) {
                    j++;
                } else {
                    ans.push_back({x, nums[j], nums[k]});
                    for (j++; j < k && nums[j] == nums[j - 1]; j++); // 跳过重复数字
                    for (k--; k > j && nums[k] == nums[k + 1]; k--); // 跳过重复数字
                }
            }
        }
        return ans;
    }
};

```

```

class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        sort(nums.begin(), nums.end());
    }
};

```

```

vector<vector<int>> result;
for(int i=0;i<nums.size()-2;i++)
{
    int left=i+1;
    int right=nums.size()-1;
    if(i&&nums[i]==nums[i-1]) continue;
    while(left<right)
    {
        if(nums[left]+nums[right]+nums[i]<0) left++;
        else if(nums[right]+nums[left]+nums[i]>0) right--;
        else if(nums[left]+nums[right]+nums[i]==0)
        {
            result.push_back({nums[i],nums[left],nums[right]});
            while (left < right && nums[left] == nums[left + 1]) left++;
            while (left < right && nums[right] == nums[right - 1]) right--;
        }
    }
}

return result;
};

```

4.盛最多水的容器

将两个指针放在两端

返回值应该是面积，所以只关心面积（保留max_area），其他量不一定要存储（组成max_area的索引）

左右指针在两端，长度已经最长了，只需要找到更高的检查面积

$(a-b) \times \min(x, y)$ $(a-b)$ 尽量长 限制于 $\min(x-y)$ $\min(x, y)$ 限制于短边 所以要找更长的边

```

class Solution {
public:
    int maxArea(vector<int>& height) {
        int left = 0;
        int right = height.size() - 1;
        int max_area = 0;

        while (left < right) {
            // 计算当前左右指针形成的面积
            int area = (right - left) * min(height[left], height[right]);
            max_area = max(max_area, area);

            // 移动较小的一边，试图找到更高的边，增加面积

```

```

        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }

    return max_area;
}

};

```

5.接雨水

```

class Solution {
public:
    int trap(vector<int>& height) {
        vector<int> left_max(height.size(),0);
        vector<int> right_max(height.size(),0);
        left_max[0]=height[0];
        right_max[height.size()-1] = height[height.size()-1];
        for(int i=1;i<height.size();i++)
        {
            left_max[i]=max(left_max[i-1],height[i]);
        }
        for(int i=height.size()-2;i>=0;i--)
        {
            right_max[i]=max(right_max[i+1],height[i]);
        }

        int sum=0;
        for(int i=0;i<height.size();i++)
        {
            sum+=(min(left_max[i],right_max[i]))-height[i];
        }

        return sum;
    }
};

```

/*

left_max 和 right_max 没有预分配大小： 在你的代码中，left_max[0] = height[0] 和 right_max[height.size()-1] = height[height.size()-1] 直接访问了未分配大小的向量，导致越界错误。你需要在使用之前分配适当的大小。

*/

你可以用双指针法来同时计算左右最大值，省去 left_max 和 right_max 数组，从而将空间复杂度降低为 $O(1)$ 。以下是优化后的代码：

```

class Solution {
public:
    int trap(vector<int>& height) {
        if (height.empty()) return 0;

```

```

int left = 0, right = height.size() - 1;
int left_max = 0, right_max = 0;
int sum = 0;

while (left < right) {
    if (height[left] < height[right]) {
        if (height[left] >= left_max) {
            left_max = height[left];
        } else {
            sum += left_max - height[left];
        }
        left++;
    } else {
        if (height[right] >= right_max) {
            right_max = height[right];
        } else {
            sum += right_max - height[right];
        }
        right--;
    }
}

return sum;
}
};
/*

```

代码的详细步骤：

边界条件： 首先，代码检查输入数组是否为空，如果为空，直接返回 0，因为没有柱子自然也无法积水。

初始化变量：

left 和 **right**：两个指针分别指向数组的最左边和最右边，表示从数组两端向中间靠拢。

left_max 和 **right_max**：用于记录从左边和右边遍历过程中遇到的最大柱子高度。用于决定当前列可以积水的高度。

sum：用于累计接到的雨水量。

主循环： 代码通过 **while(left < right)** 循环，表示在左右指针还没有相遇时，不断处理左右两侧的柱子。

比较 **height[left]** 和 **height[right]**，选择较小的高度作为处理的起点。雨水量由较小的一侧决定，所以优先移动较矮的那一侧。

处理较小的一侧： 如果 **left** 较小，首先检查当前左边柱子的高度是否大于或等于之前记录的左侧最大高度 **left_max**：

如果大于或等于，说明当前柱子不能积水，因为它成为了新的最高柱子。更新 **left_max**。

如果小于，说明它位于较高柱子之间，可以积水。积水量为 **left_max - height[left]**。

处理完后，将 **left** 向右移动一格，继续处理。

如果 **right** 较小，右侧的处理逻辑与左侧类似。比较 **height[right]** 和 **right_max**：

如果当前右边柱子的高度大于或等于 **right_max**，则更新 **right_max**，否则计算积水量并将 **right** 向左移动。

结束条件： 当 **left** 和 **right** 相遇时，循环结束，所有可能积水的位置都已经处理完毕，**sum** 中累积的值就是接到的雨水总量。

返回结果： 最终返回 **sum**，即总的雨水量。

*/

三、滑动窗口

```
class Solution:
    def problemName(self, s: str) -> int:
        # Step 1: 定义需要维护的变量们 (对于滑动窗口类题目, 这些变量通常是最小长度, 最大长度,
        # 或者哈希表)
        x, y = ..., ...
        # Step 2: 定义窗口的首尾端 (start, end), 然后滑动窗口
        start = 0
        for end in range(len(s)):
            # Step 3: 更新需要维护的变量, 有的变量需要一个if语句来维护 (比如最大最小长度)

            x = new_x
            if condition:
                y = new_y
            ...

            ----- 下面是两种情况, 读者请根据题意二选1 -----
            ...

            # Step 4 - 情况1
            # 如果题目的窗口长度固定: 用一个if语句判断一下当前窗口长度是否达到了限定长度

            # 如果达到了, 窗口左指针前移一个单位, 从而保证下一次右指针右移时, 窗口长度保持不变,

            # 左指针移动之前, 先更新Step 1定义的(部分或所有)维护变量
            if 窗口长度达
            到了限定长度:
                # 更新 (部分或所有) 维护变量
                # 窗口左指针前移一个单位保证下一次右指针右移时窗口长度保持不变
            # Step 4 - 情况2
            # 如果题目的窗口长度可变: 这个时候一般涉及到窗口是否合法的问题
            # 如果当前窗口不合法时, 用一个while去不断移动窗口左指针, 从而剔除非法元素直到窗口再次
            合法

            # 在左指针移动之前更新Step 1定义的(部分或所有)维护变量
            while 不合法:
                # 更新 (部分或所有) 维护变量
                # 不断移动窗口左指针直到窗口再次合法
            # Step 5: 返回答案
            return ...
```

1.无重复字符的最长子串

多用set map 效率高

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        vector<char> str;
        int length_max=0;
        int length;
        for(int i=0;i<s.size();i++)
        {

            while((find(str.begin(),str.end(),s[i]))!=str.end())
            {
```



```

        str.erase(str.begin());
    }
    str.emplace_back(s[i]);
    length=str.size();
    length_max = max(length,length_max);
}

return length_max;
}
};

```

你在每次循环中使用 `find(str.begin(), str.end(), s[i])` 来查找是否存在重复字符。`find` 在 `vector` 中的时间复杂度是 $O(n)$ ，这使得算法在最坏情况下的时间复杂度变成 $O(n^2)$

```

class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int n = s.length(), ans = 0, left = 0;
        unordered_set<char> window; // 维护从下标 left 到下标 right 的字符
        for (int right = 0; right < n; right++) {
            char c = s[right];
            // 如果窗口内已经包含 c，那么再加入一个 c 会导致窗口内有重复元素
            // 所以要在加入 c 之前，先移出窗口内的 c
            while (window.count(c)) { // 窗口内有 c
                window.erase(s[left++]); // 缩小窗口
            }
            window.insert(c); // 加入 c
            ans = max(ans, right - left + 1); // 更新窗口长度最大值
        }
        return ans;
    }
};

```

`unordered_set` 基于哈希表实现，`count()` 函数用于检查某个元素是否存在。由于哈希表的查找、插入和删除操作在平均情况下都是常数时间，即 $O(1)$ ，因此 `count()` 函数的平均时间复杂度也是 $O(1)$ 。

不过在 最坏情况下，如果哈希冲突非常多（例如所有元素都映射到同一个哈希桶中），`count()` 的时间复杂度可能退化为 $O(n)$

但通常我们认为哈希表操作的时间复杂度是 $O(1)$ ，因为哈希冲突的概率很低。

使得算法复杂度为 $O(n)$

四、前缀和

<https://leetcode.cn/problems/range-sum-query-immutable/solutions/2693498/qian-zhui-he-ji-qi-k-uo-zhan-fu-ti-dan-py-vaar/>

1.和为K的子数组

前缀和+哈希表

给你一个整数数组 `nums` 和一个整数 `k`，请你统计并返回 该数组中和为 `k` 的子数组的个数。

子数组是数组中元素的连续非空序列。

数组先考虑双指针和滑动窗口，想到不是单调，用不了 --》连续数的和--》想到前缀和，前缀和能得到从 left 到 right 的和 --》找 $pre[right] - pre[left] == k$ --》 $pre[left] == pre[right] - k$ --》遍历到 right 时，寻找前面 $pre[right] - k$ 的个数 --》想到用哈希表存储前缀和的个数 --》找到有几个等于 $pre[right] - k$ 的 $pre[left]$ 就把结果加几

为什么这题不适合用滑动窗口做？

答：滑动窗口需要满足单调性，当右端点元素进入窗口时，窗口元素和是不能减少的。本题 nums 包含负数，当负数进入窗口时，窗口左端点反而要向左移动，导致算法复杂度不是线性的。

```
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        unordered_map<int, int> pre_sum;
        pre_sum[0] = 1; // 初始化前缀和为0的情况

        int currentSum = 0; // 当前的前缀和
        int result = 0; // 记录和为 k 的子数组的个数

        for (int i = 0; i < nums.size(); i++) {
            currentSum += nums[i]; // 累加当前元素到前缀和

            // 如果 currentSum - k 存在于哈希表中，说明存在一个以当前元素结尾的子数组，其和为 k
            if (pre_sum.find(currentSum - k) != pre_sum.end()) {
                result += pre_sum[currentSum - k]; // 累加符合条件的子数组个数
            }

            // 将当前前缀和存入哈希表，统计出现次数
            pre_sum[currentSum]++;
        }

        return result;
    }
};
```

五、动态规划

1. 最大子数组和

方法一：前缀和

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

```

        int min_sum=0;
        int max_sum=nums[0];
        for(int i=0;i<nums.size();i++)
        {
            min_sum = min(min_sum,pre_sum[i]);
            max_sum=max(max_sum,pre_sum[i+1]-min_sum);
        }

        return max_sum;
    }
};
/*
两次遍历
*/

```

```

class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int ans = INT_MIN;
        int min_pre_sum = 0;
        int pre_sum = 0;
        for (int x : nums) {
            pre_sum += x; // 当前的前缀和
            ans = max(ans, pre_sum - min_pre_sum); // 减去前缀和的最小值
            min_pre_sum = min(min_pre_sum, pre_sum); // 维护前缀和的最小值
        }
        return ans;
    }
};
/*
一次遍历
*/

```

方法二：动态规划

定义 $f[i]$ 表示以 $nums[i]$ 结尾的最大子数组和。

分类讨论：

$nums[i]$ 单独组成一个子数组，那么 $f[i]=nums[i]$ 。

$nums[i]$ 和前面的子数组拼起来，也就是在以 $nums[i-1]$ 结尾的最大子数组和之后添加 $nums[i]$ ，那么 $f[i]=f[i-1]+nums[i]$ 。

两种情况取最大值，得

$$f[i]=\begin{cases} nums[i], & i=0 \\ \max(f[i-1],0)+nums[i], & i\geq 1 \end{cases}$$

简单地说，如果 $nums[i]$ 左边的子数组元素和是负的，就不用和左边的子数组拼在一起了。

答案为 $\max(f)$ 。

```

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

```

2.打家劫舍

看灵神视频，空间还可以优化

我的写法：

```

class Solution {
public:
    int rob(vector<int>& nums) {
        vector<int> dp(nums.size(),0);
        dp[0]=nums[0];
        if(nums.size()==1) return dp[0];
        dp[1]=max(dp[0],nums[1]);

        for(int i=2;i<nums.size();i++)
        {
            dp[i]=max(dp[i-1],dp[i-2]+nums[i]);
        }
        return dp[nums.size()-1];
    }
};

```

121.买卖股票最佳时机

方法一

赚钱最多--》right_max-left_min 最大--》两个变量 定一移一 --》依次遍历数组 维护一个左侧最小值 计算当前最大利润 --》遍历完数组可得到最大利润

```

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int min_price=prices[0];
        int max_profit = 0;

        for(int i=1;i<prices.size();i++)
        {
            max_profit=max(max_profit,prices[i]-min_price);

```

```

        min_price=min(prices[i],min_price);
    }

    return max_profit < 0 ? 0 :max_profit;
}
};

```

方法二

思路

动态规划方法最核心的就是找到**状态转移方程**，下面详细地为大家讲解一下 $dp[i] = \max(dp[i - 1] + prices[i] - prices[i - 1], 0)$ 是如何推导出来的。

解题过程

首先要明确的是，状态转移方程的核心就是**去避免重复的计算**。避免重复的计算，我们就要保存**每次计算的值**，因此通过dp数组，我们将**每天的最大利润**保存在了里面。那么问题就已经缩小至了求**每天的最大利润**。又因为dp数组的核心之一就是dp[i]要和dp[i-1]等前面的位置有着紧密的联系，所以我们要思考**当天的最大利润能否通过前一天的最大利润求出**。这里我们引入一个简化方法： $A - C == A - B - C$ 类似于化学中的盖斯定律，**第二天买入第四天卖出==第二天买入第三天卖出+第三天买入第四天卖出**，数学表达如下： $dp[2]=price[2]-price[1]=5-1=4$ $dp[3]=price[2]-price[1]+price[3]-price[2]=dp[2]+price[3]-price[2]=5-1+3-5=2$ 从而我们找到了**核心状态转移方程**： $dp[i]=dp[i - 1] + prices[i] - prices[i - 1]$ 而当我们直接使用时，结果出错，**dp数组出现了负数的情况**，说明我们不能直接使用上面的状态转移方程，接下来我们寻找**使用此方程的前置条件**。从一个简单的角度思考，**dp[i]在本题中不可能小于0**，那么我们直接加上前置条件，式子变为： $if(dp[i - 1] + prices[i] - prices[i - 1]>0) dp[i]=dp[i - 1] + prices[i] - prices[i - 1]$ 成功通过，简化状态转移方程为： $dp[i] = \max(dp[i - 1] + prices[i] - prices[i - 1], 0)$

复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$

```

int maxProfit(int* prices, int pricesSize) {
    if (pricesSize < 2)
        return 0;
    int i = 0, max = 0, dp[pricesSize];
    dp[i] = 0;
    for (i = 1; i < pricesSize; i++) { //构建dp数组
        dp[i] = fmax(dp[i - 1] + prices[i] - prices[i - 1], 0); //核心状态转移方程
        max = dp[i] > max ? dp[i] : max; //记录dp数组中出现的最大数
    }
    return max;
}

```

六、简单的数组

1.最大子数组和

2.轮转数组(用三种方法)

方法一 创建额外数组空间

方法二 不创建额外数组空间 利用取余(%)和一个中间变量完成对原数组轮转

方法三 利用reverse反转vector

七、螺旋问题

1.螺旋矩阵

方法一

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        if(matrix.size() == 0 || matrix[0].size() == 0)
            return {};

        vector<int> ans;
        int lineBegin = 0, lineEnd = matrix[0].size() - 1; //记录行的开头与结尾
        int listBegin = 0, listEnd = matrix.size() - 1; //记录列的开头与结尾

        while(true)
        {
            //从左往右
            for(int i = lineBegin; i <= lineEnd; i++)
                ans.push_back(matrix[listBegin][i]);
            if(++listBegin > listEnd) break;

            //从上往下
            for(int i = listBegin; i <= listEnd; i++)
                ans.push_back(matrix[i][lineEnd]);
            if(--lineEnd < lineBegin) break;

            //从右往左
            for(int i = lineEnd; i >= lineBegin; i--)
                ans.push_back(matrix[listEnd][i]);
            if(--listEnd < listBegin) break;

            //从下往上
            for(int i = listEnd; i >= listBegin; i--)
                ans.push_back(matrix[i][lineBegin]);
            if(++lineBegin > lineEnd) break;
        }
        return ans;
    }
};
```

