数组与字符串

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/20
// @Function    : 数组与字符串的学习
```

一、章节概述：

完成本章后，你将能够回答以下问题：

- [ ] 数组和列表、集合之间有什么不同？
- [ ] 如何理解数组的读取、查找、插入、删除等 基本操作？
- [ ] 数组在内存中是如何存放的？
- [ ] 在你常用的语言中，如何对数组执行初始化、数据访问、修改、迭代、排序、添加、删除等操作？

二、集合、列表与数组的差别

1、集合：由一个或多个确定的元素所构成的整体。通俗的说，集合就是将一组事物组号到一起。一般认为，集合有两个显著的特性：

- 集合里的元素**类型不一定相同**；
- 集合里的元素**没有顺序。**

2、列表：即线性表，线性表中分为链表和顺序表（用数组实现）。

3、数组：

列表的实现方式之一，一般而言，不仅逻辑上相邻，且物理上相邻，能够实现随机存取。**在C++、JAVA中，数组中的元素类型必须保持一致，而Python中则可以不同。**

三、数组的操作

1、读取元素：

通过索引来访问数组中的元素，索引一般从0开始。对于数组，一般能够实现数组的**随机存取和访问**，因此：时间复杂度为O(1)。

2、查找元素：

这一操作与读取元素不同的是：查找时，参数是元素的值；读取时，参数是元素的索引。因此，要查找元素，一般需要**遍历**一遍数组，其时间复杂度为：O(n)。

3、插入元素：

- 若插入到数组的末尾：直接插入即可，时间复杂度为：O(1)；
- 若插入到数组中的其他位置：首先需要将该元素要插入位置之后的**所有元素均后移**，然后执行插入查找，时间复杂度为O(n)。

4、删除元素：

删除查找与插入操作类似，与之不同的是，插入是往后移，删除是**往前移。**

四、一维数组的习题

1、计算中心下标：

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/20
// @Function    : 计算数组的中心下标，其左侧所有元素相加的和等于右侧所有元素相加的和


#include <iostream>
#include <vector>  // 添加 vector 头文件
#include <numeric> // 添加 accumulate 头文件
using namespace std;

class Solution {
public:
    int FindMidIndes(vector<int> &nums) {
        //nums.begin(), nums.end(),nums.size() 都是vector方法
        //accumulate(start, end, 0)：返回 [start, end) 范围内元素的累积和
        int total = accumulate(nums.begin(), nums.end(), 0);
        int sum = 0;
        for (int i = 0; i < nums.size(); ++i) {
            if (2 * sum + nums[i] == total) {
                return i;
            }
            sum += nums[i];
        }
        return -1;
    }
};

int main() {
    Solution solution;                          // 创建 Solution 对象
    vector<int> nums = {1, 2, 3};               // 使用 vector 代替数组
    int index = solution.FindMidIndes(nums);    // 通过对象调用方法
    cout << "该数组的中心下标为：" << index << endl;
    return 0;
}
```

2、查找元素下标：

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/20
// @Function    : 查找元素下标，若元素不在数组中，则返回该元素应插入的位置下标

#include<iostream>
#include<vector>
using namespace std;

class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int low = 0,high = nums.size() - 1;
        int mid = (low + high)/2;

        while(low <= high){
            mid = low + (high - low)/2;      //在处理大整数范围的情况下，这样处理能够避免整数溢出
            //cout<<low<<' '<<mid<<' '<<high<<endl;
            if(nums[mid] == target){         //找到目标值，返回索引
                return mid;
            }
            else if(nums[mid] < target){     //目标值在右半部分
                low = mid+1;
            }
            else if(nums[mid] > target){     //目标值在左半部分
                high = mid-1;
            }
            //cout<<low<<' '<<mid<<' '<<high<<endl;
        }
        return high+1;
    }
};

