

数据结构与算法

Quaternijkon







数据结构与算法刷题本

July 02, 2024

主要来源于力扣

目录

1. 数据结构	5
1.1. 数组	6
1.1.1. □ 数组中的重复元素 🔗	7
1.1.2. □ 轮转数组 🔗	8
1.2. 链表	10
1.2.1. □ 反转链表 🔗	11
1.3. 栈与队列	13
1.3.1. □ 中位数 🔗	14
1.4. 字符串	17
1.4.1. □ 最长回文子串 🔗	18
1.4.2. □ 轮转字符串 🔗	23
1.5. 哈希表	24
1.5.1. □ 同构字符串 🔗	25
1.5.2. □ 两数之和 🔗	27
1.6. 堆	29
1.6.1. □ 数组中的第 K 个最大元素 🔗	30
1.7. 树	33
1.7.1. □ 子结构判断 🔗	34
1.8. 图	37
1.9. 并查集	39
2. 算法	40
2.1. 二分查找	42
2.1.1. □ 寻找旋转排序数组中的最小值 🔗	43
2.2. 排序算法	46
2.3. 滑动窗口与双指针	48
2.3.1. □ 有序数组两数之和 🔗	49
2.3.2. □ 删除有序数组中的重复项 🔗	52
2.3.3. □ 移动零 🔗	54
2.4. 递归	56
2.5. 分治	58
2.5.1. □ 逆序对 🔗	59
2.6. 广度优先搜索	61
2.6.1. □ 岛屿数量 🔗	62
2.7. 深度优先搜索	67
2.7.1. □ 字母迷宫 🔗	68
2.8. 贪心算法	71
2.8.1. □ 找字符串拼接的最大值 🔗	72
2.9. 动态规划	75
2.9.1. □ 丑数 🔗	76
2.9.2. □ 买卖股票的最佳时机 II 🔗	78

2.10. 最短路径	81
2.11. 最小生成树	82
2.12. 前缀和	83
2.13. 数学	85
2.13.1.  回文数 	86
2.13.2.  只出现一次的数字 	89
2.13.3.  旋转图像 	91
3. 题集	93

1. 数据结构

数据结构是为实现对计算机数据有效使用的各种数据组织形式，服务于各类计算机操作。不同的数据结构具有各自对应的适用场景，旨在降低各种算法计算的时间与空间复杂度，达到最佳的任务执行效率。

1.1. 数组

” 书内链接

- 小节 1.4.2 —— 轮转字符串

1.1.1. 数组中的重复元素

寻找文件副本

设备中存有 n 个文件，文件 id 记于数组 `documents`。若文件 id 相同，则定义为该文件存在副本。请返回任一存在副本的文件 id。

示例 1

输入：`documents = [2, 5, 3, 0, 5, 0]`

输出：0 或 5

提示

$0 \leq \text{documents}[i] \leq n-1$

$2 \leq n \leq 100000$

方法 1: 遍历

遍历中，第一次遇到数字 x 时，将其交换至索引 x 处；而当第二次遇到数字 x 时，一定有 `documents[x]=x`，此时即可得到一组重复数字。

```
1  class Solution {
2  public:
3      int findRepeatDocument(vector<int>& documents) {
4          int i = 0;
5          while(i < documents.size()) {
6              if(documents[i] == i) {
7                  i++;
8                  continue;
9              }
10             if(documents[documents[i]] == documents[i])
11                 return documents[i];
12             swap(documents[i],documents[documents[i]]);
13         }
14         return -1;
15     }
16 };
```

1.1.2. 轮转数组

? 轮转数组

给定一个整数数组 `nums`，将数组中的元素向右轮转 `k` 个位置，其中 `k` 是非负数。

示例 1

输入: `nums = [1,2,3,4,5,6,7]`, `k = 3`

输出: `[5,6,7,1,2,3,4]`

解释:

- 向右轮转 1 步: `[7,1,2,3,4,5,6]`
- 向右轮转 2 步: `[6,7,1,2,3,4,5]`
- 向右轮转 3 步: `[5,6,7,1,2,3,4]`

示例 2

输入: `nums = [-1,-100,3,99]`, `k = 2`

输出: `[3,99,-1,-100]`

解释:

- 向右轮转 1 步: `[99,-1,-100,3]`
- 向右轮转 2 步: `[3,99,-1,-100]`

提示

- $1 \leq \text{nums.length} \leq 10^5$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- $0 \leq k \leq 10^5$

方法 1: 富有线性代数的美

我们需要将 `ab` 变成 `ba`，可以首先将 `ab` 翻转得到 $b^{-1}a^{-1}$ ，然后将 $b^{-1}a^{-1}$ 翻转得到 `ba`。

- $(ab)^{-1} = b^{-1}a^{-1}$
- $(b^{-1})^{-1} = b$
- $(a^{-1})^{-1} = a$

```
1 class Solution {  
2     public:  
3         void rotate(vector<int>& nums, int k) {  
4             reverse(nums.begin(), nums.end());  
5             reverse(nums.begin(), nums.begin() + k % nums.size());  
6             reverse(nums.begin() + k % nums.size(), nums.end());  
}
```

cpp


```
7     }
```

```
8 };
```

1.2. 链表

1.2.1. 反转链表

反转链表

给你单链表的头节点 head，请你反转链表，并返回反转后的链表。

示例 1

输入：head = [1,2,3,4,5]

输出：[5,4,3,2,1]

示例 2

输入：head = [1,2]

输出：[2,1]

示例 3

输入：head = []

输出：[]

提示

链表中节点的数目范围是 [0, 5000]

$-5000 \leq \text{Node.val} \leq 5000$

方法 1: 迭代

考虑遍历链表，并在访问各节点时修改 next 引用指向，算法流程见注释。

```
1  class Solution {
2  public:
3      ListNode* reverseList(ListNode* head) {
4          ListNode *cur = head, *pre = nullptr;
5          while(cur != nullptr) {
6              ListNode* tmp = cur->next; // 暂存后继节点 cur.next
7              cur->next = pre;           // 修改 next 引用指向
8              pre = cur;                 // pre 暂存 cur
9              cur = tmp;                 // cur 访问下一节点
10         }
11         return pre;
12     }
13 };
```

💡 方法 2: 递归

考虑使用递归法遍历链表，当越过尾节点后终止递归，在回溯时修改各节点的 `next` 引用指向。

`recur(cur, pre)` 递归函数：

1. 终止条件：当 `cur` 为空，则返回尾节点 `pre`（即反转链表的头节点）；
2. 递归后继节点，记录返回值（即反转链表的头节点）为 `res`；
3. 修改当前节点 `cur` 引用指向前驱节点 `pre`；
4. 返回反转链表的头节点 `res`；

`reverseList(head)` 函数：

调用并返回 `recur(head, null)`。传入 `null` 是因为反转链表后，`head` 节点指向 `null`；

```
1 class Solution {
2 public:
3     ListNode* reverseList(ListNode* head) {
4         return recur(head, nullptr); // 调用递归并返回
5     }
6 private:
7     ListNode* recur(ListNode* cur, ListNode* pre) {
8         if (cur == nullptr) return pre; // 终止条件
9         ListNode* res = recur(cur->next, cur); // 递归后继节点
10        cur->next = pre; // 修改节点引用指向
11        return res; // 返回反转链表的头节点
12    }
13 };
```

cpp

1.3. 栈与队列

1.3.1. 中位数

数据流中的中位数

中位数是有序整数列表中的中间值。如果列表的大小是偶数，则没有中间值，中位数是两个中间值的平均值。

例如，

- $[2,3,4]$ 的中位数是 3
- $[2,3]$ 的中位数是 $\frac{2+3}{2} = 2.5$

设计一个支持以下两种操作的数据结构：

- `void addNum(int num)` - 从数据流中添加一个整数到数据结构中。
- `double findMedian()` - 返回目前所有元素的中位数。

示例 1

输入：["MedianFinder","addNum","addNum","findMedian","addNum","findMedian"]

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

输出：[null,null,null,1.50000,null,2.00000]

示例 2

输入：["MedianFinder","addNum","addNum","findMedian","addNum","findMedian"]

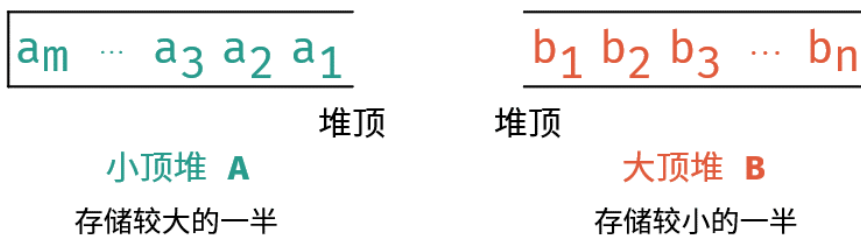
[[],[1],[2],[],[3],[]]

输出：[null,null,null,1.50000,null,2.00000]

提示

最多会对 `addNum`、`findMedian` 进行 50000 次调用。

方法 1: 堆



设共有 $N = m + n$ 个元素，规定添加元素时保证：

$$\begin{cases} m = n + 1 = \frac{N + 1}{2}, & N \text{ 为奇数} \\ m = n = \frac{N}{2}, & N \text{ 为偶数} \end{cases}$$



$$\text{中位数} = \begin{cases} a_1, & m \neq n \\ (a_1 + b_1) / 2, & m = n \end{cases}$$

图 1 堆

`addNum(num)` 函数：

- 当 $m = n$ （即 N 为偶数）：需向 A 添加一个元素。实现方法：将新元素 `num` 插入至 B ，再将 B 堆顶元素插入至 A ；
- 当 $m \neq n$ （即 N 为奇数）：需向 B 添加一个元素。实现方法：将新元素 `num` 插入至 A ，再将 A 堆顶元素插入至 B ；

`findMedian()` 函数：

- 当 $m = n$ （ N 为偶数）：则中位数为 $(A \text{ 的堆顶元素} + B \text{ 的堆顶元素}) / 2$ 。
- 当 $m \neq n$ （ N 为奇数）：则中位数为 A 的堆顶元素。

```

1  class MedianFinder {
2  public:
3      priority_queue<int, vector<int>, greater<int>> A; // 小顶堆，保存较大的一半
4      priority_queue<int, vector<int>, less<int>> B; // 大顶堆，保存较小的一半
5      MedianFinder() { }
6      void addNum(int num) {
7          if(A.size() != B.size()) {
8              A.push(num);
9              B.push(A.top());
10             A.pop();
11         } else {

```

```
12         B.push(num);
13         A.push(B.top());
14         B.pop();
15     }
16 }
17 double findMedian() {
18     return A.size() != B.size() ? A.top() : (A.top() + B.top()) / 2.0;
19 }
20 };
```


1.4. 字符串

” 书内链接

- 小节 1.1.2 —— 轮转数组

1.4.1. 最长回文子串

最长回文子串

给你一个字符串 s ，找到 s 中最长的回文子串。

示例 1

输入： $s = \text{"babad"}$

输出： "bab"

解释： "aba" 同样是符合题意的答案。

示例 2

输入： $s = \text{"cbbd"}$

输出： "bb"

