

---

# 第七课 哈希表与字符串

林沐

# 内容概述

---

## 1.6道经典**哈希表与字符串**的相关题目

预备知识:哈希表基础知识

例1:最长回文串(easy)(字符哈希)

例2:词语模式(easy)(字符串哈希)

例3:同字符词语分组(medium)(数组哈希)

例4:无重复字符的最长子串(medium)(字符哈希)

例5:重复的DNA序列(medium)(字符串哈希)

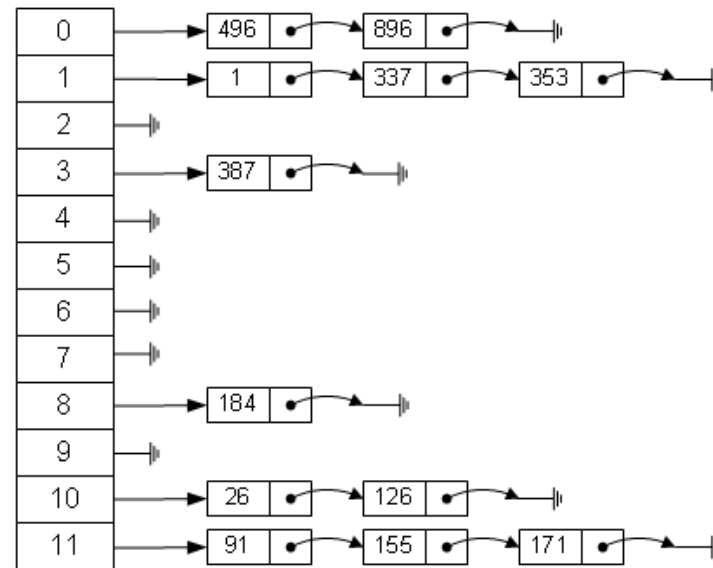
例6:最小窗口子串(hard)(哈希维护窗口)

## 2.详细讲解题目**解题方法、代码实现**

# 预备知识:哈希表定义

**哈希表**(Hash table, 也叫**散列表**), 是根据**关键字值(key)**直接进行访问的数据结构, 它通过把关键字值**映射**到表中一个位置(数组下标)来**直接访问**, 以加快查找**关键字值**的速度。这个映射函数叫做**哈希(散列)函数**, 存放记录的数组**叫做哈希(散列)表**。

给定**表M**, 存在函数 $f(key)$ , 对**任意的关键字值key**, 代入函数后若能得到包含该关键字的表中地址, 称表M为**哈希(Hash)表**, 函数 $f(key)$ 为**哈希(Hash)函数**。



# 预备知识:最简单的哈希-字符哈希

```
#include <stdio.h>
#include <string>    //ASC2码 从0 - 127 , 故使用数组
                        下标做映射 , 最大范围至128

int main() {
    int char_map[128] = {0};
    std::string str = "abcdefgaaxxy";
    //统计字符串中 , 各个字符的数量

    for (int i = 0; i < str.length(); i++) {
        char_map[str[i]]++;
        char_map['a']++;
        即char_map[97]++;
        char_map['b']++;
        即char_map[98]++;
        ...
    }

    for (int i = 0; i < 128; i++) {
        if (char_map[i] > 0) {
            printf("[%c][%d] : %d\n", i, i, char_map[i]);
        }
    }
    return 0;
}
```

[a]	[97]	:	3
[b]	[98]	:	1
[c]	[99]	:	1
[d]	[100]	:	1
[e]	[101]	:	1
[f]	[102]	:	1
[g]	[103]	:	1
[x]	[120]	:	2
[y]	[121]	:	1

# 预备知识:哈希表排序整数

```
//哈希表排序，使用数组的下标对正整数排序
//哈希表表的长度，需要超过最大待排序数字

#include <stdio.h>

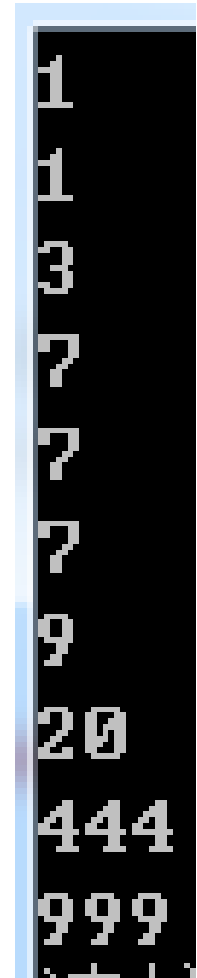
int main() {
    int random[10] = {999, 1, 444, 7, 20, 9, 1, 3, 7, 7};
    int hash_map[1000] = {0};
    for (int i = 0; i < 10; i++) {
        hash_map[random[i]]++;
    }
    for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < hash_map[i]; j++) {
            printf("%d\n", i);
        }
    }
    return 0;
}
```

**//时间复杂度 $O(\text{表长}+n)$   $n$ 为元素个数**

**hash\_map[999]++;**

**hash\_map[1]++;**

**...**



# 预备知识:问题引入1, 任意元素的映射

---

1.当遇到**负数或非常大的整数**, 如何进行**哈希(映射)**?

如: -5、99999999、...

2.当遇到**字符串**, 如何进行**哈希(映射)**?

如: abcdefg、XYZ、...

3.当遇其他到无法直接映射的数据类型, 如**浮点数、数组、对象**等等, 如何进行哈希(映射)?

如: 1.2345、[1, 2, 3]、...

**解决:**

利用**哈希函数**, 将**关键字值**(key)(大整数、字符串、浮点数等)**转换**为整数再对表长**取余**, 从而关键字值被转换为哈希表的**表长范围内**的整数。

---

# 预备知识:问题引入2, 发生冲突

```
#include <stdio.h>
#include <string>

//直接对整数取余表长再返回
int int_func(int key, int table_len){
    return key % table_len;
}

//将字符串中的字符的ASC2码相加得到整数再取余表长
int string_func(std::string key, int table_len){
    int sum = 0;
    for (int i = 0; i < key.length(); i++){
        sum += key[i];
    }
    return sum % table_len;
}

//不同的整数或字符串, 由于哈希函数的
//选择, 被映射到了同一个下标处!
//产生了冲突!

int main(){
    const int TABLE_LEN = 10;
    int hash_map[TABLE_LEN] = {0};
    hash_map[int_func(99999995, TABLE_LEN)]++;
    hash_map[int_func(5, TABLE_LEN)]++;
    hash_map[string_func("abc", TABLE_LEN)]++;
    hash_map[string_func("bac", TABLE_LEN)]++;
    for (int i = 0; i < TABLE_LEN; i++){
        printf("hash_map[%d] = %d\n", i, hash_map[i]);
    }
    return 0;
}
```

```
hash_map[0] = 0
hash_map[1] = 0
hash_map[2] = 0
hash_map[3] = 0
hash_map[4] = 2
hash_map[5] = 2
hash_map[6] = 0
hash_map[7] = 0
hash_map[8] = 0
hash_map[9] = 0
```

# 预备知识:拉链法解决冲突，构造哈希表

将所有哈希函数**结果相同**的结点连接在**同一个**单链表中。

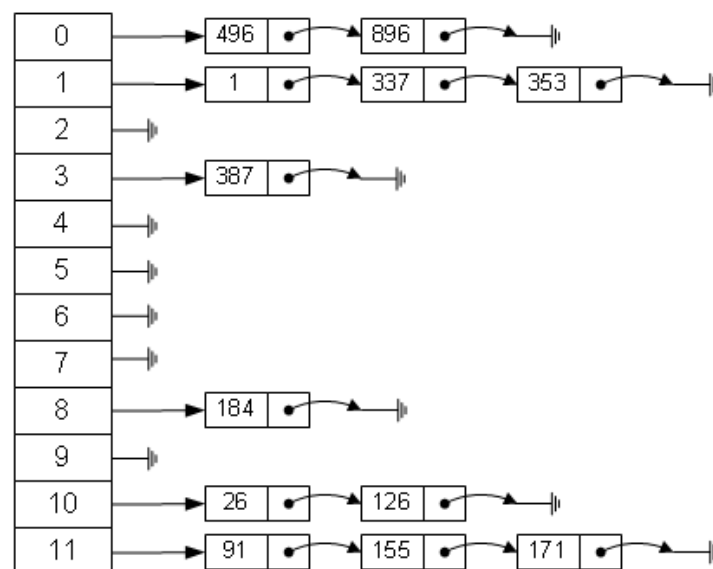
若选定的哈希表**长度**为 $m$ ，则可将哈希表定义为一个长度为 $m$ 的**指针数组** $t[0..m-1]$ ，指针数组中的每个指针指向哈希函数**结果相同**的**单链表**。

**插入value:**

将元素 $value$ 插入哈希表，若元素 $value$ 的**哈希函数值**为 $hash\_key$ ，将 $value$ 对应的节点以**头插法**的方式插入到以 $t[hash\_key]$ 为**头指针**的单链表中。

**查找value:**

若元素 $value$ 的**哈希函数值**为 $hash\_key$ ，**遍历**以 $t[hash\_key]$ 为头指针的单链表，查找链表各个节点的**值域**是否为 $value$ 。





