


力扣记录


1. 两数之和

题目描述：

1. 两数之和

提示 

简单  17.8K  

 相关企业

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。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

示例 1：

输入：nums = [2,7,11,15], target = 9
输出：[0,1]
解释：因为 nums[0] + nums[1] == 9，返回 [0, 1]。

示例 2：

输入：nums = [3,2,4], target = 6
输出：[1,2]

示例 3：

输入：nums = [3,3], target = 6
输出：[0,1]

解题代码：

```

class Solution {
    public int[] twoSum(int[] nums, int target) {
        int i = 0;
        int j = 0;
        Loop:for(i = 0; i < nums.length - 1; i++){
            int sub = target - nums[i];
            for(j = i + 1; j < nums.length; j++){
                if(nums[j] == sub){
                    break Loop;
                }
            }
        }
        return new int[]{i, j};
    }
}

```

大佬解法：

时间显著降低

- 标签：哈希映射
- 这道题本身如果通过暴力遍历的话也是很容易解决的，时间复杂度在 $O(n^2)$
- 由于哈希查找的时间复杂度为 $O(1)$ ，所以可以利用哈希容器 map 降低时间复杂度
- 遍历数组 nums，i 为当前下标，每个值都判断map中是否存在 target-nums[i] 的 key 值
- 如果存在则找到了两个值，如果不存在则将当前的 (nums[i], i) 存入 map 中，继续遍历直到找到为止
- 如果最终都没有结果则抛出异常

```

class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for(int i = 0; i < nums.length; i++) {
            if(map.containsKey(target - nums[i])) {
                return new int[] {map.get(target-nums[i]), i};
            }
            map.put(nums[i], i);
        }
        throw new IllegalArgumentException("No two sum solution");
    }
}

```

```
//::value_type 是 std::map 中的一个嵌套类型，它表示 map 中的键值对类型。对于 std::map<int, int>, ::v
//例如 a.insert(map<int,int>::value_type(nums[i],i));
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        map<int,int> a;//提供一对一的hash
        vector<int> b(2,-1);//用来承载结果，初始化一个大小为2，值为-1的容器b
        for(int i=0;i<nums.size();i++)
        {
            if(a.count(target-nums[i])>0)
            {
                b[0]=a[target-nums[i]];
                b[1]=i;
                break;
            }
            a[nums[i]]=i;//反过来放入map中，用来获取结果下标
        }
        return b;
    };
};
```

2. 两数相加

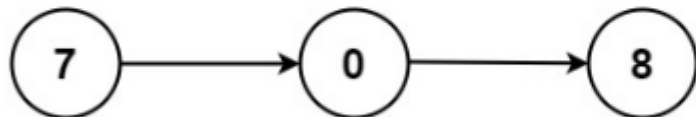
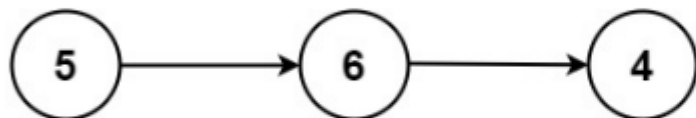
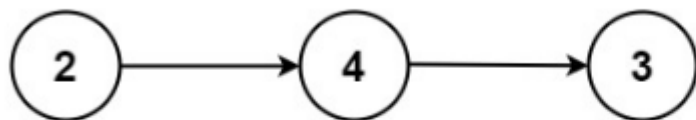
题目描述：

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。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1:



输入: $l_1 = [2,4,3]$, $l_2 = [5,6,4]$

输出: $[7,0,8]$

解释: $342 + 465 = 807$.

示例 2:

输入: $l_1 = [0]$, $l_2 = [0]$

输出: $[0]$

示例 3:

输入: $l_1 = [9,9,9,9,9,9,9]$, $l_2 = [9,9,9,9]$

输出: $[8,9,9,9,0,0,0,1]$

解题代码:

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode result = new ListNode();
        ListNode tmp = result;
        int val = (l1.val + l2.val)%10;
        int carry = (l1.val + l2.val)/10;
        result.val = val;
        l1 = l1.next;
        l2 = l2.next; // 奠基
        while(l1 != null && l2 != null){
            val = (l1.val + l2.val + carry)%10;
            carry = (l1.val + l2.val + carry)/10;
            tmp.next = new ListNode(val);
            tmp = tmp.next;
            l1 = l1.next;
            l2 = l2.next;
        }
        while(l1!= null){
            val = (l1.val + carry)%10;
            carry = (l1.val + carry)/10;
            tmp.next = new ListNode(val);
            tmp = tmp.next;
            l1 = l1.next;
        }
        while(l2!= null){
            val = (l2.val + carry)%10;
            carry = (l2.val + carry)/10;
            tmp.next = new ListNode(val);
            tmp = tmp.next;
            l2 = l2.next;
        }
        if(carry == 1){
            tmp.next = new ListNode(carry);
        }
    }
}

```

```
    }  
    return result;  
}  
}
```

大佬解法：

内存显著降低

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        return getSum(l1, l2, 0);
    }
    public ListNode getSum(ListNode node1, ListNode node2, int carry){
        if(node1 == null && node2 == null){
            if(carry != 0){
                ListNode tempNode = new ListNode(carry);
                return tempNode;
            }
            return null;
        }
        int temp = 0;
        if(node1 != null){
            temp += node1.val;
            node1 = node1.next;
        }
        if(node2 != null){
            temp += node2.val;
            node2 = node2.next;
        }
        temp += carry;
        int val = temp%10;
        ListNode node = new ListNode(val);
        carry = temp/10;
        node.next = getSum(node1, node2, carry);
        return node;
    }
}

```

3.无重复字符的最长子串

题目描述：

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。

示例 1:

输入: `s = "abcabcbb"`

输出: 3

解释: 因为无重复字符的最长子串是 `"abc"`，所以其长度为 3。

示例 2:

输入: `s = "bbbbbb"`

输出: 1

解释: 因为无重复字符的最长子串是 `"b"`，所以其长度为 1。

示例 3:

输入: `s = "pwwkew"`

输出: 3

解释: 因为无重复字符的最长子串是 `"wke"`，所以其长度为 3。

请注意，你的答案必须是 **子串** 的长度，`"pwke"` 是一个子序列，不是子串。

解题代码:

内存其实已经很好了


```

class Solution {
    public int lengthOfLongestSubstring(String s) {
        ArrayList<Character> list = new ArrayList<>();
        int length=0;
        int max=0;

        for(char character : s.toCharArray()){
            if(list.contains(character)){
                int loc = -1;
                while(loc!=list.indexOf(character)){
                    list.remove(0);
                    length--;
                }
            }
            list.add(character);
            length++;
            if(length>max){
                max = length;
            }
        }
        return max;
    }
}

```

大佬解法：

消耗更少的时间

```

class Solution {
    public int lengthOfLongestSubstring(String s) {
// String s=new String();
//         Scanner sc=new Scanner(System.in);
//         s=sc.nextLine();
        char []a= s.toCharArray();
        int []exist=new int[256];
        for (int i = 0; i < exist.length; i++) {
            exist[i]=0;
        }

        int r=0,l=0;

        int max=0;

        while(r<a.length)
        {
            if(exist[a[r]]==1)
            {
                exist[a[l]]=0;
                l++;
            }
            else {
                exist[a[r]]=1;
                r++;
                int t=r-l;
                if(t>max)max=t;
            }
        }
        //System.out.println(max);
        return max;
    }
}

```

消耗更少的内存（实测其实也没有）

```

class Solution {
    public int lengthOfLongestSubstring(String s) {
        int start = 0;
        int end = 0;
        Map<Character, Integer> indices = new HashMap<>();
        int result = 0;
        while (end < s.length()) {
            char c = s.charAt(end);
            if (!indices.containsKey(c) || indices.get(c) < start) {
                indices.put(c, end);
            } else {
                result = Math.max(result, end - start);
                start = indices.get(c) + 1;
                indices.put(c, end);
            }
            end++;
        }
        result = Math.max(result, end - start);
        return result;
    }
}