提示

$1 \leq s.length \leq 1000$

s 仅由数字和英文字母组成

方法 1: 动态规划

对于一个子串而言，如果它是回文串，并且长度大于 2，那么将它首尾的两个字母去除之后，它仍然是个回文串。例如对于字符串 "ababa" ，如果我们已经知道 "bab" 是回文串，那么 "ababa" 一定是回文串，这是因为它的首尾两个字母都是 "a" 。

根据这样的思路，我们就可以用动态规划的方法解决本题。我们用 $P(i, j)$ 表示字符串 s 的第 i 到 j 个字母组成的串（下文表示成 $s[i:j]$ ）是否为回文串：

$$\begin{cases} P(i, j) = \text{true} & \text{如果子串 } S_i \dots S_j \text{ 是回文串} \\ P(i, j) = \text{false} & \text{其它情况} \end{cases} \quad (1)$$

这里的「其它情况」包含两种可能性：

- $s[i, j]$ 本身不是一个回文串；
- $i > j$ ，此时 $s[i, j]$ 本身不合法。

那么我们就可以写出动态规划的状态转移方程：

$$P(i, j) = P(i + 1, j - 1) \wedge (S_i == S_j) \quad (2)$$

也就是说，只有 $s[i+1:j-1]$ 是回文串，并且 s 的第 i 和 j 个字母相同时， $s[i:j]$ 才会是回文串。

上文的所有讨论是建立在子串长度大于 2 的前提之上的，我们还需要考虑动态规划中的边界条件，即子串的长度为 1 或 2。对于长度为 1 的子串，它显然是个回文串；对于长度为 2 的子串，只要它的两个字母相同，它就是一个回文串。因此我们就可以写出动态规划的边界条件：

$$\begin{cases} P(i, i) = \text{true} \\ P(i, i+1) = (S_i == S_{i+1}) \end{cases} \quad (3)$$

根据这个思路，我们就可以完成动态规划了，最终的答案即为所有 $P(i, j) = \text{true}$ 中 $j - i + 1$ （即子串长度）的最大值。注意：在状态转移方程中，我们是从长度较短的字符串向长度较长的字符串进行转移的，因此一定要注意动态规划的循环顺序。

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4
5  using namespace std;
6
7  class Solution {
8  public:
9      string longestPalindrome(string s) {
10         int n = s.size();
11         if (n < 2) {
12             return s;
13         }
14
15         int maxLen = 1;
16         int begin = 0;
17         // dp[i][j] 表示 s[i..j] 是否是回文串
18         vector<vector<int>> dp(n, vector<int>(n));
19         // 初始化：所有长度为 1 的子串都是回文串
20         for (int i = 0; i < n; i++) {
21             dp[i][i] = true;
22         }
23         // 递推开始
24         // 先枚举子串长度
25         for (int L = 2; L ≤ n; L++) {
26             // 枚举左边界，左边界的上限设置可以宽松一些
27             for (int i = 0; i < n; i++) {
28                 // 由 L 和 i 可以确定右边界，即 j - i + 1 = L 得
29                 int j = L + i - 1;
30                 // 如果右边界越界，就可以退出当前循环
31                 if (j ≥ n) {
32                     break;
33                 }
34
35                 if (s[i] ≠ s[j]) {
36                     dp[i][j] = false;
37                 } else {
```

```

38         if (j - i < 3) {
39             dp[i][j] = true;
40         } else {
41             dp[i][j] = dp[i + 1][j - 1];
42         }
43     }
44
45     // 只要 dp[i][L] == true 成立，就表示子串 s[i..L] 是回文，此时记录回文长度和起始
    位置
46     if (dp[i][j] && j - i + 1 > maxlen) {
47         maxlen = j - i + 1;
48         begin = i;
49     }
50 }
51 }
52 return s.substr(begin, maxlen);
53 }
54 };

```

💡 方法 2：中心扩展算法

我们仔细观察一下方法一中的状态转移方程：

$$\begin{cases} P(i, j) = \text{true} \\ P(i, i + 1) = (S_i == S_{i+1}) \\ P(i, j) = P(i + 1, j - 1) \wedge (S_i == S_j) \end{cases} \quad (4)$$

找出其中的状态转移链：

$$P(i, j) \leftarrow P(i + 1, j - 1) \leftarrow P(i + 2, j - 2) \leftarrow \dots \leftarrow \text{某一边界情况} \quad (5)$$

可以发现，**所有的状态在转移的时候的可能性都是唯一的**。也就是说，我们可以从每一种边界情况开始「扩展」，也可以得出所有的状态对应的答案。

边界情况即为子串长度为 1 或 2 的情况。我们枚举每一种边界情况，并从对应的子串开始不断地向两边扩展。如果两边的字母相同，我们就可以继续扩展，例如从 $P(i+1, j-1)$ 扩展到 $P(i, j)$ ；如果两边的字母不同，我们就可以停止扩展，因为在这之后的子串都不能是回文串了。

聪明的读者此时应该可以发现，「边界情况」对应的子串实际上就是我们「扩展」出的回文串的「回文中心」。方法二的本质即为：我们枚举所有的「回文中心」并尝试「扩展」，直到无法扩展为止，此时的回文串长度即为此「回文中心」下的最长回文串长度。我们对所有的长度求出最大值，即可得到最终的答案。

```

1  class Solution {
2  public:
3      pair<int, int> expandAroundCenter(const string& s, int left, int right) {
4          while (left >= 0 && right < s.size() && s[left] == s[right]) {
5              --left;

```

cpp

```

6         ++right;
7     }
8     return {left + 1, right - 1};
9 }
10
11 string longestPalindrome(string s) {
12     int start = 0, end = 0;
13     for (int i = 0; i < s.size(); ++i) {
14         auto [left1, right1] = expandAroundCenter(s, i, i);
15         auto [left2, right2] = expandAroundCenter(s, i, i + 1);
16         if (right1 - left1 > end - start) {
17             start = left1;
18             end = right1;
19         }
20         if (right2 - left2 > end - start) {
21             start = left2;
22             end = right2;
23         }
24     }
25     return s.substr(start, end - start + 1);
26 }
27 };

```

💡 方法 3: Manacher 算法

略

```

1  class Solution {
2  public:
3      int expand(const string& s, int left, int right) {
4          while (left ≥ 0 && right < s.size() && s[left] == s[right]) {
5              --left;
6              ++right;
7          }
8          return (right - left - 2) / 2;
9      }
10
11     string longestPalindrome(string s) {
12         int start = 0, end = -1;
13         string t = "#";
14         for (char c: s) {
15             t += c;
16             t += '#';
17         }
18         t += '#';
19         s = t;
20
21         vector<int> arm_len;

```

cpp

```

22     int right = -1, j = -1;
23     for (int i = 0; i < s.size(); ++i) {
24         int cur_arm_len;
25         if (right ≥ i) {
26             int i_sym = j * 2 - i;
27             int min_arm_len = min(arm_len[i_sym], right - i);
28             cur_arm_len = expand(s, i - min_arm_len, i + min_arm_len);
29         } else {
30             cur_arm_len = expand(s, i, i);
31         }
32         arm_len.push_back(cur_arm_len);
33         if (i + cur_arm_len > right) {
34             j = i;
35             right = i + cur_arm_len;
36         }
37         if (cur_arm_len * 2 + 1 > end - start) {
38             start = i - cur_arm_len;
39             end = i + cur_arm_len;
40         }
41     }
42
43     string ans;
44     for (int i = start; i ≤ end; ++i) {
45         if (s[i] ≠ '#') {
46             ans += s[i];
47         }
48     }
49     return ans;
50 }
51 };

```

1.4.2. 轮转字符串

动态口令

某公司门禁密码使用动态口令技术。初始密码为字符串 password，密码更新均遵循以下步骤：

- 设定一个正整数目标值 target
- 将 password 前 target 个字符按原顺序移动至字符串末尾

请返回更新后的密码字符串。

示例 1

输入: password = "s3cur1tyC0d3", target = 4

输出: "r1tyC0d3s3cu"

示例 2

输入: password = "lrloseumgh", target = 6

输出: "umghlrlose"

提示

$1 \leq \text{target} < \text{password.length} \leq 10000$

方法 1: 富有线性代数的美

我们需要将 ab 变成 ba，可以首先将 ab 翻转得到 $b^{-1}a^{-1}$ ，然后将 $b^{-1}a^{-1}$ 翻转得到 ba。

- $(ab)^{-1} = b^{-1}a^{-1}$
- $(b^{-1})^{-1} = b$
- $(a^{-1})^{-1} = a$

```
1 class Solution {  
2     public:  
3         string dynamicPassword(string password, int target) {  
4             reverse(password.begin(), password.end());  
5             reverse(password.end() - target, password.end());  
6             reverse(password.begin(), password.end() - target);  
7             return password;  
8         }  
9     };
```

cpp

1.5. 哈希表

1.5.1. 同构字符串

? 同构字符串

给定两个字符串 s 和 t ，判断它们是否是同构的。

如果 s 中的字符可以按某种映射关系替换得到 t ，那么这两个字符串是同构的。

每个出现的字符都应当映射到另一个字符，同时不改变字符的顺序。不同字符不能映射到同一个字符上，相同字符只能映射到同一个字符上，字符可以映射到自己本身。

示例 1

输入： $s = \text{"egg"}, t = \text{"add"}$

输出：true

示例 2

输入： $s = \text{"foo"}, t = \text{"bar"}$

输出：false

示例 3

输入： $s = \text{"paper"}, t = \text{"title"}$

输出：true

提示

$1 \leq s.length \leq 5 * 10^4$

$t.length == s.length$

s 和 t 由任意有效的 ASCII 字符组成

方法 1: 散列表

使用两个映射表分别记录字符串 s 和 t 中每个字符的第一次出现位置，然后检查两个字符串对应位置的字符是否具有相同的第一次出现位置，以判断是否同构。

```
1 class Solution {  
2     public:  
3         bool isIsomorphic(string s, string t) {  
4             int n=s.size();  
5             unordered_map<char,int> smap;  
6             unordered_map<char,int> tmap;  
7         }
```

```
8     for(int i=0;i<n;++i)
9         if(smap[s[i]]==0) smap[s[i]]=i+1;
10    for(int i=0;i<n;++i)
11        if(tmap[t[i]]==0) tmap[t[i]]=i+1;
12    for(int i=0;i<n;++i)
13        if(smap[s[i]]!=tmap[t[i]]) return false;
14    return true;
15    }
16 };
```

1.5.2. 两数之和

? 两数之和

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出 和为目标值 `target` 的那两个整数，并返回它们的数组下标。

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

你可以按任意顺序返回答案。

示例 1

输入: `nums = [2,7,11,15]`, `target = 9`

输出: `[0,1]`

解释: 因为 `nums[0] + nums[1] == 9`，返回 `[0, 1]`。

示例 2

输入: `nums = [3,2,4]`, `target = 6`

输出: `[1,2]`

示例 3

输入: `nums = [3,3]`, `target = 6`

输出: `[0,1]`

提示

- $2 \leq \text{nums.length} \leq 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $-10^9 \leq \text{target} \leq 10^9$
- 只会存在一个有效答案

方法 1: 哈希表

使用哈希表 (`unordered_map`) 来记录每个元素的值及其对应的索引，以便在遍历数组时能够快速查找满足条件的配对元素。