```
}  
};
```

方法二

```
class Solution {  
public:  
    vector<int> spiralOrder(vector<vector<int>>& matrix) {  
        if (matrix.size() == 0 || matrix[0].size() == 0) {  
            return {};  
        }  
  
        int rows = matrix.size(), columns = matrix[0].size();  
        vector<int> order;  
        int left = 0, right = columns - 1, top = 0, bottom = rows - 1;  
        while (left <= right && top <= bottom) {  
            for (int column = left; column <= right; column++) {  
                order.push_back(matrix[top][column]);  
            }  
            for (int row = top + 1; row <= bottom; row++) {  
                order.push_back(matrix[row][right]);  
            }  
            if (left < right && top < bottom) {  
                for (int column = right - 1; column > left; column--) {  
                    order.push_back(matrix[bottom][column]);  
                }  
                for (int row = bottom; row > top; row--) {  
                    order.push_back(matrix[row][left]);  
                }  
            }  
            left++;  
            right--;  
            top++;  
            bottom--;  
        }  
        return order;  
    }  
};
```

我的代码:

```
class Solution {  
public:  
    vector<int> spiralOrder(vector<vector<int>>& matrix) {  
        vector<int> result;  
        int lbegin=0;  
        int lend=matrix[0].size()-1;  
  
        int rbegin=0;  
        int rend=matrix.size()-1;  
  
        while(1)
```

```

{
    for(int i=lbegin;i<lend;i++){result.emplace_back(matrix[rbegin][i]);}
    if(rbegin>rend||lbegin>lend) break;
    if(rbegin==rend) {result.emplace_back(matrix[rbegin][lend]);break;}

    for(int j=rbegin;j<rend;j++){result.emplace_back(matrix[j][lend]);}
    if(rbegin>rend||lbegin>lend) break;
    if(lbegin==lend){result.emplace_back(matrix[rend][lbegin]);break;}

    for(int i=lend;i>lbegin;i--){result.emplace_back(matrix[rend][i]);}

    for(int j=rend;j>rbegin;j--){result.emplace_back(matrix[j][lbegin]);}

    lbegin++;
    lend--;
    rbegin++;
    rend--;
    if(rbegin>rend||lbegin>lend) break;
}

return result;
}
};

```

2.螺旋矩阵II

```

class Solution {
public:
    vector<vector<int>> generateMatrix(int n) {
        int x=(n+1)/2;
        vector<vector<int>> res(n,vector<int>(n));
        int y=0;
        int t=1;
        while(x-->0)
        {
            for(int i=y;i<n-y-1;i++)
            {
                res[y][i]=t++;
            }
            for(int i=y;i<n-y-1;i++)
            {
                res[i][n-1-y]=t++;
            }
            for(int i=n-1-y;i>y;i--)
            {
                res[n-1-y][i]=t++;
            }
            for(int i=n-1-y;i>y;i--)
            {
                res[i][y]=t++;
            }
            y++;
        }
    }
};

```



```

    }
    if(n%2!=0) res[(n-1)/2][(n-1)/2]=t;

    return res;
}
};

```

3.旋转图像

我的代码：

```

class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int t=0;
        for(int i=0;i<matrix.size()-1;i++)
        {
            for(int j=0;j<matrix[0].size();j++)
            {
                swap(matrix[i][j+t+1],matrix[j+t+1][i]);
                if(j+t+1==matrix.size()-1) break;
            }
            t++;
        }

        for(int i=0;i<matrix.size();i++)
        {
            reverse(matrix[i].begin(),matrix[i].end());
        }
    }
};

```

还可以两次翻转：

```

class Solution {
public:
    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]);
            }
        }
    }
};

```

八、链表

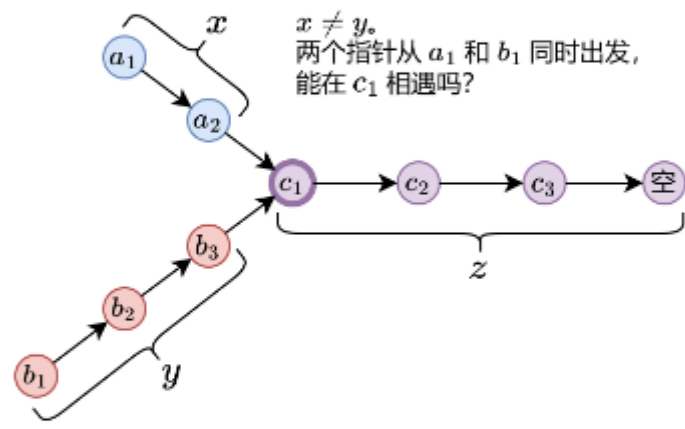
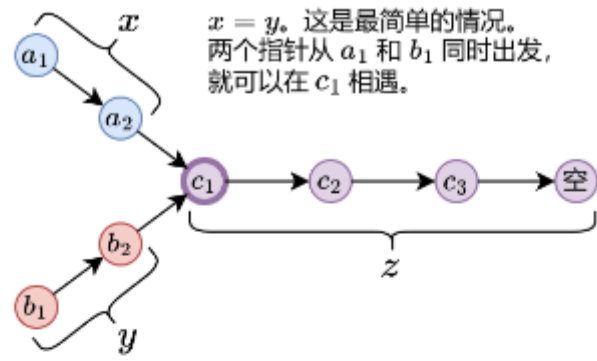
1.相交链表

方法一 利用哈希表存储节点（空间复杂度较高）

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        unordered_set<ListNode *> visited;
        ListNode *temp = headA;
        while (temp != nullptr) {
            visited.insert(temp);
            temp = temp->next;
        }
        temp = headB;
        while (temp != nullptr) {
            if (visited.count(temp)) {
                return temp;
            }
            temp = temp->next;
        }
        return nullptr;
    }
};
```

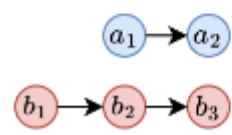
方法二

设第一条链表的长度是 $x + z$ ，
设第二条链表的长度是 $y + z$ 。
其中 z 是两条链表的公共部分的长度。
注意：为了兼容没有交点的情况，把空节点也算进来。

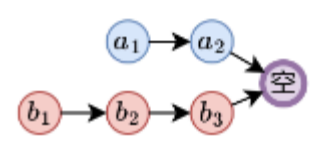


注意到 $(x + z) + y = (y + z) + x$ ，
从 a_1 出发的指针，走到空节点时，让它下一步跳到 b_1 。
从 b_1 出发的指针，走到空节点时，让它下一步跳到 a_1 。

如上图，
从 a_1 出发的指针，走了 $x + z + y = 9$ 步后到达 c_1 ，
从 b_1 出发的指针，走了 $y + z + x = 9$ 步后也到达 c_1 。



如果两条链表不相交呢？



因为 a_2 和 b_3 的
下一个节点都是空节点，
可以视作在空节点相交。
由于 $x + y = y + x$ ，
两个指针最终都会走到空节点。

```

class Solution {
public:
    ListNode* getIntersectionNode(ListNode* headA, ListNode* headB) {
        ListNode* p = headA;
        ListNode* q = headB;
        while (p != q) {
            p = p ? p->next : headB;
            q = q ? q->next : headA;
        }
        return p;
    }
};

```

`p != q`: 判断 `p` 和 `q` 这两个指针是否指向同一个地址。`p` 和 `q` 是 `ListNode` 类型的指针, 所以 `p != q` 表示 `p` 和 `q` 是否指向同一个节点, 而不是判断它们指向的节点内容是否相同。

`*p != *q`: 判断 `p` 和 `q` 指向的节点的内容是否相同。`*p` 和 `*q` 是指针的解引用操作, 即取得 `p` 和 `q` 所指向的节点内容, 因此 `*p != *q` 会比较两个节点中的数据成员是否相同。通常在这里我们并不需要 `*p != *q`, 因为这个函数只需要找到两个链表的交点, 即地址相同的节点, 而不关心节点内容是否相同。

2.反转链表

```

class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *cur = head, *pre = nullptr;
        while(cur != nullptr) {
            ListNode* tmp = cur->next; // 暂存后继节点 cur.next
            cur->next = pre;           // 修改 next 引用指向
            pre = cur;                // pre 暂存 cur
            cur = tmp;                // cur 访问下一节点
        }
        return pre;
    }
};

```

3.链表的中间节点

```

class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        ListNode *fast = head, *slow = head;
        while (fast != nullptr && fast->next != nullptr) {
            fast = fast->next->next;
            slow = slow->next;
        }
        return slow;
    }
};

```

4.回文链表

```
class Solution {
    // 876. 链表的中间结点
    ListNode* middleNode(ListNode* head) {
        ListNode* slow = head, *fast = head;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }

    // 206. 反转链表
    ListNode* reverseList(ListNode* head) {
        ListNode* pre = nullptr, *cur = head;
        while (cur) {
            ListNode* nxt = cur->next;
            cur->next = pre;
            pre = cur;
            cur = nxt;
        }
        return pre;
    }

public:
    bool isPalindrome(ListNode* head) {
        ListNode* mid = middleNode(head);
        ListNode* head2 = reverseList(mid);
        while (head2) {
            if (head->val != head2->val) { // 不是回文链表
                return false;
            }
            head = head->next;
            head2 = head2->next;
        }
        return true;
    }
};
```

我的代码:

```
/**
 * 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) {
        ListNode* p = head;
        if (p->next == nullptr) return true;

        //中间节点
        int i = 0;
        while (p != nullptr) {
            p = p->next;
            i++;
        }
        if (i % 2 != 0) i = i / 2 + 1;
        else i = i / 2;
        ListNode* t = head;
        while (i-- > 0) t = t->next;

        //反转链表
        ListNode* x = t;
        ListNode* y = nullptr;
        while (t != nullptr) {
            t = t->next;
            x->next = y;
            y = x;
            x = t;
        }

        ListNode* q = head;
        while (y != nullptr) {
            if (q->val != y->val)
                return false;
            q = q->next;
            y = y->next;
        }

        return true;
    }
};

```

5.环形链表

```

/**
 * 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) {
        ListNode* fast = head;
        ListNode* slow = head;
        if(head==nullptr) return false;
        while(fast->next!=nullptr&&fast->next->next!=nullptr)

```

```

        {
            fast=fast->next->next;
            slow=slow->next;
            if(fast==slow) return true;
        }

        return false;
    }
};

```

6.环形链表II

方法一 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 *detectCycle(ListNode *head) {
        unordered_set<ListNode*> ref;
        ListNode* p = head;
        while(p!=nullptr)
        {
            if(ref.find(p)!=ref.end())
            {
                return p;
            }
            ref.emplace(p);
            p=p->next;
        }

        return nullptr;
    }
};

```

方法二 快慢指针

相遇之处 和 链表头 距离环入口距离相等

```

class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode* fast = head;
        ListNode* slow = head;
        while (true) {
            if (fast == nullptr || fast->next == nullptr) return nullptr;
            fast = fast->next->next;
            slow = slow->next;
        }
    }
};

```

```

        if (fast == slow) break;
    }
    fast = head;
    while (slow != fast) {
        slow = slow->next;
        fast = fast->next;
    }
    return fast;
}
};

```

7.合并两个有序链表

哨兵节点的运用

8.两数相加（重要）（对于链表指针的理解）（p只是指向Node，没有真正连接到Node上）（哨兵节点）

```

class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode dummy; // 哨兵节点
        ListNode* cur = &dummy;
        int carry = 0; // 进位
        while (l1 || l2 || carry) { // 有一个不是空节点，或者还有进位，就继续迭代
            int sum = carry + (l1 ? l1->val : 0) + (l2 ? l2->val : 0); // 节点值和
进位加在一起
            cur = cur->next = new ListNode(sum % 10); // 每个节点保存一个数位
            carry = sum / 10; // 新的进位
            if (l1) l1 = l1->next; // 下一个节点
            if (l2) l2 = l2->next; // 下一个节点
        }
        return dummy.next; // 哨兵节点的下一个节点就是头节点
    }
};

```

我的代码：

```

/**
 * 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 dummy;
        ListNode* p = &dummy;
    }
};

```



```

    int carry = 0;

    while(l1 || l2 || carry)
    {
        if(l1 && l2)
        {
            int sum = l1->val+l2->val+carry;
            p->next = new ListNode(sum%10);
            carry = sum/10;
            l1=l1->next;
            l2=l2->next;
        }
        else if(l1)
        {
            int sum = l1->val+carry;
            p->next = new ListNode(sum%10);
            carry = sum/10;
            l1=l1->next;
        }
        else if(l2)
        {
            int sum = l2->val+carry;
            p->next = new ListNode(sum%10);
            carry = sum/10;
            l2=l2->next;
        }
        else if(carry)
        {
            p->next = new ListNode(carry);
            carry = 0;
        }

        p = p->next;
    }

    return dummy.next;
}
};

```

9.删除链表的倒数第 N 个结点

哨兵节点！！！

我的写法：

```

/**
 * 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) {
        int p = 0;
        ListNode* q = head;

        while(q)
        {
            p++;
            q=q->next;
        }

        p=p-n;
        ListNode dummy = ListNode(0,head);
        ListNode* pre = &dummy;

        for(;p>0;p--)
        {
            pre=pre->next;
        }

        pre->next=pre->next->next;

        return dummy.next;
    }
};

```

双指针（有争议）

```

class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        // 由于可能会删除链表头部，用哨兵节点简化代码
        ListNode dummy{0, head};
        ListNode* left = &dummy;
        ListNode* right = &dummy;
        while (n--) {
            right = right->next; // 右指针先向右走 n 步
        }
        while (right->next) {
            left = left->next;
            right = right->next; // 左右指针一起走
        }
        // 左指针的下一个节点就是倒数第 n 个节点
        ListNode* nxt = left->next;
        left->next = left->next->next;
        delete nxt;
        return dummy.next;
    }
};
/*

```

前后指针（不是快慢指针）是个好思想，但是这道题的“one-pass”要求根本不合理。原因之一，前后指针到底算不算“one-pass”属于文字游戏，从代码形式上看确实少了一个 **for**，但从操作次数上看，与单指针走两遍并无区别

九、LRU缓存

1.LRU缓存

在LRU的基础上要求增加过期时间，过期的key要删除掉，刷这题的同学不妨多一些思考

十、二叉树

1.二叉树的中序遍历 （144前序， 145后序， 102层序）

方法一 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> res;

    void tree_load(TreeNode* p)
    {
        if(p==nullptr) return;
        tree_load(p->left);
        res.emplace_back(p->val);
        tree_load(p->right);
        return;
    }

    vector<int> inorderTraversal(TreeNode* root) {
        tree_load(root);
        return res;
    }
};
/**
```

时间复杂度： $O(n)$ ，其中 n 为二叉树节点的个数。二叉树的遍历中每个节点会被访问一次且只会被访问一次。

空间复杂度： $O(n)$ 。空间复杂度取决于递归的栈深度，而栈深度在二叉树为一条链的情况下会达到 $O(n)$ 的级别。

*/

方法二 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) {
        stack<TreeNode*> load; //使用栈
        vector<int> res;
        while (!load.empty() || root != nullptr) {
            if (root == nullptr) {
                root = load.top();
                load.pop();
                res.emplace_back(root->val);
                root = root->right;
                continue;
            }
            load.push(root);
            root = root->left;
        }
        return res;
    }
};
```

方法三 Morris (了解)

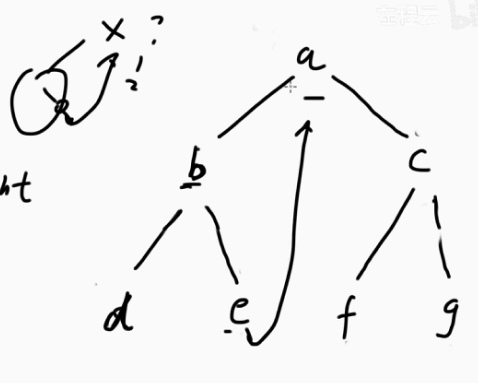
1. 开始头时 $cur = head$, cur 为空时过程停止

2. 如果 cur 无左树, cur 向右移动

3. 如果 cur 有左树, 找到左树最右节点 $mostRight$

A. 如果 $mostRight$ 的右指针指向空, 让其指向 cur , cur 向左移动

B. 如果 $mostRight$ 的右指针指向 cur , 让其指向空, cur 向右移动



1)

morris序

a b d b e a c f c g

Morris遍历

Morris遍历的理解核心

没有左子树的节点只到达一次, 有左子树的节点会到达两次
利用左子树最右节点的右指针状态, 来标记是第几次到达

Morris遍历的过程, 课上重点图解

1. 开始时 cur 来到头节点, cur 为空时过程停止
2. 如果 cur 没有左孩子, cur 向右移动
3. 如果 cur 有左孩子, 找到 cur 左子树的最右节点 $mostRight$
 - A. 如果 $mostRight$ 的右指针指向空, 让其指向 cur , 然后 cur 向左移动
 - B. 如果 $mostRight$ 的右指针指向 cur , 让其指向 $null$, 然后 cur 向右移动

额外空间复杂度很明显是 $O(1)$, 但是时间复杂度依然为 $O(n)$, 课上重点图解

2. 二叉树的层序遍历 (广度优先模版) (使用队列)

方法一: 两个数组

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode *root) {
        if (root == nullptr) return {};
        vector<vector<int>> ans;
        vector<TreeNode*> cur = {root};
        while (cur.size()) {
            vector<TreeNode*> nxt;
            vector<int> vals;
            for (auto node : cur) {
                vals.push_back(node->val);
                if (node->left) nxt.push_back(node->left);
                if (node->right) nxt.push_back(node->right);
            }
            cur = move(nxt);
        }
    }
};
```

```

        ans.emplace_back(vals);
    }
    return ans;
}
};