int main(){
    Solution solution;
    vector<int> nums = {1,3,5,6};
    int target,answer;
    cin>>target;
    answer = solution.searchInsert(nums,target);
    cout<<answer<<endl;
    return 0;
}
```

五、二维数组的习题

1、二维数组的定义：

这里需要注意的是，如果数组的大小较大，则需要将其**定义在主函数外**，否则会造成程序异常（考虑程序的**栈帧**）。

2、合并区间：

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/20
// @Function    : 合并区间

#include <vector>
#include <algorithm>
using namespace std;

vector<vector<int>> merge(vector<vector<int>>& intervals) {
    if (intervals.empty()) {
        return {};
    }

    // 按照区间的起始位置进行排序
    //Lambda 表达式的参数是两个 vector<int> 类型的引用（a 和 b），返回值是一个布尔值。
    //该 Lambda 表达式表示比较两个区间的起始位置：若 a[0] < b[0]，则返回 true，表示 a 应该排在 b 前面，否则返回 false
    sort(intervals.begin(), intervals.end(), [](const vector<int>& a, const vector<int>& b) {
        return a[0] < b[0];
    });

    vector<vector<int>> merged;
    merged.push_back(intervals[0]);     //将排序后的区间集合中的第一个区间添加到新的合并区间集合 merged 中

    for (int i = 1; i < intervals.size(); ++i) {
        //merged.back()[1]：函数返回 merged 最后一个合并的区间，并通过 [1] 取得最后一个合并区间的结束位置。
        if (merged.back()[1] >= intervals[i][0]) {
            merged.back()[1] = max(merged.back()[1], intervals[i][1]);
 // 合并重叠的区间
        }
        else {
            // 不重叠，添加新区间
            merged.push_back(intervals[i]);
        }
    }
    return merged;
}
```

3、矩阵旋转90°：

```cpp
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/20
// @Function    : 将矩阵旋转90°
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int rows = matrix.size();
        int cols = (matrix.size() > 0) ? matrix[0].size() : 0;
        vector<vector<int>> rotated(cols, vector<int>(rows));
        //cout<<rows<<' '<<cols<<endl;

        //先行后列遍历
        for(int i=0;i<rows;i++){
            for(int j=0;j<cols;j++){
                rotated[i][j] = matrix[rows-1-j][i];
            }
        }
        matrix = rotated;
        //const auto& row 表示在每次迭代中，row 将引用 rotated 中的一行。
        //matrix 是一个二维向量，所以 row 是一个一维向量（行）
        for (const auto& row : matrix) {
            //int num 表示在每次迭代中，num 将取得 row 中的一个元素。
            for (int num : row) {
                cout << num << ' ';
            }
            cout << endl;
        }
    }
};

int main(){
    Solution solution;
    vector<vector<int>> matrix = {{1,2,3},{4,5,6},{7,8,9}};
    solution.rotate(matrix);

    return 0;
}
```

4、矩阵清零：

```cpp
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/20
// @Function    : 若a[i][j]=0,则将第i行和第j列的元素清零
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;


class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        int rows = matrix.size();
        int cols = (matrix.size() > 0) ? matrix[0].size() : 0;
        vector<vector<int>> rotated(cols, vector<int>(rows));
        rotated = matrix;
        //cout<<rows<<' '<<cols<<endl;

        //先行后列遍历
        for(int i=0;i<rows;i++){                    //遍历一行
            for(int j=0;j<cols;j++){                //遍历一列
                if(matrix[i][j] == 0){
                    for(int k=0;k<cols;k++){        //修改一行
                        rotated[i][k] = 0;
                    }
                    for(int k=0;k<rows;k++){        //修改一列
                        rotated[k][j] = 0;
                    }
                }
            }
        }
        matrix = rotated;
    }
};

int main(){
    Solution solution;
    //vector<vector<int>> matrix = {{1,1,1},{1,0,1},{1,1,1}};
    vector<vector<int>> matrix = {{0,1,2,0},{3,4,5,2},{1,3,1,5}};
    solution.setZeroes(matrix);
    return 0;
}
```

