

程设错题 11-15 20241021

1.ACwing 776 字符串移位问题

对于一个字符串来说，定义一次循环移位操作为：将字符串的第一个字符移动到末尾形成新的字符串。

给定两个字符串 s_1 和 s_2 ，要求判定其中一个字符串是否是另一字符串通过若干次循环移位后的新字符串的子串。

例如 `CDAAB` 是由 `AABCD` 两次移位后产生的新串 `BCDAA` 的子串，而 `ABCD` 与 `ACBD` 则不能通过多次移位来得到其中一个字符串是新串的子串。

输入格式

共一行，包含两个字符串，中间由单个空格隔开。

字符串只包含字母和数字，长度不超过 30。

输出格式

如果一个字符串是另一字符串通过若干次循环移位产生的新串的子串，则输出 `true`，否则输出 `false`。

基本思路：

- 使用函数，判断一个字符串是不是另一个字符串的子串
- 利用for循环实现对目标数组的移位处理，并且对每个新移位的字符串调用该函数

```
#include <string>
#include <iostream>
#include <vector>
using namespace std;
bool judgestring(string teststring,string standardstring){
    if(teststring.length()<standardstring.length()){
        return false;
    }else{
        if(teststring.find(standardstring)==-1){
            return false;
        }else{
            return true;
        }
    }
}
int main(){
    string movestring,standardstring;
    cin>>movestring>>standardstring;

    for(int i=0;i<movestring.length()-1;){
        movestring.push_back(movestring[0]);
        movestring.erase(movestring.begin());
        if(judgestring(movestring,standardstring)){
            cout<<"true";
            goto end;
        }
    }
    cout<<"false";
    end;;
    return 0;
}
```

优化思路：

若将一个字符串首尾相接，则不需要进行移项操作（相当于顺序没有发生变化）

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

using namespace std;

bool check(string a, string b)
{
    int len = a.size();
    a += a; //复制字符串并连接
    //if (a.find(b) >= 0 && a.find(b) < len) return true; //判断是否包含
    if (a.find(b) != string::npos) return true; //判断a中是否包含b
    //两种判断方式，推荐第二种，若find函数未找到，则会返回一个特殊常量string::npos
    return false;
}

int main()
{
    string a, b;
    cin >> a >> b;
    if (a.size() < b.size()) swap(a, b);
    //默认a是更大的string类
    if (check(a, b)) cout << "true";
    else cout << "false";
    return 0;
}
```

2.ACwing 777 字符串的最小周期问题

给定一个字符串s,求其最小的周期格段

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    string s;
    int flag=1;
    for(int i=1;i<=s.length();++i){
        //外层循环：枚举可能的周期，从小到大枚举，第一个得到的值就是目标值
        flag=1;
        if(len%i!=0){
            continue;
        }
        for(int j=0;j<s.length();j++){
            if(s[j]!=s[j%i]){
                //核心代码：判断每一个j和j%i是否相等，判断是否符合周期性的定义。
                flag=0;
                break;
            }
        }
    }
}
```

```

        if(flag){
            cout<<s.length()/i<<endl;
            goto :end;
        }
    }
    cout<<"-1";
    end::;
}
}


```

3.Leetcode 217 存在重复元素（进阶版）

217. 存在重复元素

尝试过 

简单

 相关标签

 相关企业

Ax

给你一个整数数组 `nums`。如果任一值在数组中出现 **至少两次**，返回 `true`；如果数组中每个元素互不相同，返回 `false`。

1.最暴力方法：使用枚举筛查 $O(n^2)$

```

class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        vector<int> findnums;
        findnums.reserve(10000);
        int input=0;
        for(int i=0;i<nums.size();i++){
            auto it=std::find(findnums.begin(),findnums.end(),nums[i]);
            if(it!=findnums.end()){
                input=1;
                break;
            }else{
                findnums.push_back(nums[i]);
            }
        }
        return bool(input);
    }
};