# 预备知识:课堂练习

```
struct ListNode { //哈希表即为普通的单链表构成
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
```

```
int hash_func(int key, int table_len){
    return key % table_len; //整数哈希函数，直接取余
}
```

```
void insert(ListNode *hash_table[], ListNode *node, int table_len){
    int hash_key = hash_func(node->val, table_len);
    node->next = 1 //使用头插法插入节点
    hash_table[hash_key] = 2
}
```

```
bool search(ListNode *hash_table[], int value, int table_len){
    int hash_key = hash_func(value, table_len);
    ListNode *head = 3
    while(head){
        if ( 4 ){
            return true;
        }
        5
    }
    return false;
}
```

**3分钟**填写代码，  
**有问题随时提出！**

# 预备知识:实现

```
struct ListNode {    //哈希表即为普通的单链表构成
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
```

```
int hash_func(int key, int table_len){
    return key % table_len; //整数哈希函数，直接取余
}
```

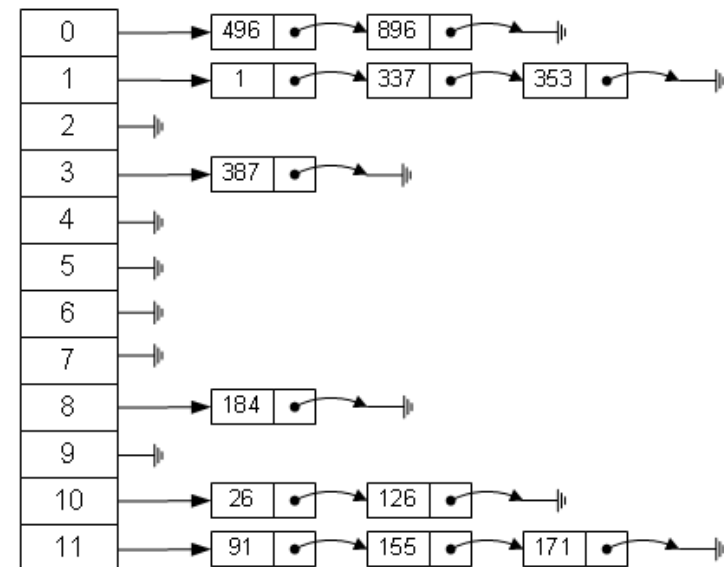
```
void insert(ListNode *hash_table[], ListNode *node, int table_len){
    int hash_key = hash_func(node->val, table_len);

    node->next = hash_table[hash_key]; //使用头插法插入节点

    hash_table[hash_key] = node;
}
```

```
bool search(ListNode *hash_table[], int value, int table_len){
    int hash_key = hash_func(value, table_len);
    ListNode *head = hash_table[hash_key];

    while(head){
        if (head->val == value){
            return true;
        }
        head = head->next;
    }
    return false;
}
```



```

int main() { //TABLE_LEN 取为质数，冲突会比其他数字少
    const int TABLE_LEN = 11;
    ListNode *hash_table[TABLE_LEN] = {0};
    std::vector<ListNode *> hash_node_vec;
    int test[8] = {1, 1, 4, 9, 20, 30, 150, 500};
    for (int i = 0; i < 8; i++){
        hash_node_vec.push_back(new ListNode(test[i]));
    }
    for (int i = 0; i < hash_node_vec.size(); i++){
        insert(hash_table, hash_node_vec[i], TABLE_LEN);
    }
    printf("Hash table:\n");
    for (int i = 0; i < TABLE_LEN; i++){
        printf("[%d]:", i);
        ListNode *head = hash_table[i];
        while(head){
            printf("->%d", head->val);
            head = head->next;
        }
        printf("\n");
    }
    printf("\n");
    printf("Test search:\n");
    for (int i = 0; i < 10; i++){
        if (search(hash_table, i, TABLE_LEN)){
            printf("%d is in the hash table.\n");
        }
        else{
            printf("%d is not in the hash table.\n");
        }
    }
    return 0;
}

```

## 预备知识:测试

```

Hash table:
[0]:
[1]:->1->1
[2]:
[3]:
[4]:->4
[5]:->500
[6]:
[7]:->150
[8]:->30
[9]:->20->9
[10]:

Test search:
0 is not in the hash table.
1 is in the hash table.
2 is not in the hash table.
3 is not in the hash table.
4 is in the hash table.
5 is not in the hash table.
6 is not in the hash table.
7 is not in the hash table.
8 is not in the hash table.
9 is in the hash table.
请按任意键继续. . .

```

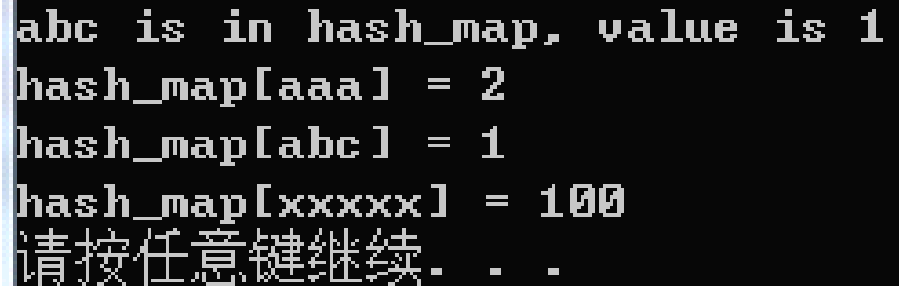
# 预备知识:哈希map与STL map

---

```
#include <stdio.h>
#include <map>
#include <string>

struct ListNode {
    std::string key;
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

int main() {
    //将字符串key映射为整数val
    std::map<std::string, int> hash_map;
    std::string str1 = "abc";
    std::string str2 = "aaa";
    std::string str3 = "xxxxx";
    hash_map[str1] = 1;
    hash_map[str2] = 2;
    hash_map[str3] = 100;
    if (hash_map.find(str1) != hash_map.end()) {
        printf("%s is in hash_map, value is %d\n",
            str1.c_str(), hash_map[str1]);
    }
    std::map<std::string, int> ::iterator it;
    for (it = hash_map.begin(); it != hash_map.end(); it++) {
        printf("hash_map[%s] = %d\n", it->first.c_str(), it->second);
    }
    return 0;
}
```



```
abc is in hash_map, value is 1
hash_map[aaa] = 2
hash_map[abc] = 1
hash_map[xxxxx] = 100
请按任意键继续. . .
```

# 例1:最长回文串

---

已知一个只包括大小写字符的**字符串**，求用**该字符串中的字符**可以生成的**最长回文字符串**长度。

例如  $s = \text{"abccccddaa"}$ ，可生成的**最长回文**字符串长度为9，如dccaaaccd、adccbccda、acdcacdca等，都是**正确**的。

```
class Solution {  
public:  
    int longestPalindrome(std::string s) {  
    }  
};
```

选自 **LeetCode 409. Longest Palindrome**

<https://leetcode.com/problems/longest-palindrome/description/>

难度:**Easy**

# 例1:思考

---

例如在s = “abccccddaa”中，有3个a，1个b，4个c，2个d。

使用字符串s中的字符，**任意组合**，生成新的字符串，若生成的字符串为回文字符串，需要除了**中心**字符，其余字符**只要头部出现，尾部就要对应出现**。

**例如：**

a...a、ccd...dcc、cc...d...cc

**思考：**

在字符串中，字符数量为**偶数**的字符，我们应该如何处理？

在字符串中，字符数量为**奇数**的字符，我们应该如何处理？

# 例1:分析

---

例如，在s = “abccccddaa”中，有3个a，1个b，4个c，2个d。

1.在字符串中，字符数量为**偶数**的字符：

**全部使用**，**头部**放一个字符，**尾部**就对应放一个。

如，4个c和2个d可**全部**用上：

...ccd...dcc...、...cdc...cdc...、...dcc...ccd...等。

2.在字符串中，字符数量为**奇数**的字符：

**丢掉**一个字符，**剩下的字符数**为偶数个，按照字符数量为**偶数**的字符处理。

如，3个a中，有2个a可以用上：...a...a...

3.若有**剩余的**字符，如：

1个a、1个b

**随便**选择1个字符当作中心字符：...a...、...b...

# 例1:算法思路

---

- 1.利用**字符哈希**方法，**统计**字符串中所有的**字符数量**；
- 2.设置最长回文串**偶数字符**长度为 $\text{max\_length} = 0$ ；
- 3.设置是否有**中心点**标记  $\text{flag} = 0$ ；
- 4.**遍历**每一个字符，**字符数**为 $\text{count}$ ，若 $\text{count}$ 为**偶数**， $\text{max\_length} += \text{count}$ ；若 $\text{count}$ 为**奇数**， $\text{max\_length} += \text{count} - 1$ ， $\text{flag} = 1$ ；
- 5.最终最长回文子串长度： $\text{max\_length} + \text{flag}$ 。

例如，在 $s = \text{"abccccddaa"}$ 中，有3个a，1个b，4个c，2个d：

1.3个a， $\text{max\_length} += 2$ ； $\text{flag} = 1$ ；如**生成**aa；

2.1个b， $\text{max\_length} += 0$ ；忽略b；

3.4个c， $\text{max\_length} += 4$ ；如**生成**ccaacc；

4.2个d， $\text{max\_length} += 2$ ；如**生成**dcceaacd；

$\text{flag} = 1$ ；

故可生成如：dcceaacd、dccabaccd

**最终长度**： $\text{max\_length} + \text{flag} = 8 + 1 = 9$



# 例1:课堂练习

```
class Solution {
public:
    int longestPalindrome(std::string s) {
        int char_map[128] = {0}; //字符哈希
        int max_length = 0; //回文串偶数部分的最大长度
        int flag = 0; //是否有中心点
        for (int i = 0; i < s.length(); i++) {
            1
        }
        for (int i = 0; i < 128; i++) {
            if (2) {
                max_length += char_map[i];
            }
            else {
                3
                flag = 1;
            }
        }
        return max_length + flag;
    }
}; //最终结果是偶数部分的最大长度+中心点字符
```

