程设错题 15~20 20241027-20241031

1.Leetcode 283 移动零

题目

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 <= nums.length <= 104`
- `-231 <= nums[i] <= 231 - 1`

解法①

使用vector数组中的push_back和erase成员函数对数组进行遍历，得到移动完成的数组

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int counter=0;
        for(auto iter=nums.begin();counter<nums.size();counter++){
            if(*(iter)==0){
                nums.erase(iter,iter+1);
                nums.push_back(0);
                //如果发现0，对数组进行操作，将0删除，并添加至末尾
            }else{
                iter++;
                //如果发现的不是0，不进行操作，迭代器后移一位
            }
        }
        for(auto num:nums){
            cout<<num;
            //遍历数组输出结果
        }

    }
};
```

时间复杂度：O(n^2)(erase操作的时间复杂度是O(n)，再加上外层的循环)

解法②

解法①的低效之处：在于erase的清除过程太繁琐

改进：操作所有非零数值，使其按顺序挪到数组的开头，覆盖开头的值，并将结尾的所有剩余值修改成0。

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int n = nums.size();
        int nonZeroIndex = 0;

        // 将所有非零元素移动到数组的前面
        for (int i = 0; i < n; ++i) {
            if (nums[i] != 0) {
                nums[nonZeroIndex++] = nums[i];
            }
        }

        // 将剩余的位置填充为零
        for (int i = nonZeroIndex; i < n; ++i) {
            nums[i] = 0;
        }

        // 打印数组
        for (auto num : nums) {
            cout << num << " ";
        }
        cout << endl;
    }
};
```

时间复杂度：O(n)

解法③ 双指针并行处理

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int n = nums.size(), left = 0, right = 0;
        while (right < n) {
            if (nums[right]) {
                swap(nums[left], nums[right]);
                left++;
                //如果nums[right]不是0，那么和左指针发生交换，右指针是试探指针，左指针代表已经检查过是非零值的元素，并将左指针右移一位
                //最坏的情况:没有0，则左指针和右指针完全同步
            }
            right++;
        }
    }
};
```

时间复杂度：O(n)

2.Leetcode 290 单词映射

给定一种规律 `pattern` 和一个字符串 `s` ，判断 `s` 是否遵循相同的规律。

这里的 **遵循** 指完全匹配，例如， `pattern` 里的每个字母和字符串 `s` 中的每个非空单词之间存在着双向连接的对应规律。

示例1:

```
输入: pattern = "abba", s = "dog cat cat dog"
输出: true
```

示例 2:

```
输入:pattern = "abba", s = "dog cat cat fish"
输出: false
```

示例 3:

```
输入: pattern = "aaaa", s = "dog cat cat dog"
输出: false
```

代码解法

思路：

- 如何获得子字符串作为单词?
  - 使用substr()
  - 使用双指针
  - `substr` 是 C++ 中 `std::string` 类的一个成员函数，用于从字符串中提取子字符串。它有两个参数：
    1. **起始位置（pos）**：要提取的子字符串的起始索引。
    2. **长度（len）**：要提取的子字符串的长度（可选）。如果不指定，则提取到字符串的末尾。

    ## 语法

    ```
    std::string substr(size_t pos = 0, size_t len = npos) const;
    ```

    - `pos`：开始提取的位置。
    - `len`：要提取的字符数。
    - `npos`：一个特殊值，表示直到字符串的末尾。

示例

```
#include <iostream>
#include <string>

int main() {
    std::string text = "Hello, World!";

    // 提取从位置 7 开始的子字符串，长度为 5
    std::string sub1 = text.substr(7, 5);
    std::cout << sub1 << std::endl;  // 输出 "World"

    // 提取从位置 7 开始的子字符串，直到字符串末尾
    std::string sub2 = text.substr(7);
    std::cout << sub2 << std::endl;  // 输出 "World!"

    return 0;
}
```

注意事项

- 如果 `pos` 超出字符串长度，会抛出 `std::out_of_range` 异常。
- 如果 `len` 超出可用长度，则提取到字符串的末尾。

`substr` 是处理字符串时非常有用的工具，尤其是在需要提取特定部分的场景中。

- 如何构建一一映射的关系
  - 使用两个unordered_map
- 如何处理两个数量不相等的情况
  - 使用计数器