```

分析：效率过低，每次插入一个新值都需要重新查找一遍findnums数组

优化：将findnums数组优化为unordered_set（使用哈希表来存储出现过的元素效率更高）

优化代码：

```

class Solution {
public:
    bool containsDuplicate(std::vector<int>& nums) {
        std::unordered_set<int> seen;
        for (int num : nums) {
            //编程习惯：遍历的简化写法
            if (seen.find(num) != seen.end()) {

```

```
        //如果找到了num,直接输出true
        return true;
    }
    seen.insert(num);
}
return false;
};
```

知识补充：哈希表

使用哈希表（如 `std::unordered_set`）存储已经遇到的元素，**时间复杂度更低**的原因主要在于哈希表的查找和插入操作的平均时间复杂度是 $O(1)$ 。这是因为哈希表通过哈希函数将元素映射到一个数组中的特定位置，从而实现了快速的查找和插入。以下是详细解释：

哈希表的工作原理

- 哈希函数**：哈希表使用一个哈希函数将每个元素的值转换为一个哈希码（通常是一个整数），这个哈希码对应哈希表中的一个位置（桶）。
- 存储位置**：元素被存储在对应的桶中，查找和插入操作都可以通过计算哈希码直接定位到相应的桶，从而避免了线性扫描。
- 冲突处理**：当多个元素映射到同一个桶时，哈希表使用冲突处理机制（如链地址法或开放地址法）来处理这些冲突。

时间复杂度分析

- 查找**：在理想情况下，哈希表的查找操作只需计算一次哈希函数并访问一个桶，时间复杂度为 $O(1)$ 。
- 插入**：插入操作也只需计算一次哈希函数并将元素放入相应的桶中，时间复杂度为 $O(1)$ 。
- 删除**：类似于查找和插入，删除操作也可以在 $O(1)$ 时间内完成。

对比线性查找

- 线性查找**：在向量或链表中查找一个元素需要遍历所有元素，时间复杂度为 $O(n)$ 。（例如vector）
- 哈希查找**：在哈希表中查找一个元素平均只需要 $O(1)$ 时间，可以显著提高查找效率。

实际应用中的考虑

虽然哈希表的平均时间复杂度是 $O(1)$ ，但在最坏情况下（例如哈希函数不均匀导致所有元素映射到同一个桶），时间复杂度可以退化到 $O(n)$ 。然而，通过选择合适的哈希函数和合理的负载因子，最坏情况发生的概率可以被大大降低。

结论

使用哈希表存储已经遇到的元素，可以利用其快速查找和插入的特性，将包含重复元素的检查操作的时间复杂度从 $O(n^2)$ 降低到 $O(n)$ ，在处理大数据集时显著提高性能。

4.Leetcode 219 存在重复元素II（进阶版）

给你一个整数数组 `nums` 和一个整数 `k`，判断数组中是否存在两个 **不同的索引** `i` 和 `j`，满足 `nums[i] == nums[j]` 且 `abs(i - j) <= k`。如果存在，返回 `true`；否则，返回 `false`。

思路：此处使用unordered_set会比较麻烦，unordered_set 键值对就是自己本身，这里可以使用unordered_map类型的容器，实现两个值之间的一一映射。

（在上一题中只是判断是否存在，对元素的索引和顺序并未提出要求，故使用unordered_set 即可）

代码实现：使用映射

```
class Solution {
public:
    bool containsNearbyDuplicate(vector<int>& nums, int k) {
        unordered_map<int, int> dictionary; // 用于存储每个元素的最新索引
        int length = nums.size(); // 获取数组的长度
        for (int i = 0; i < length; i++) { // 遍历数组
            int num = nums[i]; // 当前元素
            // 如果当前元素已经存在于字典中，并且当前索引与存储的索引之差小于等于 k
            if (dictionary.count(num) && i - dictionary[num] <= k) {
                return true; // 找到符合条件的重复元素，返回 true
            }
            dictionary[num] = i; // 更新字典中的索引 nums[i]->i的映射
        }
        return false; // 遍历完数组后，未找到符合条件的重复元素，返回 false
    }
};
```

