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);
        }
    }
}
```

5. 最长回文子串

> 给你一个字符串 s，找到 s 中最长的回文子串

```cpp
// 动态规划
// P[i,j]表示字符串 s 的第 i 到 j 个字母组成的串是否为回文串
// 状态转移方程
// p[i,j] = p[i+1,j-1] ^ (si == sj)
// 边界
// p[i,i] = 1;
// p[i,i+1] = (si == si+1)
class Solution {
public:
    string longestPalindrome(string s) {
        int n = s.size();
        if (n < 2) {
            return s;
        }

        int maxLen = 1;
        int begin = 0;
        // dp[i][j] 表示 s[i..j] 是否是回文串
        vector<vector<int>> dp(n, vector<int>(n));
        // 初始化：所有长度为 1 的子串都是回文串
        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
        }
        // 递推开始
        // 先枚举子串长度
        for (int L = 2; L <= n; L++) {
            // 枚举左边界，左边界的上限设置可以宽松一些
            for (int i = 0; i < n; i++) {
                // 由 L 和 i 可以确定右边界，即 j - i + 1 = L 得
                int j = L + i - 1;
                // 如果右边界越界，就可以退出当前循环
                if (j >= n) {
                    break;
                }

                if (s[i] != s[j]) {
                    dp[i][j] = false;
                } else {
                    if (j - i < 3) {
                        dp[i][j] = true;
                    } else {
                        dp[i][j] = dp[i + 1][j - 1];
                    }
                }

                // 只要 dp[i][L] == true 成立，就表示子串 s[i..L] 是回文，此时记录回文长度和起始
                if (dp[i][j] && j - i + 1 > maxLen) {
                    maxLen = j - i + 1;
                    begin = i;
                }
```

```
            }
        }
        return s.substr(begin, maxLen);
    }
};

// 方法二
// 我们枚举所有的「回文中心」并尝试「扩展」，直到无法扩展为止，此时的回文串长度即为此「回文中心」下的最长
class Solution {
public:
    pair<int, int> expandAroundCenter(const string& s, int left, int right) {
        while (left >= 0 && right < s.size() && s[left] == s[right]) {
            --left;
            ++right;
        }
        return {left + 1, right - 1};
    }

    string longestPalindrome(string s) {
        int start = 0, end = 0;
        for (int i = 0; i < s.size(); ++i) {
            auto [left1, right1] = expandAroundCenter(s, i, i);
            auto [left2, right2] = expandAroundCenter(s, i, i + 1);
            if (right1 - left1 > end - start) {
                start = left1;
                end = right1;
            }
            if (right2 - left2 > end - start) {
                start = left2;
                end = right2;
            }
        }
        return s.substr(start, end - start + 1);
    }
};
```

6. N字形变换

> 将一个给定字符串 s 根据给定的行数 numRows ， 以从上往下、从左到右进行 Z 字形排列

```
class Solution {
public:
    string convert(string s, int numRows) {
        string ans;
        vector<string> matric(numRows);
        int len = s.size();
        int column = 0;
        int flag = -1;

        if(numRows == 1 || len <= numRows){
            return s;
        }

        for(int i=0; i<len; i++){
            matric[column] += s[i];
            if(column==0 || column==numRows-1){ // 行首行尾变向
                flag = -flag;
            }
            column+= flag;
        }
        for(auto &row : matric){
            ans += row;
        }
        return ans;

    }
};
```

7. 整数反转

> 给你一个 32 位的有符号整数 x ， 返回将 x 中的数字部分反转后的结果.如果反转后整数超过 32 位的有符号整数的范围 `[-2^31, 2^31 - 1]` ， 就返回 0

