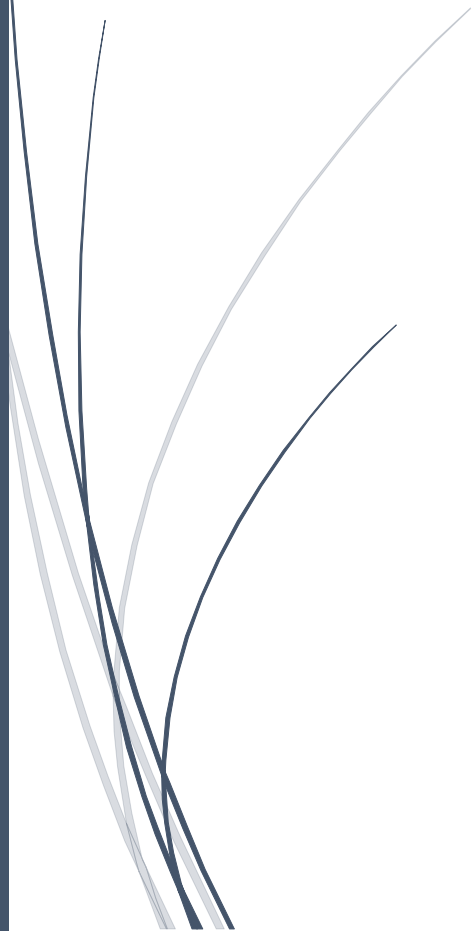
2020-4-19

数据结构与算法

腾讯精选练习 50 题 V1.0



yanpengma@163.com

华北电力大学（保定） 马燕鹏

目录

序言 4

01 两数相加 9

02 寻找两个有序数组的中位数 13

03 最长回文子串 17

04 整数反转 23

05 字符串转换整数（atoi) 26

06 回文数 30

07 盛最多水的容器 32

08 最长公共前缀 35

09 三数之和 38

10 最接近的三数之和 42

11 有效的括号 45

12 合并两个有序链表 49

13 合并 K 个排序链表 52

14 删除排序数组中的重复项 57

15 搜索旋转排序数组 60

16 字符串相乘 63

17 全排列 66

18 最大子序和 69

19 螺旋矩阵 72

20 螺旋矩阵 II 76

21 旋转链表 79

22 不同路径 82

23 爬楼梯 86

24 子集 89

25 合并两个有序数组 94

26 格雷编码 96

27 二叉树的最大深度 99

28 买卖股票的最佳时机 103

29 买卖股票的最佳时机 II 106

30 二叉树中的最大路径和 109

31 只出现一次的数字 112

32 环形链表 115

33 环形链表 II 120

34 LRU 缓存机制 124

35 排序链表 129

36 最小栈 132

37 相交链表 136

38 求众数 140

39 反转链表 142

40 数组中的第 K 个最大元素 144

41 存在重复元素 146

42 二叉搜索树中第 K 小的元素 148

43 2 的幂.....151

44 二叉搜索树的最近公共祖先.....153

45 二叉树的最近公共祖先.....156

46 删除链表中的节点.....159

47 除自身以外数组的乘积.....162

48 Nim 游戏.....164

49 反转字符串.....166

50 反转字符串中的单词 III.....168

序言

昨天在知识星球中立了一个 Flag，第一步采取的行动就是把以前刷的“**腾讯精选练习 50 题**”重新梳理一下，就有了今天这本 170 多页的小册子。

这本小册子即可以作为学习数据结构与算法课程的参考资料，也可以作为备考计算机类研究生的备考资料。希望对学习算法的同学们有所帮助。

老马的程序人生
昨天 20:33

很久没有写技术blog了，先给自己树第一个flag吧！自己已经刷过85道leetcode算法题目了，温故而知新先把自己刷过的题目总结一下，写五篇关于“位运算/双指针/字典/..... 技术在求解算法题目中的应用”的技术图文呀。知识这种东西需要不断打磨内化，希望这次的梳理能够让自己更深刻的理解这些技术，以便应用于自己日后的项目中，写出更优秀的代码。求监督。哈哈。#flag#