```
1 class Solution {cpp  
2 public:  
3     vector<int> twoSum(vector<int>& nums, int target) {  
4         unordered_map<int, int> hashtable;  
5         for (int i = 0; i < nums.size(); i++) {
```

```
6         auto it = hashtable.find(target - nums[i]);
7         if (it != hashtable.end()) {
8             return {it->second, i};
9         }
10        hashtable[nums[i]] = i;
11    }
12    return {};
13
14 }
15 };
```

1.6. 堆

1.6.1. 数组中的第 K 个最大元素

数组中的第 K 个最大元素

给定整数数组 `nums` 和整数 `k`，请返回数组中第 `k` 个最大的元素。

请注意，你需要找的是数组排序后的第 `k` 个最大的元素，而不是第 `k` 个不同的元素。

示例 1

输入: `[3,2,1,5,6,4]`, `k = 2`

输出: 5

示例 2

输入: `[3,2,3,1,2,4,5,5,6]`, `k = 4`

输出: 4

提示

$1 \leq k \leq \text{nums.length} \leq 10^5$

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

方法 1: 基于快速排序的选择方法

我们可以用快速排序来解决这个问题，先对原数组排序，再返回倒数第 `k` 个位置，这样平均时间复杂度是 $O(n \log n)$ ，但其实我们可以做的更快。

首先我们来回顾一下快速排序，这是一个典型的分治算法。我们对数组 `a[l...r]` 做快速排序的过程是（参考《算法导论》）：

- 分解：将数组 `a[l...r]` 「划分」成两个子数组 `a[l...q-1]`、`a[q+1...r]`，使得 `a[l...q-1]` 中的每个元素小于等于 `a[q]`，且 `a[q]` 小于等于 `a[q+1...r]` 中的每个元素。其中，计算下标 `q` 也是「划分」过程的一部分。
- 解决：通过递归调用快速排序，对子数组 `a[l...q-1]` 和 `a[q+1...r]` 进行排序。
- 合并：因为子数组都是原址排序的，所以不需要进行合并操作，`a[l...r]` 已经有序。
- 上文中提到的「划分」过程是：从子数组 `a[l...r]` 中选择任意一个元素 `x` 作为主元，调整子数组的元素使得左边的元素都小于等于它，右边的元素都大于等于它，`x` 的最终位置就是 `q`。

由此可以发现每次经过「划分」操作后，我们一定可以确定一个元素的最终位置，即 `x` 的最终位置为 `q`，并且保证 `a[l...q-1]` 中的每个元素小于等于 `a[q]`，且 `a[q]` 小于等于 `a[q+1...r]` 中的每个元素。所以只要某次划分的 `q` 为倒数第 `k` 个下标的时候，我们就已经找到了答案。我们只关心这一点，至于 `a[l...q-1]` 和 `a[q+1...r]` 是否是有序的，我们不关心。

因此我们可以改进快速排序算法来解决这个问题：在分解的过程当中，我们会对子数组进行划分，如果划分得到的 `q` 正好就是我们需要的下标，就直接返回 `a[q]`；否则，如果 `q` 比目标下标小，就

递归右子区间，否则递归左子区间。这样就可以把原来递归两个区间变成只递归一个区间，提高了时间效率。

```
1  class Solution {
2  public:
3      int quickselect(vector<int> &nums, int l, int r, int k) {
4          if (l == r)
5              return nums[k];
6          int partition = nums[l], i = l - 1, j = r + 1;
7          while (i < j) {
8              do i++; while (nums[i] < partition);
9              do j--; while (nums[j] > partition);
10             if (i < j)
11                 swap(nums[i], nums[j]);
12         }
13         if (k ≤ j) return quickselect(nums, l, j, k);
14         else return quickselect(nums, j + 1, r, k);
15     }
16
17     int findKthLargest(vector<int> &nums, int k) {
18         int n = nums.size();
19         return quickselect(nums, 0, n - 1, n - k);
20     }
21 };
```

💡 方法 2：基于堆排序的选择方法

我们也可以使用堆排序来解决这个问题——建立一个最大堆，做 $k-1$ 次删除操作后堆顶元素就是我们要找的答案。

```
1  class Solution {
2  public:
3      void maxHeapify(vector<int>& a, int i, int heapSize) {
4          int l = i * 2 + 1, r = i * 2 + 2, largest = i;
5          if (l < heapSize && a[l] > a[largest]) {
6              largest = l;
7          }
8          if (r < heapSize && a[r] > a[largest]) {
9              largest = r;
10         }
11         if (largest ≠ i) {
12             swap(a[i], a[largest]);
13             maxHeapify(a, largest, heapSize);
14         }
15     }
16
17     void buildMaxHeap(vector<int>& a, int heapSize) {
18         for (int i = heapSize / 2; i ≥ 0; --i) {
```

```
19         maxHeapify(a, i, heapSize);
20     }
21 }
22
23 int findKthLargest(vector<int>& nums, int k) {
24     int heapSize = nums.size();
25     buildMaxHeap(nums, heapSize);
26     for (int i = nums.size() - 1; i ≥ nums.size() - k + 1; --i) {
27         swap(nums[0], nums[i]);
28         --heapSize;
29         maxHeapify(nums, 0, heapSize);
30     }
31     return nums[0];
32 }
33 };
```

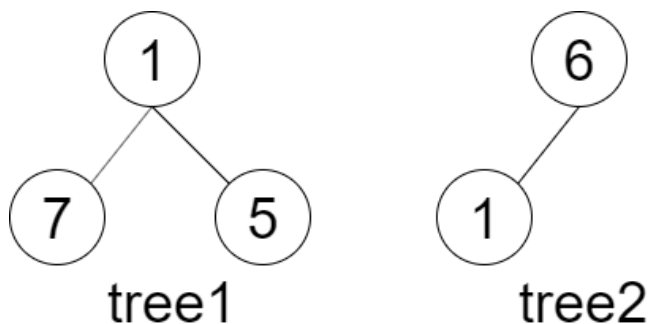

1.7. 树

1.7.1. □子结构判断 🔗

? 子结构判断

给定两棵二叉树 tree1 和 tree2，判断 tree2 是否以 tree1 的某个节点为根的子树具有相同的结构和节点值。注意，空树不会是以 tree1 的某个节点为根的子树具有相同的结构和节点值。

🌿 示例 1

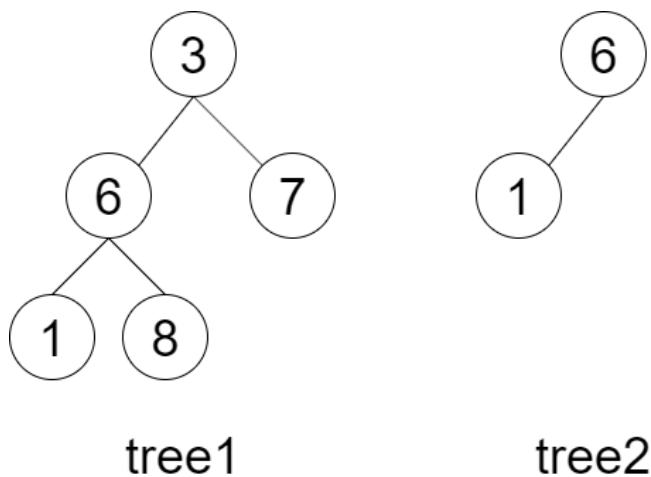


输入：tree1 = [1,7,5], tree2 = [6,1]

输出：false

解释：tree2 与 tree1 的一个子树没有相同的结构和节点值。

🌿 示例 2



输入：tree1 = [3,6,7,1,8], tree2 = [6,1]

输出：true

解释：tree2 与 tree1 的一个子树拥有相同的结构和节点值。即 6 -> 1。

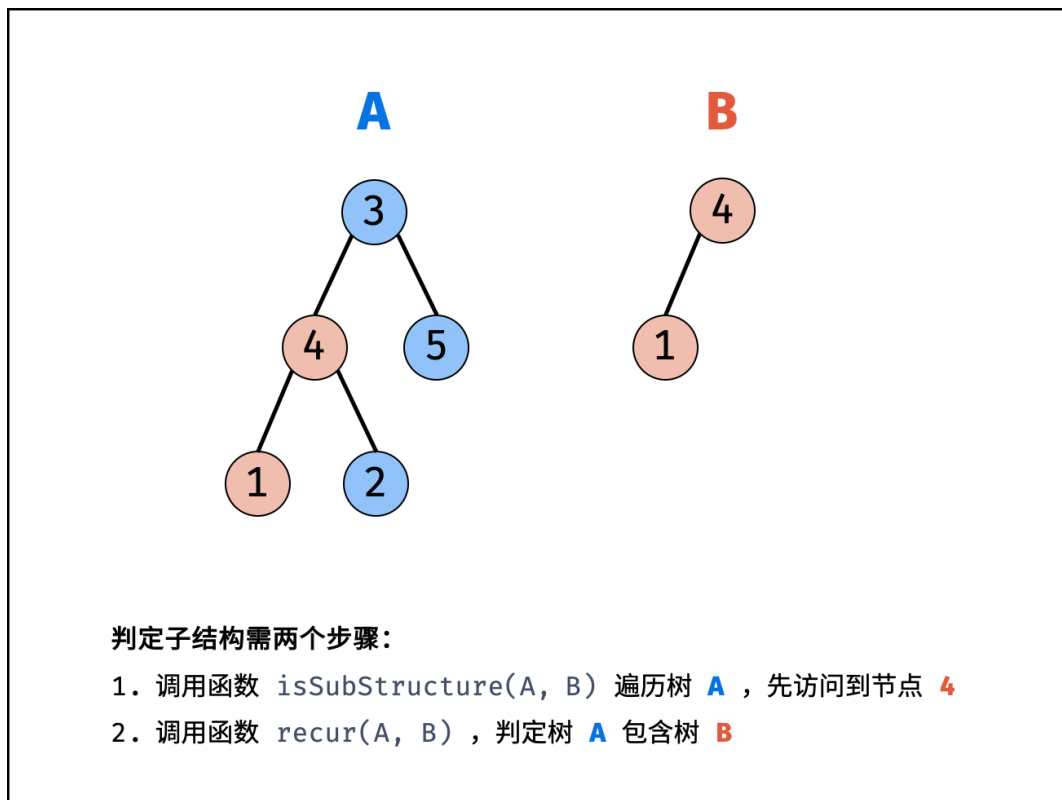
🔥 提示

0 <= 节点个数 <= 10000

方法 1: 递归遍历

若树 B 是树 A 的子结构，则子结构的根节点可能为树 A 的任意一个节点。因此，判断树 B 是否是树 A 的子结构，需完成以下两步工作：

1. 先序遍历树 A 中的每个节点 node；（对应函数 `isSubStructure(A, B)`）
2. 判断树 A 中以 node 为根节点的子树是否包含树 B。（对应函数 `recur(A, B)`）



本文名词规定：树 A 的根节点记作节点 A，树 B 的根节点称为节点 B。

`recur(A, B)` 函数：

1. 终止条件：
 1. 当节点 B 为空：说明树 B 已匹配完成（越过叶子节点），因此返回 `true`；
 2. 当节点 A 为空：说明已经越过树 A 的叶节点，即匹配失败，返回 `false`；
 3. 当节点 A 和 B 的值不同：说明匹配失败，返回 `false`；
2. 返回值：
 1. 判断 A 和 B 的左子节点是否相等，即 `recur(A.left, B.left)`；
 2. 判断 A 和 B 的右子节点是否相等，即 `recur(A.right, B.right)`；

`isSubStructure(A, B)` 函数：

1. 特例处理：当树 A 为空或树 B 为空时，直接返回 `false`；
2. 返回值：若树 B 是树 A 的子结构，则必满足以下三种情况之一，因此用或 `||` 连接：
 1. 以节点 A 为根节点的子树包含树 B，对应 `recur(A, B)`；
 2. 树 B 是树 A 左子树的子结构，对应 `isSubStructure(A.left, B)`；
 3. 树 B 是树 A 右子树的子结构，对应 `isSubStructure(A.right, B)`；