3分钟填写代码，  
有问题随时提出！

# 例1:实现

```
class Solution {
public:
    int longestPalindrome(std::string s) {
        int char_map[128] = {0}; //字符哈希
        int max_length = 0; //回文串偶数部分的最大长度
        int flag = 0; //是否有中心点
        for (int i = 0; i < s.length(); i++) {
            char_map[s[i]]++; //利用整数的数组下标实现字符哈希
                                //统计字符个数
        }
        for (int i = 0; i < 128; i++) {
            if (char_map[i] % 2 == 0) { //如果某字符为偶数个，
                                        //则均可以使用在回文串里
                max_length += char_map[i];
            }
            else {
                max_length += char_map[i] - 1; //如果某字符为奇数个，
                                                //则丢弃1个，其余的使用在回文串里
                flag = 1; //此时标记回文串可以有中心点
            }
        }
        return max_length + flag;
    };
    //最终结果是偶数部分的最大长度+中心点字符
};
```

# 例1:测试与leetcode提交结果

---

```
int main() {  
    std::string s = "abccccddaa";  
    Solution solve;  
    printf("%d\n", solve.longestPalindrome(s));  
    return 0;  
}
```

Longest Palindrome

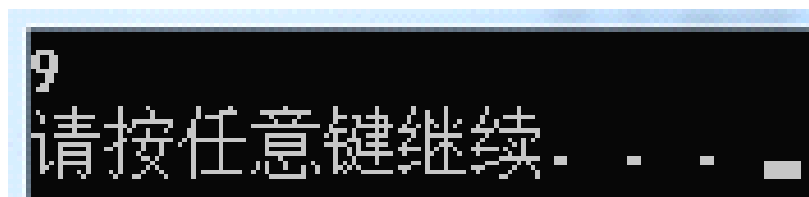
Submission Details

95 / 95 test cases passed.

Status: **Accepted**

Runtime: 6 ms

Submitted: 19 minutes ago



## 例2:词语模式

已知字符串pattern与字符串str，**确认**str是否与pattern**匹配**。str与pattern匹配代表字符串str中的单词与pattern中的字符**一一对应**。(其中pattern中只包含**小写字符**，str中的单词只包含小写字符，使用**空格分隔**。)

例如，

pattern = "abba", str = "dog cat cat dog" 匹配.

pattern = "abba", str = "dog cat cat fish" 不匹配.

pattern = "aaaa", str = "dog cat cat dog" 不匹配.

pattern = "abba", str = "dog dog dog dog" 不匹配.

```
class Solution {  
public:  
    bool wordPattern(std::string pattern, std::string str) {  
    }  
};
```

选自 **LeetCode 290. Word Pattern**

<https://leetcode.com/problems/word-pattern/description/>

难度:**Easy**

## 例2:思考与分析

**匹配**:字符串str中的单词与pattern中的字符**一一对应**。

如:

- 1)假设pattern = “abba”, str = “dog cat cat \*”; 则\*一定代表dog。
- 2)假设pattern = “abb?”, str = “dog cat cat dog”; 则?一定代表a。
- 3)假设pattern = “abb?”, str = “dog cat cat fish”; 则?一定不是a或b。
- 4)假设pattern = “abba”, str = “\*”; \*代表了4个单词。
- 5)假设pattern = “\*”, str = “dog cat cat dog”; \*代表了4个字符。

### 分析结论:

- 1.当**拆解**出一个单词时,若该单词**已出现**,则当前单词对应的pattern字符必为该单词**曾经对应**的pattern字符。
- 2.当**拆解**出一个单词时,若该单词**未曾出现**,则当前单词对应的pattern字符也必须**未曾出现**。
- 3.单词的个数与pattern字符串中的字符**数量相同**。

## 例2:算法思路

---

**pattern = “abb?”, str = “dog cat cat \*”;**

**dog -> a; cat->b**

1.设置单词(字符串)到pattern字符的**映射(哈希)**; 使用数组used[128]记录pattern字符是否使用。

2.遍历str, 按照空格**拆分单词**, 同时**对应的**向前移动指向pattern字符的指针, 每拆分出一个单词, **判断**:

如果该单词**从未出现**在哈希表中:

如果当前的pattern字符**已被使用**, 则返回false;

将单词与当前指向的pattern字符**做映射**;

**标记**当前指向的pattern字符**已使用**。

否则:

如果当前单词在哈希表中的**映射字符**不是当前指向的pattern字符, 则返回false。

3.若单词个数与pattern字符个数**不匹配**, 返回false。

# 例2:算法思路

---

图1:

str = "dog cat cat fish"



pattern = "abba"



创建: dog -> a

used = [a]

图2:

str = "dog cat cat fish"



pattern = "abba"



创建: dog -> a

cat -> b

used = [a, b]

图3:

str = "dog cat cat fish"



pattern = "abba"



查询: cat -> b == 当前pattern字符

图4:

str = "dog cat cat fish"



pattern = "abba"



创建: fish -> a

由于 used = [a, b]

返回false

# 例2:算法思路

图1:

str = "dog cat cat fish"



pattern = "abca"



创建: dog -> a  
used = [a]

图2:

str = "dog cat cat fish"



pattern = "abca"



创建: dog -> a  
cat -> b  
used = [a, b]

图3:

str = "dog cat cat fish"



pattern = "abca"



查询: cat -> b != 当前pattern字符 c  
返回false

str的单词数与pattern字符长度不一致:

str = "dog cat cat fish"



pattern = "ab "



str = "dog "



pattern = "abca"





## 例2:课堂练习

```
class Solution {
public:
    bool wordPattern(std::string pattern, std::string str) {
        std::map<std::string, char> word_map; //单词到pattern字符的映射
        char used[128] = {0}; //已被映射的pattern字符
        std::string word; //临时保存拆分出的单词
        int pos = 0; //当前指向的pattern字符 (无需特殊处理最后一个单词了)
        str.push_back(' '); //str尾部push一个空格, 使得遇到空格拆分单词
        for (int i = 0; i < str.length(); i++){
            if (str[i] == ' '){ //遇到空格, 即拆分出一个新单词
                if (pos == pattern.length()){
                    1
                }
                //若单词未出现在哈希映射中
                if (word_map.find(word) == word_map.end()){
                    if (2){
                        return false;
                    }
                    3
                    used[pattern[pos]] = 1;
                }
                else{
                    if (4){
                        return false;
                    }
                }
                word = ""; //完成一个单词的插入和查询后, 清空word
                pos++; //指向pattern字符的pos指针前移
            }
            else{
                word += str[i];
            }
        }
        if (5){
            return false;
        }
        return true;
    }
};
```

3分钟填写代码,  
有问题随时提出!

## 例2:实现

```
class Solution {
public:
    bool wordPattern(std::string pattern, std::string str) {
        std::map<std::string, char> word_map; //单词到pattern字符的映射
        char used[128] = {0}; //已被映射的pattern字符
        std::string word; //临时保存拆分出的单词
        int pos = 0; //当前指向的pattern字符 (无需特殊处理最后一个单词了)
        str.push_back(' '); //str尾部push一个空格,使得遇到空格拆分单词

        for (int i = 0; i < str.length(); i++) {
            if (str[i] == ' ') { //遇到空格,即拆分出一个新单词
                if (pos == pattern.length()) { //若分隔出一个单词,
                    return false; //但已无pattern字符对应
                }
                //若单词未出现在哈希映射中
                if (word_map.find(word) == word_map.end()) {
                    if (used[pattern[pos]]) { //如果当前pattern字符已使用
                        return false;
                    }
                    word_map[word] = pattern[pos];
                    used[pattern[pos]] = 1;
                }
                else {
                    if (word_map[word] != pattern[pos]) { //若当前word已建立映射
                        return false; //无法与当前pattern对应
                    }
                }
                word = ""; //完成一个单词的插入和查询后,清空word
                pos++; //指向pattern字符的pos指针前移
            }
            else {
                word += str[i];
            }
        }
        if (pos != pattern.length()) {
            return false; //有多余的pattern字符
        }
        return true;
    }
};
```

str = "dog cat cat fish"

pattern = "ab"

str = "dog cat cat fish"

pattern = "abba"

str = "dog cat cat fish"

pattern = "abca"

str = "dog "

pattern = "abca"

# 例2:测试与leetcode提交结果

---

```
int main() {  
    std::string pattern = "abba";  
    std::string str = "dog cat cat dog";  
    Solution solve;  
    printf("%d\n", solve.wordPattern(pattern, str));  
    return 0;  
}
```

Word Pattern

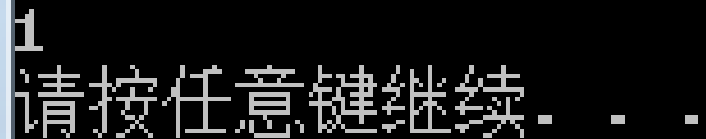
Submission Details

33 / 33 test cases passed.

Status: **Accepted**

Runtime: 0 ms

Submitted: 0 minutes ago



1  
请按任意键继续. . .