```

4. 寻找两个正序数组的中位数

题目描述：

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 $O(\log(m+n))$ 。

示例 1：

输入: `nums1 = [1,3]`, `nums2 = [2]`

输出: 2.00000

解释: 合并数组 = `[1,2,3]`，中位数 2

示例 2：

输入: `nums1 = [1,2]`, `nums2 = [3,4]`

输出: 2.50000

解释: 合并数组 = `[1,2,3,4]`，中位数 $(2 + 3) / 2 = 2.5$

解题代码：

时间100%，内存74.39%

```

class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        if(nums1.length == 0){//如果一个为空
            if(nums2.length%2 == 1){
                return nums2[nums2.length/2];
            }else{
                return ((double)nums2[nums2.length/2] + (double)nums2[nums2.length/2-1])/2;
            }
        }else if(nums2.length == 0){
            if(nums1.length%2 == 1){
                return nums1[nums1.length/2];
            }else{
                return ((double)nums1[nums1.length/2] + (double)nums1[nums1.length/2-1])/2;
            }
        }
        double num = (nums1[0] <= nums2[0])?nums1[0]:nums2[0];
        int i,j;
        if(num==nums1[0]){
            i = 1;
            j = 0;
        }else{
            i = 0;
            j = 1;
        }
        int round = 1;
        if((nums1.length + nums2.length)%2 == 1){
            for(;i<=nums1.length && j <= nums2.length;){
                if(i == nums1.length){
                    num = nums2[j];
                    j++;
                }else if(j == nums2.length){
                    num = nums1[i];
                    i++;
                }
                else if(nums1[i] < nums2[j]){
                    num = nums1[i];
                    i++;
                }else{
                    num = nums2[j];
                    j++;
                }
            }
            if(++round == (nums1.length + nums2.length)/2 + 1){
                return num;
            }
        }
    }
}

```

```

    }
}
}else{
    for(;i <= nums1.length && j <= nums2.length;){
        if(++round == (nums1.length + nums2.length)/2 + 1){
            double next;
            if(i == nums1.length){
                next = nums2[j];
            }else if(j==nums2.length){
                next=nums1[i];
            }else{
                next = (nums1[i] <= nums2[j])?nums1[i]:nums2[j];
            }

            return (num + next)/2;
        }
        if(i == nums1.length){
            num = nums2[j];
            j++;
        }else if(j == nums2.length){
            num = nums1[i];
            i++;
        }else if(nums1[i] < nums2[j]){
            num = nums1[i];
            i++;
        }else{
            num = nums2[j];
            j++;
        }
    }
}
return 0;
}
}

```

大佬解法：

```
import java.util.ArrayList;

class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int m = nums1.length, n = nums2.length, left = (m + n + 1) / 2, right = (m + n + 2) / 2;
        return (findKth(nums1, nums2, left) + findKth(nums1, nums2, right)) / 2.0;
    }
    int findKth(int[] nums1, int[] nums2, int k) {
        int m = nums1.length, n = nums2.length;
        if (m == 0) return nums2[k - 1];
        if (n == 0) return nums1[k - 1];
        if (k == 1) return Math.min(nums1[0], nums2[0]);
        int i = Math.min(m, k / 2), j = Math.min(n, k / 2);
        if (nums1[i - 1] > nums2[j - 1]) {
            return findKth(nums1, Arrays.copyOfRange(nums2, j, n), k - j);
        } else {
            return findKth(Arrays.copyOfRange(nums1, i, m), nums2, k - i);
        }
    }
}
```

5. 最长回文子串

题目描述：

给你一个字符串 `s`，找到 `s` 中最长的回文子串。

如果字符串的反序与原始字符串相同，则该字符串称为回文字符串。

示例 1:

输入: `s = "babad"`

输出: `"bab"`

解释: `"aba"` 同样是符合题意的答案。

示例 2:

输入: `s = "cbbd"`

输出: `"bb"`

提示:

- `1 <= s.length <= 1000`
- `s` 仅由数字和英文字母组成

大佬解法:

动态规划:

对于一个子串而言，如果它是回文串，并且长度大于 2，那么将它首尾的两个字母去除之后，它仍然是个回文串。例如对于字符串 “ababa”，如果我们已经知道 “bab” 是回文串，那么 “ababa” 一定是回文串，这是因为它的首尾两个字母都是 “a”。

根据这样的思路，我们就可以用动态规划的方法解决本题。我们用 $P(i, j)$ 表示字符串 s 的第 i 到 j 个字母组成的串（下文表示成 $s[i : j]$ ）是否为回文串：

$$P(i, j) = \begin{cases} \text{true}, & \text{如果子串 } S_i \dots S_j \text{ 是回文串} \\ \text{false}, & \text{其它情况} \end{cases}$$

这里的「其它情况」包含两种可能性：

- $s[i, j]$ 本身不是一个回文串；
- $i > j$ ，此时 $s[i, j]$ 本身不合法。

那么我们就可以写出动态规划的状态转移方程：

$$P(i, j) = P(i + 1, j - 1) \wedge (S_i == S_j)$$

也就是说，只有 $s[i + 1 : j - 1]$ 是回文串，并且 s 的第 i 和 j 个字母相同时， $s[i : j]$ 才会是回文串。

上文的所有讨论是建立在子串长度大于 2 的前提之上的，我们还需要考虑动态规划中的边界条件，即子串的长度为 1 或 2。对于长度为 1 的子串，它显然是个回文串；对于长度为 2 的子串，只要它的两个字母相同，它就是一个回文串。因此我们可以写出动态规划的边界条件：

$$\begin{cases} P(i, i) = \text{true} \\ P(i, i + 1) = (S_i == S_{i+1}) \end{cases}$$

根据这个思路，我们就可以完成动态规划了，最终的答案即为所有 $P(i, j) = \text{true}$ 中 $j - i + 1$ （即子串长度）的最大值。注意：在状态转移方程中，我们是从长度较短的字符串向长度较长的字符串进行转移的，因此一定要注意动态规划的循环顺序。

- 时间复杂度： $O(n^2)$ ，其中 n 是字符串的长度。动态规划的状态总数为 $O(n^2)$ ，对于每个状态，我们需要转移的时间为 $O(1)$ 。
- 空间复杂度： $O(n^2)$ ，即存储动态规划状态需要的空间。

```

#include <iostream>
#include <string>
#include <vector>

using namespace std;

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位置
        if (dp[i][j] && j - i + 1 > maxLen) {
            maxLen = j - i + 1;
            begin = i;
        }
    }
}
return s.substr(begin, maxLen);
}
};

```

中心扩展算法:

我们仔细观察一下方法一中的状态转移方程:



$$\begin{cases} P(i, i) &= \text{true} \\ P(i, i+1) &= (S_i == S_{i+1}) \\ P(i, j) &= P(i+1, j-1) \wedge (S_i == S_j) \end{cases}$$

找出其中的状态转移链:

$$P(i, j) \leftarrow P(i+1, j-1) \leftarrow P(i+2, j-2) \leftarrow \dots \leftarrow \text{某一边界情况}$$

可以发现，所有的状态在转移的时候的可能性都是唯一的。也就是说，我们可以从每一种边界情况开始「扩展」，也可以得出所有的状态对应的答案。

边界情况即为子串长度为 1 或 2 的情况。我们枚举每一种边界情况，并从对应的子串开始不断地向两边扩展。如果两边的字母相同，我们就可以继续扩展，例如从 $P(i+1, j-1)$ 扩展到 $P(i, j)$ ；如果两边的字母不同，我们就可以停止扩展，因为在这之后的子串都不能是回文串了。

聪明的读者此时应该可以发现，「边界情况」对应的子串实际上就是我们「扩展」出的回文串的「回文中心」。方法二的本质即为：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」下的最长回文串长度。我们对所有的长度求出最大值，即可得到最终的答案。

- 时间复杂度： $O(n^2)$ ，其中 n 是字符串的长度。长度为 1 和 2 的回文中心分别有 n 和 $n-1$ 个，每个回文中心最多会向外扩展 $O(n)$ 次。
- 空间复杂度： $O(1)$ 。

```

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);
    }
};

```

Manacher 算法(没看懂):