```
1 class Solution {
2     public:
3         bool isSubStructure(TreeNode* A, TreeNode* B) {
```

cpp

```

4         return (A != nullptr && B != nullptr) && (recur(A, B) || isSubStructure(A->left,
5         B) || isSubStructure(A->right, B));
6     }
7     private:
8     bool recur(TreeNode* A, TreeNode* B) {
9         if(B == nullptr) return true;
10        if(A == nullptr || A->val != B->val) return false;
11        return recur(A->left, B->left) && recur(A->right, B->right);
12    };

```

1.8. 图

1.9. 并查集

” 书内链接

- 小节 2.6.1 —— 岛屿数量

2. 算法

2.1. 二分查找

” 书内链接

- 小节 2.3.1 —— 有序数组两数之和

2.1.1. 寻找旋转排序数组中的最小值

寻找旋转排序数组中的最小值

已知一个长度为 n 的数组，预先按照升序排列，经由 1 到 n 次旋转后，得到输入数组。例如，原数组 `nums = [0,1,2,4,5,6,7]` 在变化后可能得到：

- 若旋转 4 次，则可以得到 `[4,5,6,7,0,1,2]`
- 若旋转 7 次，则可以得到 `[0,1,2,4,5,6,7]`

注意，数组 `[a[0], a[1], a[2], ..., a[n-1]]` 旋转一次的结果为数组 `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`。

给你一个元素值互不相同的数组 `nums`，它原来是一个升序排列的数组，并按上述情形进行了多次旋转。请你找出并返回数组中的最小元素。

你必须设计一个时间复杂度为 $O(\log n)$ 的算法解决此问题。

示例 1

> 输入: `nums = [3,4,5,1,2]`

> 输出: 1

> 解释: 原数组为 `[1,2,3,4,5]`，旋转 3 次得到输入数组。

示例 2

> 输入: `nums = [4,5,6,7,0,1,2]`

> 输出: 0

> 解释: 原数组为 `[0,1,2,4,5,6,7]`，旋转 3 次得到输入数组。

示例 3

> 输入: `nums = [11,13,15,17]`

> 输出: 11

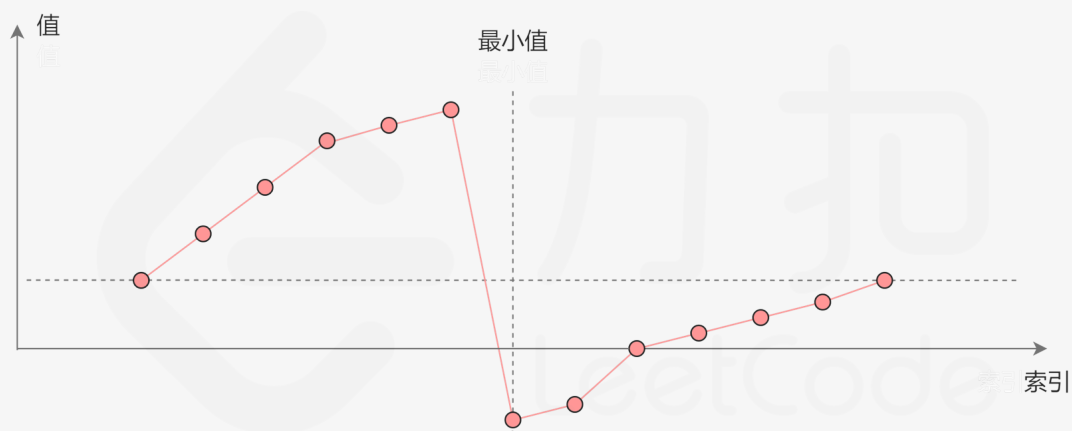
> 解释: 原数组为 `[11,13,15,17]`，旋转 4 次得到输入数组。

提示

- `N = nums.Length`
- `1 <= n <= 5000`
- `-5000 <= nums[i] <= 5000`
- `Nums` 中的所有整数互不相同
- `Nums` 原来是一个升序排序的数组，并进行了 1 至 n 次旋转

方法 1: 二分查找

一个不包含重复元素的升序数组在经过旋转之后，可以得到下面可视化的折线图：

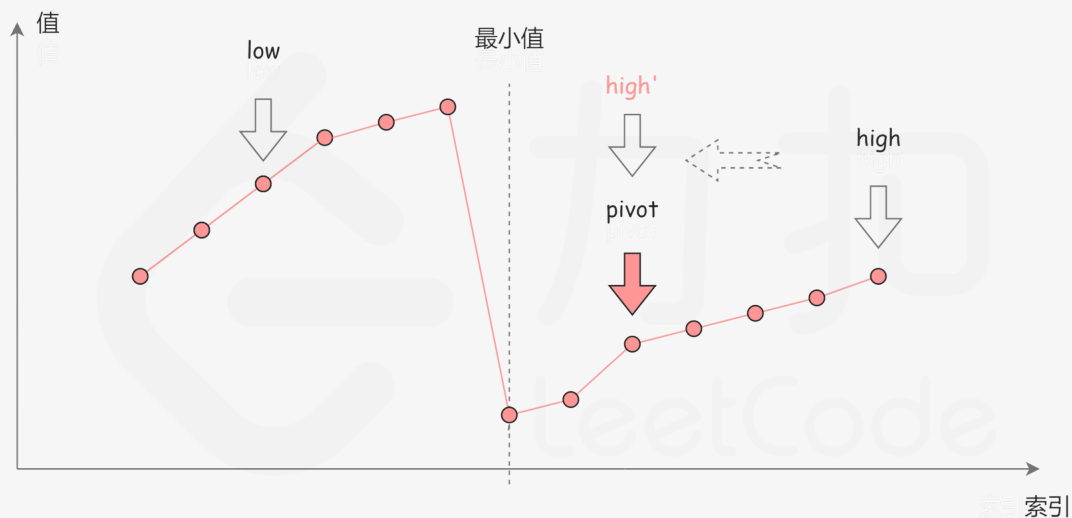


其中横轴表示数组元素的下标，纵轴表示数组元素的值。图中标出了最小值的位置，是我们需要查找的目标。

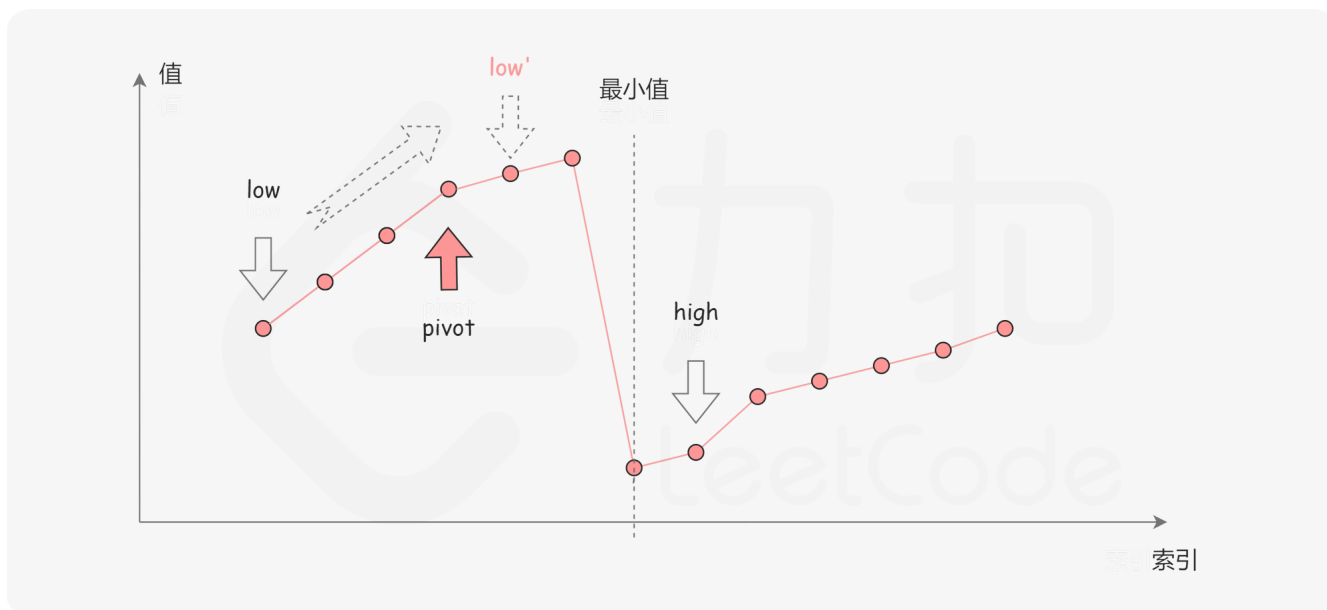
我们考虑数组中的最后一个元素 x ：在最小值右侧的元素（不包括最后一个元素本身），它们的值一定都严格小于 x ；而在最小值左侧的元素，它们的值一定都严格大于 x 。因此，我们可以根据这一条性质，通过二分查找的方法找出最小值。

在二分查找的每一步中，左边界为 low ，右边界为 $high$ ，区间的中点为 $pivot$ ，最小值就在该区间内。我们将中轴元素 $nums[pivot]$ 与右边界元素 $nums[high]$ 进行比较，可能会有以下的三种情况：

第一种情况是 $nums[pivot] < nums[high]$ 。如下图所示，这说明 $nums[pivot]$ 是最小值右侧的元素，因此我们可以忽略二分查找区间的右半部分。



第二种情况是 `nums[pivot] > nums[high]`。如下图所示，这说明 `nums[pivot]` 是最小值左侧的元素，因此我们可以忽略二分查找区间的左半部分。



由于数组不包含重复元素，并且只要当前的区间长度不为 1，`pivot` 就不会与 `high` 重合；而如果当前的区间长度为 1，这说明我们已经可以结束二分查找了。因此不会存在 `nums[pivot] = nums[high]` 的情况。

当二分查找结束时，我们就得到了最小值所在的位置。