5、对角线遍历：

思路与算法：

设矩阵为m行n列的矩阵，观察规律：

- 共有$m+n+1$条对角线，相邻对角线的遍历方向不同；
- 设对角线编号为 $i \in [0，m+n-2]$：
  - 当 i 为偶数时，对角线的走向是从下往上遍历；
  - 当 i 为奇数时，对角线的走向是从上往下遍历；
- 当对角线的从下往上遍历时，每次行号-1，列号$+1$，直至到达矩阵边缘：
  - 当 $i<m$ 时，对角线遍历起点为：$（i，0）$；
  - 当 $i \geq m$ 时，对角线遍历起点为：$（i-n+1，n-1）$；
- 当对角线的从上往下遍历时，每次行号$+1$，列号-1，直至到达矩阵边缘：
  - 当 $i<n$ 时，对角线遍历起点为：$（0，i）$；
  - 当 $i \geq n$ 时，对角线遍历起点为：$（i-n+1，n-1）$。

```
class Solution {
public:
    vector<int> findDiagonalOrder(vector<vector<int>>& mat) {
        int m = mat.size();
        int n = mat[0].size();
        vector<int> res;
        for (int i = 0; i < m + n - 1; i++) {
            if (i % 2) {                                //i为奇数，从上往下遍历
                int x = i < n ? 0 : i - n + 1;
                int y = i < n ? i : n - 1;
                while (x < m && y >= 0) {           //从上往下遍历时，x>m ||
y<0 就说明发生了越界
                    res.emplace_back(mat[x][y]);    //在数组r
                    x++;
                    y--;
                }
            }
            else {                                      //i为偶数，从上往下遍历
                int x = i < m ? i : m - 1;
                int y = i < m ? 0 : i - m + 1;
                while (x >= 0 && y < n) {           //从下往上遍历时，x<0 ||
y>=n 就说明发生了越界
                    res.emplace_back(mat[x][y]);
                    x--;
                    y++;
                }
            }
        }
        return res;
    }
};
```

六、字符数组

1、字符数组：

（1）定义和初始化字符数组：同数组的定义和初始化。我们知道，字符串与数组有很多相似之处，比如使用 名称[下标] 来得到一个字符。那么我们为什么要单独讨论字符串呢？原因主要有：

- 字符串的基本操作对象通常是**字符串整体或者其子串**；
- 字符串操作比其他数据类型更复杂（例如比较、连接操作）。

（2）字符数组的赋值与引用：只能对字符数组中的元素赋值，而不能用赋值语句对整个数组赋值。

（3）字符串结束标志：C++规定了**字符串的结束标志** `'\0'`，程序中通常依靠检测 `'\0'` 的位置来判定字符串是否结束，而不是根据数组的长度来决定字符串长度。

（4）字符串的输入输出：

- 字符数组逐个字符输入输出：`cout<<str[i][j]`；
- 字符数组一次性输入和输出：`cin>>str` 和 `cout<<str`
    - 注意：用 `cin` 从键盘输入字符串时，若输入的字符超过一定要的字符数组的长度，**系统不报错**，而是将多余的字符顺序存放到该字符数组后面的3个字节，这就有可能导致破坏其他数据，从而造成无法估计的后果。

（5）字符串处理函数：应使用 `string` 头文件。

函数原型	函数定义	函数功能
字符串连接函数 `strcat`	`strcat(char[], const char[])`	将第二个字符串连接到第一个字符串的后面
字符串复制函数 `strcpy`	`strcpy(char[], const char[])`	将第二个字符串的内容复制到第一个字符串的后面
字符串比较函数 `strcmp`	`strcmp(const char[], const char[])`	用于查验两个字符串是否相同
字符串长度函数 `strlen`	`strlen(const char[])`	测试字符串的实际长度（不包括 `'\0'` 在内）

2、字符串类：

（1）字符串变量的定义与引用：使用字符串类，需要引用头文件 `string`，而不是 `string.h`。

定义字符串变量：`string str = 'China'`。

（2）字符串变量的赋值：对字符串变量的赋值可以直接使用 `=`，如上。这使得我们在为字符串变量赋值时不变精确计算字符个数，也不必顾虑是否“超长”。

（3）字符串变量的输入输出：可以在输入输出语句中用字符串的变量名来完成字符串的输入输出。

（4）字符串变量的运算（复制、连接、比较）：

- 复制：`str1 = str;`
- 连接：`str = str1 +str2;`
- 比较：可以直接用==、 > 、 < 、！ =、>=、<=等关系运算符来进行字符串的比较。