```
class Solution {
public:
    bool wordPattern(string pattern, string str) {
        unordered_map<string, char> str2ch;
        unordered_map<char, string> ch2str;
        int m = str.length();
        int i = 0;
        for (auto ch : pattern) {
            if (i >= m) {
                return false;
                //如果 i 超过或等于 str 的长度，说明 str 中的单词数量不足以匹配 pattern，返回 false(此时ch还在遍历Pattern，但是i已经超过str的长度了，说明str不够长)
            }
            int j = i;
            //经典的双指针解法，截取一个子字符串
            while (j < m && str[j] != ' ') j++;
            //截取一个单词
            const string &tmp = str.substr(i, j - i);
            //此处已经跳出循环，故包含了' '的情况，单词长度就是j-i
            if (str2ch.count(tmp) && str2ch[tmp] != ch) {
                //如果map映射中已经有了这个单词的映射，并且这个单词的映射并不等于现在遍历得到的ch
                return false;
```

```
            }
            if (ch2str.count(ch) && ch2str[ch] != tmp) {
                //反过来，如果map映射已经有了字符的映射并且这个字符的映射并不等于现在得到的单词
                return false;
            }
            //以上两个条件都为false，故在两个映射中填充进新的值。
            str2ch[tmp] = ch;
            ch2str[ch] = tmp;
            i = j + 1;
            //更新双指针ij的位置
        }
        return i >= m;
        //如果i<m,说明还有单词未被输出，说明Pattern不够长，输出false
    }
};
```

3. Leetcode 350 两数组交集

给你两个整数数组 `nums1` 和 `nums2` ，请你以数组形式返回两数组的交集。返回结果中每个元素出现的次数，应与元素在两个数组中都出现的次数一致（如果出现次数不一致，则考虑取较小值）。可以不考虑输出结果的顺序。

解法1　最基本的思路

```
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        int list1[1001]={0};
        int list2[1001]={0};
        vector<int> succedd;
        for(auto num1:nums1){
            list1[num1]++;
        }
        for(auto num2:nums2){
            list2[num2]++;
        }
        for(int i=0;i<1001;i++){
            if(list1[i]&&list2[i]){
                for(int j=0;j<min(list1[i],list2[i]);j++){
                    succedd.push_back(i);
                }
            }
        }
        return succedd;
    }
};
```

解法2 优化：使用哈希表存储值

```
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        if (nums1.size() > nums2.size()) {
            return intersect(nums2, nums1);
        }
        //为了减少空间复杂度，优先遍历元素数较少的数组
        unordered_map <int, int> m;
        for (int num : nums1) {
            ++m[num];
        }
        //遍历nums1
        vector<int> intersection;
        for (int num : nums2) {
            if (m.count(num)) {
                intersection.push_back(num);
                //如果出现，则插入到目标序列中
                --m[num];
                //相当于用掉了num1中的一个数，故要减1
                if (m[num] == 0) {
                    m.erase(num);
                }
                //踢出num出s，代表nums1中对应的数已经被用完了
            }
        }
        return intersection;
    }
};
```

解法3 排序+双指针遍历

对于数组问题，可以先进行排序再进行操作！

```
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        sort(nums1.begin(), nums1.end());
        sort(nums2.begin(), nums2.end());
        int length1 = nums1.size(), length2 = nums2.size();
        vector<int> intersection;
        int index1 = 0, index2 = 0;
        while (index1 < length1 && index2 < length2) {
            if (nums1[index1] < nums2[index2]) {
                index1++;
            } else if (nums1[index1] > nums2[index2]) {
                index2++;
            } else {
                //nums1[]==nums2[]
                //同时向前前进一位
                intersection.push_back(nums1[index1]);
```

```
                index1++;
                index2++;
            }
        }
        return intersection;
    }
};
```

4. Leetcode 438 字符串的字母异位词

给定两个字符串 `s` 和 `p`，找到 `s` 中所有 `p` 的 **异位词** 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

示例 1:

```
输入: s = "cbaebabacd", p = "abc"
输出: [0,6]
解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的异位词。
```

示例 2:

```
输入: s = "abab", p = "ab"
输出: [0,1,2]
解释:
起始索引等于 0 的子串是 "ab", 它是 "ab" 的异位词。
起始索引等于 1 的子串是 "ba", 它是 "ab" 的异位词。
起始索引等于 2 的子串是 "ab", 它是 "ab" 的异位词。
```

解法1 使用基本思路

从i=0开始，遍历s数组，切割得到子串，然后判定s的子串和p是否为异位串。

本质上是优化的滑动窗口