```

复杂度分析

时间复杂度： $O(n)$ ，其中 n 为二叉树的节点个数。

空间复杂度： $O(n)$ 。满二叉树（每一层都填满）最后一层有大约 $n/2$ 个节点，因此数组中最多有 $O(n)$ 个元素，所以空间复杂度是 $O(n)$ 的。

方法二：一个队列

```

class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode *root) {
        if (root == nullptr) return {};
        vector<vector<int>> ans;
        queue<TreeNode *> q;
        q.push(root);
        while (!q.empty()) {
            vector<int> vals;
            for (int n = q.size(); n--;) {
                auto node = q.front();
                q.pop();
                vals.push_back(node->val);
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
            ans.emplace_back(vals);
        }
        return ans;
    }
};

```

复杂度分析

时间复杂度： $O(n)$ ，其中 n 为二叉树的节点个数。

空间复杂度： $O(n)$ 。满二叉树（每一层都填满）最后一层有大约 $n/2$ 个节点，因此队列中最多有 $O(n)$ 个元素，所以空间复杂度是 $O(n)$ 的。

3. 二叉树的最大深度

方法一 递归 深度优先

```

/**
 * 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 == nullptr) return 0;
        return max(maxDepth(root->left),maxDepth(root->right))+1;
    }
};

```

方法二 广度优先 层序遍历

4.二叉树的最小深度

方法一 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:
    int minDepth(TreeNode* root) {
        if(root == nullptr) return 0;
        if(!root->left) return minDepth(root->right)+1;
        if(!root->right) return minDepth(root->left)+1;
        return min(minDepth(root->left),minDepth(root->right))+1;
    }
};

```

方法二 广度优先 层序遍历

5.二叉树的右视图

方法一 层次遍历 广度优先搜索

```

class Solution {
public:
    vector<int> rightSideview(TreeNode* root) {
        vector<int> res;
        queue<TreeNode*> cur;
        TreeNode* p =nullptr;
        if(root == nullptr) return res;

```

```

        cur.push(root);
        while(cur.size()!=0)
        {
            int n=cur.size();//正序写不容易出错
            for(int i=0;i<n;i++) //遍历到最右节点加入
            {
                p = cur.front();
                if(i==n-1) res.emplace_back(p->val);
                cur.pop();
                if(p->left!=nullptr) cur.push(p->left);
                if(p->right!=nullptr) cur.push(p->right);
            }
        }

        return res;
    }
};

```

方法二 深度优先搜索

```

class Solution {
    vector<int> ans;

    void dfs(TreeNode* node, int depth) {
        if (node == nullptr) {
            return;
        }
        if (depth == ans.size()) { // 这个深度首次遇到
            ans.push_back(node->val);
        }
        dfs(node->right, depth + 1); // 先递归右子树，保证首次遇到的一定是最右边的节点
        dfs(node->left, depth + 1);
    }

public:
    vector<int> rightSideview(TreeNode* root) {
        dfs(root, 0);
        return ans;
    }
};

```

**6.验证二叉搜索树

方法一 递归

我的代码

你的 `isValidBST` 实现思路存在问题。当前代码仅仅检查了直接子节点与当前节点的关系，没有确保整棵树中的所有节点满足二叉搜索树 (BST) 的要求。例如，在一个有效的 BST 中，左子树的所有节点都应该小于根节点，而右子树的所有节点都应该大于根节点。仅检查左、右节点的值不足以验证这点。

解决方案是引入一个范围 (区间)，在递归时限制每个节点的值，使它落在该区间内。具体来说，每个节点应满足：

- 左子树的所有节点小于当前节点的值。
- 右子树的所有节点大于当前节点的值。

经过修改：

```
class solution {
public:
    bool isValidBST(TreeNode* root, long long min_val = LLONG_MIN, long long max_val = LLONG_MAX) {
        if (root == nullptr) return true;

        // 检查当前节点是否在合法的范围内
        if (root->val <= min_val || root->val >= max_val) return false;
        //验证不合法只需要一个条件不满足
        //验证合法需要所有条件满足
        //所有把问题转换为验证不合法更简洁

        // 对左、右子树递归检查，同时调整合法范围
        return isValidBST(root->left, min_val, root->val) &&
            isValidBST(root->right, root->val, max_val);
    }
};
```

方法二（前序中序后序）遍历

7.二叉树的最近公共祖先

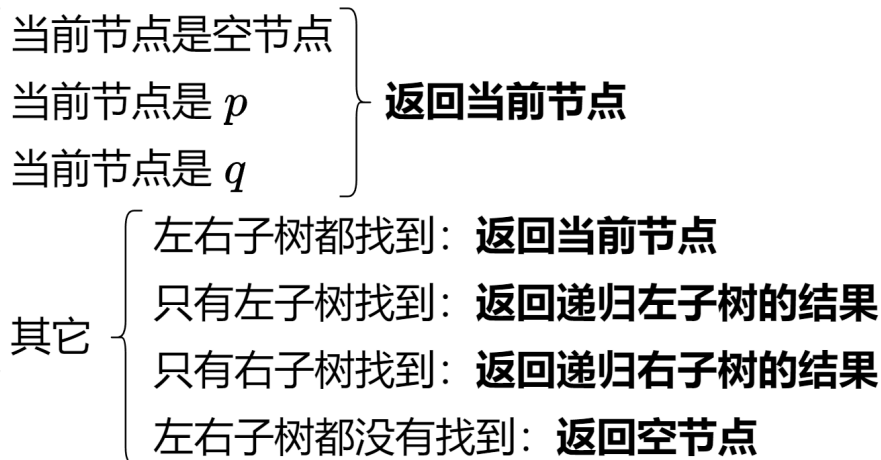
我的代码：

```
class Solution {
public:
    bool findtree(TreeNode* cur, TreeNode* find) {
        if (cur == find)
            return true;
        if (cur == nullptr)
            return false;
        return findtree(cur->left, find) || findtree(cur->right, find);
    }

    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (findtree(root->left, p) && findtree(root->left, q))
            return lowestCommonAncestor(root->left, p, q);
        else if (findtree(root->right, p) && findtree(root->right, q))
            return lowestCommonAncestor(root->right, p, q);
        return root;
    }
};
```

非常好的代码，认真理解：

分类讨论



```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root == nullptr || root == p || root == q) {
            return root;
        }
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);
        if (left && right) { // 左右都找到
            return root; // 当前节点是最近公共祖先
        }
        return left ? left : right;
    }
};
```

8. 二叉搜索树的最近公共祖先

方法一 递归

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root->val > p->val && root->val > q->val)
            return lowestCommonAncestor(root->left, p, q);
        else if (root->val < p->val && root->val < q->val)
            return lowestCommonAncestor(root->right, p, q);
        return root;
    }
};
```

方法二 迭代

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        while (root != nullptr) {
            if (root->val < p->val && root->val < q->val) // p,q 都在 root 的右子树
                root = root->right; // 遍历至右子节点
            else if (root->val > p->val && root->val > q->val) // p,q 都在 root 的
                root = root->left; // 遍历至左子节点
            else break;
        }
        return root;
    }
};
```

9.二叉树的锯齿形层序遍历

方法一 两个数组

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> res;
        if(root == nullptr) return res;
        vector<TreeNode*> next;
        vector<TreeNode*> cur;
        cur.emplace_back(root);
        int index = 1;

        while(!cur.empty()) {

            vector<int> mid;
            int n = cur.size();

            // 遍历当前层的节点
            for(int i = 0; i < n; i++) {
                TreeNode* node = cur[i];
                mid.push_back(node->val);

                // 将左右子节点加入 next 队列
                if(node->left != nullptr) next.push_back(node->left);
                if(node->right != nullptr) next.push_back(node->right);
            }

            // 如果是偶数层，则需要反转
            if(index % 2 == 0) {
                reverse(mid.begin(), mid.end());
            }

            res.push_back(mid); // 添加当前层的结果
            index++;
            cur = next;
        }

        return res;
    }
};
```

```

        cur = move(next); // 将 next 作为新的当前层
    }

    return res;
}
};
/*

```

mid 和 next 都不需要手动调用 clear()，原因如下：

mid 的作用域：

mid 是在每次 while 循环的开头被重新定义的，因此在每次循环开始时都会被重新初始化为空。也就是说，mid 的数据在每一层结束时并不会被保留到下一层。这样可以避免手动清空 mid。

next 的赋值方式：

在 cur = move(next); 这一行中，我们使用了 move 将 next 的内容转移给 cur。move 操作会将 next 的资源转移给 cur，并使 next 变为空。这种方法不仅避免了手动清空 next，还提高了效率，减少了内存的重复分配。

```
*/
```

方法二 一个队列

```

class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode *root) {
        if (root == nullptr) return {};
        vector<vector<int>> ans;
        queue<TreeNode *> q;
        q.push(root);
        while (!q.empty()) {
            vector<int> vals;
            for (int n = q.size(); n--;) {
                auto node = q.front();
                q.pop();
                vals.push_back(node->val);
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
            if (ans.size() % 2) ranges::reverse(vals);
            ans.emplace_back(vals);
        }
        return ans;
    }
};

```

10.找树左下角的值

BFS 这棵二叉树，先把右儿子入队，再把左儿子入队，这样最后一个出队的节点就是左下角的节点了。

```

class Solution {
public:
    int findBottomLeftValue(TreeNode *root) {
        TreeNode *node;
        queue<TreeNode *> q;
        q.push(root);
    }
};

```

```

        while (!q.empty()) {
            node = q.front(); q.pop();
            if (node->right) q.push(node->right);
            if (node->left) q.push(node->left);
        }
        return node->val;
    }
};

```

我的代码：

```

class Solution {
public:
    int findBottomLeftValue(TreeNode* root) {
        vector<TreeNode*> cur,next;
        cur.emplace_back(root);
        while(cur.size() != 0)
        {
            int n = cur.size();
            for(int i = 0;i < n;i++)
            {
                if(cur[i]->left != nullptr) next.emplace_back(cur[i]->left);
                if(cur[i]->right != nullptr) next.emplace_back(cur[i]->right);
            }
            if(next.size() == 0) return cur[0]->val;
            cur = move(next);
        }
        return root->val;
    }
};

```

**11.构造二叉树

a.前序中序

1. 前序遍历的首个元素即为树的 **根节点 3** 的值

preorder =

3	9	2	1	7
---	---	---	---	---

根节点



2. 根据根节点索引，可将 **中序遍历** 划分为 **左子树-根节点-右子树**

inorder =

9	3	1	2	7
---	---	---	---	---

左子树

右子树

(长度为 1) (长度为 3)



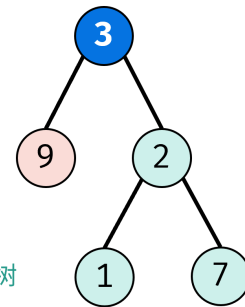
3. 根据中序遍历的左/右子树的节点数量，可将 **前序遍历** 划分 **根节点-左子树-右子树**

preorder =

3	9	2	1	7
---	---	---	---	---

左子树

右子树



```
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        int n = preorder.size();

        // 创建哈希表 index，将中序遍历数组的值和索引对应起来，便于查找
        unordered_map<int, int> index;
        for (int i = 0; i < n; i++) {
            index[inorder[i]] = i; // 记录值在 inorder 中的索引位置
        }

        // 定义递归函数 dfs，用于构建树，lambda 表达式用于局部函数定义
        function<TreeNode*(int, int, int, int)> dfs = [&](int pre_l, int pre_r,
            int in_l, int in_r) -> TreeNode* {
            // 递归基准条件：当前序遍历范围为空时，返回 nullptr
            if (pre_l == pre_r) {
                return nullptr;
            }

            // 计算左子树的节点数
            int left_size = index[preorder[pre_l]] - in_l;

            // 递归构建左子树：
            // 前序范围为 [pre_l + 1, pre_l + 1 + left_size)
            // 中序范围为 [in_l, in_l + left_size)
            TreeNode* left = dfs(pre_l + 1, pre_l + 1 + left_size, in_l, in_l +
            left_size);

            // 递归构建右子树：
            // 前序范围为 [pre_l + 1 + left_size, pre_r)
```

```

        // 中序范围为 [in_l + 1 + left_size, in_r)
        TreeNode* right = dfs(pre_l + 1 + left_size, pre_r, in_l + 1 +
left_size, in_r);

        // 创建当前子树的根节点，并将左、右子树连接上
        return new TreeNode(preorder[pre_l], left, right);
    };

    // 调用 dfs 函数，从整个树的前序和中序遍历范围开始构建
    return dfs(0, n, 0, n);
}
};

```

b.前序后序

c.中序后序

```

class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        unordered_map<int,int> index;
        int n = inorder.size();
        for(int i = 0; i < n; i++)
        {
            index[inorder[i]] = i;
        }

        function<TreeNode*(int,int,int,int)> dfs = [&](int in_l, int in_r, int
po_l, int po_r)->TreeNode*
        {
            if(po_l == po_r) return nullptr;
            int left_size = index[postorder[po_r-1]] - in_l;

            auto left = dfs(in_l, in_l + left_size, po_l, po_l + left_size);
            auto right = dfs(in_l + left_size + 1, in_r, po_l + left_size, po_r-
1);

            return new TreeNode(postorder[po_r-1], left, right);
        };

        return dfs(0, n, 0, n);
    }
};

```

12.树直径问题

链：从下面的某个节点（不一定是叶子）到当前节点的路径。把这条链的节点值之和，作为 dfs 的返回值。如果节点值之和是负数，则返回 0。

直径：等价于由两条（或者一条）链拼成的路径。我们枚举每个 node，假设直径在这里「拐弯」，也就是计算由左右两条从下面的某个节点（不一定是叶子）到 node 的链的节点值之和，去更新答案的最大值。

543.二叉树的直径

124.二叉树中的最大路径和

重点：转折点



```
class Solution {
public:
    int maxSum = INT_MIN;
    int maxPathSum(TreeNode* root) {

        dfs(root);
        return maxSum;
    }
    int dfs(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        int left = dfs(root->left);
        left = max(left, 0); // 分支最大值为负数，则丢弃考虑分支
        int right = dfs(root->right);
        right = max(right, 0);
        // 递归过程进行更新
        maxSum = max(maxSum, root->val + left + right);
        return root->val + max(left, right);
    }
};
```

2246.相邻字符不同的最长路径

递归大全

相同的树

```
class Solution {
public:
    bool isSameTree(TreeNode *p, TreeNode *q) {
        if (p == nullptr || q == nullptr)
            return p == q; // 必须都是 nullptr
        return p->val == q->val && isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};
```


对称二叉树

```
class Solution {
public:
    bool isthesame(TreeNode* p, TreeNode* q)
    {
        if(p==nullptr&&q!=nullptr) return false;
        if(p!=nullptr&&q==nullptr) return false;
        if(p==nullptr&&q==nullptr) return true;
        if(p->val==q->val)
        {
            return isthesame(p->left,q->right)&&isthesame(p->right,q->left);
        }
        return false;
    }

    bool issymmetric(TreeNode* root) {
        if(root == nullptr) return true;
        return isthesame(root->left,root->right);
    }
};
```

平衡二叉树

```
class Solution {
    int get_height(TreeNode *node) {
        if (node == nullptr) return 0;
        int leftH = get_height(node->left);
        if (leftH == -1) return -1; // 提前退出，不再递归
        int rightH = get_height(node->right);
        if (rightH == -1 || abs(leftH - rightH) > 1) return -1;
        return max(leftH, rightH) + 1;
    }

public:
    bool isBalanced(TreeNode *root) {
        return get_height(root) != -1;
    }
};
```

十一、图论

1.岛屿数量

方法一 dfs

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int x = grid.size();
        int y = grid[0].size();
        int res = 0;
        function<void(int,int)> dfs = [&](int i,int j)
        {
            grid[i][j] = '0';

            if(i-1>=0 && grid[i-1][j] == '1') dfs(i-1,j);
            if(i+1<x && grid[i+1][j] == '1') dfs(i+1,j);
            if(j-1>=0 && grid[i][j-1] == '1') dfs(i,j-1);
            if(j+1<y && grid[i][j+1] == '1') dfs(i,j+1);
        };

        for(int i = 0;i < x;i++)
        {
            for(int j = 0;j < y;j++)
            {
                if(grid[i][j] == '1')
                {
                    dfs(i,j);
                    res++;
                }
            }
        }

        return res;
    }
};
```

方法二 bfs

方法三 并查集

2.腐烂的橘子

我的代码:

```
class Solution {
public:
    int orangesRotting(vector<vector<int>>& grid) {
        int time = 0;           // 用于记录腐烂所有橘子所需的分钟数
        int remain = 0;         // 新鲜橘子数量
        int bad = 0;            // 腐烂橘子数量
```

```

int old_bad = bad;           // 上一轮的腐烂橘子数量，用于检测腐烂数量是否增加
int m = grid.size();        // 行数
int n = grid[0].size();     // 列数

// 定义一个递归 lambda 表达式 `broad`，用于向四周扩散腐烂状态
function<void(int, int)> broad = [&](int i, int j) {
    // 向上扩散
    if (i - 1 >= 0 && grid[i - 1][j] == 1) {
        grid[i - 1][j] = 3 + time; // 标记新的腐烂橘子
        remain--;                  // 新鲜橘子数量减少
        bad++;                     // 腐烂橘子数量增加
    }
    // 向下扩散
    if (i + 1 < m && grid[i + 1][j] == 1) {
        grid[i + 1][j] = 3 + time;
        remain--;
        bad++;
    }
    // 向左扩散
    if (j - 1 >= 0 && grid[i][j - 1] == 1) {
        grid[i][j - 1] = 3 + time;
        remain--;
        bad++;
    }
    // 向右扩散
    if (j + 1 < n && grid[i][j + 1] == 1) {
        grid[i][j + 1] = 3 + time;
        remain--;
        bad++;
    }
};