```
1  class Solution {
2  public:
3      int findMin(vector<int>& nums) {
4          int low = 0;
5          int high = nums.size() - 1;
6          while (low < high) {
7              int pivot = low + (high - low) / 2;
8              if (nums[pivot] < nums[high]) {
9                  high = pivot;
10             }
11             else {
12                 low = pivot + 1;
13             }
14         }
15         return nums[low];
16     }
17 };
```

2.2. 排序算法

” 书内链接

- 小节 1.6.1 —— 数组中的第 K 个最大元素

2.3. 滑动窗口与双指针

2.3.1. 有序数组两数之和

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。

示例 1

输入: `numbers = [2,7,11,15]`, `target = 9`

输出: `[1,2]`

解释: 2 与 7 之和等于目标数 9。因此 `index1 = 1`, `index2 = 2`。返回 `[1, 2]`。

示例 2

输入: `numbers = [2,3,4]`, `target = 6`

输出: `[1,3]`

解释: 2 与 4 之和等于目标数 6。因此 `index1 = 1`, `index2 = 3`。返回 `[1, 3]`。

示例 3

输入: `numbers = [-1,0]`, `target = -1`

输出: `[1,2]`

解释: -1 与 0 之和等于目标数 -1。因此 `index1 = 1`, `index2 = 2`。返回 `[1, 2]`。

提示

$2 \leq \text{numbers.length} \leq 3 * 10^4$

$-1000 \leq \text{numbers}[i] \leq 1000$

`numbers` 按非递减顺序排列

$-1000 \leq \text{target} \leq 1000$

仅存在一个有效答案

方法 1: 二分查找

在数组中找到两个数,使得它们的和等于目标值,可以首先固定第一个数,然后寻找第二个数,第二个数等于目标值减去第一个数的差。利用数组的有序性质,可以通过二分查找的方法寻找第二个数。为了避免重复寻找,在寻找第二个数时,只在第一个数的右侧寻找。

```
1 class Solution {
2 public:
3     vector<int> twoSum(vector<int>& numbers, int target) {
4         for (int i = 0; i < numbers.size(); ++i) {
5             int low = i + 1, high = numbers.size() - 1;
6             while (low ≤ high) {
7                 int mid = (high - low) / 2 + low;
8                 if (numbers[mid] == target - numbers[i]) {
9                     return {i + 1, mid + 1};
10                } else if (numbers[mid] > target - numbers[i]) {
11                    high = mid - 1;
12                } else {
13                    low = mid + 1;
14                }
15            }
16        }
17        return {-1, -1};
18    }
19 };
```

💡 方法 2: 双指针

初始时两个指针分别指向第一个元素位置和最后一个元素的位置。每次计算两个指针指向的两个元素之和,并和目标值比较。如果两个元素之和等于目标值,则发现了唯一解。如果两个元素之和小于目标值,则将左侧指针右移一位。如果两个元素之和大于目标值,则将右侧指针左移一位。移动指针之后,重复上述操作,直到找到答案。

使用双指针的实质是缩小查找范围。那么会不会把可能的解过滤掉?答案是不会。假设 $\text{numbers}[i] + \text{numbers}[j] = \text{target}$ 是唯一解,其中 $0 \leq i < j \leq \text{numbers.length} - 1$ 。初始时两个指针分别指向下标 0 和下标 $\text{numbers.length} - 1$,左指针指向的下标小于或等于 i ,右指针指向的下标大于或等于 j 。除非初始时左指针和右指针已经位于下标 i 和 j ,否则一定是左指针先到达下标 i 的位置或者右指针先到达下标 j 的位置。

如果左指针先到达下标 i 的位置,此时右指针还在下标 j 的右侧, $\text{sum} > \text{target}$,因此一定是右指针左移,左指针不可能移到 i 的右侧。

如果右指针先到达下标 j 的位置,此时左指针还在下标 i 的左侧, $\text{sum} < \text{target}$,因此一定是左指针右移,右指针不可能移到 j 的左侧。

由此可见,在整个移动过程中,左指针不可能移到 i 的右侧,右指针不可能移到 j 的左侧,因此不会把可能的解过滤掉。由于题目确保有唯一的答案,因此使用双指针一定可以找到答案。

```
1 class Solution {
2 public:
3     vector<int> twoSum(vector<int>& numbers, int target) {
```

```
4     int low = 0, high = numbers.size() - 1;
5     while (low < high) {
6         int sum = numbers[low] + numbers[high];
7         if (sum == target) {
8             return {low + 1, high + 1};
9         } else if (sum < target) {
10             ++low;
11         } else {
12             --high;
13         }
14     }
15     return {-1, -1};
16 }
17 };
```

2.3.2. 删除有序数组中的重复项

删除有序数组中的重复项

给你一个非严格递增排列的数组 `nums`，请你原地删除重复出现的元素，使每个元素只出现一次，返回删除后数组的新长度。元素的相对顺序应该保持一致。然后返回 `nums` 中唯一元素的个数。

考虑 `nums` 的唯一元素的数量为 k ，你需要做以下事情确保你的题解可以被通过：

- 更改数组 `nums`，使 `nums` 的前 k 个元素包含唯一元素，并按照它们最初在 `nums` 中出现的顺序排列。`nums` 的其余元素与 `nums` 的大小不重要。
- 返回 k 。

示例 1

输入：`nums = [1,1,2]`

输出：`2, nums = [1,2,_]`

解释：函数应该返回新的长度 2，并且原数组 `nums` 的前两个元素被修改为 1, 2。不需要考虑数组中超出新长度后面的元素。

示例 2

输入：`nums = [0,0,1,1,1,2,2,3,3,4]`

输出：`5, nums = [0,1,2,3,4]`

解释：函数应该返回新的长度 5，并且原数组 `nums` 的前五个元素被修改为 0, 1, 2, 3, 4。不需要考虑数组中超出新长度后面的元素。

提示

$1 \leq \text{nums.length} \leq 3 \times 10^4$

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

`nums` 已按非严格递增排列

方法 1: 双指针

这道题目的要求是：对给定的有序数组 `nums` 删除重复元素，在删除重复元素之后，每个元素只出现一次，并返回新的长度，上述操作必须通过原地修改数组的方法，使用 $O(1)$ 的空间复杂度完成。

由于给定的数组 `nums` 是有序的，因此对于任意 $i < j$ ，如果 `nums[i] = nums[j]`，则对于任意 $i \leq k \leq j$ ，必有 `nums[i] = nums[k] = nums[j]`，即相等的元素在数组中的下标一定是连续的。利用数组有序的特点，可以通过双指针的方法删除重复元素。

如果数组 `nums` 的长度为 0，则数组不包含任何元素，因此返回 0。

当数组 `nums` 的长度大于 0 时，数组中至少包含一个元素，在删除重复元素之后也至少剩下一个元素，因此 `nums[0]` 保持原状即可，从下标 1 开始删除重复元素。

定义两个指针 `i` 和 `j` 分别为慢指针和快指针，快指针表示遍历数组到达的下标位置，慢指针表示下一个不同元素要填入的下标位置，初始时慢指针指向下标 0，快指针指向下标 1。

假设数组 `nums` 的长度为 `n`。将快指针 `j` 依次遍历从 1 到 `n-1` 的每个位置，对于每个位置，如果 `nums[j]` 不等于 `nums[i]`，说明 `nums[j]` 和之前的元素都不同，因此将 `nums[j]` 的值复制到 `nums[i+1]` 的位置，并将 `i` 的值加 1，即指向下一个位置。

遍历结束之后，从 `nums[0]` 到 `nums[i]` 的每个元素都不相同且包含原数组中的每个不同的元素，因此新的长度即为 `i+1`，返回 `i+1` 即可。

```
1  class Solution {
2      public:
3          int removeDuplicates(vector<int>& nums) {
4              int i,j,n = nums.size();
5              if(n == 0) return 0;
6              for(i = 0, j = 1; j < n; j++){ //j 不断向后寻找
7                  if(nums[i] != nums[j]){ //i 和 j 各自后移一位
8                      nums[++i] = nums[j]; //如果 j=i+1,相当于什么都没有做。
9                  }
10             }
11             return i+1;
12         }
13     }
```

cpp

2.3.3. 移动零

? 移动零

给定一个数组 `nums`，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。
请注意，必须在不复制数组的情况下原地对数组进行操作。

示例 1

输入: `nums = [0,1,0,3,12]`

输出: `[1,3,12,0,0]`

示例 2

输入: `nums = [0]`

输出: `[0]`

提示

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

$-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

方法 1: 双指针

我们使用两个指针 `left` 和 `right` 来遍历数组：

- `right` 指针用于遍历数组的每个元素。
- `left` 指针用于记录非零元素的位置。

算法步骤：

1. 初始化两个指针 `left` 和 `right`，都指向数组的起始位置。
2. 遍历数组，直到 `right` 指针到达数组的末尾：
 - 如果 `nums[right]` 不是零，就将 `nums[left]` 和 `nums[right]` 交换，并将 `left` 指针右移一位。
 - 不管 `nums[right]` 是否为零，`right` 指针都右移一位。
3. 继续这个过程，直到 `right` 指针遍历完数组。

```
1 class Solution {  
2     public:  
3         void moveZeroes(vector<int>& nums) {  
4             int n = nums.size();  
5             int left = 0, right = 0;  
6             while (right < n) {  
7                 if (nums[right] != 0) {  
8                     swap(nums[left], nums[right]);
```

cpp

```
9         left++;  
10      }  
11      right++;  
12  }  
13  
14  }  
15 };
```

2.4. 递归

” 书内链接

- 小节 1.2.1 —— 反转链表
- 小节 1.7.1 —— 子结构判断

2.5. 分治

2.5.1. 逆序对

交易逆序对的总数

在股票交易中，如果前一天的股价高于后一天的股价，则可以认为存在一个「交易逆序对」。请设计一个程序，输入一段时间内的股票交易记录 `record`，返回其中存在的「交易逆序对」总数。

示例 1

输入：`record = [9, 7, 5, 4, 6]`

输出：8

解释：交易中的逆序对为 (9, 7), (9, 5), (9, 4), (9, 6), (7, 5), (7, 4), (7, 6), (5, 4)。

提示

$0 \leq \text{record.length} \leq 50000$

方法 1: 分治

「归并排序」与「逆序对」是息息相关的。归并排序体现了“分而治之”的算法思想，具体为：

- 分：不断将数组从中点位置划分开（即二分法），将整个数组的排序问题转化为子数组的排序问题；
- 治：划分到子数组长度为 1 时，开始向上合并，不断将较短排序数组合并为较长排序数组，直至合并至原数组时完成排序；

合并阶段本质上是合并两个排序数组的过程，而每当遇到左子数组当前元素 > 右子数组当前元素时，意味着「左子数组当前元素至末尾元素」与「右子数组当前元素」构成了若干「逆序对」。

因此，考虑在归并排序的合并阶段统计「逆序对」数量，完成归并排序时，也随之完成所有逆序对的统计。

`merge_sort()` 归并排序与逆序对统计：

1. 终止条件：当 $l \neq r$ 时，代表子数组长度为 1，此时终止划分；
2. 递归划分：计算数组中点 m ，递归划分左子数组 `merge_sort(l, m)` 和右子数组 `merge_sort(m + 1, r)`；
3. 合并与逆序对统计：
 1. 暂存数组 `record` 闭区间 $[l, r]$ 内的元素至辅助数组 `tmp`；
 2. 循环合并：设置双指针 i, j 分别指向左 / 右子数组的首元素；
 - 当 $i = m + 1$ 时：代表左子数组已合并完，因此添加右子数组当前元素 `tmp[j]`，并执行 $j = j + 1$ ；
 - 否则，当 $j = r + 1$ 时：代表右子数组已合并完，因此添加左子数组当前元素 `tmp[i]`，并执行 $i = i + 1$ ；
 - 否则，当 `tmp[i] > tmp[j]` 时：添加左子数组当前元素 `tmp[i]`，并执行 $i = i + 1$ ；

- 否则（即 $\text{tmp}[i] > \text{tmp}[j]$ ）时：添加右子数组当前元素 $\text{tmp}[j]$ ，并执行 $j=j+1$ ；此时构成 $m-i+1$ 个「逆序对」，统计添加至 res ；

4. 返回值：返回直至目前的逆序对总数 res ；

`reversePairs()` 主函数：

1. 初始化：辅助数组 tmp ，用于合并阶段暂存元素；
2. 返回值：执行归并排序 `merge_sort()`，并返回逆序对总数即可；

为简化代码，可将“当 $j=r+1$ 时”与“当 $\text{tmp}[i] \neq \text{tmp}[j]$ 时”两判断项合并。