这本小册子，主要是对我们第一次“数据结构与算法（Leetcode）刻意练习”中，在自己公众号所发打卡图文的重新梳理与总结，有部分代码进行了优化。

- Day01 两数相加
- Day02 寻找两个有序数组的中位数
- Day03 最长回文子串
- Day04 整数反转
- Day05 字符串转换整数 (atoi)
- Day06 回文数

- Day07 盛最多水的容器
- Day08 最长公共前缀
- Day09 三数之和
- Day10 最接近的三数之和
- Day11 有效的括号
- Day12 合并两个有序链表
- Day13 合并 K 个排序链表
- Day14 删除排序数组中的重复项
- Day15 搜索旋转排序数组
- Day16 字符串相乘
- Day17 全排列
- Day18 最大子序和
- Day19 螺旋矩阵
- Day20 螺旋矩阵 II
- Day21 旋转链表
- Day22 不同路径
- Day23 爬楼梯
- Day24 子集
- Day25 合并两个有序数组
- Day26 格雷编码

- Day27 二叉树的最大深度
- Day28 买卖股票的最佳时机
- Day29 买卖股票的最佳时机 II
- Day30 二叉树中的最大路径和
- Day31 只出现一次的数字
- Day32 环形链表
- Day33 环形链表 II
- Day34 LRU 缓存机制
- Day35 排序链表
- Day36 最小栈
- Day37 相交链表
- Day38 求众数
- Day39 反转链表
- Day40 数组中的第 K 个最大元素
- Day41 存在重复元素
- Day42 二叉搜索树中第 K 小的元素
- Day43 2 的幂
- Day44 二叉搜索树的最近公共祖先
- Day45 二叉树的最近公共祖先
- Day46 删除链表中的节点

- Day47 除自身以外数组的乘积
- Day48 Nim 游戏
- Day49 反转字符串
- Day50 反转字符串中的单词 III

我是 终身学习者“**老马**”，一个长期践行“结伴式学习”理念的 **中年大叔。**

我崇尚分享，渴望成长，于 2010 年创立了“**LSGO 软件技术团队**”，并加入了国内著名的开源组织“**Datawhale**”，也是“**Dre@mtech**”、“**智能机器人研究中心**”和“**大数据与哲学社会科学实验室**”的一员。

愿我们一起学习，一起进步，相互陪伴，共同成长。

欢迎关注公众号，请扫描二维码：

01 两数相加

- 题号：2
- 难度：中等
- https://leetcode-cn.com/problems/add-two-numbers/

给出两个 **非空** 的链表用来表示两个非负的整数。其中，它们各自的位数是按照 **逆序** 的方式存储的，并且它们的每个节点只能存储 **一位** 数字。

如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

您可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1：

```
输入：(2 -> 4 -> 3) + (5 -> 6 -> 4)
输出：7 -> 0 -> 8
原因：342 + 465 = 807
```

示例 2：

```
输入：(3 -> 7) + (9 -> 2)
输出：2 -> 0 -> 1
原因：73 + 29 = 102
```

思路：模仿我们小学时代学的加法运算。个位相加超过十进一，十位相加有进位则加上进位，依次类推。

C# 实现

- 状态：通过
- 执行用时: 144 ms, 在所有 C# 提交中击败了 97.98% 的用户
- 内存消耗: 26.7 MB, 在所有 C# 提交中击败了 5.07% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public ListNode AddTwoNumbers(ListNode l1, ListNode l2)
    {
        ListNode result = new ListNode(-1);
        ListNode l3 = result;
        int flag = 0;
        while (l1 != null && l2 != null)
        {
            int a = l1.val;
            int b = l2.val;
            int c = a + b + flag;
            l3.next = new ListNode(c%10);

            flag = c >= 10 ? 1 : 0;
            l1 = l1.next;
            l2 = l2.next;
            l3 = l3.next;
        }

        while (l1 != null)
        {
            int a = l1.val + flag;

            l3.next = new ListNode(a%10);

            flag = a >= 10 ? 1 : 0;
            l1 = l1.next;
            l3 = l3.next;
```

```
        }

        while (l2 != null)
        {
            int b = l2.val + flag;

            l3.next = new ListNode(b%10);
            flag = b >= 10 ? 1 : 0;
            l2 = l2.next;
            l3 = l3.next;
        }