（5）字符串数组：用`string`定义字符串数组

```
string name[5] = {"Zhang", "Li", "Sun","Wang", "Tan"};
```

- 一个字符串数组中包含若干个元素，每个元素都是一个string类型的变量，且每个元素的长度可以不同；
- 在字符串数组的每一元素中存放一个字符串，每个元素的值只包含字符串本身，不包含`'\0'`。

七、字符数组的习题：

1、最长公共前缀：

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/22
// @Function    : 查找字符串数组中的最长公共前缀;如果不存在公共前缀，返回空字符串 ""

#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {
        string prefix = strs[0];                                    // 将第一个字符串设为初始的最长前缀

        if (strs.empty()) return "";                                // 如果输入为空，返回空字符串

        // 遍历后续字符串，不断更新最长前缀
        for (int i = 1; i < strs.size(); i++) {
            int j = 0;
            while (j < prefix.size() && j < strs[i].size() && prefix[j] == strs[i][j]){
                ++j;
            }
```

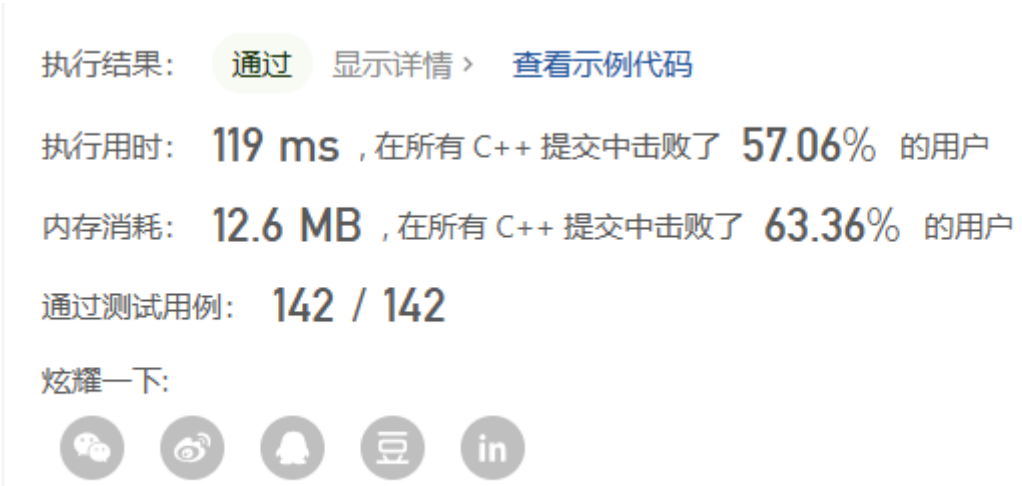
```
            prefix = prefix.substr(0, j);                    // 截取出当前最长前缀
            if (prefix.empty())  return "";                  // 如果最长前缀为空，说明不存在公共前缀，直接返回
        }
        return prefix;
    }
};

int main() {
    Solution solution;
    vector<string> strs = {"flower","flow","flight"};
    string maxstr = solution.longestCommonPrefix(strs);
    cout << maxstr << endl;
    return 0;
}
```

2、最长回文子串：

解决最长回文子串需要运用动态规划的思想，这对当前进度的算法学习来说，无疑是一道难题。因此我们首先提出一种暴力解决方法：求从字符串的所有子串，然后对每个子串进行判断。

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/22
// @Function    : 最长回文子串

#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    string longestPalindrome(string s) {
        string strs = "";
        int len = s.length();
        int maxlen = 0;

        for(int i=0;i<len;i++){                              //遍历所有的子串
            for(int j=i;j<len;j++){                          //遍历从i开始的所有子串
                int len = j - i + 1;                         //记录当前子串的长度

                if(len > maxlen){
                    int start = i,end = j;
                    while(start < end){                      //判断子串是否是回文串

                        if(s[start] ==  s[end]){
                            start++;end--;
                        }
```

```
                        else break;
                    }

                    if(start >= end){                    //若子串是回文串
                        maxlen = len;                    //更新最长回文子串长
度记录
                        strs = s.substr(i,len);          //更新回文串记录
                    }
                }
            }
        }
        return strs;
    }
};

int main() {
    Solution solution;
    string s = "babad";
    string strs = solution.longestPalindrome(s);
    cout <<"最大回文子串为: "<< strs << endl;
    return 0;
}
```