```
class Solution {
public:
    int reverse(int x) {
        int rev = 0;
        while (x != 0) {
            if (rev < INT_MIN / 10 || rev > INT_MAX / 10) {
                return 0;
            }
            int digit = x % 10;
            x /= 10;
            rev = rev * 10 + digit;
        }
        return rev;
    }
};
```

8. 字符串转换整数 (atoi)

> 函数 myAtoi(string s) 的算法如下：
>
> - 读入字符串并丢弃无用的前导空格
> - 检查下一个字符（假设还未到字符末尾）为正还是负号，读取该字符（如果有）。 确定最终结果是负数还是正数。 如果两者都不存在，则假定结果为正。
> - 读入下一个字符，直到到达下一个非数字字符或到达输入的结尾。字符串的其余部分将被忽略。
> - 将前面步骤读入的这些数字转换为整数（即，"123" -> 123， "0032" -> 32）。如果没有读入数字，则整数为 0 。必要时更改符号（从步骤 2 开始）。
> - 如果整数数超过 32 位有符号整数范围 [–2^31，2^31 – 1] ，需要截断这个整数，使其保持在这个范围内。具体来说，小于 –2^31 的整数应该被固定为 –2^31 ，大于 2^31 – 1 的整数应该被固定为 2^31 – 1 。
> - 返回整数作为最终结果。

```cpp
class Solution {
public:
    int myAtoi(string str) {
        unsigned long len = str.length();

        // 去除前导空格
        int index = 0;
        while (index < len) {
            if (str[index] != ' ') {
                break;
            }
            index++;
        }

        if (index == len) {
            return 0;
        }

        int sign = 1;
        // 处理第 1 个非空字符为正负符号，这两个判断需要写在一起
        if (str[index] == '+') {
            index++;
        } else if (str[index] == '-') {
            sign = -1;
            index++;
        }

        // 根据题目限制，只能使用 int 类型
        int res = 0;
        while (index < len) {
            char curChar = str[index];
            if (curChar < '0' || curChar > '9') {
                break;
            }

            if (res > INT_MAX / 10 || (res == INT_MAX / 10 && (curChar - '0') > INT_MAX
                return INT_MAX;
            }
            if (res < INT_MIN / 10 || (res == INT_MIN / 10 && (curChar - '0') > -(INT_MI
                return INT_MIN;
            }

            res = res * 10 + sign * (curChar - '0');
            index++;
        }
        return res;
    }
};
```

9. 回文数

> 给你一个整数 x ，如果 x 是一个回文整数，返回 true ；否则，返回 false
> 回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

```
// class Solution {
// public:
//     bool isPalindrome(int x) {
//         // 转成字符串进行判断
//         string s = to_string(x);
//         // 定义双指针
//         int left = 0, right = s.length() - 1;
//         while(left < right) {
//             // 判断不是回文，返回false
//             if(s[left] != s[right]) {
//                 return false;
//             }
//             // 左右指针移动
//             left ++;
//             right --;
//         }
//         return true;
//     }
// };

class Solution {
    public:
    bool isPalindrome(int x) {
        if (x < 0) return false; // 负数就是false
        long long y = 0; // 防止溢出
        int flag = x; // 留一个做比较
        while(x != 0){  // 常见处理，如上面的力扣7.整数反转
            y = y * 10 + x % 10;
            x = x / 10;
        }
        return flag == y;  // 写成逻辑式，判断是否回文
    }
};
```

10. 正则表达式匹配

> 给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。
>
> - '.' 匹配任意单个字符
> - '*' 匹配零个或多个前面的那一个元素

```
const isMatch = (s, p) => {
  if (s == null || p == null) return false;

  const sLen = s.length, pLen = p.length;

  const dp = new Array(sLen + 1);
  for (let i = 0; i < dp.length; i++) {
    dp[i] = new Array(pLen + 1).fill(false); // 将项默认为false
  }
  // base case
  dp[0][0] = true;
  for (let j = 1; j < pLen + 1; j++) {
    if (p[j - 1] == "*") dp[0][j] = dp[0][j - 2];
  }
  // 迭代
  for (let i = 1; i < sLen + 1; i++) {
    for (let j = 1; j < pLen + 1; j++) {

      if (s[i - 1] == p[j - 1] || p[j - 1] == ".") {
        dp[i][j] = dp[i - 1][j - 1];
      } else if (p[j - 1] == "*") {
        if (s[i - 1] == p[j - 2] || p[j - 2] == ".") {
          dp[i][j] = dp[i][j - 2] || dp[i - 1][j - 2] || dp[i - 1][j];
        } else {
          dp[i][j] = dp[i][j - 2];
        }
      }
    }
  }
  return dp[sLen][pLen]; // 长sLen的s串 是否匹配 长pLen的p串
};
// 作者：xiao_ben_zhu
// 链接：https://leetcode.cn/problems/regular-expression-matching/solution/shou-hui-tu-j
// 来源：力扣（LeetCode）
// 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```