## 例3:同字符词语分组

---

已知一组**字符串**，将所有**anagram**(由颠倒字母顺序而构成的字)**放到一起**输出。

例如:["eat", "tea", "tan", "ate", "nat", "bat"]

返回:[ ["ate", "eat", "tea"], ["nat", "tan"], ["bat"] ]

```
class Solution {
public:
    std::vector<std::vector<std::string> > groupAnagrams (
        std::vector<std::string>& strs) {
    }
};
```

选自 **LeetCode 49. Group Anagrams**

<https://leetcode.com/problems/group-anagrams/description/>

难度:**Medium**

## 例3:思考

---

**anagram分组**: 若某两个字符串, 出现的各个字符数相同, 则它们应该为同一分组。

例如: `["eat", "tea", "tan", "ate", "nat", "bat"]`

返回: `[ ["ate", "eat", "tea"], ["nat", "tan"], ["bat"] ]`

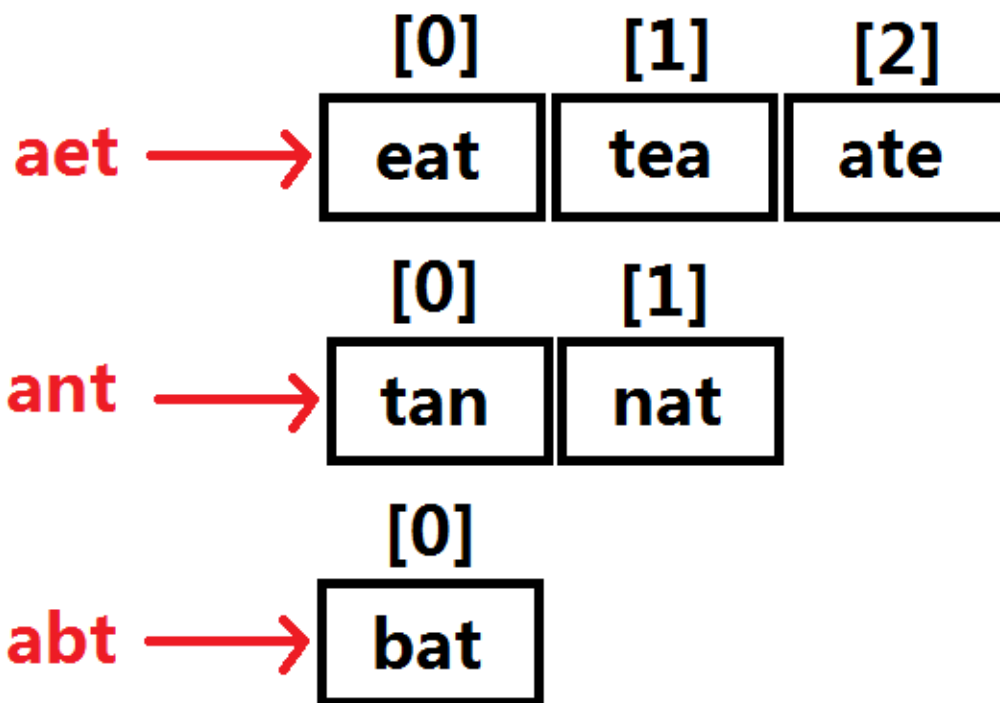
**思考:**

如何建立**哈希表**, 怎样设计哈希表的**key**与**value**, 就可将**各个字符数相同**的字符串映射到一起?

# 例3:方法1设计

**哈希表**以内部进行排序的各个单词为 **key**，以 **字符串向量** (vector<string>)为 **value**，存储各个字符**数量相同**的字符串(anagram)。

**["eat", "tea", "tan", "ate", "nat", "bat"]**



# 例3:方法1算法思路

设置**字符串到字符串向量**的哈希表anagram，遍历字符串向量strs中的单词strs[i]:

- 1)设置临时变量str = strs[i]，对str进行**排序**。
- 2)若str**未出现**在anagram中，设置str到一个**空字符串向量**的**映射**。
- 3)将strs[i]**添加**至字符串向量anagram[str]中。

**遍历**哈希表anagram，将全部key对应的value push至最终结果中。

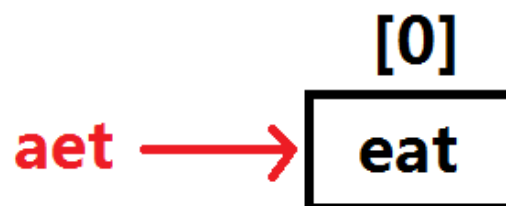
**图1:**

["eat", "tea", "tan", "ate", "nat", "bat"]



i=0

哈希表anagram:



# 例3:方法1算法思路

图2:

["eat", "tea", "tan", ...]

↑ i=1

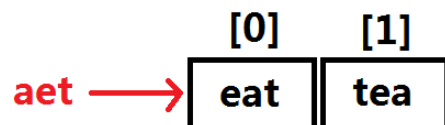


图4:

["eat", "tea", "tan", "ate", "nat", "bat"]

↑ i=3

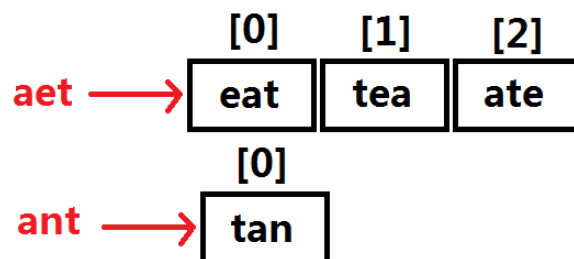


图6:

[ ... , "ate", "nat", "bat"]

i=5 ↑

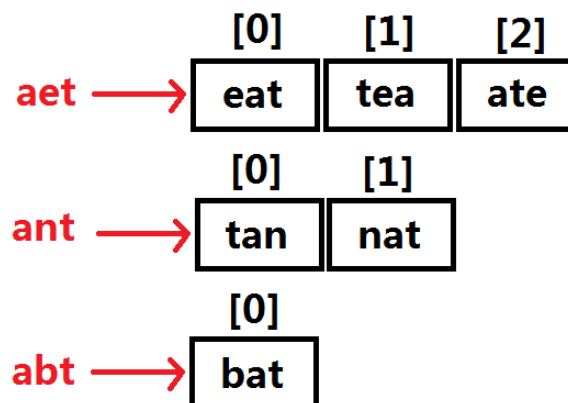


图3:

["eat", "tea", "tan", ...]

↑ i=2

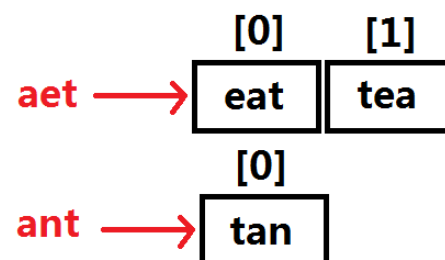
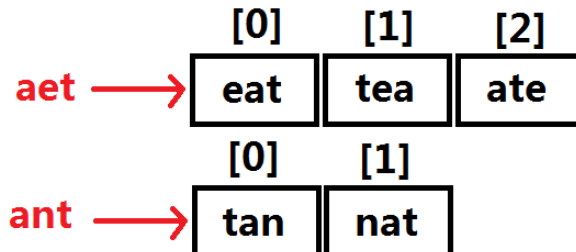


图5:

["eat", "tea", "tan", "ate", "nat", "bat"]

↑ i=4





# 例3:方法1课堂练习

```
class Solution {
public:
    std::vector<std::vector<std::string> > groupAnagrams (
        std::vector<std::string>& strs) {
        std::map<std::string, std::vector<std::string> > anagram;
        //内部进行排序的各个单词为key，以字符串向量(vector<string>)为
        //value，存储各个字符数量相同的字符串(anagram)
        std::vector<std::vector<std::string> > result; //存储最终的结果
        for (int i = 0; i < strs.size(); i++) { //遍历各个单词
            std::string str = 1
            std::sort(str.begin(), str.end()); //对str内部进行排序
            if (anagram.find(str) == anagram.end()) { //若无法在哈希表中找到str
                std::vector<std::string> item; //设置一个空的字符串向量
                2
            }
            3
        }
        std::map<std::string, std::vector<std::string> > ::iterator it;
        for (it = anagram.begin(); it != anagram.end(); it++) {
            result.push_back((*it).second);
        }
        return result; //遍历哈希表，将哈希表的value push进入最终结果
    }
};
```