3、子串倒序:

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/22
// @Function    : 子串倒序

#include <iostream>
#include <cstring>
#include <vector>
using namespace std;
```

```
class Solution {
public:
    string reverseWords(string s) {
        string reverse_str = "";
        string str[10000],reverse[10000];
        int len = s.size();
        int i = 0,j = 0,num = 0;

        for(int k=0;k<len;k++){                                  // 以空格为中断符，
将原字符串变为字符数组
            if(s[k] != ' '){
                str[i] = str[i] + s[k];
                j++;
                //cout<<str[i]<<' '<<s[k]<<endl;
            }
            else
                i++;j=0;
        }

        for(int m=0;m<=i;m++){                                   //消去字符串数组中的
空白内容
            if(!str[m].empty()){
                reverse[num] = str[m];
                num++;
            }
        }

        for(int m=num-1;m>=0;m--){
            if(m != 0)
                reverse_str = reverse_str + reverse[m] +' ';
            else
                reverse_str = reverse_str + reverse[m];
        }
        return reverse_str;
    }
};

int main() {
    Solution solution;
    string s = "a good   example";
    //string s = "the sky is blue";
    //string s = "  hello world  ";
    string strs = solution.reverseWords(s);
    cout <<"新的字符串为:"<<"\""<<strs<<"\""<<endl;
    return 0;
}
```

4、字符串匹配算法KMP：

（1）朴素模式匹配：暴力解法。依次对比字符串s的各个子串，直至找到子串t为止。最坏时间复杂度：O(mn)。

```
int findindex_violent(string s, string t){
        int i,j;
        for( i=0;i<s.size();i++){
            for( j=0;j<t.size();j++){
                if(s[i+j] != t[j]){
                    break;
                }
            }
            if(j == t.size()){
                break;
            }
        }
        if(j == t.size())
            return i;
        else
            return -1;
    }
```

（2）KMP算法：最坏时间复杂度：O(m+n)。

```
void get_next(int next[],string t){
        int i=1,j=0;
        next[1] = 0;
        while(i<t.size()){
            if(j == 0 || t[i] == t[j]){
                i++;j++;
            }
            else{
                j = next[j];
            }
        }
    }
    int Index_KMP(string s,string t){
        int i,j;
        int next[t.size()+1];
        get_next(next,t);
        while(i<=s.size() && j<=t.size()){
            if(j == 0 || s[i] == t[j]){
                i++;j++;
            }
            else
                j = next[j];
        }
        if(j > t.size())
            return i - t.size();
        else
```

```
            return 0;
    }
```

八、双指针

1、原地反转字符串：

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/24
// @Function    : 原地字符数组倒序

#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    void reverseString(vector<char>& s) {
        int len = s.size();
        int i = 0,j = len-1,temp ;
        while(i < j){
            temp = s[i];
            s[i] = s[j];
            s[j] = temp;
            i++;j--;
        }
    }
};

int main(){
    Solution soluton;
    vector<char> s = {'l', 'e', 'e', 't', 'c', 'o', 'd', 'e'};
    soluton.reverseString(s);
    for(char c : s){
        cout<<c;
    }
    cout<<endl;
}
```

2、数组拆分：

给定长度为 2n 的整数数组 nums ，你的任务是将这些数分成 n 对, 例如 (a1, b1), (a2, b2), ..., (an, bn) ，**使得从 1 到 n 的 min(ai, bi) 总和最大。**返回该最大总和 。