        if (flag == 1)
        {
            l3.next = new ListNode(flag);
        }
        return result.next;
    }
}
```

Python 实现

- 执行结果：通过
- 执行用时 :108 ms, 在所有 Python 提交中击败了 16.54% 的用户
- 内存消耗 :13.6 MB, 在所有 Python 提交中击败了 8.73 %的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        result = ListNode(-1)
        l3 = result
        flag = 0
        while l1 is not None and l2 is not None:
            a = l1.val
            b = l2.val
            c = a + b + flag
```

```
            l3.next = ListNode(c % 10)
            flag = 1 if c >= 10 else 0
            l1 = l1.next
            l2 = l2.next
            l3 = l3.next

        while l1 is not None:
            a = l1.val + flag
            l3.next = ListNode(a % 10)
            flag = 1 if a >= 10 else 0
            l1 = l1.next
            l3 = l3.next

        while l2 is not None:
            b = l2.val + flag
            l3.next = ListNode(b % 10)
            flag = 1 if b >= 10 else 0
            l2 = l2.next
            l3 = l3.next

        if flag == 1:
            l3.next = ListNode(flag)

        return result.next
```

02 寻找两个有序数组的中位数

- 题号：4
- 难度：困难
- https://leetcode-cn.com/problems/median-of-two-sorted-arrays/

给定两个大小为 m 和 n 的有序数组 nums1 和 nums2。

请你找出这两个有序数组的中位数，并且要求算法的时间复杂度为 O(log(m + n))。

你可以假设 nums1 和 nums2 不会同时为空。

示例 1:

```
nums1 = [1, 3]
nums2 = [2]

则中位数是 2.0
```

示例 2:

```
nums1 = [1, 2]
nums2 = [3, 4]

则中位数是 (2 + 3)/2 = 2.5
```

第一种：利用二分策略

中位数：用来将一个集合划分为两个长度相等的子集，其中一个子集中的元素总是大于另一个子集中的元素。

由于题目要求时间复杂度为 O(log(m + n))，所以不能从两个有序数组的首尾挨个遍历来找到中位数（复杂度 O(m + n)）；而是要通过二分策略，通过每次比较，能够直接按比例的刷掉一组数字，最后找到中位数（复杂度 O(log(m + n))）。

```
nums1: [a1,a2,a3,...am]
nums2: [b1,b2,b3,...bn]

[nums1[:i],nums2[:j] | nums1[i:], nums2[j:]]

nums1 取 i 个数的左半边
nums2 取 j = (m+n+1)/2 - i 的左半边
```

只要保证左右两边 **个数** 相同，中位数就在 | 这个边界旁边产生，从而可以利用二分法找到合适的 i。

- 状态：通过
- 2085 / 2085 个通过测试用例
- 执行用时: 160 ms, 在所有 C# 提交中击败了 99.18% 的用户
- 内存消耗: 26.8 MB, 在所有 C# 提交中击败了 5.05% 的用户

```
public class Solution {
    public double FindMedianSortedArrays(int[] nums1, int[] nums2) {
        int m = nums1.Length;
        int n = nums2.Length;
        if (m > n)
            return FindMedianSortedArrays(nums2, nums1);

        int k = (m + n + 1)/2;
        int left = 0;
        int right = m;
        while (left < right)
        {
            int i = (left + right)/2;
            int j = k - i;
            if (nums1[i] < nums2[j - 1])
                left = i + 1;
```

```
            else
                right = i;
        }
        int m1 = left;
        int m2 = k - left;
        int c1 = Math.Max(m1 == 0 ? int.MinValue : nums1[m1 - 1],
            m2 == 0 ? int.MinValue : nums2[m2 - 1]);

        if ((m + n)%2 == 1)
            return c1;

        int c2 = Math.Min(m1 == m ? int.MaxValue : nums1[m1],
            m2 == n ? int.MaxValue : nums2[m2]);
        return (c1 + c2)*0.5;
    }
}
```

第二种：通过合并为一个有序数组的方式

思路：首先把两个有序数组合并为一个有序数组，然后根据数组长度来确定中位数，如果数组长度为偶数，那么返回两个中位数的平均值，如果数组长度为奇数，那么返回中位数。

python 实现

- 执行结果：通过
- 执行用时 :128 ms, 在所有 Python3 提交中击败了 34.35%的用户
- 内存消耗 :12.8 MB, 在所有 Python3 提交中击败了 99.43%的用户

```
class Solution:
    def findMedianSortedArrays(self, nums1, nums2):
        m = len(nums1)
        n = len(nums2)
        nums1.extend(nums2)# 合并
        nums1.sort()# 排序
        if (m + n) & 1:# 如果是奇数 返回中位数
            return nums1[(m + n - 1) >> 1]
```

```
        else:# 返回两个中位数的平均值
            return (nums1[(m + n - 1) >> 1] + nums1[(m + n) >> 1]) / 2
```