**3分钟**填写代码，  
**有问题随时提出！**

## 例3:方法1实现

```
class Solution {
public:
    std::vector<std::vector<std::string> > groupAnagrams (
        std::vector<std::string>& strs) {

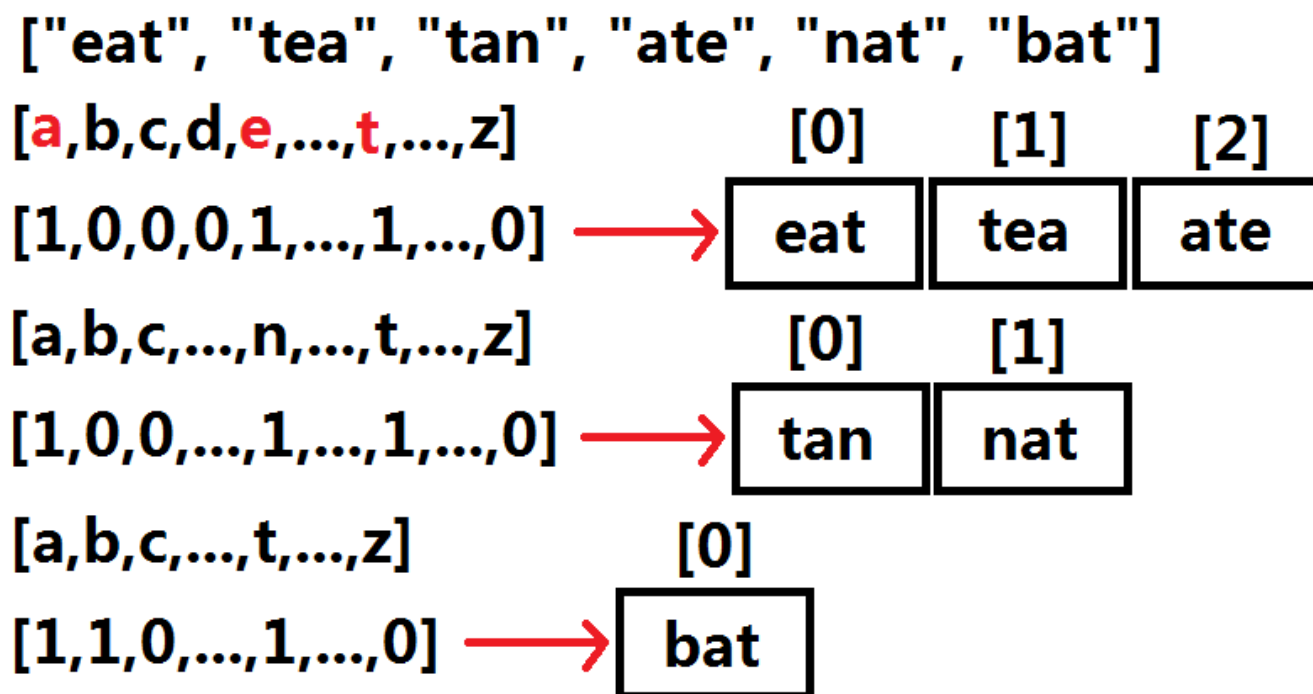
        std::map<std::string, std::vector<std::string> > anagram;
        //内部进行排序的各个单词为key，以字符串向量(vector<string>)为
        //value，存储各个字符数量相同的字符串(anagram)
        std::vector<std::vector<std::string> > result; //存储最终的结果
        for (int i = 0; i < strs.size(); i++) { //遍历各个单词

            std::string str = strs[i];

            std::sort(str.begin(), str.end()); //对str内部进行排序
            if (anagram.find(str) == anagram.end()) { //若无法在哈希表中找到str
                std::vector<std::string> item; //设置一个空的字符串向量
                anagram[str] = item; //以排序后的strs[i]作为key
            }
            anagram[str].push_back(strs[i]); //在对应的字符串向量中push结果
        }
        std::map<std::string, std::vector<std::string> > ::iterator it;
        for (it = anagram.begin(); it != anagram.end(); it++) {
            result.push_back((*it).second);
        }
        return result; //遍历哈希表，将哈希表的value push进入最终结果
    }
};
```

# 例3:方法2设计

**哈希表**以26个字母的字符数量(一个长度为26的vector, 统计单词中各个字符的数量)为**key**, 以**字符串向量**(vector<string>)为**value**, 存储各个字符**数量相同**的字符串(anagram)。



# 例3:方法2算法思路

设置 **vector到字符串向量** 的哈希表 `anagram`，遍历字符串向量 `strs` 中的单词 `strs[i]`:

- 1) 统计 `strs[i]` 中的 **各个字符数量**，存储至 `vec`。
- 2) 若 `vec` **未出现** 在 `anagram` 中，设置 `vec` 到一个 **空字符串向量** 的 **映射**。
- 3) 将 `strs[i]` **添加** 至字符串向量 `anagram[vec]` 中。

**遍历** 哈希表 `anagram`，将全部 `key` 对应的 `value` `push` 至最终结果中。

`["eat", "tea", "tan", "ate", "nat", "bat"]`

↑ `i=0`

哈希表 `anagram`:

`[a,b,c,d,e,...,t,...,z]`

`[1,0,0,0,1,...,1,...,0]`

`[0]`

`eat`

**//将字符串str中的各个字符数量进行统计，存储至vec**

```
void change_to_vector(std::string &str, std::vector<int> &vec) {  
    for (int i = 0; i < 26; i++) {  
        vec.push_back(0);  
    }  
    for (int i = 0; i < str.length(); i++) {  
        vec[str[i]-'a']++;  
    }  
}
```

```
class Solution {  
public:  
    std::vector<std::vector<std::string> > groupAnagrams (  
        std::vector<std::string>& strs) {  
        std::map<std::vector<int>, std::vector<std::string> > anagram;  
        std::vector<std::vector<std::string> > result;  
        for (int i = 0; i < strs.size(); i++) {  
            std::vector<int> vec;  
            change_to_vector(strs[i], vec);  
            if (anagram.find(vec) == anagram.end()) {  
                std::vector<std::string> item;  
                anagram[vec] = item;  
            }  
            anagram[vec].push_back(strs[i]);  
        }  
        std::map<std::vector<int>,  
            std::vector<std::string> > ::iterator it;  
        for (it = anagram.begin(); it != anagram.end(); it++) {  
            result.push_back((*it).second);  
        }  
        return result;  
    }  
};
```

## 例3:方法2实现

**//各个单词中字符出现的  
数量的vector到字符串向  
量(存储结果)的映射**

# 例3:测试与leetcode提交结果

```
int main() {
    std::vector<std::string> strs;
    strs.push_back("eat");
    strs.push_back("tea");
    strs.push_back("tan");
    strs.push_back("ate");
    strs.push_back("nat");
    strs.push_back("bat");
    Solution solve;
    std::vector<std::vector<std::string> > result
        = solve.groupAnagrams(strs);
    for (int i = 0; i < result.size(); i++) {
        for (int j = 0; j < result[i].size(); j++) {
            printf("[%s]", result[i][j].c_str());
        }
        printf("\n");
    }
    return 0;
}
```

Group Anagrams

Submission Details

101 / 101 test cases passed.

Status: **Accepted**

Runtime: 45 ms

Submitted: 0 minutes ago

```
[bat]
[eat][tea][ate]
[tan][nat]
请按任意键继续. . .
```

# 课间休息10分钟

---

## 有问题提出！

# 例4:无重复字符的最长子串

---

已知一个**字符串**，求用**该字符串**的**无重复字符**的**最长子串**的长度。  
例如：

s = "abcabcbb" -> "abc", 3

s = "bbbbbb" -> "b", 1

s = "pwwkew" -> "wke", 3 注意 "pwke"是子序列而非子串。

```
class Solution {  
public:  
    int lengthOfLongestSubstring(std::string s) {  
    }  
};
```

选自 **LeetCode 3. Longest Substring Without Repeating Characters**

<https://leetcode.com/problems/longest-substring-without-repeating-characters/description/>

难度:**Medium**



## 例4:思考

---

**枚举**  $S = \text{"pwwkew"}$  的**所有子字符串**:

p	w	w	k	e	w
pw	ww	wk	ke	ew	
pww	wwk	wke	kew		
pwwk	wwke	wkew			
pwwke	wwkew				
pwwkew					

**检查**以上所有子字符串, 是否**满足无重复**字符的条件, 取**最长的**满足条件的子串作为结果。

思考: 该算法的**复杂度**是多少? 是否有**更好的**方法?