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/24
// @Function    :
```

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    void qSortArray(vector<int>& array, int start, int last){           //
快速排序
        int low = start;
        int high = last;
        if (low < high){
            while (low < high){
                while (array[low] <= array[start] && low < last){       //
满足小于基准的条件，指针右移
                    low++;
                }
                while (array[high] >= array[start] && high > start){    //
满足大于基准的条件，指针左移
                    high--;
                }
                if (low < high){
                    swap(array[low], array[high]);                      //
交换两个不满足条件的元素
                }
                else{
                    break;
                }
            }
            swap(array[start], array[high]);                            //
插入基准元素
            qSortArray(array, start, high - 1);
            qSortArray(array, high + 1, last);
        }
    }
    int arrayPairSum(vector<int>& nums) {
        int len = nums.size();
        int min = 0;
        qSortArray(nums,0,len-1);
        for(int i=0;i<len;i+=2){
            min += nums[i];
        }
        return min;
    }
};

int main(){
    Solution solution;
    vector<int> a = {1,4,3,2};
    int min = solution.arrayPairSum(a);
    cout<<min<<endl;
    return 0;
}
```

3、两数之和：

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 <= index1 < index2 <= 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。

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/24
// @Function    : 略

#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        int len = numbers.size(), i = 0, j = len - 1;
        vector<int> sum;
        while (i < j) {
            if (numbers[i] + numbers[j] < target) {
                i++;
            } else if (numbers[i] + numbers[j] > target) {
                j--;
            } else {
                sum.push_back(i + 1);
                sum.push_back(j + 1);
                return sum;
            }
        }
        for(j=i;j<len;j++){
            if(numbers[i] + numbers[j] == target){
                sum.push_back(i+1);
                sum.push_back(j+1);
                return sum;
            }
        }
        return sum;
    }
};

int main() {
    Solution solution;
    vector<int> a = {-1,0};
    vector<int> b = solution.twoSum(a,-1);
```

```
    // 输出结果
    cout << "Indices: ";
    for (int num : b) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

4、原地删除值为 val 的数组元素：

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/24
// @Function    : 快慢指针

#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int len = nums.size();
        int fast=0,slow=0;
        for(fast=0;fast<len;fast++){
            if(nums[fast] != val){
                nums[slow] = nums[fast];
                slow++;
            }
        }
        return slow;
    }
};

int main(){
    Solution solution ;
    vector<int> nums = {3,2,2,3};
    int val = 3;
    int len = solution.removeElement(nums,val);
    for(int i=0;i<len;i++){
        cout<<nums[i];
    }
    cout<<endl;
    return 0;
}
```

5、最大连续1的个数：

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/24
// @Function    : 最大连续 1的个数

#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    int findMaxConsecutiveOnes(vector<int>& nums) {
        int len = nums.size();
        int fast = 0,maxnum = 0,num = 0;
        for(fast=0;fast<len;fast++){
            if(nums[fast] == 1){
                num++;
            }
            else{
                if(num > maxnum){
                    maxnum = num;
                }
                num = 0;
            }
            if(num > maxnum){
                maxnum = num;
            }
        }
        return maxnum;
    }
};

int main(){
    Solution solution;
    vector<int> nums = {1,0,1,1,0,1};
    int maxnum = solution.findMaxConsecutiveOnes(nums);
    cout<<"该数组最大的连续1的个数为："<<maxnum<<endl;
    return 0;
}
```

6、长度最小的子数组：

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/24
// @Function    : 长度最小的子数组(暴力解，时间复杂度为：O(n^2))
int minSubArrayLen(int target, vector<int>& nums) {
```

```
        int len = nums.size();
        if (len == 0) {                                  // 或者返回一个特殊值，
表示空数组
            return 0;
        }

        int minLen = 100000000; // 初始化为一个足够大的值
        for (int i = 0; i < len; i++) {
            int sum = 0;
            for (int j = i; j < len; j++) {
                sum += nums[j];
                if (sum >= target) {
                    minLen = min(minLen, j - i + 1);
                    break; // 如果已经找到一个满足条件的子数组，可以提前结束内
部循环
                }
            }
        }
        return (minLen == 100000000) ? 0 : minLen;
    }
```

为了降低时间复杂度，可以使用**滑动窗口**的方法。

定义两个指针 $start$ 和 end 分别表示子数组（滑动窗口窗口）的开始位置和结束位置，维护变量 sum 存储子数组中的元素和（即从 $nums[start]$到 $nums[end]$的元素和）。

初始状态下，$start$ 和 end 都指向下标 0，sum 的值为 0。

每一轮迭代，将 $nums[end]$ 加到 sum，如果 $sum \geq s$，则更新子数组的最小长度（此时子数组的长度是 $end - start + 1$），然后将 $nums[start]$ 从 sum 中减去并将 $start$ 右移，直到 $sum < s$，在此过程中同样更新子数组的最小长度。在每一轮迭代的最后，将 end 右移。

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/1/24
// @Function    : 长度最小的子数组（滑动窗口，时间复杂度为：O(n)）
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int n = nums.size();
        if (n == 0) {
            return 0;
        }
        int ans = INT_MAX;
        int start = 0, end = 0;
        int sum = 0;
        while (end < n) {
            sum += nums[end];
            while (sum >= s) {
                ans = min(ans, end - start + 1);
                sum -= nums[start];
```

```
                start++;
            }
            end++;
        }
        return ans == INT_MAX ? 0 : ans;
    }
};
```