相关代码解释

- 基本思路：创建dictionary的map映射，并对nums数组不断进行操作，如果找到两个相同的元素并且索引小于k，直接return true，如果没有，则储存一对新的映射（元素唯一）
- unordered_map容器中键值保持唯一性，使用 [] 运算符插入相同键时，新值会覆盖旧值。使用 insert 方法插入相同键时，插入操作会失败，原值保持不变。
- count函数：

std::unordered_map 的 count 函数用于检查特定键是否存在于映射中。它返回一个整数值，表示具有指定键的元素数量。在 std::unordered_map 中，每个键是唯一的，因此 count 函数的返回值只能是 0 或 1。

```
size_t count(const Key& key) const;
```

- key：要检查的键。
- 返回值：如果键存在，则返回 1；否则返回 0。

```
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<int, std::string> myMap;
    myMap[1] = "one";
    myMap[2] = "two";
    myMap[3] = "three";

    int key = 2;
```

```
// 使用 count 函数检查键是否存在
if (myMap.count(key)) {
    std::cout << "key " << key << " exists in the map with value: " <<
myMap[key] << std::endl;
} else {
    std::cout << "key " << key << " does not exist in the map." <<
std::endl;
}

return 0;
}
```

```
key 2 exists in the map with value: two
```

容器：unordered_map简介

`unordered_map` 是 C++ 标准库中的一个关联容器，用于存储键值对。它基于哈希表实现，提供了平均常数时间复杂度的插入、删除和查找操作。

特性

1. **无序存储**：元素存储在哈希表中，因此没有特定的顺序。
2. **唯一键**：每个键在容器中是唯一的。
3. **高效操作**：插入、删除和查找操作的平均时间复杂度为 $O(1)$ ，最坏情况下为 $O(n)$ ，但这种情况很少发生。

常用操作

- **插入元素**：使用 `insert` 方法或 `[]` 运算符。
- **删除元素**：使用 `erase` 方法。
- **查找元素**：使用 `find` 方法或 `count` 方法。
- **访问元素**：使用 `[]` 运算符。

示例

```
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<int, std::string> myMap;

    // 插入元素
    myMap[1] = "one";
    myMap[2] = "two";
    myMap.insert({3, "three"});

    // 访问元素
    std::cout << "key 1: " << myMap[1] << std::endl;
    std::cout << "key 2: " << myMap[2] << std::endl;

    // 查找元素
    auto it = myMap.find(3);
```

```

//find函数用来返回一个迭代器，如果找到就返回特定键值的迭代器，如果没有找到就返回对应的.end()迭代器
if (it != myMap.end()) {
    std::cout << "Found key 3 with value: " << it->second << std::endl;
} else {
    std::cout << "Key 3 not found." << std::endl;
}

// 删除元素
myMap.erase(2);

// 检查元素是否存在
if (myMap.count(2)) {
    std::cout << "Key 2 exists in the map." << std::endl;
} else {
    std::cout << "Key 2 does not exist in the map." << std::endl;
}

return 0;
}

```

输出：

```

Key 1: one
Key 2: two
Found key 3 with value: three
Key 2 does not exist in the map.

```

思路2：滑动窗口

根据题目要求，并不是所有的题目的数据都要用一个数组进行储存再查找的过程。

思路：设置滑动窗口的双枚举i, j

```

class Solution {
public:
    bool containsNearbyDuplicate(vector<int>& nums, int k) {
        for(int i=0;i<nums.size();i++){
            for(int j=i+1;j<=i+k&& j<nums.size();j++){
                if(nums[i]==nums[j]){
                    return true;
                }
            }
        }
        return false;
    }
};

```

时间复杂度：O(Kn) 空间复杂度O(1)

算法优化：

考虑数组 `nums` 中的每个长度不超过 `k+1` 的滑动窗口，同一个滑动窗口中的任意两个下标差的绝对值不超过 `k`。如果存在一个滑动窗口，其中有重复元素，则**存在两个不同的下标 `i` 和 `j` 满足 `nums[i]=nums[j]` 且 `|i-j|≤k`**。如果所有滑动窗口中都没有重复元素，则不存在符合要求的下标。因此，只要**遍历每个滑动窗口，判断滑动窗口中是否有重复元素即可**。

