1. 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。

```
var twoSum = function(nums, target) {
    const hashNums = {};
    let result = [];
    nums.forEach((item,index)=>{
        const targetNum = target - item;
        if(hashNums[targetNum] === undefined ){
            hashNums[item] = index;
        }else{
            result =  [index,hashNums[targetNum]];
        }
    });
    return result;
};
```

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int,int> hashNums;
        for(int i=0; i<nums.size();i++){
            auto item = hashNums.find(target-nums[i]);
            if(item != hashNums.end()){
                return {item->second,i};
            }
            hashNums[nums[i]]=i;
        }
        return {};
    }
};
```

2. 两数相加

> 给你两个**非空**的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储 一位 数字。请你将两个数相加，并以相同形式返回一个表示和的链表。

```javascript
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
var addTwoNumbers = function(l1, l2) {
    let initListNode = new ListNode('0');
    let theOne = 0;
    let answer = initListNode;
    while(theOne || l1 || l2){
        let val1 = l1?.val ?? 0;
        let val2 = l2?.val ?? 0;
        let addAll = theOne + val1 + val2;
        theOne = addAll >= 10 ? 1 : 0;
        initListNode.next = new ListNode(addAll % 10);
        initListNode = initListNode.next;
        l1 = l1 ? l1.next : l1;
        l2 = l2 ? l2.next : l2;
    }
    return answer.next;
};
```

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode *head = nullptr, *tail = nullptr;
        int carry = 0;
        while(l1 || l2){
            int num1 = l1 ? l1->val : 0;
            int num2 = l2 ? l2->val : 0;
            int sum = num1 + num2 + carry;
            if(head){
                tail->next = new ListNode(sum % 10);
                tail = tail->next;
            }else{
                head = tail = new ListNode(sum % 10);
            }
            carry = sum / 10;
            l1 = l1 ? l1->next : nullptr;
            l2 = l2 ? l2->next : nullptr;
        }
        if(carry){
            tail->next = new ListNode(carry);
        }
        return head;
    }
};
```

3. 无重复字符的最长子串 [滑动数组]

> 给定一个字符串 `s` ，请你找出其中不含有重复字符的 `最长子串` 的长度。

需要使用一种数据结构来判断 **是否有重复的字符**，常用的数据结构为哈希集合（即 C++ 中的 `std::unordered_set` ，Java 中的 `HashSet` ，Python 中的 `set` , JavaScript 中的 `Set` ）

依次递增地枚举子串的起始位置，那么子串的结束位置也是递增的！这里的原因在于，假设我们选择字符串中的第k个字符作为起始位置，并且得到了不包含重复字符的最长子串的结束位置为rk。那么当我们选择第k+1个字符作为起始位置时，首先从k+1到rk+1的字符显然是不重复的，并且由于少了原本的第k个字符，我们可以尝试继续增大rk，直到右侧出现了重复字符为止。

```javascript
var lengthOfLongestSubstring = function(s) {
    let left = 0;
    let ans = 0;
    const hashSet = new Set();
    for(let i=0;i<s.length; i++){
        while(hashSet.has(s[i])){
            hashSet.delete(s[left]);
            left++;
        }
        ans = Math.max(ans,i-left+1);
        hashSet.add(s[i]);
    }
    return ans;
};
```

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int len = s.size();
        int left = 0;
        int ans = 0;
        unordered_set<char> hashSet;
        for(int i=0; i<len; i++){
            while(hashSet.find(s[i])!=hashSet.end()){
                hashSet.erase(s[left]);
                left++;
            }
            ans = max(ans,i-left+1);
            hashSet.insert(s[i]);
        }
        return ans;
    }
};
```

4. 寻找两个正序数组的中位数 [二分法]

> 给定两个大小分别为 m 和 n 的正序（从小到大）数组 nums1 和 nums2。请你找出并返回这两个正序数组的 中位数 。
> 算法的时间复杂度应该为 O(log (m+n)) 。