// 遍历网格，统计新鲜橘子（1）和腐烂橘子（2）的数量
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (grid[i][j] == 1)
            remain++;
        if (grid[i][j] == 2)
            bad++;
    }
}

// 如果没有新鲜橘子，直接返回 0，因为不需要腐烂时间
if (remain == 0)
    return 0;

// 如果没有腐烂橘子且有新鲜橘子，返回 -1，因为无法使新鲜橘子腐烂
if (bad == 0)
    return -1;

old_bad = bad; // 记录初始的腐烂橘子数量

// 开始模拟腐烂过程
while (true) {
    // 遍历所有腐烂橘子，调用 `broad` 扩散腐烂
    for (int i = 0; i < m; i++) {

```

```

        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 2 + time) // 只对当前时间腐烂的橘子进行扩散
                broad(i, j);
        }

        // 如果腐烂橘子数量没有变化, 说明无法再扩散, 返回 -1
        if (old_bad == bad)
            return -1;
        else
            old_bad = bad; // 更新腐烂橘子数量

        time++; // 时间加一

        // 如果新鲜橘子数量为 0, 返回所需时间
        if (remain == 0)
            return time;
    }

    return -1; // 不可达的代码, 仅为安全性
}
};

```

多源广度优先搜索 (多源bfs)

```

class Solution {
    int DIRECTIONS[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}}; // 四方向

public:
    int orangesRotting(vector<vector<int>>& grid) {
        int m = grid.size(), n = grid[0].size();
        int fresh = 0;
        vector<pair<int, int>> q;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) {
                    fresh++; // 统计新鲜橘子个数
                } else if (grid[i][j] == 2) {
                    q.emplace_back(i, j); // 一开始就腐烂的橘子
                }
            }
        }

        int ans = 0;
        while (fresh && !q.empty()) {
            ans++; // 经过一分钟
            vector<pair<int, int>> nxt;
            for (auto& [x, y] : q) { // 已经腐烂的橘子 //注意这种写法
                for (auto d : DIRECTIONS) { // 四方向
                    int i = x + d[0], j = y + d[1];
                    if (0 <= i && i < m && 0 <= j && j < n && grid[i][j] == 1) {
                        // 新鲜橘子

                        fresh--;
                        grid[i][j] = 2; // 变成腐烂橘子
                        nxt.emplace_back(i, j);
                    }
                }
            }
            q = nxt;
        }

        return ans;
    }
};

```

```

        }
    }
}
q = move(nxt);
}

return fresh ? -1 : ans;
}
};

```

两种不同：

一个每次遍历寻找新腐烂的橘子

一个在找到腐烂的橘子同时把新腐烂的橘子存储以便下一次扩散

向四个方向扩散写法学习

相似题目 928.尽量减少恶意软件的传播II

十二、二分查找(三种二分查找都得熟悉)

- 核心要素

关键不在于区间里的元素具有什么性质，而是区间外面的元素具有什么性质

- 注意区间开闭，三种都可以
- 循环结束条件：当前区间内没有元素
- 下一次二分查找区间：不能再查找(区间不包含)mid，防止死循环
- 返回值：大于等于target的第一个下标（注意循环不变量）

有序数组中二分查找的四种类型（下面的转换仅适用于数组中都是整数）

- 第一个大于等于x的下标： `low_bound(x)`
- 第一个大于x的下标：可以转换为 第一个大于等于 `x+1` 的下标， `low_bound(x+1)`
- 最后一个一个小于x的下标：可以转换为 第一个大于等于 `x` 的下标 的 左边位置， `low_bound(x) - 1`
- 最后一个小于等于x的下标：可以转换为 第一个大于等于 `x+1` 的下标 的 左边位置， `low_bound(x+1) - 1`

注意二分查找的区别

`upper_bound` 和 `lower_bound` 的主要区别在于：

- `lower_bound(nums, target)`: 返回 第一个 `>= target` 的元素索引（即大于等于 `target` 的最左侧位置）。
- `upper_bound(nums, target)`: 返回 第一个 `> target` 的元素索引（即严格大于 `target` 的最左侧位置）。

```

int lower_bound(vector<int>& nums, int target) {
    int left = 0, right = (int) nums.size() - 1; // 闭区间 [left, right]
    while (left <= right) { // 区间不为空
        // 循环不变量：

```

```

        // nums[left-1] < target
        // nums[right+1] >= target
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1; // 范围缩小到 [mid+1, right]
        } else {
            right = mid - 1; // 范围缩小到 [left, mid-1]
        }
    }
    return left;
}

int upper_bound(vector<int>& nums, int target) {
    int left = 0, right = (int) nums.size() - 1; // 闭区间 [left, right]
    while (left <= right) { // 区间不为空
        int mid = left + (right - left) / 2;
        if (nums[mid] <= target) { // 注意这里是 <=
            left = mid + 1; // 继续查找右侧 [mid+1, right]
        } else {
            right = mid - 1; // 继续查找左侧 [left, mid-1]
        }
    }
    return left;
}

```

1.搜索插入位置

`int mid = left + (right - left) / 2;`防止溢出

```

class Solution {
    // lower_bound 返回最小的满足 nums[i] >= target 的 i
    // 如果数组为空, 或者所有数都 < target, 则返回 nums.size()
    // 要求 nums 是非递减的, 即 nums[i] <= nums[i + 1]

    // 闭区间写法
    int lower_bound(vector<int>& nums, int target) {
        int left = 0, right = (int) nums.size() - 1; // 闭区间 [left, right]
        while (left <= right) { // 区间不为空
            // 循环不变量:
            // nums[left-1] < target
            // nums[right+1] >= target
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) {
                left = mid + 1; // 范围缩小到 [mid+1, right]
            } else {
                right = mid - 1; // 范围缩小到 [left, mid-1]
            }
        }
        return left;
    }

    // 左闭右开区间写法

```

```

int lower_bound2(vector<int>& nums, int target) {
    int left = 0, right = nums.size(); // 左闭右开区间 [left, right)
    while (left < right) { // 区间不为空
        // 循环不变量:
        // nums[left-1] < target
        // nums[right] >= target
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1; // 范围缩小到 [mid+1, right)
        } else {
            right = mid; // 范围缩小到 [left, mid)
        }
    }
    return left;
}

// 开区间写法
int lower_bound3(vector<int>& nums, int target) {
    int left = -1, right = nums.size(); // 开区间 (left, right)
    while (left + 1 < right) { // 区间不为空
        // 循环不变量:
        // nums[left] < target
        // nums[right] >= target
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid; // 范围缩小到 (mid, right)
        } else {
            right = mid; // 范围缩小到 (left, mid)
        }
    }
    return right;
}

public:
    int searchInsert(vector<int>& nums, int target) {
        return lower_bound(nums, target); // 选择其中一种写法即可
    }
};

```

2.在排序数组中查找元素的第一个和最后一个位置

```

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

```

```

    }
    vector<int> searchRange(vector<int>& nums, int target) {
        int begin = lower_bound(nums, target);
        if (nums.size() == begin || nums[begin] != target)
            return {-1, -1};
        int end = lower_bound(nums, target + 1);
        return {begin, end - 1};
    }
};

```

3. 寻找旋转排序数组中的最小值

```

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

```

4. 搜索旋转排序数组

方法一 一次二分查找（为什么我想不到？）（数学分段函数问题）

```

class Solution {
public:
    int search(vector<int>& nums, int target) {
        int m = 0;
        int n = nums.size() - 1;

        while(m <= n){
            int mid = (m + n) / 2;
            if(nums[mid] >= nums[0]){
                if(target == nums[mid]) return mid;
                else if(target < nums[mid] && target >= nums[0]){
                    n = mid - 1;
                }
            }
            else {
                m = mid + 1;
            }
        }
    }
};

```



```

        }
    } else {
        if(target == nums[mid]) return mid;
        if(target >= nums[mid] && target < nums[0]){
            m = mid + 1;
        } else {
            n = mid - 1;
        }
    }
}

return -1;
}
};

```

方法二 两次二分查找

**5.寻找两个正序数组的中位数

问题概述

我们有两个有序数组 `nums1` 和 `nums2`，需要找到它们合并后的中位数。合并排序的常规方法是 $O(m + n)$ 的时间复杂度，但在这种情况下，可以通过二分查找降低到 $O(\log(\min(m, n)))$ 。

核心思想：分割两数组，找到中位数位置

1. 理解“中位数”在有序数组中的位置

对于一个数组，**中位数是中间位置的元素**：

- 若数组长度为奇数，中位数是正中间的元素。
- 若数组长度为偶数，中位数是中间两个元素的平均值。

在两个数组的总长度是 `totalLength = m + n` 的情况下：

- 若 `totalLength` 为奇数，则中位数是合并后数组第 $\text{totalLength} / 2 + 1$ 小的元素。
- 若 `totalLength` 为偶数，则中位数是第 $\text{totalLength} / 2$ 和第 $\text{totalLength} / 2 + 1$ 小的元素的平均值。

2. 寻找“分割点”来保证两部分平衡

为了找到中位数，我们不必真正合并数组，而是通过“分割”的方式来在逻辑上确定中间位置。设想将两个数组分割成两部分，使得左侧部分的所有元素都小于右侧部分的所有元素，并且**左侧和右侧的元素数量尽可能相等**。

举个例子：

- 假设 `nums1` 和 `nums2` 的总长度是 9（奇数）。则左侧部分包含 5 个元素，右侧部分包含 4 个元素。
- 假设 `nums1` 和 `nums2` 的总长度是 8（偶数）。则左右各包含 4 个元素。

通过这样的分割，我们可以利用左侧的最大元素和右侧的最小元素来直接得到中位数。

3. 定义两个数组的分割位置

我们定义两个数组 `nums1` 和 `nums2` 的分割位置为 `partition1` 和 `partition2`。这样：

- `partition1` 左边的所有元素属于“左侧部分”；
- `partition2` 左边的所有元素也属于“左侧部分”。

我们希望这两个分割点满足以下条件：

- 左侧部分的元素总数等于右侧部分的元素总数（或者在奇数长度下，左侧比右侧多一个元素）。
- 左侧的最大值小于等于右侧的最小值，即 `maxLeft1 <= minRight2` 且 `maxLeft2 <= minRight1`。

4. 通过二分查找寻找分割位置

为了高效地找到分割点，我们对较短的数组 `nums1` 进行二分查找：

- 选择 `nums1` 的某个位置 `partition1` 将其分为左、右两部分。
- 确定 `partition2 = (halfLength - partition1)` 以保证左右两部分元素数量接近平衡。

在每一步检查：

- 若满足 `maxLeft1 <= minRight2` 且 `maxLeft2 <= minRight1`，则找到了合适的分割位置，可以计算中位数。
- 如果 `maxLeft1 > minRight2`，说明 `partition1` 选得太大，`partition1` 左移。
- 如果 `maxLeft2 > minRight1`，说明 `partition1` 选得太小，`partition1` 右移。

中位数的计算

一旦找到合适的分割点：

- 如果总长度 `totalLength` 为奇数，中位数为左侧的最大值 `max(maxLeft1, maxLeft2)`。
- 如果 `totalLength` 为偶数，中位数为左侧最大值和右侧最小值的平均值，即 `(max(maxLeft1, maxLeft2) + min(minRight1, minRight2)) / 2.0`。

总结

这个算法的关键在于：

1. 利用二分查找的方式确定分割位置。
2. 通过两个分割点将两个有序数组逻辑上分成“左侧”和“右侧”，使得只需在两数组的一部分范围内确定中位数位置，降低了时间复杂度。
3. 利用分割后的左侧最大值和右侧最小值，直接确定中位数的值，而不需实际合并数组。

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int len1 = nums1.size();
        int len2 = nums2.size();

        // 保证 len1 <= len2, 这样我们只对较短数组进行二分
        if (len1 > len2) {
            return findMedianSortedArrays(nums2, nums1);
        }

        int totalLength = len1 + len2;
        int halfLength = (totalLength + 1) / 2; // 中位数位置

        int left = 0, right = len1;
        while (left <= right) {
            int partition1 = (left + right) / 2;
            int partition2 = halfLength - partition1;
```

```

int maxLeft1 = (partition1 == 0) ? INT_MIN : nums1[partition1 - 1];
int minRight1 = (partition1 == len1) ? INT_MAX : nums1[partition1];

int maxLeft2 = (partition2 == 0) ? INT_MIN : nums2[partition2 - 1];
int minRight2 = (partition2 == len2) ? INT_MAX : nums2[partition2];

if (maxLeft1 <= minRight2 && maxLeft2 <= minRight1) {
    // 找到合适的分割
    if (totalLength % 2 == 0) {
        return (max(maxLeft1, maxLeft2) + min(minRight1, minRight2))
/ 2.0;
    } else {
        return max(maxLeft1, maxLeft2);
    }
} else if (maxLeft1 > minRight2) {
    // 说明 partition1 需要左移
    right = partition1 - 1;
} else {
    // 说明 partition1 需要右移
    left = partition1 + 1;
}
}

throw invalid_argument("Input arrays are not sorted.");
}
};

```

十三、排除法

1.搜索二维矩阵

最优解（二分查找） $O(\log mn)$

排除法：

```

class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m = matrix.size(), n = matrix[0].size();
        int i = 0, j = n - 1;
        while (i < m && j >= 0) { // 还有剩余元素
            if (matrix[i][j] == target) {
                return true; // 找到 target
            }
            if (matrix[i][j] < target) {
                i++; // 这一行剩余元素全部小于 target, 排除
            } else {
                j--; // 这一列剩余元素全部大于 target, 排除
            }
        }
        return false;
    }
};

```

```
    }  
};
```

- 时间复杂度: $O(m+n)$, 其中 m 和 n 分别为 *matrix* 的行数和列数。
- 空间复杂度: $O(1)$ 。

2. [搜索二维矩阵 II](#)。

十四、栈

1.有效的括号

```
class Solution {  
    unordered_map<char, char> mp = {{'}', '('}, {'}', '['}, {'}', '{'}, {'}', '{'}};  
public:  
    bool isValid(string s) {  
        if (s.length() % 2) { // s 长度必须是偶数  
            return false;  
        }  
        stack<char> st;  
        for (char c : s) {  
            if (!mp.contains(c)) { // c 是左括号  
                st.push(c); // 入栈  
            } else { // c 是右括号  
                if (st.empty() || st.top() != mp[c]) {  
                    return false; // 没有左括号, 或者左括号类型不对  
                }  
                st.pop(); // 出栈  
            }  
        }  
        return st.empty(); // 所有左括号必须匹配完毕  
    }  
};
```

//使用map利用空间换取代码可维护性

2.最小栈

使用链表

```
class MinStack {  
public:  
    class Node{  
    public:  
        int val;  
        int min;  
        Node* next;  
        Node(int val_ , int min_) : val(val_),min(min_),next(nullptr){}  
    };  
    Node* node;  
    MinStack() {  
        node = nullptr;  
    }  
};
```

```

}

void push(int val) {
    if(node == nullptr){
        node = new Node(val, val);
    } else {
        Node* cur = new Node(val, min(node->min, val));
        cur->next = node;
        node = cur;
    }
}

void pop() {
    Node* temp = node;
    node = node->next;
    delete temp;
}

int top() {
    return node->val;
}

int getMin() {
    return node->min;
}
};

```

使用辅助栈

```

class MinStack {
    stack<pair<int, int>> st;

public:
    MinStack() {
        // 添加栈底哨兵 INT_MAX
        // 这里的 0 写成任意数都可以，反正用不到
        st.emplace(0, INT_MAX);
    }

    void push(int val) {
        st.emplace(val, min(getMin(), val));
    }

    void pop() {
        st.pop();
    }

    int top() {
        return st.top().first;
    }

    int getMin() {
        return st.top().second;
    }
}

```

```
};
```

`st.push(pair(0, INT_MAX));` 和 `st.emplace(0, INT_MAX);` 这两个方法的不同主要体现在其内部实现机制和性能优化上。以下是它们的详细区别：

1. **调用方式与内部行为**

- **`push`**

- `push` 方法需要一个已经构造好的对象作为参数。
- 在 `st.push(pair(0, INT_MAX))` 中, `pair(0, INT_MAX)` 会先调用 `std::make_pair` 或构造函数创建一个临时的 `std::pair` 对象。
- 然后将这个临时对象拷贝（或移动，如果可能）到容器中。

- **`emplace`**

- `emplace` 方法直接在容器的内部构造对象。
- 在 `st.emplace(0, INT_MAX)` 中, `std::pair` 的构造函数直接在栈内被调用，从而避免了额外的临时对象的创建和拷贝/移动。