如果一个滑动窗口的结束下标是 `i`，则该滑动窗口的开始下标是 `max(0,i-k)`。可以使用哈希集合存储滑动窗口中的元素。从左到右遍历数组 `nums`，当遍历到下标 `i` 时，具体操作如下：

如果 `i>k`，则下标 `i-k-1` 处的元素被移出滑动窗口，因此将 `nums[i-k-1]` 从哈希集合中删除；

判断 `nums[i]` 是否在哈希集合中，如果在哈希集合中则在同一个滑动窗口中有重复元素，返回 `true`，如果不在哈希集合中则将其加入哈希集合。

当遍历结束时，如果所有滑动窗口中都没有重复元素，返回 `false`。

```
class Solution {
public:
    bool containsNearbyDuplicate(vector<int>& nums, int k) {
        unordered_set<int> s;
        //s就是滑动窗口，用一个unordered_set存储，可以减少一个for循环，降低时间复杂度
        int length = nums.size();
        for (int i = 0; i < length; i++) {
            if (i > k) {
                s.erase(nums[i - k - 1]);
            }
            //当i>k时，每次执行i++时都需要将滑动窗口的最后一个元素驱逐出窗口
            if (s.count(nums[i])) {
                return true;
            }
            s.emplace(nums[i]);
            //在每一次i++前，都会插入一个新元素，之后会执行命令count，判断新插入的元素是否已经存在在滑动窗口中，如果是，直接return true
        }
        return false;
    }
};
```

时间复杂度：O(n)

emplace函数

`emplace` 是 C++11 引入的一个成员函数，用于在容器中原地构造元素。对于 `unordered_map`，`emplace` 可以避免不必要的临时对象创建和复制，从而提高性能。`emplace` 方法接受键和值的构造参数，并直接在容器中构造元素。

`emplace` 的用法

`emplace` 的函数签名如下：

```
template <class... Args>
std::pair<iterator, bool> emplace(Args&&... args);
```

- **参数：**`args` 是传递给元素构造函数的参数。

- **返回值**: 返回一个 `std::pair`, 其中第一个元素是指向插入元素的迭代器, 第二个元素是一个布尔值, 表示插入是否成功。

示例

以下示例展示了如何使用 `emplace` 在 `unordered_map` 中插入元素:

```
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<int, std::string> myMap;

    // 使用 emplace 插入元素
    myMap.emplace(1, "one");
    myMap.emplace(2, "two");

    // 打印初始内容
    std::cout << "Initial map:" << std::endl;
    for (const auto& pair : myMap) {
        std::cout << "Key: " << pair.first << ", Value: " << pair.second <<
std::endl;
    }

    // 尝试使用 emplace 插入相同键
    auto result = myMap.emplace(1, "uno");

    if (result.second) {
        std::cout << "Emplace succeeded." << std::endl;
    } else {
        std::cout << "Emplace failed. Key 1 already exists with value: " <<
myMap[1] << std::endl;
    }

    // 打印更新后的内容
    std::cout << "Updated map:" << std::endl;
    for (const auto& pair : myMap) {
        std::cout << "Key: " << pair.first << ", Value: " << pair.second <<
std::endl;
    }

    return 0;
}
```

输出:

```
Initial map:
Key: 1, Value: one
Key: 2, Value: two
Emplace failed. Key 1 already exists with value: one
Updated map:
Key: 1, Value: one
Key: 2, Value: two
```

`emplace` 与 `insert` 的区别

- `insert`: 需要先构造一个临时对象, 然后再插入到容器中, 可能会有额外的复制或移动操作。
- `emplace`: 直接在容器中构造对象, 避免了不必要的临时对象创建和复制操作, 提高了性能。

总结

- `emplace` 方法用于在 `unordered_map` 中原地构造元素, 避免不必要的临时对象创建和复制操作。
- 如果插入的键已经存在, `emplace` 操作将失败, 原值保持不变。