```
1  class Solution {
2  public:
3      int reversePairs(vector<int>& record) {
4          vector<int> tmp(record.size());
5          return mergeSort(0, record.size() - 1, record, tmp);
6      }
7  private:
8      int mergeSort(int l, int r, vector<int>& record, vector<int>& tmp) {
9          // 终止条件
10         if (l ≥ r) return 0;
11         // 递归划分
12         int m = (l + r) / 2;
13         int res = mergeSort(l, m, record, tmp) + mergeSort(m + 1, r, record, tmp);
14         // 合并阶段
15         int i = l, j = m + 1;
16         for (int k = l; k ≤ r; k++)
17             tmp[k] = record[k];
18         for (int k = l; k ≤ r; k++) {
19             if (i == m + 1)
20                 record[k] = tmp[j++];
21             else if (j == r + 1 || tmp[i] ≤ tmp[j])
22                 record[k] = tmp[i++];
23             else {
24                 record[k] = tmp[j++];
25                 res += m - i + 1; // 统计逆序对
26             }
27         }
28         return res;
29     }
30 };
```

2.6. 广度优先搜索

2.6.1. 岛屿数量

岛屿数量

给你一个由 '1'（陆地）和 '0'（水）组成的二维网格，请你计算网格中岛屿的数量。岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。此外，你可以假设该网格的四条边均被水包围。

示例 1

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

示例 2

```
输入：grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
输出：3
```

提示

```
m == grid.length
n == grid[i].length
1 <= m, n <= 300
grid[i][j] 的值为 '0' 或 '1'
```

方法 1: 深度优先搜索

我们可以将二维网格看成一个无向图，竖直或水平相邻的 1 之间有边相连。

为了求出岛屿的数量，我们可以扫描整个二维网格。如果一个位置为 1，则以其为起始节点开始进行深度优先搜索。在深度优先搜索的过程中，每个搜索到的 1 都会被重新标记为 0。

最终岛屿的数量就是我们进行深度优先搜索的次数。

```
1  class Solution {
2  private:
3      const char LAND='1';
4      const char SEA='0';
5      const vector<pair<int,int>> DIRECTIONS={{1,0},{-1,0},{0,1},{0,-1}};
6
7      void dfs(vector<vector<char>>& grid,int r, int c){
8          int m=grid.size();
9          int n=grid[0].size();
10         grid[r][c]=SEA;
11         for(auto direction:DIRECTIONS){
12             int row=r+direction.first;
13             int col=c+direction.second;
14             if(row<0||row>=m||col<0||col>=n||grid[row][col]==SEA)
15                 continue;
16             dfs(grid,row,col);
17         }
18     }
19
20 public:
21     int numIslands(vector<vector<char>>& grid) {
22         int m=grid.size();
23         if(m==0) return 0;
24         int n=grid[0].size();
25         int cnt=0;
26         for(int i=0;i<m;i++){
27             for(int j=0;j<n;j++){
28                 if(grid[i][j]==LAND){
29                     cnt++;
30                     dfs(grid, i, j);
31                 }
32             }
33         }
34         return cnt;
35     }
36 };
```

💡 方法 2: 广度优先搜索

同样地，我们也可以使用广度优先搜索代替深度优先搜索。

为了求出岛屿的数量，我们可以扫描整个二维网格。如果一个位置为 1，则将其加入队列，开始进行广度优先搜索。在广度优先搜索的过程中，每个搜索到的 1 都会被重新标记为 0。直到队列为空，搜索结束。

最终岛屿的数量就是我们进行广度优先搜索的次数。

```
1  class Solution {
2  private:
3      const char LAND='1';
4      const char SEA='0';
5      const char VISITED='2';
6      const vector<pair<int,int>> DIRECTIONS={{1,0},{-1,0},{0,1},{0,-1}};
7
8  public:
9      int numIslands(vector<vector<char>>& grid) {
10         int m=grid.size();
11         if(m==0) return 0;
12         int n=grid[0].size();
13         int cnt=0;
14         for(int i=0;i<m;i++){
15             for(int j=0;j<n;j++){
16                 if(grid[i][j]==LAND){
17                     cnt++;
18                     grid[i][j]=VISITED;
19                     queue<pair<int,int>> que;
20                     que.push({i,j});
21                     while(!que.empty()){
22                         auto point=que.front();
23                         que.pop();
24                         int row=point.first;
25                         int col=point.second;
26                         for(auto direction:DIRECTIONS){
27                             int r=row+direction.first;
28                             int c=col+direction.second;
29                             if(r<0||r>=m||c<0||c>=n||grid[r][c]!=LAND)
30                                 continue;
31                             que.push({r,c});
32                             grid[r][c]=VISITED;
33                         }
34                     }
35                 }
36             }
37         }
38         return cnt;
39     }
40 }
41 };
```

💡 方法 3: 并查集

同样地，我们也可以使用并查集代替搜索。

为了求出岛屿的数量，我们可以扫描整个二维网格。如果一个位置为 1，则将其与相邻四个方向上的 1 在并查集中进行合并。

最终岛屿的数量就是并查集中连通分量的数目。

```
1  class UnionFind { cpp
2  public:
3      UnionFind(vector<vector<char>>& grid) {
4          count = 0;
5          int m = grid.size();
6          int n = grid[0].size();
7          for (int i = 0; i < m; ++i) {
8              for (int j = 0; j < n; ++j) {
9                  if (grid[i][j] == '1') {
10                     parent.push_back(i * n + j);
11                     ++count;
12                 }
13                 else {
14                     parent.push_back(-1);
15                 }
16                 rank.push_back(0);
17             }
18         }
19     }
20
21     int find(int i) {
22         if (parent[i] != i) {
23             parent[i] = find(parent[i]);
24         }
25         return parent[i];
26     }
27
28     void unite(int x, int y) {
29         int rootx = find(x);
30         int rooty = find(y);
31         if (rootx != rooty) {
32             if (rank[rootx] < rank[rooty]) {
33                 swap(rootx, rooty);
34             }
35             parent[rooty] = rootx;
36             if (rank[rootx] == rank[rooty]) rank[rootx] += 1;
37             --count;
38         }
39     }
40
41     int getCount() const {
42         return count;
43     }
44 }
```

```

45 private:
46     vector<int> parent;
47     vector<int> rank;
48     int count;
49 };
50
51 class Solution {
52 public:
53     int numIslands(vector<vector<char>>& grid) {
54         int nr = grid.size();
55         if (!nr) return 0;
56         int nc = grid[0].size();
57
58         UnionFind uf(grid);
59         int num_islands = 0;
60         for (int r = 0; r < nr; ++r) {
61             for (int c = 0; c < nc; ++c) {
62                 if (grid[r][c] == '1') {
63                     grid[r][c] = '0';
64                     if (r - 1 ≥ 0 && grid[r-1][c] == '1') uf.unite(r * nc + c, (r-1) *
nc + c);
65                     if (r + 1 < nr && grid[r+1][c] == '1') uf.unite(r * nc + c, (r+1) *
nc + c);
66                     if (c - 1 ≥ 0 && grid[r][c-1] == '1') uf.unite(r * nc + c, r * nc
+ c - 1);
67                     if (c + 1 < nc && grid[r][c+1] == '1') uf.unite(r * nc + c, r * nc
+ c + 1);
68                 }
69             }
70         }
71
72         return uf.getCount();
73     }
74 };

```

2.7. 深度优先搜索

” 书内链接

- 小节 2.6.1 —— 岛屿数量

2.7.1. 字母迷宫

? 字母迷宫

A	B	C	E
S	F	C	S
A	D	E	E

字母迷宫游戏初始界面记作 $m \times n$ 二维字符串数组 `grid`，请判断玩家是否能在 `grid` 中找到目标单词 `target`。注意：寻找单词时必须按照字母顺序，通过水平或垂直方向相邻的单元格内的字母构成，同时，同一个单元格内的字母不允许被重复使用。

示例 1

输入：`grid = [["A","B","C","E"], ["S","F","C","S"], ["A","D","E","E"]]`, `target = "ABCCED"`

输出：true

示例 2

输入：`grid = [["A","B","C","E"], ["S","F","C","S"], ["A","D","E","E"]]`, `target = "ABCB"`

输出：false

提示

`m == grid.length`

`n = grid[i].length`

`1 <= m, n <= 6`

`1 <= target.length <= 15`

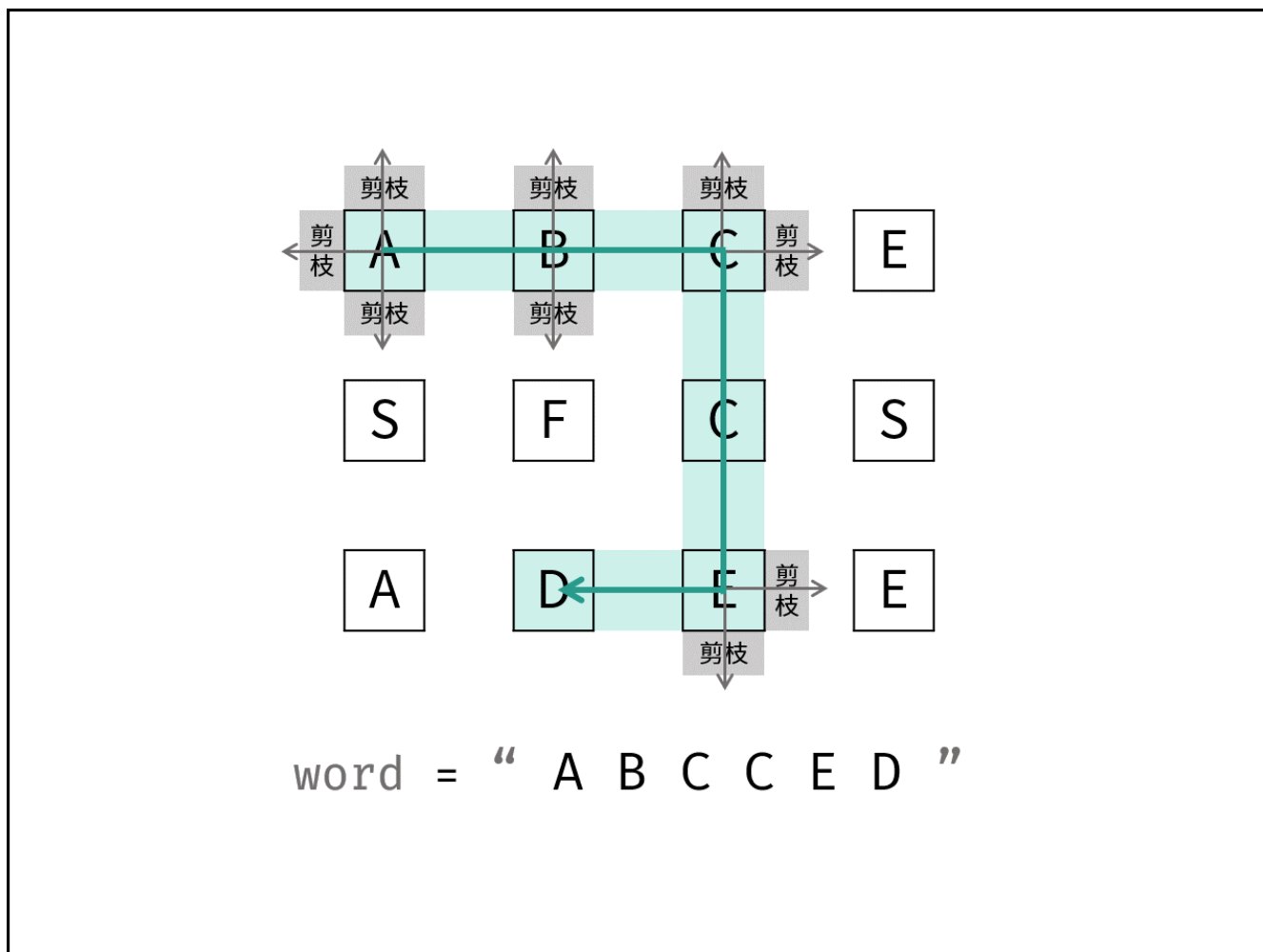
`grid` 和 `target` 仅由大小写英文字母组成

方法 1: 深度优先搜索+剪枝

本问题是典型的回溯问题，可使用深度优先搜索（DFS）+ 剪枝 解决。

- 深度优先搜索：可以理解为暴力法遍历矩阵中所有字符串可能性。DFS 通过递归，先朝一个方向搜到底，再回溯至上个节点，沿另一个方向搜索，以此类推。
- 剪枝：在搜索中，遇到这条路不可能和目标字符串匹配成功的情况（例如：此矩阵元素和目标字符不同、此元素已被访问），则应立即返回，称之为可行性剪枝。

下图中的 word 对应本题的 target。



1. 递归参数：当前元素在矩阵 grid 中的行列索引 i 和 j，当前目标字符在 target 中的索引 k。
2. 终止条件：
 1. 返回 false：(1) 行或列索引越界 或 (2) 当前矩阵元素与目标字符不同 或 (3) 当前矩阵元素已访问过（(3) 可合并至 (2)）。
 2. 返回 true：k = len(target) - 1，即字符串 target 已全部匹配。
3. 递推工作：
 1. 标记当前矩阵元素：将 grid[i][j] 修改为空字符“”，代表此元素已访问过，防止之后搜索时重复访问。
 2. 搜索下一单元格：朝当前元素的上、下、左、右四个方向开启下层递归，使用或连接（代表只需找到一条可行路径就直接返回，不再做后续 DFS），并记录结果至 res。
 3. 还原当前矩阵元素：将 grid[i][j] 元素还原至初始值，即 target[k]。
4. 返回值：返回布尔量 res，代表是否搜索到目标字符串。