C# 实现

- 状态：通过
- 2085 / 2085 个通过测试用例
- 执行用时: 188 ms, 在所有 C# 提交中击败了 72.14% 的用户
- 内存消耗: 26.9 MB, 在所有 C# 提交中击败了 5.05% 的用户

```
public class Solution {
    public double FindMedianSortedArrays(int[] nums1, int[] nums2) {
        int len1 = nums1.Length;
        int len2 = nums2.Length;
        int len = len1 + len2;
        int[] nums = new int[len];
        Array.Copy(nums1, nums, len1);
        Array.Copy(nums2, 0, nums, len1, len2);
        Array.Sort(nums);
        if (len%2 == 0)
        {
            return (nums[len/2] + nums[len/2 - 1])*0.5;
        }
        return nums[len/2];
    }
}
```

03 最长回文子串

- 题号：5
- 难度：中等
- https://leetcode-cn.com/problems/longest-palindromic-substring/

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1：

```
输入："babad"
输出："bab"
注意："aba" 也是一个有效答案。
```

示例 2：

```
输入："cbbd"
输出："bb"
```

示例 3：

```
输入："a"
输出："a"
```

回文是一个正读和反读都相同的字符串，例如，“aba”是回文，而“abc”不是。

第一种：暴力法，列举所有的子串，判断该子串是否为回文。

- 执行结果：超出时间限制

```
public class Solution
{
    public string LongestPalindrome(string s)
```

```
    {
        if (string.IsNullOrEmpty(s))
            return string.Empty;
        if (s.Length == 1)
            return s;

        int start = 0;
        int end = 0;
        int len = int.MinValue;
        for (int i = 0; i < s.Length; i++)
        {
            for (int j = i + 1; j < s.Length; j++)
            {
                string str = s.Substring(i, j - i + 1);
                if (isPalindrome(str) && str.Length > len)
                {
                    len = str.Length;
                    start = i;
                    end = j;
                }
            }
        }
        return s.Substring(start, end - start + 1);
    }

    public bool isPalindrome(string s)
    {
        for (int i = 0, len = s.Length / 2; i < len; i++)
        {
            if (s[i] != s[s.Length - 1 - i])
                return false;
        }
        return true;
    }
}
```

第二种：动态规划

动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。

与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。

我们可以用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。

具体的动态规划算法多种多样，但它们具有相同的填表格式。

使用记号 `s[l，r]` 表示原始字符串的一个子串，`l`、`r` 分别是区间的左右边界的索引值，使用左闭、右闭区间表示左右边界可以取到。

`dp[l，r]` 表示子串 `s[l，r]`（包括区间左右端点）是否构成回文串，是一个二维布尔型数组。

- 当子串只包含 1 个字符，它一定是回文子串；
- 当子串包含 2 个以上字符的时候：`s[l，r]` 是一个回文串，那么这个回文串两边各往里面收缩一个字符（如果可以的话）的子串 `s[l + 1，r - 1]` 也一定是回文串。