7、双指针技巧总结：

（1）使用情景一：要从两端向中间迭代数组。这时可以使用双指针技巧：一个指针从头部开始，而另一个指针从尾部开始。这种技巧经常在排序数组中使用。

（2）使用情景二：快慢指针，有时，我们可以使用两个不同步的指针来解决问题，即**快慢指针**。与情景一不同的是，两个指针的运动方向是相同的，而非相反。也可以用来解决**滑动窗口**问题。

九、小结习题

1、杨辉三角：

```
vector<vector<int>> generate(int numRows) {
    vector<vector<int>> array(numRows); // 先定义二维数组的行数
    for(int i=0;i<numRows;i++){         // 初始化数组
        array[i].resize(i+1);           // 动态定义二维数组每一行的列数
        array[i][0] = 1;
        array[i][i] = 1;
    }
    if(numRows <= 2){
        return array;
    }
    for(int i=2;i<numRows;i++){
        for(int j=1;j<i;j++){
            array[i][j] = array[i-1][j] + array[i-1][j-1];
        }
    }
    return array;
}
```

2、杨辉三角II：

```
vector<int> generate(int rowIndex) {
    int numRows = rowIndex + 1;
    vector<vector<int>> array(numRows); // 先定义二维数组的行数
    vector<int> array_index(numRows);
    for(int i=0;i<numRows;i++){         // 初始化数组
        array[i].resize(i+1);           // 动态定义二维数组每一行的列数
```

```
            array[i][0] = 1;
            array[i][i] = 1;
        }
        if(numRows <= 2){
            for(int i=0;i<numRows;i++){
                array_index[i] = array[rowIndex][i];
            }
            return array_index;
        }
        for(int i=2;i<numRows;i++){
            for(int j=1;j<i;j++){
                array[i][j] = array[i-1][j] + array[i-1][j-1];
            }
        }
        for(int i=0;i<numRows;i++){
            array_index[i] = array[rowIndex][i];
        }
        return array_index;
    }
```

3、反转字符串中的单词：

```
void reverse_word(string& s, int i, int j){
    for(int t = i; t <= (j+i)/2;t++){
        swap(s[t], s[j+i-t]);
    }
}

string reverseWords(string s) {
    int i = 0, j = 0;
    for(;j< s.size();j++){
        if(s[j] == ' '){
            reverse_word(s, i, j-1);
            i = j+1;
        }
    }
    reverse_word(s, i, j-1);
    return s;
}
```

4、原地删除重复元素：

```
int removeDuplicates(vector<int>& nums) {
    int r=0;
    int fast = 0, slow = 0;
    for(fast = 1;fast<nums.size();fast++){
        if(nums[fast] == nums[slow]){
            r++;
        }
        else{
```

```
            nums[fast - r] = nums[fast];
            slow = fast;
        }
    }
    return nums.size() - r;
}
```

5、原地删除0：

```
 void moveZeroes(vector<int>& nums) {
    int k = 0;
    int fast = 0,slow = 0;
    for(fast = 0;fast<nums.size();fast++){
        if(nums[fast] == 0){
            k++;
        }
        else{
            nums[fast - k] = nums[fast];
        }
    }
    for(fast = nums.size() - k;fast<nums.size();fast++){
        nums[fast] = 0;
    }
}
```