```
1 class Solution {
```

```
cpp
```

```

2  public:
3      bool wordPuzzle(vector<vector<char>>& grid, string target) {
4          rows = grid.size();
5          cols = grid[0].size();
6          for(int i = 0; i < rows; i++) {
7              for(int j = 0; j < cols; j++) {
8                  if(dfs(grid, target, i, j, 0)) return true;
9              }
10         }
11         return false;
12     }
13 private:
14     int rows, cols;
15     bool dfs(vector<vector<char>>& grid, string target, int i, int j, int k) {
16         if(i ≥ rows || i < 0 || j ≥ cols || j < 0 || grid[i][j] ≠ target[k])
17         return false;
18         if(k == target.size() - 1) return true;
19         grid[i][j] = '\0';
20         bool res = dfs(grid, target, i + 1, j, k + 1) || dfs(grid, target, i - 1, j, k
21         + 1) ||
22                     dfs(grid, target, i, j + 1, k + 1) || dfs(grid, target, i , j - 1,
23         k + 1);
24         grid[i][j] = target[k];
25         return res;
26     }
27 };

```

2.8. 贪心算法

” 书内链接

小节 2.9.2 —— 买卖股票的最佳时机 II

2.8.1. 找字符串拼接的最大值

破解闯关密码

闯关游戏需要破解一组密码，闯关组给出的有关密码的线索是：

- 一个拥有密码所有元素的非负整数数组 `password`
- 密码是 `password` 中所有元素拼接后得到的最小的一个数

请编写一个程序返回这个密码。

示例 1

输入: `password = [15, 8, 7]`

输出: “1578”

示例 2

输入: `password = [0, 3, 30, 34, 5, 9]`

输出: “03033459”

提示

$0 < \text{password.length} \leq 100$

输出结果可能非常大，所以你需要返回一个字符串而不是整数

拼接起来的数字可能会有前导 0，最后结果不需要去掉前导 0

方法 1: 快速排序

此题求拼接起来的最小数字，本质上是一个排序问题。设数组 `password` 中任意两数字的字符串为 x 和 y ，则规定排序判断规则为：

- 若拼接字符串 $x + y > y + x$ ，则 x “大于” y ；
- 反之，若 $x + y < y + x$ ，则 x “小于” y ；

x “小于” y 代表：排序完成后，数组中 x 应在 y 左边；“大于”则反之。

根据以上规则，套用任何排序方法对 `password` 执行排序即可。

nums
字符串列表

"3"	"30"	"34"	"5"	"9"
^	^			
x	y			

拼接的最小值 "3033459" = "30" + "3" + "34" + "5" + "9"

排序判断规则 {
若 $x + y > y + x$, 则 x "大于" y
若 $x + y < y + x$, 则 x "小于" y

例如:

∴ "330" > "303" , > 是整数大小判断

∴ "30" 小于 "3" , "小于" 意味着"30"应排在"3"的前面

1. 初始化: 字符串列表 strs , 保存各数字的字符串格式;
2. 列表排序: 应用以上“排序判断规则”, 对 strs 执行排序;
3. 返回值: 拼接 strs 中的所有字符串, 并返回。

需修改快速排序函数中的排序判断规则。字符串大小(字典序)对比的实现方法:

- 在 Python 和 C++ 中可直接用 < , > ;
- 在 Java 中使用函数 A.compareTo(B);

```
1  class Solution {  
2      public:  
3          string crackPassword(vector<int>& password) {  
4              vector<string> strs;  
5              for(int i = 0; i < password.size(); i++)  
6                  strs.push_back(to_string(password[i]));  
7              quickSort(strs, 0, strs.size() - 1);  
8              string res;  
9              for(string s : strs)  
10                  res.append(s);  
11              return res;  
12          }  
13      private:  
14          void quickSort(vector<string>& strs, int l, int r) {  
15              if(l ≥ r) return;  
16              int i = l, j = r;
```

cpp

```

17     while(i < j) {
18         while(strs[j] + strs[l] ≥ strs[l] + strs[j] && i < j) j--;
19         while(strs[i] + strs[l] ≤ strs[l] + strs[i] && i < j) i++;
20         swap(strs[i], strs[j]);
21     }
22     swap(strs[i], strs[l]);
23     quickSort(strs, l, i - 1);
24     quickSort(strs, i + 1, r);
25 }
26 };

```

💡 方法 2: 内置函数

需定义排序规则:

- Python 定义在函数 `sort_rule(x, y)` 中;
- Java 定义为 `(x, y) → (x + y).compareTo(y + x)` ;
- C++ 定义为 `(string& x, string& y){ return x + y < y + x; }` ;

```

1  class Solution {
2  public:
3      string crackPassword(vector<int>& password) {
4          vector<string> strs;
5          string res;
6          for(int i = 0; i < password.size(); i++)
7              strs.push_back(to_string(password[i]));
8          sort(strs.begin(), strs.end(), [](string& x, string& y){ return x + y < y + x; });
9          for(int i = 0; i < strs.size(); i++)
10             res.append(strs[i]);
11         return res;
12     }
13 };

```

cpp

2.9. 动态规划

2.9.1. 丑数

丑数

给你一个整数 n ，请你找出并返回第 n 个丑数。

说明：丑数是只包含质因数 2、3 和/或 5 的正整数；1 是丑数。

示例 1

输入: $n = 10$

输出: 12

解释: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。

提示

$1 \leq n \leq 1690$

方法 1: 动态规划

根据题意，每个丑数都可以由其他较小的丑数通过乘以 2 或 3 或 5 得到。

所以，可以考虑使用一个优先队列保存所有的丑数，每次取出最小的那个，然后乘以 2, 3, 5 后放回队列。然而，这样做会出现重复的丑数。例如：

> 初始化丑数列表 [1]

> 第一轮：1 -> 2, 3, 5，丑数列表变为 [1, 2, 3, 5]

> 第二轮：2 -> 4, 6, 10，丑数列表变为 [1, 2, 3, 4, 6, 10]

> 第三轮：3 -> 6, 9, 15，出现重复的丑数 6

为了避免重复，我们可以用三个指针 a, b, c ，分别表示下一个丑数是当前指针指向的丑数乘以 2, 3, 5。

利用三个指针生成丑数的算法流程：

1. 初始化丑数列表 res ，首个丑数为 1，三个指针 a, b, c 都指向首个丑数。
2. 开启循环生成丑数：
 1. 计算下一个丑数的候选集 $res[a] \cdot 2, res[b] \cdot 3, res[c] \cdot 5$ 。
 2. 选择丑数候选集中最小的那个作为下一个丑数，填入 res 。
 3. 将被选中的丑数对应的指针向右移动一格。
3. 返回 res 的最后一个元素即可。

```
1 class Solution {
2     public:
3         int nthUglyNumber(int n) {
4             int a = 0, b = 0, c = 0;
5             int res[n];
```

cpp