```cpp
class Solution {
public:
    int getKthElement(const vector<int>& nums1, const vector<int>& nums2, int k) {
        /* 主要思路：要找到第 k (k>1) 小的元素，那么就取 pivot1 = nums1[k/2-1] 和 pivot2 = num
         * 这里的 "/" 表示整除
         * nums1 中小于等于 pivot1 的元素有 nums1[0 .. k/2-2] 共计 k/2-1 个
         * nums2 中小于等于 pivot2 的元素有 nums2[0 .. k/2-2] 共计 k/2-1 个
         * 取 pivot = min(pivot1, pivot2)，两个数组中小于等于 pivot 的元素共计不会超过 (k/2-1)
         * 这样 pivot 本身最大也只能是第 k-1 小的元素
         * 如果 pivot = pivot1，那么 nums1[0 .. k/2-1] 都不可能是第 k 小的元素。把这些元素全部
         * 如果 pivot = pivot2，那么 nums2[0 .. k/2-1] 都不可能是第 k 小的元素。把这些元素全部
         * 由于我们 "删除" 了一些元素（这些元素都比第 k 小的元素要小），因此需要修改 k 的值，减去删除
         */

        int m = nums1.size();
        int n = nums2.size();
        int index1 = 0, index2 = 0;

        while (true) {
            // 边界情况
            // 如果一个数组为空，说明该数组中的所有元素都被排除，我们可以直接返回另一个数组中第 k 小的
            if (index1 == m) {
                return nums2[index2 + k - 1];
            }
            if (index2 == n) {
                return nums1[index1 + k - 1];
            }
            // 如果 k=1，我们只要返回两个数组首元素的最小值即可
            if (k == 1) {
                return min(nums1[index1], nums2[index2]);
            }

            // 正常情况
            // k/2 对k进行二次对半划分
            int newIndex1 = min(index1 + k / 2 - 1, m - 1);
            int newIndex2 = min(index2 + k / 2 - 1, n - 1);
            int pivot1 = nums1[newIndex1];
            int pivot2 = nums2[newIndex2];
            if (pivot1 <= pivot2) {
                k -= newIndex1 - index1 + 1;
                index1 = newIndex1 + 1;
            }
            else {
                k -= newIndex2 - index2 + 1;
                index2 = newIndex2 + 1;
            }
        }
    }

    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
```

```
        int totalLength = nums1.size() + nums2.size();
        if (totalLength % 2 == 1) {
            return getKthElement(nums1, nums2, (totalLength + 1) / 2);
        }
        else {
            return (getKthElement(nums1, nums2, totalLength / 2) + getKthElement(nums1,
        }
    }
};


class Solution {
  public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int m = nums1.length;
        int n = nums2.length;
        int left = (m + n + 1) / 2;
        int right = (m + n + 2) / 2;
        return (findKth(nums1, 0, nums2, 0, left) + findKth(nums1, 0, nums2, 0, right))
        // 使用一个小trick，我们分别找第 (m+n+1) / 2 个，和 (m+n+2) / 2 个，然后求其平均值即可，j
    }
    //i: nums1的起始位置 j: nums2的起始位置
    public int findKth(int[] nums1, int i, int[] nums2, int j, int k){
        if( i >= nums1.length) return nums2[j + k - 1];//nums1为空数组
        if( j >= nums2.length) return nums1[i + k - 1];//nums2为空数组
        if(k == 1){
            return Math.min(nums1[i], nums2[j]);
        }
        int midVal1 = (i + k / 2 - 1 < nums1.length) ? nums1[i + k / 2 - 1] : Integer.MA
        int midVal2 = (j + k / 2 - 1 < nums2.length) ? nums2[j + k / 2 - 1] : Integer.MA
        // 二分法的核心，比较这两个数组的第K/2小的数字midVal1和midVal2的大小，如果第一个数组的第K/2
        if(midVal1 < midVal2){
            return findKth(nums1, i + k / 2, nums2, j , k - k / 2);
        }else{
            return findKth(nums1, i, nums2, j + k / 2 , k - k / 2);
        }
    }
}
```