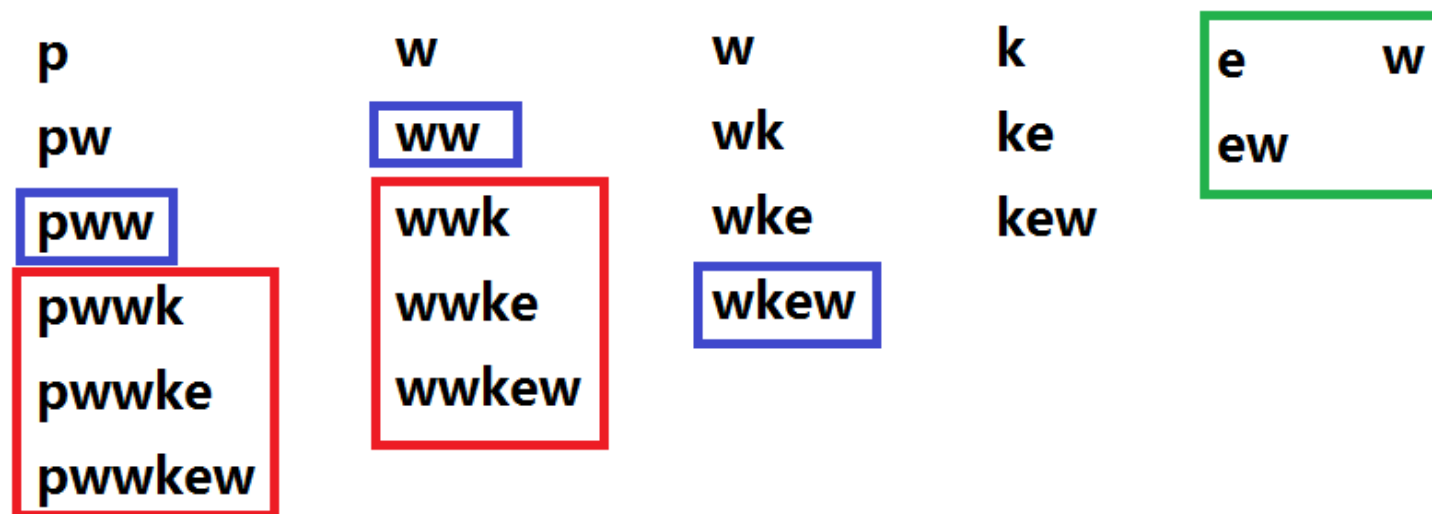
---

# 例4:分析

该题目的时间复杂度**优化目标**是从 $O(n^2)$ 优化至 $O(n)$ (中间没有优化至 $n\log n$ 的条件)。

所以需要将两重循环的**枚举**扫描**修改**为一层扫描。

题目中最关键的条件，**无重复字符**的子串，观察子串：



**红色方框**中的枚举均无意义，因为**蓝色方框**中已出现了**重复**字符。

**绿色方框**中的也无意义，因为后续的枚举不会出现**满足条件的更优**子串。

# 例4:算法思路

1. 设置一个记录字符数量的**字符哈希**, char\_map;
  2. 设置一个记录当前满足条件的**最长子串**变量 word;
  3. 设置**两个指针** (记作指针 i 与指针 begin) 指向字符串第一个字符;
  4. 设置最长满足条件的子串的**长度** result;
  5. i 指针向后**逐个扫描**字符串中的字符, 在这个过程中, 使用 char\_map 记录字符数量  
**如果** word 中没出现过该字符: 对 word 尾部**添加字符并检查** result 是否需要更新;  
**否则**: begin 指针向前移动, 更新 char\_map 中的字符数量, **直到** 字符 s[i] 的数量为 1; **更新** word, 将 word**赋值**为 begin 与 i 之间的子串。
- 在整个过程中, 使用 begin 与 i 维护一个**窗口**, 该窗口中的子串**满足题目条件**(无重复的字符), 窗口**线性向前滑动**, 整体时间复杂度为 **O(n)**。

abcbadab      abcbadab

begin ↑      ↑ i      begin ↑↑ i

# 例4:算法思路

---

图1: ↓ begin = 0

**abcbadab**

↑ i = 0

word = "a", result = 1

图2: ↓ begin = 0

**abcbadab**

↑ i = 1

word = "ab", result = 2

图3: ↓ begin = 0

**abcbadab**

↑ i = 2

word = "abc", result = 3

图4: ↓ begin = 0

**abcbadab**

↑ i = 3

word = "abc", result = 3

图5: ↓ begin = 2

**abcbadab**

↑ i = 3

word = "cb", result = 3

图6: ↓ begin = 2

**abcbadab**

↑ i = 4

word = "cba", result = 3

## 例4:算法思路

---

图7:  begin = 2

**abcbadab**

 i = 5

word = "cbad", result = 4

图8:  begin = 2

**abcbadab**

 i = 6

word = "cbad", result = 4

图9:  begin = 5

**abcbadab**

 i = 6

word = "da", result = 4

图10:  begin = 5

**abcbadab**

 i = 7

word = "dab", result = 4

## 例4:课堂练习

```
class Solution {
public:
    int lengthOfLongestSubstring(std::string s) {
        int begin = 0; //窗口的头指针
        int result = 0;
        std::string word = "";
        int char_map[128] = {0};
        for (int i = 0; i < s.length(); i++) {
            1
            if (char_map[s[i]] == 1) { //word中没出现过该字符
                2
                if (result < word.length()) {
                    result = word.length();
                }
            }
            else { //将重复的字符s[i]删去
                while (3) {
                    4
                    begin++;
                }
                word = ""; //重新更新word
                for (int j = begin; j <= i; j++) {
                    5
                }
            }
        }
        return result;
    }
};
```

5分钟填写代码，  
有问题随时提出！

# 例4:实现

```
class Solution {
public:
    int lengthOfLongestSubstring(std::string s) {
        int begin = 0; //窗口的头指针
        int result = 0;
        std::string word = "";
        int char_map[128] = {0};
        for (int i = 0; i < s.length(); i++) {
            char_map[s[i]]++;
            if (char_map[s[i]] == 1) { //word中没出现过该字符
                word += s[i];
                if (result < word.length()) {
                    result = word.length();
                }
            } else { //将重复的字符s[i]删去
                while (begin < i && char_map[s[i]] > 1) {
                    char_map[s[begin]]--;
                    begin++;
                }
                word = ""; //重新更新word
                for (int j = begin; j <= i; j++) {
                    word += s[j];
                }
            }
        }
        return result;
    }
};
```

图3: ↓ begin = 0

abcbadab  
↑ i = 2

word = "abc", result = 3

图4: ↓ begin = 0

abcbadab  
↑ i = 3

word = "abc", result = 3

图5: ↓ begin = 2

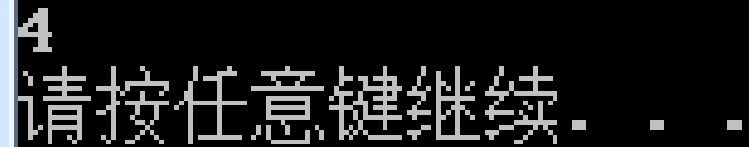
abcbadab  
↑ i = 3

word = "cb", result = 3

# 例4:测试与leetcode提交结果

---

```
int main() {  
    std::string s = "abcbadab";  
    Solution solve;  
    printf("%d\n", solve.lengthOfLongestSubstring(s));  
    return 0;  
}
```



4  
请按任意键继续. . .

Longest Substring Without Repeating Characters

## Submission Details

983 / 983 test cases passed.

Status: **Accepted**

Runtime: 29 ms

Submitted: 0 minutes ago



# 例5:重复的DNA序列

---

将**DNA序列**看作是只包含['A', 'C', 'G', 'T']4个字符的**字符串**，给一个DNA字符串，找到所有**长度为10**的且出现**超过1次**的子串。

例如：

s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",

Return: ["AAAAACCCCC", "CCCCCAAAAA"].

s = "AAAAAAAAAAAA",

Return: ["AAAAAAAAAA"].

```
class Solution {  
public:  
    std::vector<std::string> findRepeatedDnaSequences(std::string s) {  
    }  
};
```

选自 **LeetCode 187. Repeated DNA Sequences**

<https://leetcode.com/problems/repeated-dna-sequences/description/>

难度:**Medium**

---

## 例5:算法思路(方法1)

---

枚举**DNA字符串**中的所有长度为10的子串，将其插入到**哈希Map**中，并**记录子串数量**；遍历哈希map，将所有**出现超过1次**的子串存储到结果中。算法复杂度 $O(n)$ 。

`s = "AAAAACCCCCAAAAACCCCCAAAAAGGGTTT"`

AAAAACCCCC

AAAACCCCCA

AAACCCCCAA

...

AAAAAGGGTT

AAAAGGGTTT

# 例5:课堂练习(方法1)

```
class Solution {
public:
    std::vector<std::string> findRepeatedDnaSequences(std::string s) {
        std::map<std::string, int> word_map; //<单词, 单词数量>的映射
        std::vector<std::string> result;
        for (int i = 0; i < s.length(); i++) {
            std::string word = s.substr(i, 10); //若word在哈希中出现
            if (word_map.find(word) != word_map.end()) {
                1
            }
            else{
                2
            }
        }
        //遍历哈希表中的所有单词
        std::map<std::string, int> ::iterator it;
        for (it = word_map.begin(); it != word_map.end(); it++) {
            if ( 3 ) {
                result.push_back(it->first);
            }
        }
        return result;
    }
};
```

**3分钟**填写代码,  
**有问题随时提出!**

# 例5:实现(方法1)

```
class Solution {
public:
    std::vector<std::string> findRepeatedDnaSequences(std::string s) {
        std::map<std::string, int> word_map; //<单词, 单词数量> 的映射
        std::vector<std::string> result;
        for (int i = 0; i < s.length(); i++) {
            std::string word = s.substr(i, 10); //若word在哈希中出现
            if (word_map.find(word) != word_map.end()) {
                word_map[word] += 1;
            }
            else {
                word_map[word] = 1;
            }
        }
        std::map<std::string, int> ::iterator it; //遍历哈希表中的所有单词
        for (it = word_map.begin(); it != word_map.end(); it++) {
            if (it->second > 1) {
                result.push_back(it->first);
            }
        }
        return result;
    }
};
```

## 例5:算法思路2

---

将长度为10的**DNA序列**进行整数编码：

['A', 'C', 'G', 'T'] 4个**字符**分别用[0, 1, 2, 3](二进制形式(00, 01, 10, 11))所表示，故长度为10的DNA序列可以用20个比特位的**整数**所表示，如：

00	00	00	00	00	01	01	01	01	11
<b>AAAAACCCCT</b>									
最低位					最高位				

使用整数**11010101010000000000 = 873472**表示

**AAAAACCCCT**

# 例5:算法思路2

- 1.设置全局整数哈希int g\_hash\_map[1048576];  $1048576 = 2^{20}$ , 表示**所有的长度为10**的DNA序列。
- 2.将DNA字符串的前10个字符**使用左移位运算**转换为整数key, g\_hash\_map[key]++。
- 3.从DNA的第11个字符开始, **按顺序**遍历各个字符, 遇到1个字符**即将key右移2位(去掉最低位)**, 并且将新的DNA字符s[i]转换为整数后, **或到最高位(第19、20位)**, g\_hash\_map[key]++。
- 4.遍历**哈希表**g\_hash\_map, 若g\_hash\_map[i] > 1, 将i从低到高位转换为10个字符的**DNA序列**, push至结果数组。