2. **性能差异**

- `push` 可能涉及：

1. **对象构造**：先在调用 `push` 的地方构造一个临时对象。
2. **对象拷贝/移动**：将构造好的临时对象拷贝或移动到容器中。

- `emplace` 只涉及：

1. **对象原地构造**：直接在栈或容器内调用构造函数。

因此, `emplace` 可以避免一次额外的构造和拷贝/移动操作，在性能上通常优于 `push`。

3. **适用场景**

- **`push`**：

- 更适合用于已经有一个现成的对象时，例如：

```
```cpp
std::pair<int, int> p(0, INT_MAX);
st.push(p);
```

- 如果直接使用 `pair(0, INT_MAX)`，性能上略逊于 `emplace`。
- `emplace`：

- 更适合直接在容器中构造对象，尤其是需要传递构造函数参数时，例如：

```
st.emplace(0, INT_MAX);
```

- 避免了临时对象的创建和拷贝/移动。

总结

方法	是否构造临时对象	是否涉及拷贝/移动	性能
push	是	是	较慢
emplace	否	否	较快

**结论：**  
在你的例子中，如果只需要构造临时对象并直接插入容器，推荐使用 `emplace`，因为它更高效。

```
不使用额外空间

```cpp
class MinStack {
private:
    std::stack<long long> st;
    long long minValue; // 用于记录当前最小值

public:
    MinStack() : minValue(0) {}

    void push(int val) {
        if (st.empty()) {
            st.push(0);          // 计算差值
            minValue = val;      // 更新最小值
        } else {
            st.push((long long)val - minValue); // 保存差值
            if (val < minValue) {
                minValue = val; // 更新最小值
            }
        }
    }

    void pop() {
        if (st.empty()) return;

        long long diff = st.top();
        st.pop();

        if (diff < 0) {
            // 弹出的是一个小于0的差值，恢复上一个最小值
            minValue -= diff; // 恢复到上一个最小值
        }
    }

    int top() {
        long long diff = st.top();
        if (diff > 0) {
            return diff + minValue; // 当前值 = 最小值 + 差值
        } else {
            return minValue; // 当前值就是最小值
        }
    }
}
```

```
int getMin() {  
    return minValue;  
}  
};
```

3.字符串解码

方法一 双栈

方法二 单栈

```
class Solution {  
public:  
    string decodeString(string s) {  
        stack<pair<int,string>> stk;  
        string ch = "";  
        int nums = 0;  
        for(char c : s){  
            if(c>='0'&&c<='9'){  
                nums = nums*10 + c - '0';  
            }  
            else if((c>='a' && c<='z') || (c>='A' && c<='Z')){  
                ch = ch + c;  
            }  
            else if(c == '['){  
                stk.emplace(nums,ch);  
                ch = "";  
                nums = 0;  
            }  
            else if(c == ']){  
                int num = stk.top().first - 1;  
                string temp = ch;  
                while(num--){  
                    ch = ch + temp;  
                }  
                ch = stk.top().second + ch;  
                stk.pop();  
            }  
        }  
        return ch;  
    }  
};
```

方法三 递归

```
class Solution {  
public:  
    string decodeString(string s) {  
        int i = 0; // 当前s字符串遍历位置
```



```

auto help = [&](auto &&help) -> string { // lambda
    int k = 0; // 用于存储重复次数
    string res = "";
    while (i < s.length()) {
        char ch = s[i++]; // 取当前字符，并跳到下一个字符
        if (isdigit(ch)) {
            k = k * 10 + (ch - '0');
        } else if (ch == '[') {
            auto sub = help(help); // 递归解码
            while (k--) res += sub;
            k = 0; // 重置 k（感觉没必要这句，但是不知道为什么在力扣上，少了这句过不去，在自己的电脑上少了这句又没问题）
        } else if (ch == ']') {
            return res; // 返回结果到上一层递归
        } else res += ch;
    }
    return res;
};
return help(help);
}
};

```

4.单调栈

在一维数组中对每一个数找到第一个比自己小的元素。这类“在一维数组中找第一个满足某种条件的数”的场景就是典型的单调栈应用场景。

739.每日温度

从左到右:

```

class Solution {
public:
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        stack<int> st;
        int n = temperatures.size();
        vector<int> res(n, 0);
        for (int i = 0; i < n; i++) {
            while (!st.empty() && temperatures[i] > temperatures[st.top()]) {
                int temp = st.top();
                res[temp] = i - temp;
                st.pop();
            }
            st.push(i);
        }
        return res;
    }
};

```

从右到左:

```

class Solution {
public:
    vector<int> dailyTemperatures(vector<int> &temperatures) {

```

```

int n = temperatures.size();
vector<int> ans(n);
stack<int> st;
for (int i = n - 1; i >= 0; i--) {
    int t = temperatures[i];
    while (!st.empty() && t >= temperatures[st.top()]) {
        st.pop();
    }
    if (!st.empty()) {
        ans[i] = st.top() - i;
    }
    st.push(i);
}
return ans;
}
};

```

42.接雨水

```

class Solution {
public:
    int trap(vector<int>& height) {
        stack<int> st;
        int ans = 0;
        int n = height.size();

        for(int i = 0; i < n; i++){
            while(!st.empty() && height[i] > height[st.top()]){
                int mid = st.top();
                st.pop();
                if(!st.empty()){
                    int left = st.top();
                    ans = ans + (i-left-1) * (min(height[i],height[left]) - height[mid]);
                }
            }
            st.push(i);
        }
        return ans;
    }
};

```

**84.柱状图中的最大矩形(自己写一遍才能感觉到)

我的代码:

```

class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        int n = heights.size();

        stack<int> st_l;
        vector<int> left(n, -1); //这里的初始化要保证边界
        for(int i = 0; i < n; i++){

```

```

        // < 还是 > 还是 = 要认真考虑
        while(!st_l.empty() && heights[i] <= heights[st_l.top()]){
            st_l.pop();
        }
        if(!st_l.empty()) left[i] = st_l.top();
        st_l.push(i);
    }

    stack<int> st_r;
    vector<int> right(n,n); //这里的初始化要保证边界
    for(int i = n-1; i>=0; i--){
        // < 还是 > 还是 = 要认真考虑
        while(!st_r.empty() && heights[i] <= heights[st_r.top()]){
            st_r.pop();
        }
        if(!st_r.empty()) right[i] = st_r.top();
        st_r.push(i);
    }

    int ans = 0;
    for(int i = 0; i < n; i++){
        ans = max(ans, heights[i]*(right[i]-left[i]-1));
    }
    return ans;
}
};

```

两个单调栈

```

class Solution {
public:
    int largestRectangleArea(vector<int> &heights) {
        int n = heights.size();
        vector<int> left(n, -1);
        stack<int> st;
        for (int i = 0; i < n; i++) {
            while (!st.empty() && heights[i] <= heights[st.top()]) {
                st.pop();
            }
            if (!st.empty()) {
                left[i] = st.top();
            }
            st.push(i);
        }

        vector<int> right(n, n);
        st = stack<int>();
        for (int i = n - 1; i >= 0; i--) {
            // < 还是 > 还是 = 要认真考虑
            while (!st.empty() && heights[i] <= heights[st.top()]) {
                st.pop();
            }
            if (!st.empty()) {

```

```

        right[i] = st.top();
    }
    st.push(i);
}

int ans = 0;
for (int i = 0; i < n; i++) {
    ans = max(ans, heights[i] * (right[i] - left[i] - 1));
}
return ans;
}
};

```

一个单调栈(非常妙)

```

class Solution {
public:
    int largestRectangleArea(vector<int>& heights)
    {
        int ans = 0;
        stack<int> st;
        heights.insert(heights.begin(), 0);
        heights.push_back(0);
        for (int i = 0; i < heights.size(); i++)
        {
            while (!st.empty() && heights[st.top()] > heights[i])
            {
                int cur = st.top();
                st.pop();
                int left = st.top() + 1;
                int right = i - 1;
                ans = max(ans, (right - left + 1) * heights[cur]);
            }
            st.push(i);
        }
        return ans;
    }
};

```

十五、异或运算

异或运算有一个重要的性质： $a \oplus a = 0$ 和 $a \oplus 0 = a$ 。这意味着，如果我们对数组中的所有元素进行异或运算，所有出现两次的数字会相互抵消，最终剩下的就是那个只出现一次的数字。

1.找单独的数 (MarsCode)

十六、排序问题

排序算法	时间复杂度			空间复杂度	排序方式	稳定性
	最好	最坏	平均			
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	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	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n^2)$	-	$O(1)$	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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	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	$O(n+k)$	$O(n^2)$	$O(n+k)$	$O(n+k)$	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

快速排序 双路快排 三路快排

堆排序：STL和自己实现大根堆和小根堆

桶排序

1.数组中的第k个最大元素

方法一：基于快速选择方法

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        return selectNum(nums,k);
    }

    int selectNum(vector<int>& nums, int k){
        int pos_num = nums[rand() % nums.size()];
        vector<int> big,small,equal;

        for(int num : nums){
            if(num < pos_num) small.push_back(num);
            if(num > pos_num) big.push_back(num);
            if(num == pos_num) equal.push_back(num);
        }

        if(k <= big.size()){
            return selectNum(big,k);
        }
        if(k > big.size() && k > big.size()+equal.size()){
            return selectNum(small, k - big.size() - equal.size());
        }
        return pos_num;
    }
};
```

```
    }  
};
```

方法二：基于堆排序的选择方法

```
int findKthLargest(vector<int>& nums, int k) {  
    priority_queue<int, vector<int>, greater<int>> minHeap; // 小顶堆  
    for (int num : nums) {  
        minHeap.push(num);  
        if (minHeap.size() > k) { // 堆中仅保留 k 个最大元素  
            minHeap.pop();  
        }  
    }  
    return minHeap.top(); // 堆顶是第 k 大的元素  
}
```

方法三：桶排序

```
int findKthLargest(vector<int>& nums, int k) {  
    int maxVal = *max_element(nums.begin(), nums.end());  
    int minVal = *min_element(nums.begin(), nums.end());  
    int bucketSize = maxVal - minVal + 1;  
  
    vector<int> bucket(bucketSize, 0);  
  
    // 统计每个数字出现的次数  
    for (int num : nums) {  
        bucket[num - minVal]++;  
    }  
  
    // 从大到小遍历桶，找到第 k 大的元素  
    for (int i = bucketSize - 1; i >= 0; i--) {  
        if (bucket[i] > 0) {  
            k -= bucket[i];  
            if (k <= 0) {  
                return i + minVal;  
            }  
        }  
    }  
    return -1; // 不会到达这里  
}
```

2.前K个高频元素

利用堆

```
class Solution {  
public:  
    class tmp{
```

```

public:
    bool operator()(const pair<int,int>& p1,const pair<int,int>& p2){
        //return p1.second < p2.second;
        return p1.second > p2.second;
    }
};

vector<int> topKFrequent(vector<int>& nums, int k) {
    unordered_map<int,int> mp;
    for(int num : nums)
    {
        mp[num]++;
    }
    priority_queue<pair<int,int>,vector<pair<int,int>>,tmp> qe;
    /*for(auto i = mp.begin();i != mp.end();i++){
        qe.push({i->first,i->second});
    } 时间复杂度 nlogn */

    //时间复杂度 nlogk
    //也可以for(auto& [m,n] : mp)
    for(auto n : mp){
        if(qe.size() == k){
            if(n.second > qe.top().second){
                qe.pop();
                qe.push({n.first,n.second});
            }
        }
        else {
            qe.push({n.first,n.second});
        }
    }
    vector<int> res;
    while(!qe.empty()){
        res.push_back(qe.top().first);
        qe.pop();
    }
    /*for(int i = 0; i < k;i++){
        res.push_back(qe.top().first);
        qe.pop();
    }*/
    return res;
}
};

```

3.数据流中的中位数（中位数利用两个大小堆）

我的写法(未利用堆来比较顶端元素 分类讨论较复杂)

```

class MedianFinder {
public:
    MedianFinder() {}

    void addNum(int num) {

```

```

        if (size_b == size_s) {
            if (size_b != 0 && num > small.top()) {
                int tep = small.top();
                small.pop();
                small.push(num);
                big.push(tep);
                size_b++;
            } else {
                big.push(num);
                size_b++;
            }
        } else {
            if (num <= big.top()) {
                int tmp = big.top();
                big.pop();
                big.push(num);
                small.push(tmp);
                size_s++;
            } else {
                small.push(num);
                size_s++;
            }
        }
    }

    double findMedian() {
        return (size_b + size_s) % 2 == 0 ? (big.top() + small.top()) / 2.0
            : big.top();
    }
    priority_queue<int, vector<int>, less<int>> big;
    int size_b = 0;
    priority_queue<int, vector<int>, greater<int>> small;
    int size_s = 0;
};

/**
 * Your MedianFinder object will be instantiated and called as such:
 * MedianFinder* obj = new MedianFinder();
 * obj->addNum(num);
 * double param_2 = obj->findMedian();
 */

```

最优写法 充分利用堆的排序性质

```

class MedianFinder {
    priority_queue<int> left; // 最大堆
    priority_queue<int, vector<int>, greater<>> right; // 最小堆

public:
    void addNum(int num) {
        if (left.size() == right.size()) {
            right.push(num);
            left.push(right.top());
            right.pop();
        } else {

```



```

        left.push(num);
        right.push(left.top());
        left.pop();
    }
}

double findMedian() {
    if (left.size() > right.size()) {
        return left.top();
    }
    return (left.top() + right.top()) / 2.0;
}
};

```

十七、字符串

3.数字字符串格式化

问题描述

小M在工作时遇到了一个问题，他需要将用户输入的不带千分位逗号的数字字符串转换为带千分位逗号的格式，并且保留小数部分。小M还发现，有时候输入的数字字符串前面会有无用的 0，这些也需要精简掉。请你帮助小M编写程序，完成这个任务。

测试样例

样例1：

输入：s = "1294512.12412"
 输出：'1,294,512.12412'

样例2：

输入：s = "0000123456789.99"
 输出：'123,456,789.99'

样例3：

输入：s = "987654321"
 输出：'987,654,321'

```

std::string solution(const std::string& s){
    std::string str = s;

    //删除前置0
    str.erase(0,str.find_first_not_of('0'));
    if(s.empty()) str = "0";

    size_t pos = str.find('.');

    //分出小数和整数

```

```

std::string int_part = str.substr(0,pos);
std::string dec_part = (pos != std::string::npos ? str.substr(pos+1) : "");

int index = 0;
//添加逗号
/*size_t i = int_part.size() - 1; i >= 0; i-- 是不正确的，
因为 i 是 size_t 类型（无符号类型），在 i 为 0 时减去 1 会导致发生下溢，变为非常大的值。
可以改为：*/
for(int i = int_part.size()-1;i >= 0;i--){
    index = (++index)%3;
    if(index == 0 && i !=0) {
        int_part.insert(i,",");
    }
}

return dec_part != "" ? int_part + '.' +dec_part : int_part ;
}

```

十八、正则表达式

在广告平台中，为了给广告主一定的自由性和效率，允许广告主在创造标题的时候以通配符的方式进行创意提交。线上服务的时候，会根据用户的搜索词触发的 bidword 对创意中的通配符（通配符是用成对 {} 括起来的字符串，可以包含 0 个或者多个字符）进行替换，用来提升广告投放体验。例如：“{末日血战} 上线送 SSR 英雄，三天集齐无敌阵容！”，会被替换成“帝国时代游戏下载上线送 SSR 英雄，三天集齐无敌阵容！”。给定一个含有通配符的创意和 n 个标题，判断这句标题是否从该创意替换生成的。

测试样例

样例1：

输入：n = 4, template = "ad{xyz}cdc{y}f{x}e", titles = ["adcdcefdfeffe",
"adcdcefdfeff", "dcdcefdfeffe", "adcdce"]
输出："True,False,False,True"

样例2：

输入：n = 3, template = "a{bdc}efg", titles = ["abcdefg", "abefg", "efg"]
输出："True,True,False"

样例3：

输入：n = 5, template = "{abc}xyz{def}", titles = ["xyzdef", "abcdef",
"abxyzdef", "xyz", "abxyz"]
输出："True,False,True,True,True"

```

#include <iostream>
#include <regex>
#include <string>
#include <vector>

std::string solution(int n, const std::string& template_, const
std::vector<std::string>& titles) {
    // 将模板转换为正则表达式

```

```

std::string pattern = std::regex_replace(template_, std::regex("\\{[^}]*\\}"), ".*");
std::string res;

// 对每个标题进行匹配
for (const auto& title : titles) {
    if (std::regex_match(title, std::regex("^" + pattern + "$"))) {
        res += (res.empty() ? "True" : ",True");
    } else {
        res += (res.empty() ? "False" : ",False");
    }
}
return res;
}

int main() {
    std::vector<std::string> testTitles1 = {"adcdcefdfeffe", "adcdcefdfeff",
"dc dcefdfeffe", "adcdcefe"};
    std::vector<std::string> testTitles2 = {
        "CLS omGhcQNVFuzENTAMLCqxBdj", "CLS omNVFuXTASzENTAMLCqxBdj",
        "CLS omFuXTASzEXBdj", "CLSoQNVFuMLCqxBdj",
        "SovFuXTASzENTAMLCq", "mGhcQNVFuXTASzENTAMLCqx"};
    std::vector<std::string> testTitles3 = {"abcdefg", "abefg", "efg"};

    std::cout << (solution(4, "ad{xyz}cdc{y}f{x}e", testTitles1) ==
"True,False,False,True") << std::endl;
    std::cout << (solution(6, "{xxx}h{cQ}N{vF}u{XTA}S{NTA}MLCq{yyy}", testTitles2) ==
        "False,False,False,False,False,True")
        << std::endl;
    std::cout << (solution(3, "a{bdc}efg", testTitles3) == "True,True,False") <<
    std::endl;

    return 0;
}

```

十九、贪心算法

跳跃游戏1 2 见leetcode

1.灌溉花园的最小水龙头数目(和跳跃游戏II一样)(建桥问题)

将每个水龙头点上的问题 转换为区间段上的问题