还有一个复杂度为 $O(n)$ 的 Manacher 算法。然而本算法十分复杂，一般不作为面试内容。这里给出，仅供有兴趣的同学挑战自己。

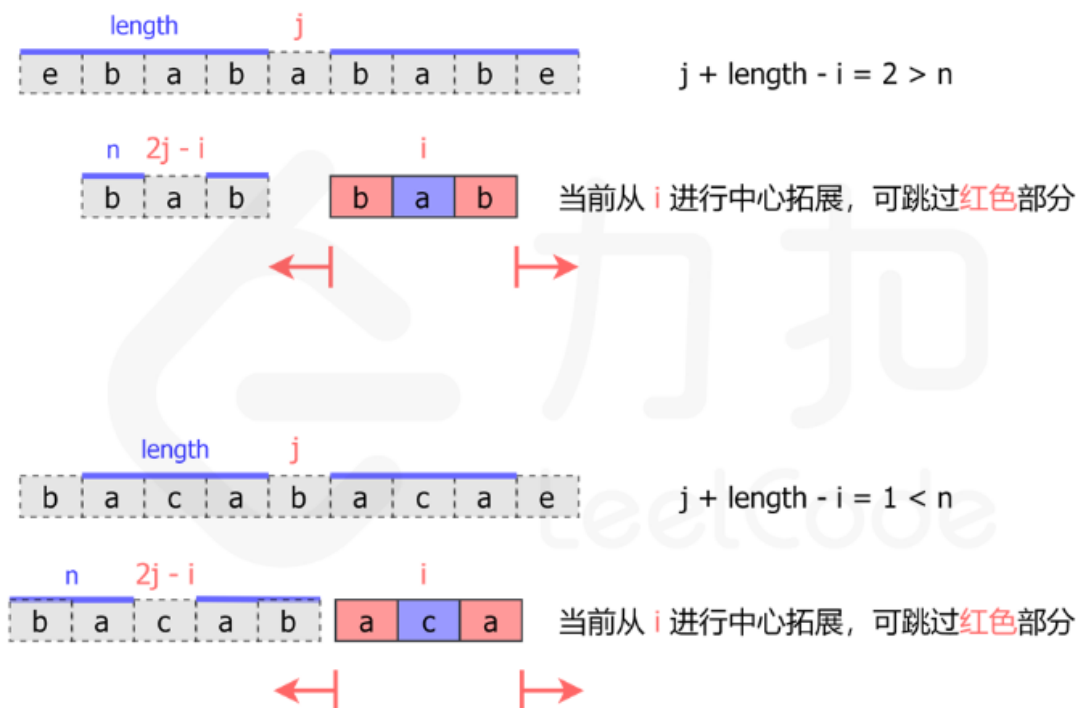
为了表述方便，我们定义一个新概念**臂长**，表示中心扩展算法向外扩展的长度。如果一个位置的最大回文字符串长度为 $2 * length + 1$ ，其臂长为 $length$ 。

下面的讨论只涉及长度为奇数的回文字符串。长度为偶数的回文字符串我们将会在最后与长度为奇数的情况统一起来。

思路与算法

在中心扩展算法的过程中，我们能够得出每个位置的臂长。那么当我们要得出以下一个位置 i 的臂长时，能不能利用之前得到的信息呢？

答案是肯定的。具体来说，如果位置 j 的臂长为 $length$ ，并且有 $j + length > i$ ，如下图所示：



当在位置 i 开始进行中心拓展时，我们可以先找到 i 关于 j 的对称点 $2 * j - i$ 。那么如果点 $2 * j - i$ 的臂长等于 n ，我们就可以知道，点 i 的臂长至少为 $\min(j + length - i, n)$ 。那么我们就可以直接跳过 i 到 $i + \min(j + length - i, n)$ 这部分，从 $i + \min(j + length - i, n) + 1$ 开始拓展。

我们只需要在中心扩展法的过程中记录右臂在最右边的回文字符串，将其中心作为 j ，在计算过程中就能最大限度地避免重复计算。

那么现在还有一个问题：如何处理长度为偶数的回文字符串呢？

我们可以通过一个特别的操作将奇偶数的情况统一起来：我们向字符串的头尾以及每两个字符中间添加一个特殊字符 $\#$ ，比如字符串 `aaba` 处理后会变成 `#a#a#b#a#`。那么原先长度为偶数的回文字符串 `aa` 会变成长度为奇数的回文字符串 `#a#a#`，而长度为奇数的回文字符串 `aba` 会变成长度仍然为奇数的回文字符串 `#a#b#a#`，我们就

个需要再考虑长度为偶数的回文子字符串了。

注意这里的特殊字符不需要是没有出现过的字母，我们可以使用任何一个字符来作为这个特殊字符。这是因为，当我们只考虑长度为奇数的回文字符串时，每次我们比较的两个字符奇偶性一定是相同的，所以原来字符串中的字符不会与插入的特殊字符互相比对，不会因此产生问题。

- 时间复杂度： $O(n)$ ，其中 n 是字符串的长度。由于对于每个位置，扩展要么从当前已知的回文子串扩展，要么从起始位置扩展，而不会重复，因此算法复杂度为 $O(n)$ 。
- 空间复杂度： $O(n)$ ，我们需要 $O(n)$ 的空间记录每个位置的回文长度。

```

class Solution {
public:
    int expand(const string& s, int left, int right) {
        while (left >= 0 && right < s.size() && s[left] == s[right]) {
            --left;
            ++right;
        }
        return (right - left - 2) / 2;
    }

    string longestPalindrome(string s) {
        int start = 0, end = -1;
        string t = "#";
        for (char c: s) {
            t += c;
            t += '#';
        }
        t += '#';
        s = t;

        vector<int> arm_len;
        int right = -1, j = -1;
        for (int i = 0; i < s.size(); ++i) {
            int cur_arm_len;
            if (right >= i) {
                int i_sym = j * 2 - i;
                int min_arm_len = min(arm_len[i_sym], right - i);
                cur_arm_len = expand(s, i - min_arm_len, i + min_arm_len);
            } else {
                cur_arm_len = expand(s, i, i);
            }
            arm_len.push_back(cur_arm_len);
            if (i + cur_arm_len > right) {
                j = i;
                right = i + cur_arm_len;
            }
            if (cur_arm_len * 2 + 1 > end - start) {
                start = i - cur_arm_len;
                end = i + cur_arm_len;
            }
        }

        string ans;

```

```
    for (int i = start; i <= end; ++i) {  
        if (s[i] != '#') {  
            ans += s[i];  
        }  
    }  
    return ans;  
}  
};
```

6.N 字形变换

题目描述：

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。

比如输入字符串为 `"PAYPALISHIRING"` 行数为 `3` 时，排列如下：

```
P   A   H   N
A P L S I I G
Y   I   R
```

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如：`"PAHNAPLSIIGYIR"`。

请你实现这个将字符串进行指定行数变换的函数：

```
string convert(string s, int numRows);
```

示例 1:

```
输入: s = "PAYPALISHIRING", numRows = 3
输出: "PAHNAPLSIIGYIR"
```

示例 2:

```
输入: s = "PAYPALISHIRING", numRows = 4
输出: "PINALSIGYAHRPI"
解释:
P       I       N
A   L S   I   G
Y A   H R
P       I
```

示例 3:

```
输入: s = "A", numRows = 1
输出: "A"
```

解题代码:

时间27.46%，内存34.24%

```

class Solution {
    public String convert(String s, int numRows) {
        if(numRows == 1){
            return s;
        }

        String result = "";
        for(int i=0;i < numRows; i++){
            boolean first_flag = true;
            for(int loc = i;loc < s.length();loc+=(2*(numRows-i-1))){
                if(first_flag){
                    result=result + s.charAt(loc);
                    first_flag = false;
                }else{
                    if((numRows-i-1)!=0 )
                        result=result + s.charAt(loc);
                    if(i!=0){
                        loc+=(2*i);
                        if(loc < s.length())
                            result=result + s.charAt(loc);
                    }
                }
            }
        }
        return result;
    }
}

```

大佬解法：

更少的时间

```

class Solution {
    public String convert(String s, int numRows) {

        if (numRows == 1) return s;

        char[] string = s.toCharArray();
        char[] ans = new char[string.length];
        int add = (numRows - 1) * 2; // add: 每个 V 字形的长度
        int i = 1; // i: 第一个 V 字形的起始位置
        int j = add; // j: 第一个 V 字形的结束位置
        int k = i; // k: 下一个 V 字形的起始位置
        int l = j; // l: 下一个 V 字形的结束位置
        int n = 1; // n: 存储字符数组 ans 中下一个要填充的位置
        ans[0] = string[0];
        // 第一行
        while (l < string.length) {
            ans[n] = string[l];
            l += add;
            n++;
        }
        j--;
        l = j;

        // 第i行和倒数第i行
        while (i < j) {
            while (l < string.length) {
                ans[n++] = string[k];
                ans[n++] = string[l];
                k += add;
                l += add;
            }
            if (k < string.length) ans[n++] = string[k];
            i++;
            j--;
            k = i;
            l = j;
        }
        // 最后一行
        while (l < string.length) {
            ans[n] = string[l];
            l += add;
            n++;
        }
    }
}