```
6     res[0] = 1;
7     for(int i = 1; i < n; i++) {
8         int n2 = res[a] * 2, n3 = res[b] * 3, n5 = res[c] * 5;
9         res[i] = min(min(n2, n3), n5);
10        if (res[i] == n2) a++;
11        if (res[i] == n3) b++;
12        if (res[i] == n5) c++;
13    }
14    return res[n - 1];
15 }
16 };
```

2.9.2. 买卖股票的最佳时机 II

买卖股票的最佳时机 II

给你一个整数数组 `prices`，其中 `prices[i]` 表示某支股票第 `i` 天的价格。

在每一天，你可以决定是否购买和/或出售股票。你在任何时候 **最多** 只能持有一股股票。你也可以先购买，然后在 **同一天** 出售。

返回你能获得的 **最大** 利润。

示例 1

输入: `prices = [7,1,5,3,6,4]`

输出: 7

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = $5 - 1 = 4$ 。随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出, 这笔交易所能获得利润 = $6 - 3 = 3$ 。最大总利润为 $4 + 3 = 7$ 。

示例 2

输入: `prices = [1,2,3,4,5]`

输出: 4

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

示例 3

输入: `prices = [7,6,4,3,1]`

输出: 0

解释: 在这种情况下, 交易无法获得正利润, 所以不参与交易可以获得最大利润, 最大利润为 0。

提示

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

方法 1: 动态规划

这是一个经典的动态规划问题, 可以通过动态规划的方法来求解。我们定义一个二维数组 `dp`, 其中:

- `dp[i][0]` 表示在第 `i` 天不持有股票的最大利润。
- `dp[i][1]` 表示在第 `i` 天持有股票的最大利润。

状态转移方程

- $dp[i][0]$ 可以通过以下两种情况得到:

1. 前一天也不持有股票, $dp[i-1][0]$
2. 前一天持有股票, 并在今天卖出了, $dp[i-1][1] + prices[i]$

所以, $dp[i][0] = \max(dp[i-1][0], dp[i-1][1] + prices[i])$

- $dp[i][1]$ 可以通过以下两种情况得到:

1. 前一天也持有股票, $dp[i-1][1]$
2. 前一天不持有股票, 并在今天买入了, $dp[i-1][0] - prices[i]$

所以, $dp[i][1] = \max(dp[i-1][1], dp[i-1][0] - prices[i])$

初始条件

- $dp[0][0] = 0$: 第一天不持有股票, 利润为 0
- $dp[0][1] = -prices[0]$: 第一天持有股票, 利润为负的股票价格

最终结果

我们需要返回在最后一天不持有股票的最大利润, 即 $dp[n-1][0]$ 。

```
1 class Solution {
2 public:
3     int maxProfit(vector<int>& prices) {
4         int n = prices.size();
5         if (n == 0) return 0;
6         int dp[n][2];
7         dp[0][0] = 0;
8         dp[0][1] = -prices[0];
9         for (int i = 1; i < n; i++) {
10             dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i]);
11             dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i]);
12         }
13         return dp[n-1][0];
14     }
15 };
```

方法 2: 贪心算法

对于单独交易日: 设今天价格 p_1 、明天价格 p_2 , 则今天买入、明天卖出可赚取金额 $p_2 - p_1$ (负值代表亏损)。

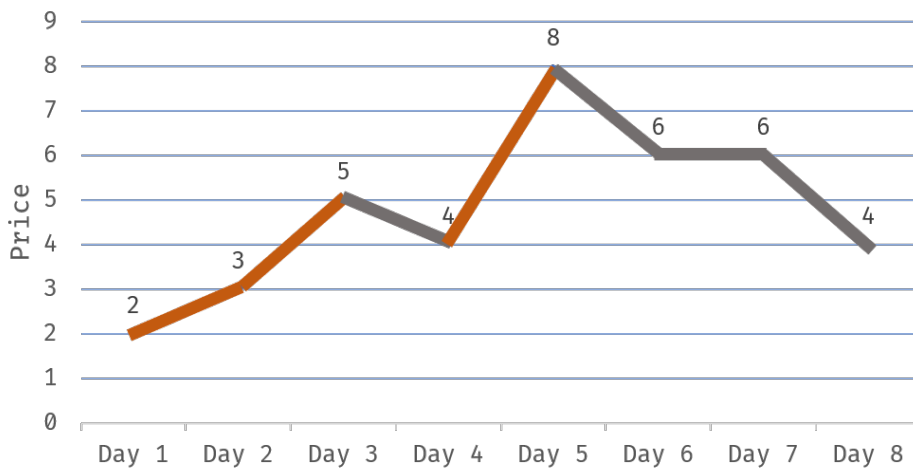
对于连续上涨交易日: 设此上涨交易日股票价格分别为 p_1, p_2, \dots, p_n , 则第一天买最后一天卖收益最大, 即 $p_n - p_1$; 等价于每天都买卖, 即 $p_n - p_1 = (p_2 - p_1) + (p_3 - p_2) + \dots + (p_n - p_{n-1})$ 。

对于连续下降交易日: 则不买卖收益最大, 即不会亏钱。

算法流程

遍历整个股票交易日价格列表 `price`，并执行贪心策略：所有上涨交易日都买卖（赚到所有利润），所有下降交易日都不买卖（永不亏钱）。

1. 设 `tmp` 为第 $i-1$ 日买入与第 i 日卖出赚取的利润，即 `tmp = prices[i] - prices[i - 1]`；
2. 当该天利润为正 `tmp > 0`，则将利润加入总利润 `profit`；当利润为 0 或为负，则直接跳过；
3. 遍历完成后，返回总利润 `profit`。



`tmp = -2`

`profit = 7` → `return 7`

```
1 class Solution {
2 public:
3     int maxProfit(vector<int>& prices) {
4         int profit = 0;
5         for (int i = 1; i < prices.size(); i++) {
6             int tmp = prices[i] - prices[i - 1];
7             if (tmp > 0) profit += tmp;
8         }
9         return profit;
10    }
11 };
```

cpp

2.10. 最短路径

2.11. 最小生成树

2.12. 前缀和

2.13. 数学

2.13.1. 回文数

回文数

给你一个整数 x ，如果 x 是一个回文整数，返回 `true`；否则，返回 `false`。

回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

例如，121 是回文，而 123 不是。

你能不将整数转为字符串来解决这个问题吗？

示例 1

输入： $x = 121$

输出：true

示例 2

输入： $x = -121$

输出：false

解释：从左向右读，为 -121 。从右向左读，为 $121-$ 。因此它不是一个回文数。

示例 3

输入： $x = 10$

输出：false

解释：从右向左读，为 01。因此它不是一个回文数。

提示

$$-2^{31} \leq x \leq 2^{31} - 1$$

方法 1: 组

重新组成一个新数，判断是否和原数相等。

```
1 class Solution {
2 public:
3     bool isPalindrome(int x) {
4         if(x < 0) return false;
5         int temp = x;
6         long long res = 0;
7         while(x){
```

cpp

```

8         res = res * 10 + x % 10;
9         x /= 10;
10    }
11    return res == temp;
12 }
13 };

```

💡 方法 2: 拆

将数按头尾拆分，判断是否相等。

```

1  class Solution {
2  public:
3      bool isPalindrome(int x) {
4          if (x < 0) return false;
5          int len = log10(x) + 1;
6          int num = x;
7          while (num) {
8              int left = num / pow(10, len - 1);
9              int right = num % 10;
10             if (left != right) return false;
11             num %= static_cast<int>(pow(10, len - 1));
12             num /= 10;
13             len -= 2;
14         }
15         return true;
16     }
17 };

```

💡 方法 3: 只反转一半

为了避免数字反转可能导致的溢出问题，考虑只反转 int 数的一半。

对于数字 1221，如果执行 $1221 \% 10$ ，我们将得到最后一位数字 1，要得到倒数第二位数字，我们可以先通过除以 10 把最后一位数字从 1221 中移除， $\frac{1221}{10} = 122$ ，再求出上一步结果除以 10 的余数， $122 \% 10 = 2$ ，就可以得到倒数第二位数字。如果我们把最后一位数字乘以 10，再加上倒数第二位数字， $1 * 10 + 2 = 12$ ，就得到了我们想要的反转后的数字。如果继续这个过程，我们将得到更多位数的反转数字。

现在的问题是，我们如何知道反转数字的位数已经达到原始数字位数的一半？

由于整个过程我们不断将原始数字除以 10，然后给反转后的数字乘上 10，所以，当原始数字小于或等于反转后的数字时，就意味着我们已经处理了一半位数的数字了。

```

1  class Solution {
2  public:
3      bool isPalindrome(int x) {
4          if (x < 0 || (x % 10 == 0 && x != 0)) {

```

```
5         return false;
6     }
7     int revertedNumber = 0;
8     while (x > revertedNumber) {
9         revertedNumber = revertedNumber * 10 + x % 10;
10        x /= 10;
11    }
12
13    // 当数字长度为奇数时，我们可以通过 revertedNumber/10 去除处于中位的数字。
14    // 例如，当输入为 12321 时，在 while 循环的末尾我们可以得到 x = 12, revertedNumber = 12
15    // 由于处于中位的数字不影响回文（它总是与自己相等），所以我们可以简单地将其去除。
16    return x == revertedNumber || x == revertedNumber / 10;
17 }
18 };
```


2.13.2. 只出现一次的数字

只出现一次的数字

给你一个非空整数数组 `nums`，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

你必须设计并实现线性时间复杂度的算法来解决此问题，且该算法只使用常量额外空间。

示例 1

输入: `nums = [2,2,1]`

输出: 1

示例 2

输入: `nums = [4,1,2,1,2]`

输出: 4

示例 3

输入: `nums = [1]`

输出: 1

提示

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $-3 * 10^4 \leq \text{nums}[i] \leq 3 * 10^4$
- 除了某个元素只出现一次以外，其余每个元素均出现两次。

方法 1: 位运算

异或运算有个重要的性质，两个相同数字异或为 0，即对于任意整数 a 有 $a \oplus a = 0$ 。因此，若将 `nums` 中所有数字执行异或运算，留下的结果则为出现一次的数字 x ，即：

$$a \oplus a \oplus b \oplus b \oplus \dots \oplus x$$

$$= 0 \oplus 0 \oplus \dots \oplus x$$

$$= x$$

```
1 class Solution {  
2     public:  
3         int singleNumber(vector<int>& nums) {  
4             int res = 0;  
5             for (int i = 0; i < nums.size(); i++) {
```

cpp

```
6         res ^= nums[i];  
7     }  
8     return res;  
9 }  
10};
```

2.13.3. 旋转图像


? 旋转图像

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

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

示例 1

1	2	3
4	5	6
7	8	9




7	4	1
8	5	2
9	6	3

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

输出: `[[7,4,1],[8,5,2],[9,6,3]]`

示例 2

5	1	9	11
2	4	8	10
13	3	6	7
15	14	12	16



15	13	2	5
14	3	4	1
12	6	8	9
16	7	10	11

输入: `matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]`

输出: `[[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]`

提示

- $n == \text{matrix.length} == \text{matrix}[i].\text{length}$
- $1 \leq n \leq 20$
- $-1000 \leq \text{matrix}[i][j] \leq 1000$

方法 1: 用翻转代替旋转

```
1 class Solution {  
2     public:  
3         void rotate(vector<vector<int>>& matrix) {  
4             int n = matrix.size();
```

cpp

```
5      // 水平翻转
6      for (int i = 0; i < n / 2; i++) {
7          for (int j = 0; j < n; j++) {
8              swap(matrix[i][j], matrix[n - i - 1][j]);
9          }
10     }
11     // 主对角线翻转
12     for (int i = 0; i < n; i++) {
13         for (int j = 0; j < i; j++) {
14             swap(matrix[i][j], matrix[j][i]);
15         }
16     }
17
18 }
19 };
```

3. 题集