```

class Solution {
public:
    int minTaps(int n, vector<int>& ranges) {
        //将水龙头点的问题转化为区间段问题
        vector<int> max_right(n+1,0);
        for(int i = 0; i <= n; i++){
            //if(ranges[i] == 0) continue;
            int left = max(i - ranges[i], 0);
            int right = min(i + ranges[i], n);

```

```

        max_right[left] = max(max_right[left], right);
    }

    int num = 0;
    int pos = 0;
    int next_pos = 0;
    for(int i = 0; i < n; i++){
        next_pos = max(next_pos, max_right[i]);
        if(i == pos){
            if(i == next_pos) return -1;
            pos = next_pos;
            num++;
        }
    }
    return num;
}
};

```

2.视频拼接

```

class Solution {
public:
    int videoStitching(vector<vector<int>>& clips, int time) {
        vector<int> rightMax(time + 1, 0);
        for(int i = 0; i < clips.size(); i++){
            if(clips[i][0] < time)
                rightMax[clips[i][0]] = max(rightMax[clips[i][0]], clips[i][1]);
        }
        int num = 0;
        int pos = 0;
        int next_pos = 0;
        for(int i = 0; i < time; i++){
            next_pos = max(next_pos, rightMax[i]);
            if(pos == i){
                if(i == next_pos) return -1;
                pos = next_pos;
                num++;
            }
        }
        return num;
    }
};

```

3.划分字母区间

类似于合并区间端点问题 贪心算法

```

class Solution {
public:
    vector<int> partitionLabels(string s) {
        int max_right = 0;
        int pos = 0;
        vector<int> result;
        for(int i = 0; i < s.size(); i++){

```

```

        max_right = max(max_right, (int)s.find_last_of(s[i]));
        if(i == max_right){
            result.push_back(i - pos + 1);
            pos = max_right + 1;
        }
    }
    return result;
}
};

```

4.合并区间

```

class Solution {
public:
    // struct sortinter{
    //     bool operator()(const vector<int>& n1 ,const vector<int>& n2){
    //         return n2[0] > n1[0];
    //     }
    // };
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        //排序默认按第一个元素从小到大
        sort(intervals.begin(),intervals.end());
        int left = intervals[0][0];
        int right = intervals[0][1];
        vector<vector<int>> res;
        for(int i = 0; i < intervals.size(); i++){
            if(intervals[i][0] <= right) right = max(right,intervals[i][1]);
            else {
                res.push_back({left,right});
                left = intervals[i][0];
                right = intervals[i][1];
            }
            if(i == intervals.size() - 1) res.push_back({left,right});
        }
        return res;
    }
};

```

##

二十、回溯(通常用到递归)

[总结了回溯问题类型，带你搞懂回溯算法](#)

回溯法 采用试错的思想，它尝试分步的去解决一个问题。在分步解决问题的过程中，当它通过尝试发现现有的分步答案不能得到有效的正确的解答的时候，它将取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问题的答案。回溯法通常用最简单的递归方法来实现，在反复重复上述的步骤后可能出现两种情况：

找到一个可能存在的正确的答案；

在尝试了所有可能的分步方法后宣告该问题没有答案。

分为三种类型：子集、组合 排列 搜索

回溯也有不同写法

在递归函数中

1.只有最后if(index == nums.size()) res.push_back(current) + 不含for循环

2.每次res.push_back() + 含for循环

1.电话号码的字母组合(组合)

```
// 回溯函数
void backtrack(const string &digits, const vector<string> &mapping, int index,
string &current, vector<string> &result) {
    // 如果当前组合长度等于输入数字长度，加入结果
    if (index == digits.size()) {
        result.push_back(current);
        return;
    }

    // 当前数字对应的字母集
    string letters = mapping[digits[index] - '0'];
    for (char letter : letters) {
        current.push_back(letter);    // 选择当前字母
        backtrack(digits, mapping, index + 1, current, result); // 递归
        current.pop_back();          // 撤销选择
    }
}

vector<string> letterCombinations(string digits) {
    if (digits.empty()) return {}; // 输入为空的边界情况

    // 数字到字母的映射表，0和1对应空字符串
    vector<string> mapping = {
        "", "", "abc", "def", "ghi", "jkl",
        "mno", "pqrs", "tuv", "wxyz"
    };

    vector<string> result; // 存放结果
    string current;       // 当前路径
    backtrack(digits, mapping, 0, current, result);
    return result;
}
```

2.全排列(排列)

在上一题基础上传参优化

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        dfs(nums, 0);
        return res;
    }
private:
    vector<vector<int>> res;
    void dfs(vector<int> nums, int x) {
        if (x == nums.size() - 1) {
            res.push_back(nums);    // 添加排列方案
            return;
        }
        for (int i = x; i < nums.size(); i++) {
            swap(nums[i], nums[x]);    // 交换, 将 nums[i] 固定在第 x 位
            dfs(nums, x + 1);          // 开启固定第 x + 1 位元素
            swap(nums[i], nums[x]);    // 恢复交换
        }
    }
};
```

3.子集(子集)

与上面两题有点不同之处

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> current;
    void runback(vector<int> nums, int index) {
        if(index == nums.size()) {    // 递归终止条件: 到达数组末尾
            res.push_back(current);    // 将当前路径生成的子集加入结果集
            return;                    // 结束当前递归
        }

        // 选择当前元素, 加入当前路径
        current.push_back(nums[index]);
        runback(nums, index + 1);    // 递归处理下一个元素
        current.pop_back();          // 回溯: 移除当前元素

        // 不选择当前元素, 直接递归处理下一个元素
        runback(nums, index + 1);
    }

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

```

class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> ans;
        vector<int> path;
        int n = nums.size();
        auto dfs = [&](this auto&& dfs, int i) -> void {
            ans.emplace_back(path);
            for (int j = i; j < n; j++) { // 枚举选择的数字
                path.push_back(nums[j]);
                dfs(j + 1);
                path.pop_back(); // 恢复现场
            }
        };
        dfs(0);
        return ans;
    }
};

```

```

class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        int n = nums.size();
        vector<vector<int>> ans(1 << n);
        for (int i = 0; i < (1 << n); i++) { // 枚举全集 U 的所有子集 i
            for (int j = 0; j < n; j++) {
                if (i >> j & 1) { // j 在集合 i 中
                    ans[i].push_back(nums[j]);
                }
            }
        }
        return ans;
    }
};

```

4.子集II(子集)

在不选 `nums[i]` 时，要跳过后续所有等于 `nums[i]` 的数。如果不跳过这些数，

设 `x=nums[i]`, `x'=nums[i+1]`，那么「选 `x` 不选 `x'`」和「不选 `x` 选 `x'`」这两种情况都会加到答案中，这就重复了。

去重需要排序

```

class Solution {
public:
    vector<vector<int>> res;
    vector<int> current;
    void runback(vector<int>& nums, int index){
        if(index == nums.size()){
            res.push_back(current);
            return;
        }
        current.push_back(nums[index]);
    }
};

```



```

        runback(nums, index+1);
        current.pop_back();
        while(index + 1 < nums.size() && nums[index + 1] == nums[index]){
            index++;
        }
        runback(nums, index+1);
        return;
    }
    vector<vector<int>> subsetValueWithDup(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        runback(nums, 0);
        return res;
    }
};

```

```

class Solution {
public:
    vector<vector<int>> res;
    vector<int> current;

    void runback(vector<int>& nums, int index) {
        res.emplace_back(current); // 每个节点的状态都可以构成一个子集
        for (int i = index; i < nums.size(); ++i) {
            // 跳过同一层的重复元素
            if (i > index && nums[i] == nums[i - 1]) continue;

            current.push_back(nums[i]);
            runback(nums, i + 1);
            current.pop_back();
        }
    }

    vector<vector<int>> subsetValueWithDup(vector<int>& nums) {
        sort(nums.begin(), nums.end()); // 排序以方便去重
        runback(nums, 0);
        return res;
    }
};

```

5.组合总和

```

class Solution {
public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> res;
        vector<int> current;
        int sum = 0;
        function<void(int)> runback = [&](int index){
            if(sum == target) {
                res.push_back(current);
                return;
            }
            if(index >= candidates.size() || sum > target) return;

```

```

        runback(index+1);
        sum += candidates[index];
        current.push_back(candidates[index]);
        runback(index);
        //下面这句会重复
        //runback(index+1);

        sum -= candidates[index];
        current.pop_back();
    };
    runback(0);
    return res;
}
};

```

6.括号生成

```

class Solution {
public:
    string current;
    vector<string> res;
    void backtrack(int open, int close, int& n){
        if(current.size() == n * 2){
            res.push_back(current);
            return;
        }
        if(open < n){
            current.push_back('(');
            backtrack(open+1, close, n);
            current.pop_back();
        }
        if(close < open){
            current.push_back(')');
            backtrack(open, close+1, n);
            current.pop_back();
        }
    }
    vector<string> generateParenthesis(int n) {
        backtrack(0, 0, n);
        return res;
    }
};

```

7.单词搜索(搜索)

方法一 利用临时标记

```

class Solution {
public:
    bool exist(vector<vector<char>>& board, string word) {
        int rows = board.size();
        int cols = board[0].size();
    }
};

```

```

// 辅助函数：深度优先搜索
function<bool(int, int, int)> dfs = [&](int row, int col, int index) ->
bool {
    // 如果匹配到单词的所有字符
    if (index == word.size()) return true;
    // 如果越界或字符不匹配
    if (row < 0 || row >= rows || col < 0 || col >= cols || board[row]
[col] != word[index])
        return false;

    // 临时标记当前单元格为访问过
    char temp = board[row][col];
    board[row][col] = '#';

    // 搜索四个方向
    bool found = dfs(row + 1, col, index + 1) ||
        dfs(row - 1, col, index + 1) ||
        dfs(row, col + 1, index + 1) ||
        dfs(row, col - 1, index + 1);

    // 回溯：恢复单元格的原始状态
    board[row][col] = temp;

    return found;
};

// 遍历每个单元格作为起点
for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
        if (dfs(r, c, 0)) return true; // 从当前位置开始搜索
    }
}

return false; // 未找到单词
};

```

8.分割回文串

```

class Solution {
public:
    bool isPalindrome(string &s, int left, int right) {
        while (left < right) {
            if (s[left] != s[right]) return false;
            left++;
            right--;
        }
        return true;
    }

    vector<vector<string>> partition(string s) {
        vector<vector<string>> res;
        vector<string> current;

        function<void(int)> dfs = [&](int left) {

```

```

        if (left == s.size()) {
            res.push_back(current);
            return;
        }

        for (int right = left; right < s.size(); ++right) {
            if (isPalindrome(s, left, right)) {
                current.push_back(s.substr(left, right - left + 1)); // 当前
回文子串加入当前路径
                dfs(right + 1); // 递归寻找剩下的部分
                current.pop_back(); // 回溯，移除当前子串
            }
        }
    };

    dfs(0); // 从索引 0 开始进行回溯
    return res;
}
};

```

9.N皇后

```

class Solution {
public:
    vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> res; // 存储最终的解
        vector<string> board(n, string(n, '.')); // 初始化棋盘
        vector<bool> cols(n, false); // 用于标记列是否被占用
        vector<bool> diag1(2 * n - 1, false); // 用于标记 "/" 方向的对角线是否被占用
        vector<bool> diag2(2 * n - 1, false); // 用于标记 "\" 方向的对角线是否被占用
        backtrack(res, board, 0, n, cols, diag1, diag2); // 开始回溯
        return res;
    }

private:
    // 回溯函数
    void backtrack(vector<vector<string>>& res, vector<string>& board, int row,
int n,
                vector<bool>& cols, vector<bool>& diag1, vector<bool>& diag2)
    {
        if (row == n) { // 所有行都已放置皇后
            res.push_back(board);
            return;
        }

        for (int col = 0; col < n; ++col) {
            // 检查当前列和对角线是否已被占用
            if (cols[col] || diag1[row + col] || diag2[row - col + n - 1])
                continue;

            // 放置皇后
            board[row][col] = 'Q';
            cols[col] = true;
            diag1[row + col] = true;

```

```

        diag2[row - col + n - 1] = true;

        // 递归放置下一行的皇后
        backtrack(res, board, row + 1, n, cols, diag1, diag2);

        // 回溯
        board[row][col] = '.'; // 恢复棋盘
        cols[col] = false;
        diag1[row + col] = false;
        diag2[row - col + n - 1] = false;
    }
}
};

```

10.解数独

```

class Solution {
public:
    void solveSudoku(vector<vector<char>>& board) {
        backtrack(board);
    }

private:
    // 回溯法：尝试填充每个空格
    bool backtrack(vector<vector<char>>& board) {
        for (int row = 0; row < 9; ++row) {
            for (int col = 0; col < 9; ++col) {
                if (board[row][col] == '.') { // 如果当前格子是空的
                    for (char num = '1'; num <= '9'; ++num) { // 尝试每个数字
                        if (isValid(board, row, col, num)) { // 检查是否合法
                            board[row][col] = num; // 填充当前数字
                            if (backtrack(board)) { // 继续递归填充下一个空格
                                return true; // 找到一个解
                            }
                        }
                        board[row][col] = '.'; // 回溯，撤销当前填充
                    }
                }
            }
            return false; // 如果没有找到合法数字，回溯
        }
        return true; // 所有格子都已填充完成，返回 true
    }

    // 检查填充的数字是否在当前行、列和 3x3 宫格中合法
    bool isValid(vector<vector<char>>& board, int row, int col, char num) {
        for (int i = 0; i < 9; ++i) {
            // 检查行和列
            if (board[row][i] == num || board[i][col] == num) {
                return false;
            }
            // 检查 3x3 宫格
            // 一个/ 一个%
            if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == num) {

```

```

        return false;
    }
}
return true;
}
};

```

```

class Solution {
public:
    void solveSudoku(vector<vector<char>>& board) {
        function<bool()> dfs = [&]()->bool{
            for(int i = 0; i < 9; i++){
                for(int j = 0; j < 9; j++){
                    if(board[i][j] != '.') continue;
                    for(char num = '1'; num <= '9'; num++){
                        if(isValid(board, i, j, num)){
                            board[i][j] = num;
                            if(dfs()) return true;
                        }
                    }
                    board[i][j] = '.';
                    return false;
                }
            }
            return true;
        };
        dfs();
    }

    bool isValid(vector<vector<char>>& board, int row, int col, char num){
        for(int i = 0; i < 9; i++){
            if(board[row][i] == num || board[i][col] == num ||
                board[i / 3 + row / 3 * 3][i % 3 + col / 3 *
3] == num)
                return false;
        }
        return true;
    }
};

```

二十一、大数问题

1. 字符串相加

二十二、动态规划(画树)(基础)(01背包)(完全背包)(选还是不选)(多重背包)(分组背包)

背包问题(至多 恰好 至少) 背包问题就用一维数组来做!