```

```
        return String.valueOf(ans);  
    }  
}
```

7. 整数反转

题目描述：

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。

假设环境不允许存储 64 位整数（有符号或无符号）。

示例 1：

输入： $x = 123$

输出：321

示例 2：

输入： $x = -123$

输出：-321

示例 3：

输入： $x = 120$

输出：21

示例 4：

输入： $x = 0$

输出：0

解题代码：

时间17.61%，内存52.49%

```

class Solution {
    public int reverse(int x) {
        boolean flag = true;
        if(x == 0){
            return x;
        }else if(x<0){
            x = Math.abs(x);
            flag = false;
        }
        ArrayList<Integer> list = new ArrayList<>();
        while(x >= 10){
            list.add(x%10);
            x = x/10;
        }
        if(x>0)
            list.add(x);
        int ans = 0;
        int i = (int)Math.pow(10,list.size()-1);

        for(int ele:list){
            int tmp = ans;
            ans = ans + ele*i;
            i = i/10;
        }
        if(list.size()== 10){
            if(ans < 2000000000){// 这儿判断超阈值其实有问题
                return 0;
            }
        }

        if(flag)
            return ans;
        else
            return -ans;
    }
}

```

大佬解法：

```
class Solution {  
    public int reverse(int x) {  
        long ans = 0;  
        while (x != 0) {  
            ans = ans * 10 + x % 10;  
            x /= 10;  
        }  
        return ans < Integer.MIN_VALUE || ans > Integer.MAX_VALUE ? 0 : (int) ans;  
    }  
}
```

8. 字符串转换整数 (atoi)

题目描述：

请你来实现一个 `myAtoi(string s)` 函数，使其能将字符串转换成一个 32 位有符号整数（类似 C/C++ 中的 `atoi` 函数）。

函数 `myAtoi(string s)` 的算法如下：

1. 读入字符串并丢弃无用的前导空格
2. 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。
3. 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。
4. 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）。
5. 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 $2^{31} - 1$ 。
6. 返回整数作为最终结果。

注意：

- 本题中的空白字符只包括空格字符 ' '。
- 除前导空格或数字后的其余字符串外，请勿忽略任何其他字符。

示例 1：

输入：s = "42"

输出：42

解释：加粗的字符串为已经读入的字符，插入符号是当前读取的字符。

第 1 步："42"（当前没有读入字符，因为没有前导空格）

^

第 2 步："42"（当前没有读入字符，因为这里不存在 '-' 或者 '+'）

^

第 3 步："42"（读入 "42"）

^

解析得到整数 42。

由于 "42" 在范围 $[-2^{31}, 2^{31} - 1]$ 内，最终结果为 42。

示例 2：

输入：s = " -42"

输出：-42

解释：

第 1 步: " -42" (读入前导空格, 但忽视掉)

第 2 步: " - 42" (读入 '-' 字符, 所以结果应该是负数)

第 3 步: " -42" (读入 "42")

解析得到整数 -42 。

由于 "-42" 在范围 $[-2^{31}, 2^{31} - 1]$ 内, 最终结果为 -42 。

示例 3:

输入: s = "4193 with words"

输出: 4193

解释:

第 1 步: "4193 with words" (当前没有读入字符, 因为没有前导空格)

第 2 步: "4193 with words" (当前没有读入字符, 因为这里不存在 '-' 或者 '+')

第 3 步: "4193 with words" (读入 "4193"; 由于下一个字符不是一个数字, 所以读入停止)

解析得到整数 4193 。

由于 "4193" 在范围 $[-2^{31}, 2^{31} - 1]$ 内, 最终结果为 4193 。

提示:

- `0 <= s.length <= 200`
- s 由英文字母 (大写和小写)、数字 (`0-9`)、`' '`、`'+'`、`'-'` 和 `'.'` 组成

解题代码:

时间6.67%, 内存8.60%


```

class Solution {
    public int myAtoi(String s) {
        char[] string = s.toCharArray();
        String ans = "";
        boolean flag = true;
        boolean flag_symbol = true;
        boolean flag_space = true;
        for(char c:string){
            if(c == ' ' && flag_space){
                continue;
            }else if((c == '+' || c == '-') && ans == "" && flag_symbol){
                if(c == '-'){
                    flag = false;
                }
                flag_symbol = false;
            }else if(c >= '0' && c <= '9'){
                ans = ans + c;
                if(ans.equals("0")){
                    ans = "";
                }
            }else{
                break;
            }
            flag_space = false;
            flag_symbol = false;
        }
        if(ans == ""){
            return 0;
        }
        if(ans.length() > 10){
            if(flag)
                return Integer.MAX_VALUE;
            else
                return Integer.MIN_VALUE;
        }
        if(flag){
            if(Long.parseLong(ans) > Integer.MAX_VALUE){
                return Integer.MAX_VALUE;
            }else{
                return Integer.parseInt(ans);
            }
        }else{
            if(-Long.parseLong(ans) <= Integer.MIN_VALUE){

```

```

        return Integer.MIN_VALUE;
    }else{
        return -Integer.parseInt(ans);
    }
}
}
}
}

```

大佬解法：

```

class Solution {
    public int myAtoi(String s) {
        long sum=0;
        s=s.trim();//它调用了字符串的 trim 方法，将字符串两端的空格去除
        int f=1;

        for(int i=0;i<s.length();i++){

            if(i==0 && s.charAt(0)=='-')f=-1;
            else if (i==0 && s.charAt(0)=='+')f=1;
            else if(s.charAt(i)<'0' || s.charAt(i)>'9')break;
            else{
                sum=sum*10+(s.charAt(i)-'0');
            }
            if(sum>Integer.MAX_VALUE && f>0)return Integer.MAX_VALUE;
            if(f*sum<Integer.MIN_VALUE && f<0)return Integer.MIN_VALUE;
        }
        return (int)sum*f;
    }
}

```

9. 回文数

题目描述：

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。

- 例如，`121` 是回文，而 `123` 不是。

示例 1:

输入: `x = 121`

输出: `true`

示例 2:

输入: `x = -121`

输出: `false`

解释: 从左向右读，为 `-121`。从右向左读，为 `121-`。因此它不是一个回文数。

示例 3:

输入: `x = 10`

输出: `false`

解释: 从右向左读，为 `01`。因此它不是一个回文数。

解题代码:

时间98.39%，内存58.12%

```
class Solution {
    public boolean isPalindrome(int x) {
        if(x < 0){
            return false;
        }
        if(x == reverse(x))
            return true;
        else
            return false;
    }
    public int reverse(int x) {
        long ans = 0;
        while (x != 0) {
            ans = ans * 10 + x % 10;
            x /= 10;
        }
        return ans < Integer.MIN_VALUE || ans > Integer.MAX_VALUE ? 0 : (int) ans;
    }
}
```

大佬解法：

更小的内存

```
class Solution {  
    public boolean isPalindrome(int x) {  
        if(x < 0) {  
            return false;  
        }  
        List<Integer> list = new ArrayList<>();  
        while(x > 0) {  
            list.add(x % 10);  
            x = x / 10;  
        }  
        int left = 0, right = list.size() - 1;  
        while(left < right) {  
            if(list.get(left) != list.get(right)) {  
                return false;  
            }  
            left++;  
            right--;  
        }  
        return true;  
    }  
}
```

10. 正则表达式匹配（还不会）

题目描述：

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

所谓匹配，是要涵盖 整个 字符串 `s` 的，而不是部分字符串。

示例 1:

```
输入: s = "aa", p = "a"
输出: false
解释: "a" 无法匹配 "aa" 整个字符串。
```

示例 2:

```
输入: s = "aa", p = "a*"
输出: true
解释: 因为 '*' 代表可以匹配零个或多个前面的那一个元素，在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。
```

示例 3:

```
输入: s = "ab", p = ".*"
输出: true
解释: ".*" 表示可匹配零个或多个 ('*') 任意字符 ('.').
```

官方解题代码：

动态规划

题目中的匹配是一个「逐步匹配」的过程：我们每次从字符串 p 中取出一个字符或者「字符 + 星号」的组合，并在 s 中进行匹配。对于 p 中一个字符而言，它只能在 s 中匹配一个字符，匹配的方法具有唯一性；而对于 p 中字符 + 星号的组合而言，它可以在 s 中匹配任意自然数个字符，并不具有唯一性。因此我们可以考虑使用动态规划，对匹配的方案进行枚举。