```
class Solution {
public:
    vector<int> findAnagrams(string s, string p) {
        vector<int> ans;
        if (s.length() < p.length()) {
        } else {
            for (int i = 0; i + p.length() - 1 < s.length(); i++) {
                string s2 = s.substr(i, p.length());
                if (judgeyiwei(s2, p)) {
                    ans.push_back(i);
                }
            }
        }
        return ans;
    }
    bool judgeyiwei(string s1, string s2) {
        int numlist[26] = {0};
```

```
        for (int i = 0; i < s1.length(); i++) {
            numlist[s1[i] - 'a']++;
            numlist[s2[i] - 'a']--;
        }
        for (int i = 0; i < 26; i++) {
            if (numlist[i]) {
                return false;
            }
        }
        return true;
    }
};
```

解法2 不定长滑动窗口优化

基本原理：枚举子串 s '的右端点，如果发现 s '其中一种字母的出现次数大于 p 的这种字母的出现次数，则右移 s '的左端点。如果发现 s '的长度等于 p 的长度，则说明 s '的每种字母的出现次数，和 p 的每种字母的出现次数都相同，那么 s '是 p 的异位词。

有点类似于双指针

```
class Solution {
public:
    vector<int> findAnagrams(string s, string p) {
        vector<int> ans;
        int cnt[26]{0}; // 统计 p 的每种字母的出现次数
        for (char c : p) {
            cnt[c - 'a']++;
        }
        //接下来对s字符串进行遍历操作(遍历右端点作为外层循环，在内部用while循环控制左端点)
        int left = 0;
        for (int right = 0; right < s.size(); right++) {
            int c = s[right] - 'a';
            cnt[c]--; // 右端点字母进入窗口，相当于把p中减掉一个对应的字符
            while (cnt[c] < 0) { // c对应的字符(ASCII)太多了
                cnt[s[left] - 'a']++; // 左端点字母离开窗口
                left++;
            }
            if (right - left + 1 == p.length()) { // s' 和 p 的每种字母的出现次数都相
同
                ans.push_back(left); // s' 左端点下标加入答案
            }
        }
        return ans;
    }
};
```

5.Leetcode 443 字符串压缩问题

给你一个字符数组 `chars` ，请使用下述算法压缩：

从一个空字符串 `s` 开始。对于 `chars` 中的每组 **连续重复字符** ：

- 如果这一组长度为 `1` ，则将字符追加到 `s` 中。

- 否则，需要向 `s` 追加字符，后跟这一组的长度。

压缩后得到的字符串 `s` **不应该直接返回**，需要转储到字符数组 `chars` 中。需要注意的是，如果组长度为 `10` 或 `10` 以上，则在 `chars` 数组中会被拆分为多个字符。

请在 **修改完输入数组后**，返回该数组的新长度。

你必须设计并实现一个只使用常量额外空间的算法来解决此问题。

必须在原来的数组上进行修改！

解法：双指针

为了实现原地压缩，我们可以使用双指针分别标志我们在字符串中读和写的位置。每次当读指针 read 移动到某一段连续相同子串的最右侧，我们就在写指针 write 处依次写入该子串对应的字符和子串长度即可。

在实际代码中，当读指针 read 位于字符串的末尾，或读指针 read 指向的字符不同于下一个字符时，我们就认为**读指针 read 位于某一段连续相同子串的最右侧**。该子串对应的字符即为读指针 read 指向的字符串。我们使用变量 left 记录该子串的最左侧的位置，这样子串长度即为 read−left+1。

特别地，为了达到 O(1) 空间复杂度，我们需要自行实现将数字转化为字符串写入到原字符串的功能。这里我们采用短除法将子串长度倒序写入原字符串中，然后再将其反转即可。

```
class Solution {
public:
    int compress(vector<char>& chars) {
        int n = chars.size();
        int write = 0, left = 0;
        for (int read = 0; read < n; read++) {
            if (read == n - 1 || chars[read] != chars[read + 1]) {
                //由于写的速度肯定没有读的快，故不用担心write会覆盖未被读取的初始值
                chars[write++] = chars[read];
                int num = read - left + 1;
                if (num > 1) {
                    int anchor = write;
                    while (num > 0) {
                        chars[write++] = num % 10 + '0';
                        num /= 10;
                    }
                    reverse(&chars[anchor], &chars[write]);
                    //实现输入一个数字倒序插入char数组中（也可以使用stringstream）
                }
                left = read + 1;
                //后面还会执行read++，本质上就是更新read和left的位置使其对齐
            }
        }
        return write;
    }
};
```