AAAAACCCCT AA      A AAAACCCCTA A

00 00 00 00 00 01 01 01 01 11

AAAAACCCCT

00 00 00 00 01 01 01 01 11 00

AAAAACCCCTA

key = key >> 2  
key = key | (A(00) << 18);

## 例5:课堂练习(方法2)

```
int g_hash_map[1048576] = {0}; //哈希太大了, 需要全局数组

std::string change_int_to_DNA(int DNA){
    static const char DNA_CHAR[] = {'A', 'C', 'G', 'T'};
    std::string str;
    for (int i = 0; i < 10; i++){
        str +=  //将一个长度为10的片段, 从整数
        DNA =  转为字符串
    }
    return str;
}

class Solution {
public:
    std::vector<std::string> findRepeatedDnaSequences(std::string s)
    std::vector<std::string> result;
    if (s.length() < 10){
        return result;
    }
    for (int i = 0; i < 1048576; i++){ //每次调用时需要更新全局数组
        g_hash_map[i] = 0;
    }
    int char_map[128] = {0};
    char_map['A'] = 0;
    char_map['C'] = 1; //设置字符到整数的转换数组
    char_map['G'] = 2;
    char_map['T'] = 3;
    int key = 0; //将DNA字符串的前10个字符
    for (int i = 9; i >= 0; i--){
        key = 
    }
    g_hash_map[key] = 1;
    for (int i = 10; i < s.length(); i++){
        key = key >> 2;
        key = 
        g_hash_map[key]++;
    }
    for (int i = 0; i < 1048576; i++){
        if () {
            result.push_back(change_int_to_DNA(i));
        }
    }
    //将出现次数大于2的片段push进入结果
    return result;
}
};
```

3分钟填写代码,  
有问题随时提出!

# 例5:实现(方法2)

```
int g_hash_map[1048576] = {0}; //哈希太大了，需要全局数组

std::string change_int_to_DNA(int DNA){
    static const char DNA_CHAR[] = {'A', 'C', 'G', 'T'};
    std::string str;
    for (int i = 0; i < 10; i++){
        str += DNA_CHAR[DNA & 3]; //将一个长度为10的片段，从整数
                                   转为字符串
        DNA = DNA >> 2;
    }
    return str;
}

class Solution {
public:
    std::vector<std::string> findRepeatedDnaSequences(std::string s) {
        std::vector<std::string> result;
        if (s.length() < 10){
            return result;
        }
        for (int i = 0; i < 1048576; i++){ //每次调用时需要更新全局数组
            g_hash_map[i] = 0;
        }
        int char_map[128] = {0};
        char_map['A'] = 0;
        char_map['C'] = 1; //设置字符到整数的转换数组
        char_map['G'] = 2;
        char_map['T'] = 3;
        int key = 0; //将DNA字符串的前10个字符
        for (int i = 9; i >= 0; i--){
            key = (key << 2) + char_map[s[i]];
        }
        g_hash_map[key] = 1;
        for (int i = 10; i < s.length(); i++){
            key = key >> 2;
            key = key | (char_map[s[i]] << 18);
            g_hash_map[key]++;
        }
        for (int i = 0; i < 1048576; i++){
            if (g_hash_map[i] > 1){
                result.push_back(change_int_to_DNA(i));
            }
        }
        return result;
    }
};
```

AAAAACCCCT AA      A AAAACCCCTA A

00 00 00 00 00 01 01 01 01 11

**AAAAACCCCT**

00 00 00 00 01 01 01 01 11 00

**AAAACCCCTA**



# 例5:测试与leetcode提交结果

---

```
int main() {
    std::string s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT";
    Solution solve;
    std::vector<std::string> result = solve.findRepeatedDnaSequences(s);
    for (int i = 0; i < result.size(); i++) {
        printf("%s\n", result[i].c_str());
    }
    return 0;
}
```

Repeated DNA Sequences

## Submission Details

32 / 32 test cases passed.

Status: Accepted

Runtime: 189 ms

Submitted: 0 minutes ago

AAAAACCCCC

CCCCCAAAA

请按任意键继续. . .

Repeated DNA Sequences

## Submission Details

32 / 32 test cases passed.

Status: Accepted

Runtime: 59 ms

Submitted: 0 minutes ago

# 例6:最小窗口子串

---

已知字符串S与字符串T，求在S中的**最小窗口(区间)**，使得这个区间中**包含**了字符串T中的**所有字符**。

例如: S = “ADOBECODEBANC” ; T = “ABC ”

包含T的**子区间**中，有“ADOBEC”，“CODEBA”，“BANC”等等；**最小窗口区间**是“BANC”

```
class Solution {  
public:  
    std::string minWindow(std::string s, std::string t) {  
  
    }  
};
```

选自 **LeetCode 76. Minimum Window Substring**

<https://leetcode.com/problems/minimum-window-substring/description/>

难度:**Hard**

# 例6:思考

---

枚举  $S = \text{"ADOBECODEBANC"}$  的**所有子字符串**:

A	D	A	N	C
AD	DO	AN	NC	
ADO	DOB	...	ANC	
ADOB	DOBE			
...	...			
ADOBECODEBANC	DOBECODEBANC			

**检查**以上所有子字符串, 是否**包含**字符串  $T = \text{"ABC"}$

**思考**: 本题是否同样和例4一样, 有一个 **$O(n)$** 的最佳算法?

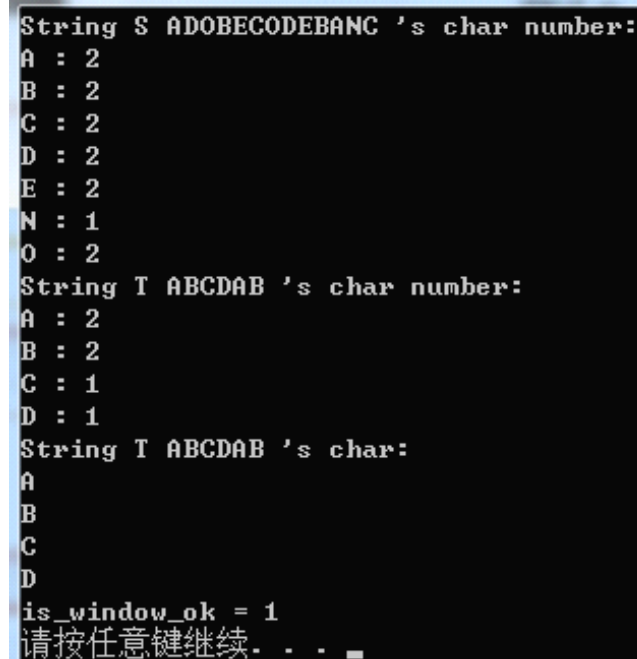
# 例6:预备知识

```
bool is_window_ok(int map_s[], int map_t[], std::vector<int> &vec_t){
    for (int i = 0; i < vec_t.size(); i++){ //利用vec_t遍历t中出现的字符
        if (map_s[vec_t[i]] < map_t[vec_t[i]]){
            return false; //如果s出现该字符的数量小于t中出现该字符的数量
        }
    }
    return true;
}

int main(){
    std::string s = "ADOBECODEBANC";
    std::string t = "ABCDAB";
    const int MAX_ARRAY_LEN = 128; //char 0-127, 利用数组下标记录字符个数
    int map_t[MAX_ARRAY_LEN] = {0}; //记录t字符串各字符个数
    int map_s[MAX_ARRAY_LEN] = {0}; //记录s字符串各字符个数
    std::vector<int> vec_t; //记录t字符串中有哪些字符

    for (int i = 0; i < s.length(); i++){
        map_s[s[i]]++; //遍历s, 记录s字符串个字符个数
    }
    for (int i = 0; i < t.length(); i++){
        map_t[t[i]]++; //遍历t, 记录s字符串个字符个数
    }
    for (int i = 0; i < MAX_ARRAY_LEN; i++){
        if (map_t[i] > 0){
            vec_t.push_back(i);
        } //遍历, 将字符串t中出现的字符存储到vec_t中
    }
}
```

互联网新技术在线教育领航者



```
String S ADOBECODEBANC 's char number:
A : 2
B : 2
C : 2
D : 2
E : 2
N : 1
O : 2
String T ABCDAB 's char number:
A : 2
B : 2
C : 1
D : 1
String T ABCDAB 's char:
A
B
C
D
is_window_ok = 1
请按任意键继续. . .
```

# 例6:算法思路

1. 设置两个**字符哈希**数组,  $\text{map\_s}$ 与 $\text{map\_t}$ ,  $\text{map\_s}$ 代表**当前处理的**窗口区间中的字符数量,  $\text{map\_t}$ 代表子串T的字符数量。

2. 设置**两个指针**(记作指针i与指针begin)指向字符串第一个字符;

3. i指针向后**逐个扫描**字符串中的字符, 在这个过程中, **循环检查begin**指针是否可以向前移动:

**如果**当前begin指向的字符T中**没出现**, 直接**前移**begin;

**如果**begin指向的字符T中出现了, 但是当前区间窗口中的该字符**数量足够**, 向前移动begin, 并更新 $\text{map\_s}$ ;

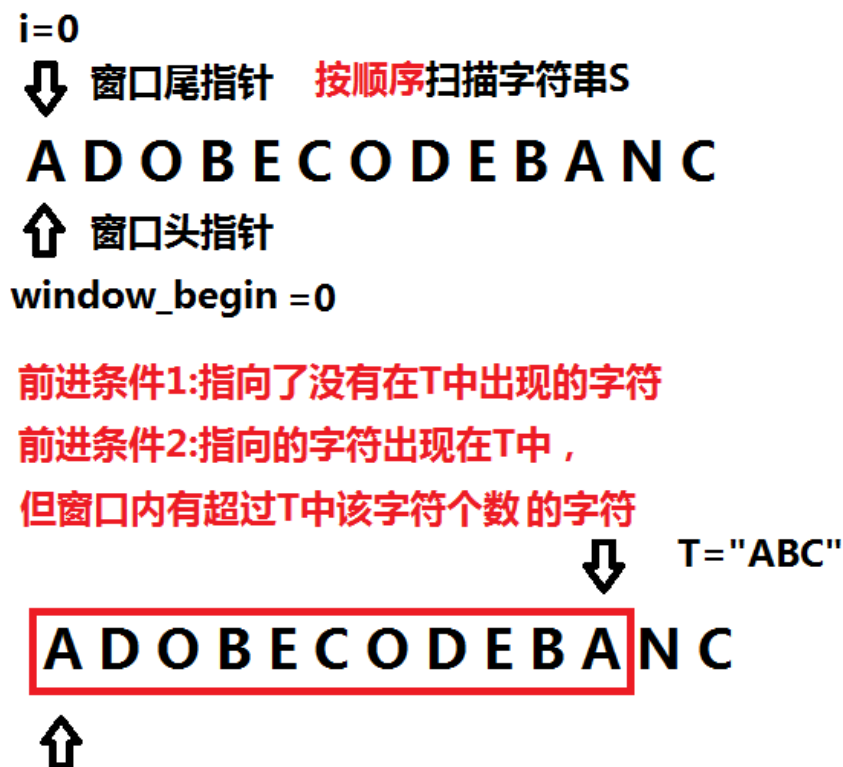
**否则**不能移动begin, 跳出检查。

4. 指针i每向前扫描一个字符, 即**检查**一下是否可以**更新**最终结果(找到最小的包含T中各个字符的窗口)。

在整个过程中, 使用begin与i维护一个**窗口**, 该窗口中的子串**满足题目条件**(包含T中所有字符), 窗口**线性向前滑动**, 整体时间复杂度为 **$O(n)$** 。



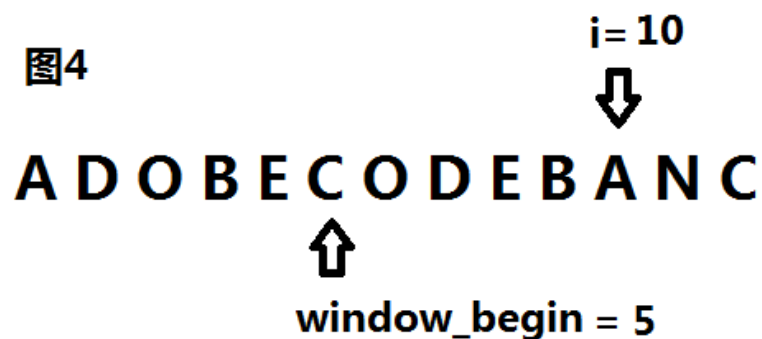
# 例6:算法思路



# 例6:算法思路

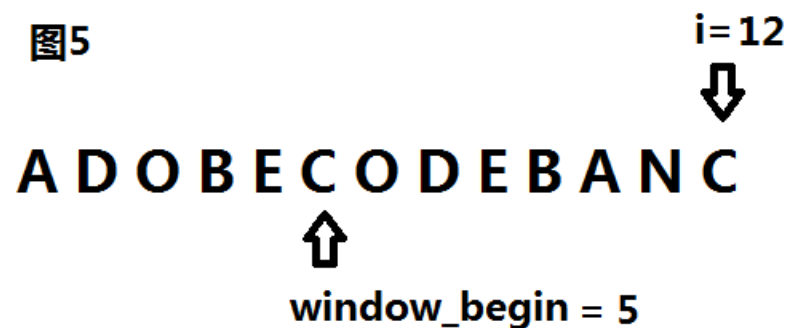
---

图4



发现**可能的**结果"CODEBA", 但不更新  
继续移动尾指针

图5



此时可以移动头指针了!

图6



发现**可能的**结果 **BANC** 最终求的结果!

# 例6:实现(统计T字符串、检查包含函数)

```
class Solution {
private:
    bool is_window_ok(int map_s[], int map_t[], std::vector<int> &vec_t) {
        for (int i = 0; i < vec_t.size(); i++) { //利用vec_t遍历t中出现的字符
            if (map_s[vec_t[i]] < map_t[vec_t[i]]) {
                return false; //如果s出现该字符的数量小于t中出现该字符的数量
            }
        }
        return true;
    }
public:
    std::string minWindow(std::string s, std::string t) {
        const int MAX_ARRAY_LEN = 128; //char 0-127, 利用数组下标记录字符个数
        int map_t[MAX_ARRAY_LEN] = {0}; //记录t字符串各字符个数
        int map_s[MAX_ARRAY_LEN] = {0}; //记录s字符串各字符个数

        std::vector<int> vec_t; //记录t字符串中有哪些字符
        for (int i = 0; i < t.length(); i++) {
            map_t[t[i]]++; //遍历t, 记录s字符串个字符个数
        }

        for (int i = 0; i < MAX_ARRAY_LEN; i++) {
            if (map_t[i] > 0) {
                vec_t.push_back(i);
            } //遍历, 将字符串t中出现的字符存储到vec_t中
        }
    }
};
```



# 例6:实现(课堂练习, 双指针求最小窗口)

```
int window_begin = 0; //最小窗口起始指针
std::string result; //最小窗口对应的字符串

for (int i = 0; i < s.length(); i++) { //i代表了窗口的尾指针
    1
    while(window_begin < i) { //窗口的头指针不能超过尾指针
        char begin_ch = s[window_begin];
        if (map_t[begin_ch] == 0) { //如果当前头指针指向的字符没有
            2
            在字符串t中出现
        } //头指针指向的字符出现在T中, 窗口内有超过T中该字符个数的字符
        else if (map_s[begin_ch] > map_t[begin_ch]) {
            3
            window_begin++; //窗口头指针前移
        }
        else {
            4
        }
    } //检查此时的窗口是否包含字符串t
    if (is_window_ok(map_s, map_t, vec_t)) { //计算新字符串长度
        int new_window_len = i - window_begin + 1;
        if (
            5
        ) {
            result = s.substr(window_begin, new_window_len);
            //替换窗口所对应的字符串
        }
    }
}
return result;
}
```

5分钟时间填写代码,  
有问题随时提出!

# 例6:实现(双指针求最小窗口)

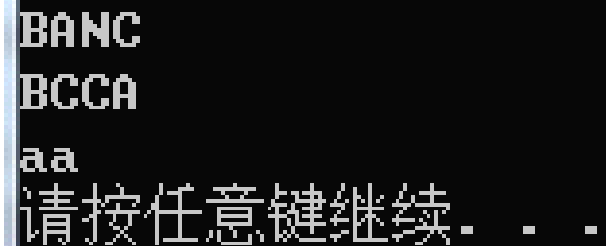
```
int window_begin = 0; //最小窗口起始指针
std::string result; //最小窗口对应的字符串

for (int i = 0; i < s.length(); i++) { //i代表了窗口的尾指针
    map_s[s[i]]++; //将尾指针指向的字符添加到表示窗口的map中
    while (window_begin < i) { //窗口的头指针不能超过尾指针
        char begin_ch = s[window_begin];
        if (map_t[begin_ch] == 0) { //如果当前头指针指向的字符没有
//窗口头指针前移 window_begin++; //在字符串t中出现
        } //头指针指向的字符出现在T中，窗口内有超过T中该字符个数的字符
        else if (map_s[begin_ch] > map_t[begin_ch]) {
            map_s[begin_ch]--; //头指针前移了，它指向的字符减少1个
            window_begin++; //窗口头指针前移
        }
        else {
            break; //除了1,2两个条件，其他情况都跳出循环
        }
        //检查此时的窗口是否包含字符串t
    }
    if (is_window_ok(map_s, map_t, vec_t)) { //计算新字符串长度
        int new_window_len = i - window_begin + 1;
        if (result == "" || result.length() > new_window_len) {
            result = s.substr(window_begin, new_window_len);
        }
        //替换窗口所对应的字符串 //结果字符串为空
        //或者当前窗口字符串更小的时候
        //更新结果
    }
    return result;
};
```

# 例6:测试与leetcode提交结果

```
int main() {  
  
    Solution solve;  
    std::string result = solve.minWindow("ADOBECODEBANC", "ABC");  
    printf("%s\n", result.c_str());  
    result = solve.minWindow("MADOBCCABEC", "ABCC");  
    printf("%s\n", result.c_str());  
    result = solve.minWindow("aa", "aa");  
    printf("%s\n", result.c_str());  
  
    return 0;  
}
```

Minimum Window Substring



```
BANC  
BCCA  
aa  
请按任意键继续. . .
```

## Submission Details

268 / 268 test cases passed.

Runtime: 22 ms

Status: Accepted

Submitted: 0 minutes ago

# 结束

---

非常感谢大家！

林沐