我们用 $f[i][j]$ 表示 s 的前 i 个字符与 p 中的前 j 个字符是否能够匹配。在进行状态转移时，我们考虑 p 的第 j 个字符的匹配情况：

- 如果 p 的第 j 个字符是一个小写字母，那么我们必须要在 s 中匹配一个相同的小写字母，即

$$f[i][j] = \begin{cases} f[i-1][j-1], & s[i] = p[j] \\ \text{false}, & s[i] \neq p[j] \end{cases}$$

也就是说，如果 s 的第 i 个字符与 p 的第 j 个字符不相同，那么无法进行匹配；否则我们可以匹配两个字符串的最后一个字符，完整的匹配结果取决于两个字符串前面的部分。

- 如果 p 的第 j 个字符是 `*`，那么就表示我们可以对 p 的第 $j-1$ 个字符匹配任意自然数次。在匹配 0 次的情况下，我们有

$$f[i][j] = f[i][j-2]$$

也就是我们「浪费」了一个字符 + 星号的组合，没有匹配任何 s 中的字符。

在匹配 1, 2, 3, ... 次的情况下，类似地我们有

$$\begin{aligned} f[i][j] &= f[i-1][j-2], & \text{if } s[i] &= p[j-1] \\ f[i][j] &= f[i-2][j-2], & \text{if } s[i-1] &= s[i] = p[j-1] \\ f[i][j] &= f[i-3][j-2], & \text{if } s[i-2] &= s[i-1] = s[i] = p[j-1] \\ &\dots\dots \end{aligned}$$

如果我们通过这种方法进行转移，那么我们就需要枚举这个组合到底匹配了 s 中的几个字符，会增导致时间复杂度增加，并且代码编写起来十分麻烦。我们不妨换个角度考虑这个问题：字母 + 星号的组合在匹配的过程中，本质上只会有两种情况：

- 匹配 s 末尾的一个字符，将该字符扔掉，而该组合还可以继续进行匹配；
- 不匹配字符，将该组合扔掉，不再进行匹配。

如果按照这个角度进行思考，我们可以写出很精巧的状态转移方程：

$$f[i][j] = \begin{cases} f[i-1][j] \text{ or } f[i][j-2], & s[i] = p[j-1] \\ f[i][j-2], & s[i] \neq p[j-1] \end{cases}$$

- 在任意情况下，只要 $p[j]$ 是 `.`，那么 $p[j]$ 一定成功匹配 s 中的任意一个小写字母。

最终的状态转移方程如下：

$$f[i][j] = \begin{cases} \text{if } (p[j] \neq '*') = \begin{cases} f[i-1][j-1], & \text{matches}(s[i], p[j]) \\ \text{false}, & \text{otherwise} \end{cases} \\ \text{otherwise} = \begin{cases} f[i-1][j] \text{ or } f[i][j-2], & \text{matches}(s[i], p[j-1]) \\ f[i][j-2], & \text{otherwise} \end{cases} \end{cases}$$

其中 $\text{matches}(x, y)$ 判断两个字符是否匹配的辅助函数。只有当 y 是 `.` 或者 x 和 y 本身相同时，这两个字符才会匹配。

细节

动态规划的边界条件为 $f[0][0] = \text{true}$ ，即两个空字符串是可以匹配的。最终的答案即为 $f[m][n]$ ，其中 m 和 n 分别是字符串 s 和 p 的长度。由于大部分语言中，字符串的字符下标是从 0 开始的，因此在实现上面的状态转移方程时，需要注意状态中每一维下标与实际字符下标的对应关系。

在上面的状态转移方程中，如果字符串 p 中包含一个「字符 + 星号」的组合（例如 `a*`），那么在进行状态转移时，会先将 `a` 进行匹配（当 $p[j]$ 为 `a` 时），再将 `a*` 作为整体进行匹配（当 $p[j]$ 为 `*` 时）。然而，在题目描述中，我们必须将 `a*` 看成一个

整体，因此将 `a` 进行匹配是不符合题目要求的。看来我们进行了额外的状态转移，这样会对最终的答案产生影响吗？这个问题留给读者进行思考。

- 时间复杂度： $O(mn)$ ，其中 m 和 n 分别是字符串 s 和 p 的长度。我们需要计算出所有的状态，并且每个状态在进行转移时的时间复杂度为 $O(1)$ 。
- 空间复杂度： $O(mn)$ ，即为存储所有状态使用的空间。

```
class Solution {
    public boolean isMatch(String s, String p) {
        int m = s.length();
        int n = p.length();

        boolean[][] f = new boolean[m + 1][n + 1];
        f[0][0] = true;
        for (int i = 0; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (p.charAt(j - 1) == '*') {
                    f[i][j] = f[i][j - 2];
                    if (matches(s, p, i, j - 1)) {
                        f[i][j] = f[i][j] || f[i - 1][j];
                    }
                } else {
                    if (matches(s, p, i, j)) {
                        f[i][j] = f[i - 1][j - 1];
                    }
                }
            }
        }
        return f[m][n];
    }

    public boolean matches(String s, String p, int i, int j) {
        if (i == 0) {
            return false;
        }
        if (p.charAt(j - 1) == '.') {
            return true;
        }
        return s.charAt(i - 1) == p.charAt(j - 1);
    }
}
```



```

class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.size();
        int n = p.size();

        auto matches = [&](int i, int j) {
            if (i == 0) {
                return false;
            }
            if (p[j - 1] == '.') {
                return true;
            }
            return s[i - 1] == p[j - 1];
        };

        vector<vector<int>> f(m + 1, vector<int>(n + 1));
        f[0][0] = true;
        for (int i = 0; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (p[j - 1] == '*') {
                    f[i][j] |= f[i][j - 2];
                    if (matches(i, j - 1)) {
                        f[i][j] |= f[i - 1][j];
                    }
                }
                else {
                    if (matches(i, j)) {
                        f[i][j] |= f[i - 1][j - 1];
                    }
                }
            }
        }
        return f[m][n];
    }
};

```

11. 盛最多水的容器

题目描述：

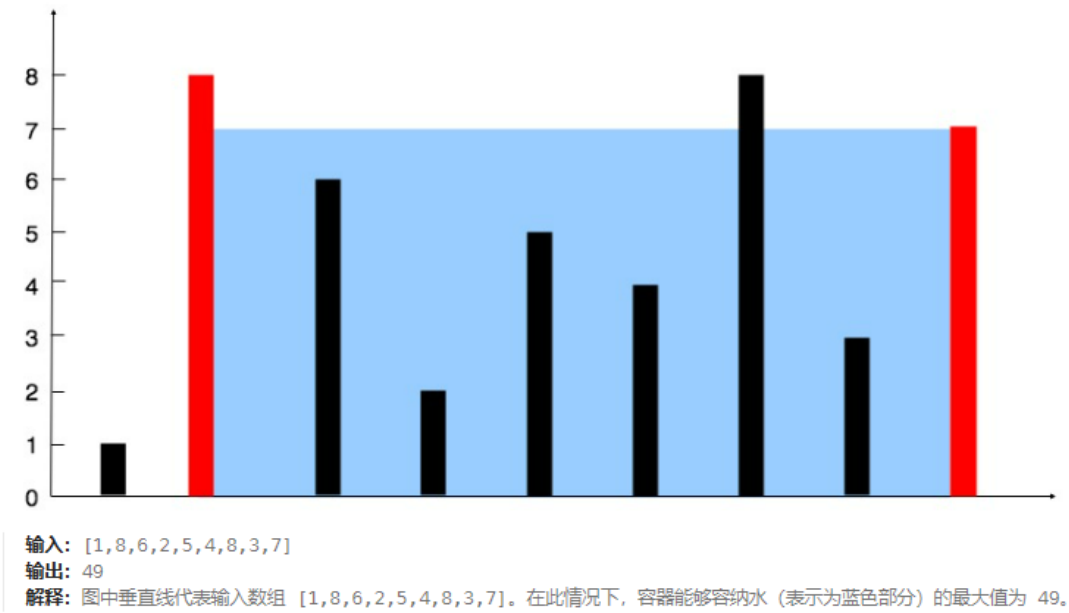
给定一个长度为 `n` 的整数数组 `height`。有 `n` 条垂线，第 `i` 条线的两个端点是 `(i, 0)` 和 `(i, height[i])`。

找出其中的两条线，使得它们与 `x` 轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

说明：你不能倾斜容器。

示例 1:



示例 2:

输入: height = [1,1]
输出: 1

解题代码:

时间91.47%，内存81.38%

```
class Solution {  
    public int maxArea(int[] height) {  
        int max = 0;  
        int i = 0;  
        int j = height.length-1;  
        while(i < j){  
            int tmp = (j-i)*Math.min(height[i],height[j]);  
            if(tmp > max){  
                max = tmp;  
            }  
            if(height[i] < height[j]){  
                i++;  
            }else{  
                j--;  
            }  
        }  
        return max;  
    }  
}
```

12. 整数转罗马数字

题目描述：

罗马数字包含以下七种字符： I， V， X， L， C， D 和 M。

字符	数值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

例如， 罗马数字 2 写做 II ，即为两个并列的 1。12 写做 XII ，即为 X + II 。 27 写做 XXVII, 即为 XX + V + II 。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

- I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
- X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
- C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给你一个整数，将其转为罗马数字。

示例 1:

```
输入: num = 3
输出: "III"
```

示例 2:

```
输入: num = 4
输出: "IV"
```

示例 3:

```
输入: num = 9
输出: "IX"
```

示例 4:

```
输入: num = 58
输出: "LVIII"
解释: L = 50, V = 5, III = 3.
```

示例 5:

```
输入: num = 1994
输出: "MCMXCIV"
解释: M = 1000, CM = 900, XC = 90, IV = 4.
```

解题代码：

时间43.84%，内存11.00%

```
class Solution {
    public String intToRoman(int num) {
        String ans="";
        while(num >= 1000){
            ans = ans + "M";
            num = num - 1000;
        }
        while(num >= 900){
            ans = ans + "CM";
            num = num - 900;
        }
        while(num >= 500){
            ans = ans + "D";
            num = num - 500;
        }
        while(num >= 400){
            ans = ans + "CD";
            num = num - 400;
        }
        while(num >= 100){
            ans = ans + "C";
            num = num - 100;
        }
        while(num >= 90){
            ans = ans + "XC";
            num = num - 90;
        }
        while(num >= 50){
            ans = ans + "L";
            num = num - 50;
        }
        while(num >= 40){
            ans = ans + "XL";
            num = num - 40;
        }
        while(num >= 10){
            ans = ans + "X";
            num = num - 10;
        }
        while(num >= 9){
            ans = ans + "IX";
            num = num - 9;
        }
    }
}
```

```
    while(num >= 5){  
        ans = ans + "V";  
        num = num - 5;  
    }  
    while(num >= 4){  
        ans = ans + "IV";  
        num = num - 4;  
    }  
    while(num >= 1){  
        ans = ans + "I";  
        num = num - 1;  
    }  
    return ans;  
}  
}
```

大佬解法：

本质一样

```

class Solution {
    public String intToRoman(int num) {
        StringBuilder res=new StringBuilder();
        if(num>=1000){for(int i=num/1000;i>0;i--){res.append('M');}num%=1000;}
        if(num>=500){
            if(num>=900){res.append("CM");num%=100;}
            else{res.append('D');num-=500;}
        }
        if(num>=100){
            if(num>=400){res.append("CD");num%=100;}
            else{for(int i=num/100;i>0;i--){res.append('C');}num%=100;}
        }
        if(num>=50){
            if(num>=90){res.append("XC");num%=10;}
            else{res.append('L');num-=50;}
        }
        if(num>=10){
            if(num>=40){res.append("XL");num%=10;}
            else{for(int i=num/10;i>0;i--){res.append('X');}num%=10;}
        }
        if(num>=5){
            if(num>=9){res.append("IX");num%=1;}
            else{res.append('V');num-=5;}
        }
        if(num>=1){
            if(num>=4){res.append("IV");num%=1;}
            else{for(int i=num;i>0;i--){res.append('I');}num%=1;}
        }
        return res.toString();
    }
}

```

13. 罗马数字转整数

题目描述：

罗马数字包含以下七种字符: I, V, X, L, C, D 和 M。

字符	数值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

例如，罗马数字 2 写做 II，即为两个并列的 1。12 写做 XII，即为 X + II。27 写做 XXVII, 即为 XX + V + II。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

- I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
- X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
- C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。

示例 1:

输入: s = "III"
输出: 3

示例 2:

输入: s = "IV"
输出: 4

示例 3:

输入: s = "IX"
输出: 9

示例 4:

输入: s = "LVIII"
输出: 58
解释: L = 50, V= 5, III = 3。

示例 5:

输入: s = "MCMXCIV"
输出: 1994
解释: M = 1000, CM = 900, XC = 90, IV = 4。

解题代码：

时间70.36%，内存73.22%


```

class Solution {
    public int romanToInt(String s) {
        int num = 0;
        s = s + "I";
        for(int i = 0; i < s.length() - 1; i++){
            if(s.charAt(i) == 'M'){
                num += 1000; continue;
            }
            if(s.charAt(i) == 'D'){
                num += 500; continue;
            }
            if(s.charAt(i) == 'C'){
                if(s.charAt(i + 1) == 'M'){
                    i++; num += 900; continue;
                }
                if(s.charAt(i + 1) == 'D'){
                    i++; num += 400; continue;
                }
                num += 100; continue;
            }
            if(s.charAt(i) == 'L'){
                num += 50; continue;
            }
            if(s.charAt(i) == 'X'){
                if(s.charAt(i + 1) == 'C'){
                    i++; num += 90; continue;
                }
                if(s.charAt(i + 1) == 'L'){
                    i++; num += 40; continue;
                }
                num += 10; continue;
            }
            if(s.charAt(i) == 'V'){
                num += 5; continue;
            }
            if(s.charAt(i) == 'I'){
                if(s.charAt(i + 1) == 'X'){
                    i++; num += 9; continue;
                }
                if(s.charAt(i + 1) == 'V'){
                    i++; num += 4; continue;
                }
                num += 1; continue;
            }
        }
    }
}

```

```

    }
}
return num;
}
}

```

大佬解法:

```

class Solution {
    public int romanToInt(String s) {
        int sum = 0;
        int preNum = getValue(s.charAt(0));
        for(int i = 1; i < s.length(); i++) {
            int num = getValue(s.charAt(i));
            if(preNum < num) {
                sum -= preNum;
            } else {
                sum += preNum;
            }
            preNum = num;
        }
        sum += preNum;
        return sum;
    }

    private int getValue(char ch) {
        switch(ch) {
            case 'I': return 1;
            case 'V': return 5;
            case 'X': return 10;
            case 'L': return 50;
            case 'C': return 100;
            case 'D': return 500;
            case 'M': return 1000;
            default: return 0;
        }
    }
}

```

14. 最长公共前缀

题目描述：

编写一个函数来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 `""`。

示例 1：

输入: strs = ["flower","flow","flight"]
输出: "fl"

示例 2：

输入: strs = ["dog","racecar","car"]
输出: ""
解释: 输入不存在公共前缀。

解题代码：

时间9.54%，内存10.22%

```
class Solution {
    public String longestCommonPrefix(String[] strs) {
        String prefix = "";
        Loop:for(int i = 0;;i++){
            if(i >= strs[0].length()){
                break;
            }
            char c = strs[0].charAt(i);
            for(String s:strs){
                if(i >= s.length()){
                    break Loop;
                }
                if(s.charAt(i)!=c){
                    break Loop;
                }
            }
            prefix = prefix + c;
        }
        return prefix;
    }
}
```

大佬解法：

更小内存

```

class Solution {

    public String longestCommonPrefix(String[] strs) {
        // 3 ptrs O(N)
        // 10^2 => n^3
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < strs[0].length(); i++) {
            char curr = strs[0].charAt(i);

            // 同层比较 arr[] 其他人的, 类BFS
            for (int j = 1; j < strs.length; j++) {
                if (i >= strs[j].length() || strs[j].charAt(i) != curr) {
                    return sb.toString();
                }
            }
            sb.append(curr);
        }

        return sb.toString();
    }
}

```

更快时间

```

class Solution {
    public String longestCommonPrefix(String[] strs) {

        if(strs == null || strs.length == 0){
            return "";
        }
        String prefix = strs[0];
        int count = strs.length;
        for(int i = 1; i < count; i++){
            prefix = longestCommonPrefix(prefix, strs[i]);

        }
        return prefix;
    }

    public String longestCommonPrefix(String str1, String str2){
        int length = Math.min(str1.length(), str2.length());
        int index = 0;
        while(index < length && str1.charAt(index) == str2.charAt(index)){
            index++;
        }
        return str1.substring(0, index);
    }
}