动态规划的时间复杂度 = 状态个数 × 单个状态的计算时间

01背包 先遍历所有物品 再逆序遍历背包容量

完全背包问题中，你的遍历顺序是不同的。完全背包问题允许每个物品被多次选择，因此，你需要先遍历背包容量，然后再遍历物品。这和 0-1 背包问题的遍历顺序是相反的。

动态规划注意事项：①一维数组初始值对不对

②两个for循环有没有反

③需不需要max min

④递归是从最后往前算 而动态规划是从开始往最后算

注：动态规划有「选或不选」和「枚举选哪个」这两种基本思考方式。在做题时，可根据题目要求，选择适合题目的一种来思考。

在完全背包、0-1 背包、以及多重背包和分组背包问题中，遍历顺序有所不同。具体来说：

1. 完全背包问题

在完全背包问题中，背包的容量是可以多次装入同一个物品的。因此，遍历顺序是 **先遍历背包容量，再遍历物品**。这种遍历顺序保证了每个物品能够被多次使用。

2. 0-1 背包问题

在 0-1 背包问题中，每个物品只能使用一次，因此遍历顺序是 **先遍历物品，再遍历背包容量**。这样做可以确保物品的每次选择只影响当前背包容量的计算，而不干扰之前的物品选择。

3. 多重背包问题

在多重背包问题中，每个物品的使用次数是有限制的，因此，需要通过动态规划的方法来处理。通常，遍历顺序是 **先遍历物品，再遍历背包容量**，但是在更新 dp 数组时要根据物品的数量来处理选择次数。

常见的做法是通过将一个物品分成多个“虚拟”物品进行处理，或采用二进制拆分方法来避免重复计算。遍历顺序和 0-1 背包类似，但需要在选择物品时考虑物品的数量限制。

4. 分组背包问题

在分组背包问题中，物品被分为若干组，在每组内可以选择任意物品，但每组只能选择一个物品。通常，遍历顺序是 **先遍历背包容量，再遍历每组物品**。这种遍历方式确保了在背包容量已经确定的情况下，选择每个组内的物品时不受干扰。

总结：

- **完全背包**：先遍历背包容量，再遍历物品。
- **0-1 背包**：先遍历物品，再遍历背包容量。
- **多重背包**：先遍历物品，再遍历背包容量，但考虑物品数量限制。
- **分组背包**：先遍历背包容量，再遍历每组物品。

1.爬楼梯

首先想到常规递归

```
// 会超时的递归代码
class Solution {
    int dfs(int i) {
        if (i <= 1) { // 递归边界
            return 1;
        }
        return dfs(i - 1) + dfs(i - 2);
    }

public:
    int climbStairs(int n) {
        return dfs(n);
    }
};
```

递归 + 记录返回值 = 记忆化搜索

```
class Solution {
    vector<int> memo;

    int dfs(int i) {
        if (i <= 1) { // 递归边界
            return 1;
        }
        int& res = memo[i]; // 注意这里是引用
        if (res) { // 之前计算过
```



```

        return res;
    }
    return res = dfs(i - 1) + dfs(i - 2); // 记忆化
}

public:
    int climbStairs(int n) {
        memo.resize(n + 1);
        return dfs(n);
    }
};

```

递推

```

class Solution {
public:
    int climbStairs(int n) {
        vector<int> f(n + 1);
        f[0] = f[1] = 1;
        for (int i = 2; i <= n; i++) {
            f[i] = f[i - 1] + f[i - 2];
        }
        return f[n];
    }
};

```

空间优化

```

class Solution {
public:
    int climbStairs(int n) {
        int f0 = 1, f1 = 1;
        for (int i = 2; i <= n; i++) {
            int new_f = f1 + f0;
            f0 = f1;
            f1 = new_f;
        }
        return f1;
    }
};

```

2.打家劫舍

```

class Solution {
public:
    int dfs(vector<int>& nums, int num){
        if(num > nums.size() - 1) return 0;
        if(num == nums.size() - 1) return nums[num];
        if(num == nums.size() - 2) return max(nums[num], nums[num + 1]);
        int temp = 0;
        //选第一个
        temp += nums[num];
        int first = dfs(nums, num + 2) + temp;
    }
};

```

```

        //恢复
        temp = 0;
        //选第二个
        temp += nums[num+1];
        int second = dfs(nums,num + 3) + temp;
        return max(first,second);
    }
    int rob(vector<int>& nums) {
        int res = dfs(nums,0);
        return res;
    }
};

```

存在以下几个问题和优化点：

问题分析：

1. 重复子问题导致性能低下：

- `dfs` 函数在同一个 `num` 上会被多次调用，导致大量的重复计算。例如：在计算 `dfs(nums, 2)` 时，它可能会多次计算相同的子问题。
- 这是典型的 **重叠子问题**，可以使用动态规划或记忆化搜索来优化。

2. 不必要的变量 `temp`：

- 变量 `temp` 用于保存 `nums[num]` 或 `nums[num + 1]`，但实际上并不需要，它可以直接用 `nums[num]` 和 `nums[num + 1]` 替代。

3. 没有对输入进行边界检查：

- 当 `nums` 的长度为 0 或 1 时，代码虽然能处理，但未显式说明行为。

优化思路：

1. 记忆化搜索 (Memoization)：

- 使用一个数组 `memo` 来保存 `dfs(nums, num)` 的结果。如果某个位置的值已经计算过，则直接返回，避免重复计算。

2. 去掉冗余变量：

- 直接使用递归结果，避免中间变量。

3. 动态规划 (非递归)：

- 可进一步优化为自底向上的动态规划，空间复杂度从 $O(n)O(n)$ 降到 $O(1)O(1)$ 。

修复后的代码：

使用记忆化搜索优化递归版本：

```

class Solution {
public:
    int dfs(vector<int>& nums, int num, vector<int>& memo) {
        if (num >= nums.size()) return 0; // 超出数组范围返回 0
        if (memo[num] != -1) return memo[num]; // 返回已计算的结果
    }
};

```

```

        // 选当前房子或不选当前房子的最大值
        int res = max(nums[num] + dfs(nums, num + 2, memo), dfs(nums, num + 1, memo));
        memo[num] = res; // 保存结果
        return res;
    }

    int rob(vector<int>& nums) {
        if (nums.empty()) return 0; // 边界情况
        vector<int> memo(nums.size(), -1); // 初始化记忆化数组
        return dfs(nums, 0, memo); // 从第 0 个房子开始
    }
};

```

动态规划优化：

动态规划的核心是将递归转换为迭代，避免函数调用栈的开销。

```

class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.empty()) return 0; // 边界情况
        if (nums.size() == 1) return nums[0]; // 只有一个房子

        int prev2 = 0, prev1 = 0; // `prev2` 表示两步之前, `prev1` 表示一步之前的最大值

        for (int num : nums) {
            int curr = max(prev1, prev2 + num); // 当前房子的选择
            prev2 = prev1; // 更新两步之前的值
            prev1 = curr; // 更新一步之前的值
        }
        return prev1; // 返回最终结果
    }
};

```

优化后的复杂度：

1. 时间复杂度：

- 记忆化搜索和动态规划都是 $O(n)$ ，其中 n 是房子的数量。

2. 空间复杂度：

- 记忆化搜索为 $O(n)$ （递归栈和 `memo` 数组）。
- 动态规划优化版本为 $O(1)$ 。

示例:

输入:

```
vector<int> nums = {2, 7, 9, 3, 1};
```

输出:

最大金额为 12。

解释:

- 选择第 1、3、5 个房子 (金额为 2、9、1) 。
- 总金额为 $2+9+1=12$ 。

**3.目标和

暴力回溯(超时)

错误代码

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int res = 0;
        int ans = 0;
        function<void(int)> dfs = [&](int n){
            if(n == -1){
                if(res == target) ans++;
                return;
            }
            res -= nums[n];
            dfs(n - 1);
            res = res + 2*nums[n];
            dfs(n - 1);
        };
        dfs(nums.size() - 1);
        return ans;
    }
};
```

回溯时要把中间变化的量作为局部变量传递

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int ans = 0;
        function<void(int,int)> dfs = [&](int n,int sum){
            if(n == -1){
                if(sum == target) ans++;
                return;
            }

            dfs(n - 1,sum - nums[n]);
        };
    }
};
```

```

        dfs(n - 1, sum + nums[n]);
    };
    dfs(nums.size() - 1, 0);
    return ans;
}
};

```

暴力回溯时尽量使用函数，使用lambda捕获参数会更加耗时，下面代码能通过

```

class Solution {
public:
    int ans = 0;
    void dfs(int n, int res, int& target, vector<int>& nums) {
        if(n == nums.size()) {
            if(res == target) ans++;
            return ;
        }
        dfs(n + 1, res + nums[n], target, nums);
        dfs(n + 1, res - nums[n], target, nums);
    }
    int findTargetSumWays(vector<int>& nums, int target) {
        dfs(0, 0, target, nums);
        return ans;
    }
};

```

DP(01背包问题)

```

class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum = 0;
        for(int num : nums){
            sum += num;
        }
        if((target + sum) % 2 != 0 || (target + sum) < 0) return 0;
        vector<int> memo((target + sum) / 2 + 1, 0);
        memo[0] = 1;
        for(int num : nums){
            for(int i = (target + sum) / 2; i >= 0; i--){
                if((i - num) >= 0)
                    memo[i] += memo[i - num];
            }
        }
        return memo[(target + sum) / 2];
    }
};

```

4.零钱兑换

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> memo(amount + 1,-1);
        memo[0] = 0;
        for(int i = 1;i <= amount;i++){
            for(int co : coins){
                if(co <= i && memo[i - co] != -1){
                    if(memo[i] != -1)
                        memo[i] = min(memo[i - co] + 1,memo[i]);
                    else
                        memo[i] = memo[i - co] + 1;
                }
            }
        }
        return memo[amount];
    }
};
```

5.完全平方数

```
class Solution {
public:
    int numSquares(int n) {
        vector<int> memo(n + 1,-1);
        function<int(int)> dfs = [&](int n)->int{
            if(n == 1) return 1;
            if(memo[n] != -1) return memo[n];
            int res = n;
            for(int i = 1;i*i <= n;i++){
                res = min(dfs(n - i*i) + 1,res);
            }
            memo[n] = res;
            return res;
        };
        return dfs(n);
    }
};
```

```

class Solution {
public:
    int numSquares(int n) {
        vector<int> memo(n + 1, n);
        memo[0] = 0;          //从0开始初始化
        for(int i = 1; i <= n; i++){
            for(int j = 1; j*j <= i; j++){
                memo[i] = min(memo[i], memo[i-j*j] + 1) ;
            }
        }
        return memo[n];
    }
};

```

6.组合总数IV

```

class Solution {
public:
    int combinationSum4(vector<int>& nums, int target) {
        vector<int> memo(target + 1, -1); // -1 表示没有计算过//这里是-1
        auto dfs = [&](this auto&& dfs, int i) {
            if (i == 0) { // 爬完了
                return 1;
            }
            int &res = memo[i]; // 注意这里是引用
            if (res != -1) { // 之前计算过
                return res;
            }
            res = 0;
            for (int x : nums) {
                if (x <= i) {
                    res += dfs(i - x);
                }
            }
            return res;
        };
        return dfs(target);
    }
};

```

```

class Solution {
public:
    int combinationSum4(vector<int>& nums, int target) {
        // 使用 unsigned 可以让溢出不报错
        // 对于溢出的数据, 不会影响答案的正确性 (题目保证)
        vector<unsigned> f(target + 1, 0); //这里是0
        f[0] = 1;
        for (int i = 1; i <= target; i++) {
            for (int x : nums) {
                if (x <= i) {
                    f[i] += f[i - x];
                }
            }
        }
    }
};

```

```

    }
}
return f[target];
}
};

```

7.单词划分

主要注意substr函数的使用

```

class Solution {
public:
    int wordBreak(string s, vector<string>& wordDict) {
        vector<bool> memo(s.size() + 1, 0);
        memo[0] = 1; // 初始化，表示空字符串总是能被分词

        for (int i = 1; i <= s.size(); i++) {
            for (string ch : wordDict) {
                // 如果当前索引 i 能匹配到字典中的单词，并且 memo[i - ch.size()] 是
可行的
                if (i >= ch.size() && s.substr(i - ch.size(), ch.size()) ==
ch) {
                    memo[i] = memo[i] | memo[i - ch.size()];
                }
            }
        }

        return memo[s.size()];
    }
};

```

8.最小路径和(网格dp)(优化空间?)

9.下降路径最小和(网格dp)(优化空间?)

```

class Solution {
public:
    int minFallingPathSum(vector<vector<int>>& matrix) {
        int row = matrix.size();
        int col = matrix[0].size();
        vector<vector<int>> memo(row + 1, vector<int>(col, INT_MAX));
        for(int i = 0; i < col; i++){
            memo[0][i] = 0;
        }
        for(int i = 1; i <= row; i++){
            for(int j = 0; j < col; j++){
                memo[i][j] = min(min((j - 1 >= 0 ? memo[i - 1][j - 1] : INT_MAX),
(j + 1 < col ? memo[i - 1][j + 1] : INT_MAX)),
(memo[i - 1][j])) + matrix[i - 1][j];
            }
        }
        return *min_element(memo[row].begin(), memo[row].end());
    }
};

```



```
}  
};
```

10.最长公共子序列(线性dp 多维)

***11.编辑距离(线性dp 多维)

[类似字符串增删操作总结](#)

#1. 确定dp数组（dp table）以及下标的含义

$dp[i][j]$ 表示以下标 $i-1$ 为结尾的字符串word1，和以下标 $j-1$ 为结尾的字符串word2，最近编辑距离为 $dp[i][j]$ 。

有同学问了，为啥要表示下标 $i-1$ 为结尾的字符串呢，为啥不表示下标 i 为结尾的字符串呢？

为什么这么定义我在 [718. 最长重复子数组 \(opens new window\)](#)中做了详细的讲解。

其实用 i 来表示也可以！用 $i-1$ 就是为了方便后面dp数组初始化的。

#2. 确定递推公式

在确定递推公式的时候，首先要考虑清楚编辑的几种操作，整理如下：

```
if (word1[i - 1] == word2[j - 1])  
    不操作  
if (word1[i - 1] != word2[j - 1])  
    增  
    删  
    换
```

也就是如上4种情况。

```
if (word1[i - 1] == word2[j - 1])` 那么说明不用任何编辑，`dp[i][j]` 就应该是 `dp[i - 1][j - 1]`，即`dp[i][j] = dp[i - 1][j - 1]`；
```

此时可能有同学有点不明白，为啥要即 $dp[i][j] = dp[i - 1][j - 1]$ 呢？

那么就在回顾上面讲过的 $dp[i][j]$ 的定义， $word1[i - 1]$ 与 $word2[j - 1]$ 相等了，那么就不用编辑了，以下标 $i-2$ 为结尾的字符串word1和以下标 $j-2$ 为结尾的字符串word2 的最近编辑距离 $dp[i - 1][j - 1]$ 就是 $dp[i][j]$ 了。

在下面的讲解中，如果哪里看不懂，就回想一下 $dp[i][j]$ 的定义，就明白了。

在整个动规的过程中，最为关键就是正确理解 $dp[i][j]$ 的定义！

$if (word1[i - 1] != word2[j - 1])$ ，此时就需要编辑了，如何编辑呢？

- 操作一：word1删除一个元素，那么就是以下标 $i - 2$ 为结尾的word1 与 $j-1$ 为结尾的word2的最近编辑距离 再加上一个操作。

即 $dp[i][j] = dp[i - 1][j] + 1;$

- 操作二: word2删除一个元素, 那么就是以下标 $i-1$ 为结尾的word1 与 $j-2$ 为结尾的word2的最近编辑距离 再加上一个操作。

即 $dp[i][j] = dp[i][j - 1] + 1;$

这里有同学发现了，怎么都是删除元素，添加元素去哪了。

word2添加一个元素，相当于word1删除一个元素，例如 `word1 = "ad"`，`word2 = "a"`，`word1` 删除元素 'd' 和 `word2` 添加一个元素 'd'，变成 `word1="a"`，`word2="ad"`，最终的操作数是一样！dp 数组如下图所示的：

```

      a
+-----+-----+
|  0  |  1  |
+-----+-----+
a |  1  |  0  |
+-----+-----+
d |  2  |  1  |
+-----+-----+

      a      d
+-----+-----+-----+
|  0  |  1  |  2  |
+-----+-----+-----+
a |  1  |  0  |  1  |
+-----+-----+-----+

```

操作三：替换元素，word1 替换 word1[i - 1]，使其与 word2[j - 1] 相同，此时不用增删加元素。

可以回顾一下，if (word1[i - 1] == word2[j - 1]) 的时候我们的操作是 dp[i][j] = dp[i - 1][j - 1] 对吧。

那么只需要一次替换的操作，就可以让 word1[i - 1] 和 word2[j - 1] 相同。

所以 `dp[i][j] = dp[i - 1][j - 1] + 1;`

综上, 当 `if (word1[i - 1] != word2[j - 1])` 时取最小的, 即: `dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]}) + 1;`

递归公式代码如下：

```
if (word1[i - 1] == word2[j - 1]) {
    dp[i][j] = dp[i - 1][j - 1];
}
else {
    dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]}) + 1;
}
```

#3. dp数组如何初始化

再回顾一下 $dp[i][j]$ 的定义:

$dp[i][j]$ 表示以下标 $i-1$ 为结尾的字符串 $word1$ ，和以下标 $j-1$ 为结尾的字符串 $word2$ ，最近编辑距离为 $dp[i][j]$ 。

那么 $dp[i][0]$ 和 $dp[0][i]$ 表示什么呢?

dp[i][0]：以下标i-1为结尾的字符串word1，和空字符串word2，最近编辑距离为dp[i][0]。

那么dp[i][0]就应该是i, 对word1里的元素全部做删除操作, 即: dp[i][0] = i;

同理 $dp[0][i] = i$;

所以C++代码如下:

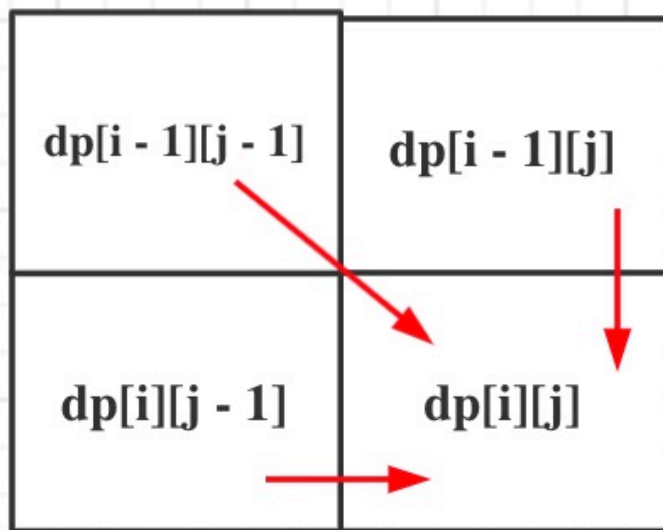
```
for (int i = 0; i <= word1.size(); i++) dp[i][0] = i;
for (int j = 0; j <= word2.size(); j++) dp[0][j] = j;
```

#4. 确定遍历顺序

从如下四个递推公式：

- $dp[i][j] = dp[i - 1][j - 1]$
- $dp[i][j] = dp[i - 1][j - 1] + 1$
- $dp[i][j] = dp[i][j - 1] + 1$
- $dp[i][j] = dp[i - 1][j] + 1$

可以看出 $dp[i][j]$ 是依赖左方，上方和左上方元素的，如图：



D
公众号：代码随想录

所以在dp矩阵中一定是从左到右从上到下去遍历。

代码如下：