```

15. 三数之和

题目描述：

给你一个整数数组 `nums`，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足 `i != j`、`i != k` 且 `j != k`，同时还满足 `nums[i] + nums[j] + nums[k] == 0`。请

你返回所有和为 `0` 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例 1:

输入: `nums = [-1,0,1,2,-1,-4]`

输出: `[[-1,-1,2],[-1,0,1]]`

解释:

`nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0`。

`nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0`。

`nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0`。

不同的三元组是 `[-1,0,1]` 和 `[-1,-1,2]`。

注意，输出的顺序和三元组的顺序并不重要。

示例 2:

输入: `nums = [0,1,1]`

输出: `[]`

解释: 唯一可能的三元组和不为 `0`。

示例 3:

输入: `nums = [0,0,0]`

输出: `[[0,0,0]]`

解释: 唯一可能的三元组和为 `0`。

大佬思路:

- 首先对数组进行排序，排序后固定一个数`nums[i]`，再使用左右指针指向`nums[i]`后面的两端，数字分别为`nums[L]`和`nums[R]`，计算三个数的和`sum`判断是否满足为0，满足则添加进结果集
- 如果`nums[i]`大于0，则三数之和必然无法等于0，结束循环
- 如果`nums[i] == nums[i-1]`，则说明该数字重复，会导致结果重复，所以应该跳过
- 当`sum == 0`时，`nums[L] == nums[L+1]`则会导致结果重复，应该跳过，`L++`
- 当`sum == 0`时，`nums[R] == nums[R-1]`则会导致结果重复，应该跳过，`R--`
- 时间复杂度： $O(n^2)$ ，`n`为数组长度

```

class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        int n = nums.length;
        Arrays.sort(nums);
        List<List<Integer>> ans = new ArrayList<List<Integer>>();
        // 枚举 a
        for (int first = 0; first < n; ++first) {
            // 需要和上一次枚举的数不相同
            if (first > 0 && nums[first] == nums[first - 1]) {
                continue;
            }
            // c 对应的指针初始指向数组的最右端
            int third = n - 1;
            int target = -nums[first];
            // 枚举 b
            for (int second = first + 1; second < n; ++second) {
                // 需要和上一次枚举的数不相同
                if (second > first + 1 && nums[second] == nums[second - 1]) {
                    continue;
                }
                // 需要保证 b 的指针在 c 的指针的左侧
                while (second < third && nums[second] + nums[third] > target) {
                    --third;
                }
                // 如果指针重合，随着 b 后续的增加
                // 就不会有满足 a+b+c=0 并且 b<c 的 c 了，可以退出循环
                if (second == third) {
                    break;
                }
                if (nums[second] + nums[third] == target) {
                    List<Integer> list = new ArrayList<Integer>();
                    list.add(nums[first]);
                    list.add(nums[second]);
                    list.add(nums[third]);
                    ans.add(list);
                }
            }
        }
        return ans;
    }
}

```


16. 最接近的三数之和

题目描述：

给你一个长度为 `n` 的整数数组 `nums` 和一个目标值 `target`。请你从 `nums` 中选出三个整数，使它们的和与 `target` 最接近。

返回这三个数的和。

假定每组输入只存在恰好一个解。

示例 1：

输入：`nums = [-1,2,1,-4]`，`target = 1`

输出：`2`

解释：与 `target` 最接近的和是 `2` ($-1 + 2 + 1 = 2$)。

示例 2：

输入：`nums = [0,0,0]`，`target = 1`

输出：`0`

解题代码：

时间47.90%，内存30.32%

```

class Solution {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int result = nums[0] + nums[1] + nums[2];
        for(int i=0;i<nums.length-2;i++){
            int left = i+1;
            int right = nums.length - 1;
            while(left != right){
                int sum = nums[i] + nums[left] + nums[right];
                if(Math.abs(sum - target) < Math.abs(result - target))
                    result = sum;
                if(sum > target){
                    right--;
                }
                else{
                    left++;
                }
            }
        }
        return result;
    }
}

```

大佬思路：

- 可以添加去重的操作
- 如果 target 的值比 $\text{nums}[i] + \text{nums}[\text{left}] + \text{nums}[\text{left} + 1]$ 的值还小，那么双指针无论怎么取，最后都会取到 $\text{nums}[i] + \text{nums}[\text{left}] + \text{nums}[\text{left} + 1]$ 。同理可证 target 的值比 $\text{nums}[i] + \text{nums}[\text{right}] + \text{nums}[\text{right} - 1]$ 的值还大的情况。所以可以增加一个判断，满足条件的情况下就可以直接取值，而不需要双指针一步步的判断来进行取值，减少了双指针的移动。
- 有些时候，可能会直接找到三数之和等于 target 的情况，此时直接返回结果即可，不需要在进行之后的循环，因为不可能有数比他自己更接近自己了。

```

class Solution {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int result = nums[0] + nums[1] + nums[2];
        for(int i=0;i<nums.length-2;i++){
            int left = i+1;
            int right = nums.length - 1;
            while(left != right){
                int min = nums[i] + nums[left] + nums[left + 1];
                if(target < min){
                    if(Math.abs(result - target) > Math.abs(min - target))
                        result = min;
                    break;
                }
                int max = nums[i] + nums[right] + nums[right - 1];
                if(target > max){
                    if(Math.abs(result - target) > Math.abs(max - target))
                        result = max;
                    break;
                }
                int sum = nums[i] + nums[left] + nums[right];
                // 判断三数之和是否等于target
                if(sum == target)
                    return sum;
                if(Math.abs(sum - target) < Math.abs(result - target))
                    result = sum;
                if(sum > target){
                    right--;
                    while(left != right && nums[right] == nums[right+1])
                        right--;
                }
                else{
                    left++;
                    while(left != right && nums[left] == nums[left-1])
                        left++;
                }
            }
            while(i<nums.length-2 && nums[i] == nums[i+1])
                i++;
        }
        return result;
    }
}

```

```
}  
}
```

17. 电话号码的字母组合

题目描述：

给定一个仅包含数字 `2-9` 的字符串，返回所有它能表示的字母组合。答案可以按 **任意顺序** 返回。

给出数字到字母的映射如下（与电话按键相同）。注意 `1` 不对应任何字母。



示例 1：

输入: `digits = "23"`

输出: `["ad","ae","af","bd","be","bf","cd","ce","cf"]`

示例 2：

输入: `digits = ""`

输出: `[]`

示例 3：

输入: `digits = "2"`

输出: `["a","b","c"]`

解题代码：

时间15.05%，内存7.79%

```

class Solution {
    public List<String> letterCombinations(String digits) {
        List<String> ans = new ArrayList<>();
        if(digits.length() == 0){
            return ans;
        }
        String[] map = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
        ans.add("");
        for(int i = 0; i < digits.length(); i++){
            String tmp = map[digits.charAt(i) - '0'];
            List<String> tmp_ans = new ArrayList<>();
            for(int j = 0; j < tmp.length(); j++){
                for(String s:ans){
                    tmp_ans.add(s + tmp.charAt(j));
                }
            }
            ans = tmp_ans;
        }
        return ans;
    }
}

```

大佬解法：

更快的时间

```

class Solution {
    String[] MAPPING=new String[]{"","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};
    ArrayList<String> ans=new ArrayList<String>();
    char[] words,path;
    public List<String> letterCombinations(String digits) {
        int len=digits.length();
        if(len==0)
            return Arrays.asList(new String[0]);
        words=digits.toCharArray();
        path=new char[len];
        dfs(0);
        return ans;
    }
    public void dfs(int i){
        if(i==path.length){
            ans.add(new String(path));
            return;
        }
        else{
            for(char c:MAPPING[words[i]-'0'].toCharArray()){
                path[i]=c;
                dfs(i+1);
            }
        }
    }
}

```

18. 四数之和

题目描述：

给你一个由 `n` 个整数组成的数组 `nums`，和一个目标值 `target`。请你找出并返回满足下述全部条件且不重复的四元组 `[nums[a], nums[b], nums[c], nums[d]]`（若两个四元组元素一一对应，则认为两个四元组重复）：

- `0 <= a, b, c, d < n`
- `a`、`b`、`c` 和 `d` 互不相同
- `nums[a] + nums[b] + nums[c] + nums[d] == target`

你可以按 任意顺序 返回答案。

示例 1：

输入：`nums = [1,0,-1,0,-2,2]`，`target = 0`
输出：`[[-2,-1,1,2], [-2,0,0,2], [-1,0,0,1]]`