```
for (int i = 1; i <= word1.size(); i++) {
    for (int j = 1; j <= word2.size(); j++) {
        if (word1[i - 1] == word2[j - 1]) {
            dp[i][j] = dp[i - 1][j - 1];
        }
        else {
            dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]}) + 1;
        }
    }
}
```

#5. 举例推导dp数组

以示例1为例，输入：word1 = "horse", word2 = "ros" 为例，dp矩阵状态图如下：

		r	o	s	
		0	1	2	3
h		1	1	2	3
o		2	2	1	2
r		3	2	2	2
s		4	3	3	2
e		5	4	4	3

公众号：代码随想录

以上动规五部分分析完毕，C++代码如下：

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        vector<vector<int>> dp(word1.size() + 1, vector<int>(word2.size() + 1, 0));

        for (int i = 0; i <= word1.size(); i++) dp[i][0] = i;
        for (int j = 0; j <= word2.size(); j++) dp[0][j] = j;
        for (int i = 1; i <= word1.size(); i++) {
            for (int j = 1; j <= word2.size(); j++) {
                if (word1[i - 1] == word2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                }
                else {
                    dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]}) + 1;
                }
            }
        }
    }
};
```

```

        }
    }
}
return dp[word1.size()][word2.size()];
}
};

```

- 时间复杂度: $O(n * m)$
- 空间复杂度: $O(n * m)$

12.最长递增子序列

动态规划

```

class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        vector<int> memo(nums.size(),1);
        int res = 0;
        for(int i = 0;i < nums.size();i++){
            for(int j = 0;j < i;j++){
                if(nums[i] > nums[j]){
                    memo[i] = max(memo[i],memo[j] + 1);
                }
            }
            res = max(res,memo[i]);
        }
        return res;
    }
};

```

贪心+二分

```

class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        vector<int> memo;
        memo.push_back(nums[0]);
        for (int i = 1; i < nums.size(); i++) {
            int left = 0;
            int right = memo.size() - 1;
            int mid = 0;
            if (nums[i] > memo[memo.size() - 1]) {
                memo.push_back(nums[i]);
            } else {
                while (left <= right) {
                    mid = (left + right) / 2;
                    if (nums[i] <= memo[mid]) {
                        right = mid - 1;
                    } else {
                        left = mid + 1;
                    }
                }
                memo[mid] = nums[i];
            }
        }
        return memo.size();
    }
};

```

```

        }
    }

    memo[left] = nums[i];
}
}
return memo.size();
}
};

```

***13.将三个组排序

状态机dp

贪心+二分

```

class Solution {
public:
    int minimumOperations(vector<int>& nums) {
        vector<int> memo;
        memo.push_back(nums[0]);
        for(int i = 1; i < nums.size(); i++){
            int left = 0;
            int right = memo.size() - 1;
            if(nums[i] >= memo[memo.size() - 1]){
                memo.push_back(nums[i]);
            } else {
                while(left <= right){
                    int mid = (left + right) / 2;
                    if(memo[mid] <= nums[i]){
                        left = mid + 1;
                    } else {
                        right = mid - 1;
                    }
                }
                memo[left] = nums[i];
            }
        }
        return nums.size() - memo.size();
    }
};

```

注意二分查找的区别

`upper_bound` 和 `lower_bound` 的主要区别在于：

- `lower_bound(nums, target)`: 返回 **第一个** `>= target` 的元素索引（即**大于等于 target**的最左侧位置）。
- `upper_bound(nums, target)`: 返回 **第一个** `> target` 的元素索引（即**严格大于 target**的最左侧位置）。

```

int lower_bound(vector<int>& nums, int target) {
    int left = 0, right = (int) nums.size() - 1; // 闭区间 [left, right]
    while (left <= right) { // 区间不为空
        // 循环不变量:
        // nums[left-1] < target
        // nums[right+1] >= target
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1; // 范围缩小到 [mid+1, right]
        } else {
            right = mid - 1; // 范围缩小到 [left, mid-1]
        }
    }
    return left;
}

int upper_bound(vector<int>& nums, int target) {
    int left = 0, right = (int) nums.size() - 1; // 闭区间 [left, right]
    while (left <= right) { // 区间不为空
        int mid = left + (right - left) / 2;
        if (nums[mid] <= target) { // 注意这里是 <=
            left = mid + 1; // 继续查找右侧 [mid+1, right]
        } else {
            right = mid - 1; // 继续查找左侧 [left, mid-1]
        }
    }
    return left;
}

```

二十三、优先队列(堆)

1.使用服务器处理任务

N、妙数组

4.数字分组求偶数和

小M面对一组从 1 到 9 的数字，这些数字被分成多个小组，并从每个小组中选择一个数字组成一个新的数。目标是使得这个新数的各位数字之和为偶数。任务是计算出有多少种不同的分组和选择方法可以达到这一目标。

- `numbers`: 一个由多个整数字符串组成的列表，每个字符串可以视为一个数字组。小M需要从每个数字组中选择一个数字。

例如对于 [123, 456, 789]，14个符合条件的数为：147 149 158 167 169 248 257 259 268 347 349 358 367 369。

```

int solution(std::vector<int> numbers) {
    // Please write your code here
    int count_ou = 1;
}

```

```

int count_ji = 0;

int m = numbers.size();
for(int i = 0; i < m; i++){
    int temp = numbers[i];
    int group_ou = 0;
    int group_ji = 0;

    while(temp > 0){
        int tep = temp % 10;
        if(tep % 2 == 0){
            group_ou++;
        } else {
            group_ji++;
        }
        temp /= 10;
    }

    int ji = count_ou * group_ji + count_ji * group_ou;
    int ou = count_ou * group_ou + count_ji * group_ji;
    count_ji = ji;
    count_ou = ou;
}

return count_ou;
}

```

如果我们用 n 表示所有数字的位数的总和（即所有数字组中数字的总位数），那么时间复杂度可以简化为：

$O(n)$

5.寻找最大葫芦

在一场经典的德州扑克游戏中，有一种牌型叫做“葫芦”。“葫芦”由五张牌组成，其中包括三张相同牌面值的牌 aa 和另外两张相同牌面值的牌 bb 。如果两个人同时拥有“葫芦”，我们会优先比较牌 aa 的大小，若牌 aa 相同则再比较牌 bb 的大小，牌面值的大小规则为：1 (A) > K > Q > J > 10 > 9 > ... > 2，其中 1 (A) 的牌面值为1，K 为13，依此类推。

在这个问题中，我们对“葫芦”增加了一个限制：组成“葫芦”的五张牌牌面值之和不能超过给定的最大值 $maxma^{**}x$ 。

给定一组牌，你需要找到符合规则的最大的“葫芦”组合，并输出其中三张相同的牌面和两张相同的牌面。如果找不到符合条件的“葫芦”，则输出 “0, 0”。

测试样例

样例1：

输入：n = 9, max = 34, array = [6, 6, 6, 8, 8, 8, 5, 5, 1]

输出：[8, 5]

说明：array 数组中可组成4个葫芦，分别为[6,6,6,8,8],[6,6,6,5,5],[8,8,8,6,6],[8,8,8,5,5]。其中[8,8,8,6,6]的牌面值为36，大于34不符合要求。剩下的3个葫芦的大小关系为[8,8,8,5,5]>[6,6,6,8,8]>[6,6,6,5,5],故返回[8,5]

不难但麻烦，考验基本功

```
std::vector<int> solution(int n, int max, const std::vector<int>& array) {
    // Edit your code here
    std::unordered_map<int, int> count_map;
    for(auto a : array){
        count_map[a]++;
    }

    std::vector<int> sec = {};
    std::vector<int> third = {};

    for(auto a : count_map){
        if(a.second >=3) third.push_back(a.first);
        if(a.second >=2) sec.push_back(a.first);
    }

    sort(sec.rbegin(),sec.rend());
    sort(third.rbegin(),third.rend());
    int i =third.size();
    int j = sec.size();
    if(i == 0 || j == 0) return {0,0};
    if(sec[sec.size()-1] == 1) {
        sec.insert(sec.begin(),1);
        sec.erase(sec.end());
    }
    if(third[third.size()-1] == 1){
        third.insert(third.begin(),1);
        third.erase(third.end());
    }

    for(int m = 0; m < i;m++){
        for(int k = 0; k < j;k++){
            if(third[m] == sec[k]) continue;
            int sum = third[m] * 3 + sec[k] * 2;
            if(sum <= max) return {third[m],sec[k]};
        }
    }

    return {0, 0};
}
```

15.最少前缀操作

169.多数元素

哈希表统计法：遍历数组 nums，用 HashMap 统计各数字的数量，即可找出众数。此方法时间和空间复杂度均为 $O(N)$ 。

数组排序法：将数组 nums 排序，数组中点的元素一定为众数。

摩尔投票法：核心理念为 票数正负抵消。此方法时间和空间复杂度分别为 $O(N)$ 和 $O(1)$ ，为本题的最

佳解法。

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int x = 0, votes = 0;
        for (int num : nums){
            if (votes == 0) x = num;
            votes += num == x ? 1 : -1;
        }
        return x;
    }
};
```

124.有限制的楼梯攀登

问题

全局变量：可以在整个程序中直接访问，不需要传参。

局部变量：只能在定义它的函数或代码块内使用，若需在其他函数中使用，需要通过传参或引用/指针传递。

类中的数据成员：

- **公共数据成员：**可以在类的任何成员函数中直接访问，不需要传参。
- **私有数据成员：**只能通过类的成员函数访问，而不能在类的外部访问，需要通过公有或私有成员函数访问，若需在其他函数中使用，仍然需要通过传参或引用/指针传递。

手撕 池组件 智能指针 stl容器等 网络编程
锁 各种锁乐观读写等 运算符重载 设计模式
lambda 大小根堆 排序

手撕题

1.三个线程交替打印abc

```
#include<iostream>
#include<thread>
```

```

#include<condition_variable>
#include<mutex>
using namespace std;
mutex mtx;
condition_variable cv;
int flag = 0;
void PrintABC(char ch,int order){
    while(1){
        unique_lock<mutex> lock(mtx);
        cv.wait(lock,[order]{return flag == order;});
        cout<<ch;
        flag = (flag+1)%3;
        cv.notify_all();
    }
}
int main(){
    thread t1(PrintABC,'A',0);
    thread t2(PrintABC,'B',1);
    thread t3(PrintABC,'C',2);
    t1.join();
    t2.join();
    t3.join();
    return 0;
}

```

2.设计一个定时任务调用类

单次添加一个任务

```

#include <iostream>
#include <thread>
#include <functional>
#include <atomic>
#include <chrono>

class TimerTask {
public:
    TimerTask() : running(false) {}

    // 启动定时任务
    void start(std::function<void()> task, int interval_ms) {
        if (running.load()) return; // 避免重复启动
        running.store(true);
        worker = std::thread([this, task, interval_ms]() {
            while (running.load()) {
                auto nextRun = std::chrono::steady_clock::now() +
std::chrono::milliseconds(interval_ms);
                task(); // 执行任务
                std::this_thread::sleep_until(nextRun); // 等待到下次执行时间点
            }
        });
    }
}

```

```

// 停止任务
void stop() {
    running.store(false);
    if (worker.joinable()) {
        worker.join();
    }
}

~TimerTask() {
    stop(); // 确保退出时释放资源
}

private:
    std::thread worker;
    std::atomic<bool> running;
};

void exampleTask() {
    std::cout << "定时任务执行: " <<
std::chrono::system_clock::now().time_since_epoch().count() << std::endl;
}

int main() {
    TimerTask timer;
    timer.start(exampleTask, 1000); // 每 1000ms 执行一次任务

    std::this_thread::sleep_for(std::chrono::seconds(5)); // 让主线程等一会
    timer.stop(); // 停止任务

    return 0;
}

```

可同时添加多个任务

```

#include <iostream>
#include <thread>
#include <functional>
#include <unordered_map>
#include <atomic>
#include <chrono>
#include <mutex>

class TimersScheduler {
public:
    TimersScheduler() : running(true) {}

    // 添加定时任务, 返回任务 ID
    int addTask(std::function<void()> task, int interval_ms) {
        std::lock_guard<std::mutex> lock(mtx);
        int taskId = nextTaskId++;
        workers[taskId] = std::thread([this, taskId, task, interval_ms]() {
            while (running.load() && taskRunning[taskId]) {
                auto nextRun = std::chrono::steady_clock::now() +
std::chrono::milliseconds(interval_ms);

```

```

        task(); // 执行任务
        std::this_thread::sleep_until(nextRun); // 等待到下次执行时间点
    }
});
taskRunning[taskId] = true;
return taskId;
}

// 移除定时任务
void removeTask(int taskId) {
    std::lock_guard<std::mutex> lock(mtx);
    if (workers.find(taskId) != workers.end()) {
        taskRunning[taskId] = false;
        if (workers[taskId].joinable()) {
            workers[taskId].join();
        }
        workers.erase(taskId);
        taskRunning.erase(taskId);
    }
}

// 关闭所有任务
void stopAll() {
    running.store(false);
    for (auto& [taskId, thread] : workers) {
        taskRunning[taskId] = false;
        if (thread.joinable()) {
            thread.join();
        }
    }
    workers.clear();
    taskRunning.clear();
}

~TimerScheduler() {
    stopAll(); // 确保退出时清理所有线程
}

private:
    std::unordered_map<int, std::thread> workers;
    std::unordered_map<int, std::atomic<bool>> taskRunning;
    std::atomic<bool> running;
    std::mutex mtx;
    int nextTaskId = 0;
};

// 示例任务
void taskA() {
    std::cout << "Task A running at " <<
std::chrono::system_clock::now().time_since_epoch().count() << std::endl;
}

void taskB() {
    std::cout << "Task B running at " <<
std::chrono::system_clock::now().time_since_epoch().count() << std::endl;
}

```

```

int main() {
    TimerScheduler scheduler;

    int taskA_id = scheduler.addTask(taskA, 1000); // 每 1 秒执行一次
    int taskB_id = scheduler.addTask(taskB, 2000); // 每 2 秒执行一次

    std::this_thread::sleep_for(std::chrono::seconds(6)); // 让任务跑一段时间

    scheduler.removeTask(taskA_id); // 移除 Task A
    std::this_thread::sleep_for(std::chrono::seconds(4)); // 让 Task B 继续执行

    scheduler.stopAll(); // 停止所有任务
    return 0;
}

```

3.c++模拟rpc调用

4.c++模拟服务发现

5.c++模拟负载均衡

6.手写shared_ptr

```

#include <iostream>
#include <atomic>

template <typename T>
class SharedPtr {
private:
    T* ptr; // 资源指针
    std::atomic<int>* ref_count; // 引用计数

public:
    // ✅ 构造函数
    explicit SharedPtr(T* p = nullptr) : ptr(p), ref_count(new std::atomic<int>(p
? 1 : 0)) {}

    // ✅ 拷贝构造（增加引用计数）
    SharedPtr(const SharedPtr& other) : ptr(other.ptr),
ref_count(other.ref_count) {
        if (ptr) (*ref_count)++;
    }

    // ✅ 移动构造（转移所有权）
    SharedPtr(SharedPtr&& other) noexcept : ptr(other.ptr),
ref_count(other.ref_count) {
        other.ptr = nullptr;
        other.ref_count = nullptr;
    }
}

```

```

// ✔️ 赋值运算符（拷贝）
SharedPtr& operator=(const SharedPtr& other) {
    if (this != &other) {
        release();
        ptr = other.ptr;
        ref_count = other.ref_count;
        if (ptr) (*ref_count)++;
    }
    return *this;
}

// ✔️ 赋值运算符（移动）
SharedPtr& operator=(SharedPtr&& other) noexcept {
    if (this != &other) {
        release();
        ptr = other.ptr;
        ref_count = other.ref_count;
        other.ptr = nullptr;
        other.ref_count = nullptr;
    }
    return *this;
}

// ✔️ 析构函数（减少引用计数，必要时释放资源）
~SharedPtr() {
    release();
}

// ✔️ 释放资源（减少引用计数）
void release() {
    if (ptr && --(*ref_count) == 0) {
        delete ptr;
        delete ref_count;
    }
    ptr = nullptr;
    ref_count = nullptr;
}

// ✔️ 获取原始指针
T* get() const { return ptr; }


// ✔️ 解引用
T& operator*() const { return *ptr; }
T* operator->() const { return ptr; }

// ✔️ 获取引用计数
int use_count() const { return ptr ? *ref_count : 0; }

// ✔️ 判断是否唯一
bool unique() const { return use_count() == 1; }

// ✔️ 显式重置
void reset(T* p = nullptr) {
    release();
    ptr = p;
    ref_count = new std::atomic<int>(p ? 1 : 0);
}

```

```
    }  
};  
  
//  辅助函数 `make_shared`  
template <typename T, typename... Args>  
SharedPtr<T> make_shared(Args&&... args) {  
    return SharedPtr<T>(new T(std::forward<Args>(args)...));  
}
```

7.手写堆排序或堆操作(大小根堆)

8.各种排序 主要快速排序

9.手写lambda表达式