示例 2：

输入：`nums = [2,2,2,2,2]`，`target = 8`
输出：`[[2,2,2,2]]`

解题代码：

超出内存限制了

```

class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        int n = nums.length;
        Arrays.sort(nums);
        List<List<Integer>> ans = new ArrayList<List<Integer>>();
        // 枚举 a
        for (int first = 0; first < n; ++first) {
            // 需要和上一次枚举的数不相同
            if (first > 0 && nums[first] == nums[first - 1]) {
                continue;
            }
            for (int second = first + 1; second < n; ++second) {
                // 需要和上一次枚举的数不相同
                if (second > first + 1 && nums[second] == nums[second - 1]) {
                    continue;
                }
                // d 对应的指针初始指向数组的最右端
                int forth = n - 1;
                int target_tmp = target - (nums[first] + nums[second]);
                // 枚举 c
                for (int third = second + 1; third < n; ++third) {
                    // 需要和上一次枚举的数不相同
                    if (third > second + 1 && nums[third] == nums[third - 1]) {
                        continue;
                    }
                    // 需要保证 c 的指针在 d 的指针的左侧
                    while (third < forth && nums[third] + nums[forth] > target_tmp) {
                        --forth;
                    }
                    // 如果指针重合，随着 c 后续的增加
                    // 就不会有满足 a+b+c+d=0 并且 c<d 的 d 了，可以退出循环
                    if (third == forth) {
                        break;
                    }
                    if (nums[third] + nums[forth] == target_tmp) {
                        List<Integer> list = new ArrayList<Integer>();
                        list.add(nums[first]);
                        list.add(nums[second]);
                        list.add(nums[third]);
                        list.add(nums[forth]);
                        ans.add(list);
                    }
                }
            }
        }
    }
}

```



```

    }
}
return ans;
}
}

```

大佬解法:

```

class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        Arrays.sort(nums);
        List<List<Integer>> ans = new ArrayList<>();
        int n = nums.length;
        for (int a = 0; a < n - 3; a++) { // 枚举第一个数
            long x = nums[a]; // 使用 long 避免溢出
            if (a > 0 && x == nums[a - 1]) continue; // 跳过重复数字
            if (x + nums[a + 1] + nums[a + 2] + nums[a + 3] > target) break; // 优化一
            if (x + nums[n - 3] + nums[n - 2] + nums[n - 1] < target) continue; // 优化二
            for (int b = a + 1; b < n - 2; b++) { // 枚举第二个数
                long y = nums[b];
                if (b > a + 1 && y == nums[b - 1]) continue; // 跳过重复数字
                if (x + y + nums[b + 1] + nums[b + 2] > target) break; // 优化一
                if (x + y + nums[n - 2] + nums[n - 1] < target) continue; // 优化二
                int c = b + 1, d = n - 1;
                while (c < d) { // 双指针枚举第三个数和第四个数
                    long s = x + y + nums[c] + nums[d]; // 四数之和
                    if (s > target) d--;
                    else if (s < target) c++;
                    else { // s == target
                        ans.add(List.of((int) x, (int) y, nums[c], nums[d]));
                        for (c++; c < d && nums[c] == nums[c - 1]; c++) ; // 跳过重复数字
                        for (d--; d > c && nums[d] == nums[d + 1]; d--) ; // 跳过重复数字
                    }
                }
            }
        }
        return ans;
    }
}

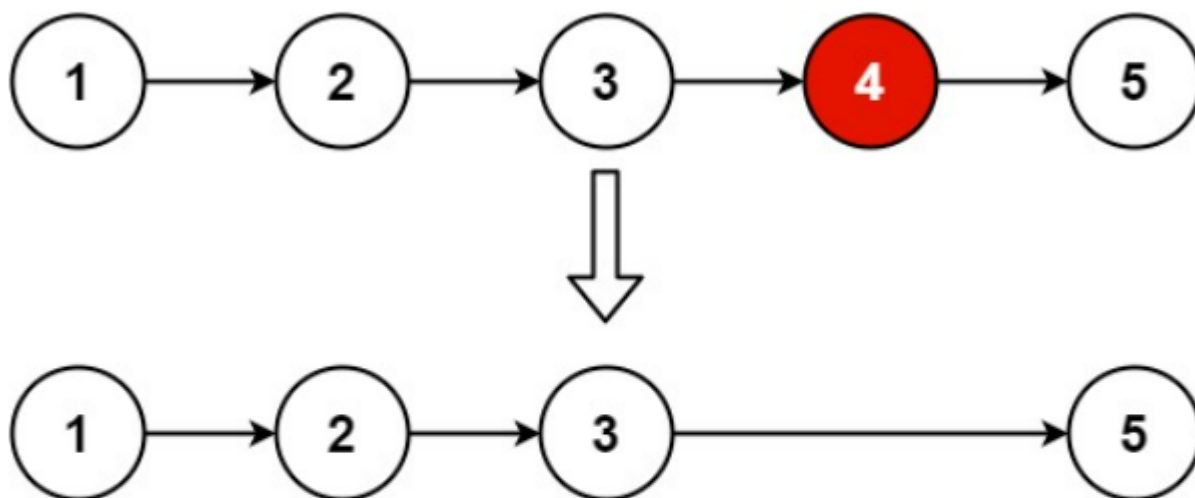
```

19. 删除链表的倒数第N个节点

题目描述：

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

示例 1：



输入: `head = [1,2,3,4,5]`, `n = 2`

输出: `[1,2,3,5]`

示例 2：

输入: `head = [1]`, `n = 1`

输出: `[]`

示例 3：

输入: `head = [1,2]`, `n = 1`

输出: `[1]`

解题代码：

时间100%，内存84.61%

```

class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        //添加表头
        ListNode res = new ListNode(-1);
        res.next = head;
        //当前节点
        ListNode cur = head;
        //前序节点
        ListNode pre = res;
        ListNode fast = head;
        //快指针先行n步
        while(n-- > 0)
            fast = fast.next;
        //快慢指针同步，快指针到达末尾，慢指针就到了倒数第n个位置
        while(fast != null){
            fast = fast.next;
            pre = cur;
            cur = cur.next;
        }
        //删除该位置的节点
        pre.next = cur.next;
        //返回去掉头
        return res.next;
    }
}

```

20. 有效的括号

题目描述：

给定一个只包括 '(' , ')' , '{' , '}' , '[' , ']' 的字符串 `s` , 判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。
3. 每个右括号都有一个对应的相同类型的左括号。

示例 1:

输入: `s = "()"`

输出: `true`

示例 2:

输入: `s = "()[]{}"`

输出: `true`

示例 3:

输入: `s = "]"`

输出: `false`

解题代码:

时间51.53%，内存54.52%

```

class Solution {
    public boolean isValid(String s) {
        if(s.equals("")){
            return true;
        }
        Stack<Character> st = new Stack<Character>();
        for(char c:s.toCharArray()){
            if(c=='(' || c=='{' || c=='['){
                st.push(c);
                continue;
            }
            if(!st.empty()){
                if(st.peek() == '(' && c == ')'){
                    st.pop();
                    continue;
                }
                else if(st.peek() == '[' && c == ']'){
                    st.pop();
                    continue;
                }
                else if(st.peek() == '{' && c == '}'){
                    st.pop();
                    continue;
                }
            }
            st.push(c);
            break;
        }
        if(st.empty())
            return true;
        else
            return false;
    }
}

```

大佬解法：

更快的时间

```

class Solution {
    public boolean isValid(String s) {
        char []c= s.toCharArray();
        char []d =new char[c.length];
        int j=0;
        for(int i=0;i<c.length;i++){
            switch(c[i]){
                case '(':;
                case '[':;
                case '{':d[j]=c[i];j++;break;
                case '}':
                    if (j==0)
                        return false;
                    if(d[j-1]!='{')
                        return false;
                    if(d[j-1]=='{')
                        j--;
                    break;
                case ']':
                    if (j==0)
                        return false;
                    if(d[j-1]!='[')
                        return false;
                    if(d[j-1]=='[')
                        j--;
                    break;
                case ')':
                    if (j==0)
                        return false;
                    // System.out.println(d[j]);
                    if(d[j-1]!='(')
                        return false;
                    if(d[j-1]=='(')
                        j--;
                    break;
            }
        }
        if(j==0)
            return true;
        else
            return false;
    }
}

```

}

}