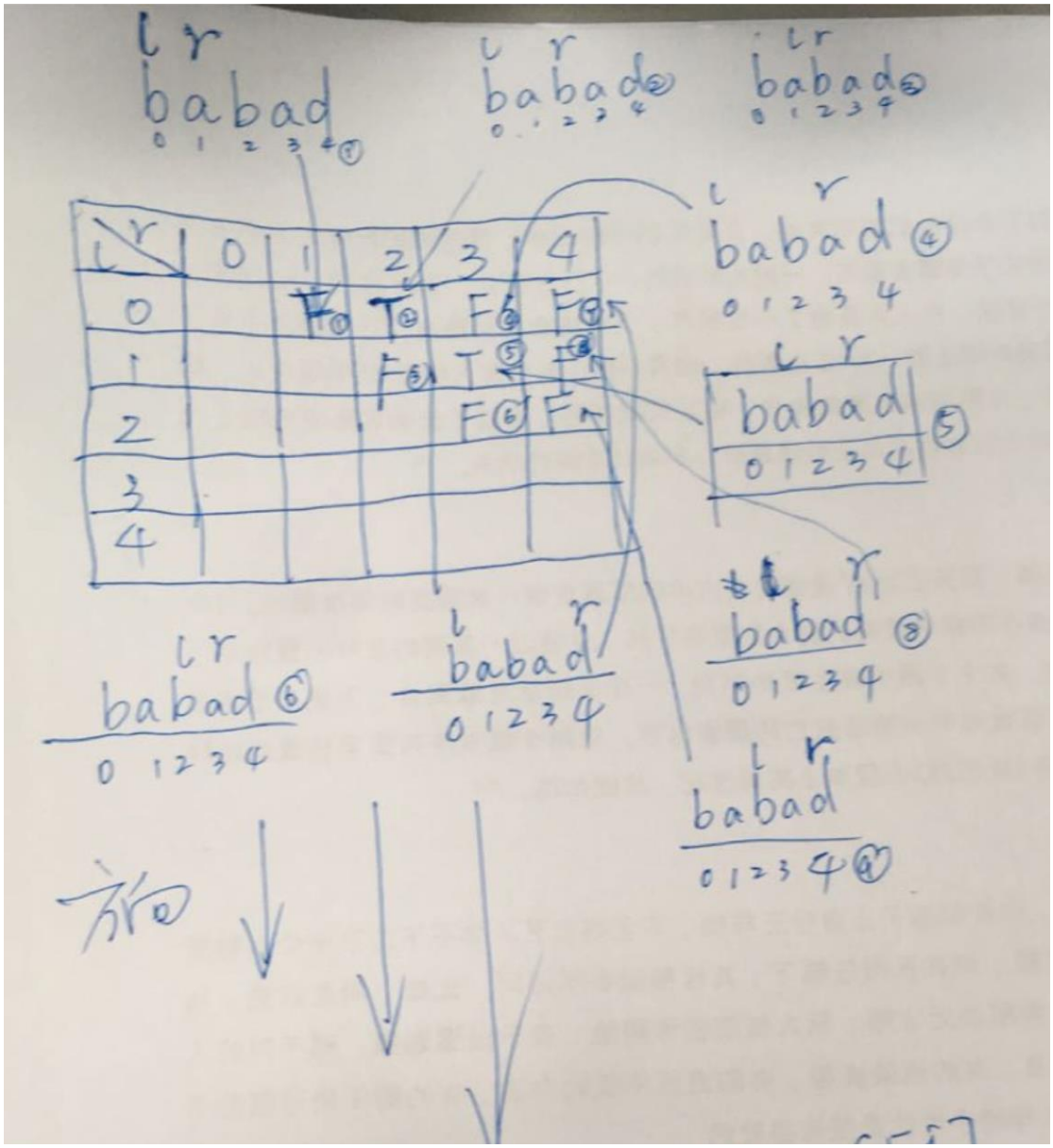
故，当 `s[l] == s[r]` 成立的时候，`dp[l，r]`的值由 `dp[l + 1，r - l]` 决定，这里还需要再多考虑一点点：“原字符串去掉左右边界”的子串的边界情况。

- 当原字符串的元素个数为 3 的时候，如果左右边界相等，那么去掉它们以后，只剩下 1 个字符，它一定是回文串，故原字符串也一定是回文串；

- 当原字符串的元素个数为 2 的时候，如果左右边界相等，那么去掉它们以后，只剩下 0 个字符，显然原字符串也一定是回文串。

综上，如果一个字符串的左右边界相等，判断为回文只需以下二者之一成立即可：

- 去掉左右边界以后的字符串不构成区间，即 `s[l + 1, r - 1]`包含元素少于 2 个，即：`r - l <= 2`。
- 去掉左右边界以后的字符串是回文串，即 `dp[l + 1, r - 1] == true`。



C# 实现

- 状态：通过
- 103 / 103 个通过测试用例
- 执行用时: 232 ms, 在所有 C# 提交中击败了 46.79% 的用户
- 内存消耗: 40.9 MB, 在所有 C# 提交中击败了 5.43% 的用户

```
public class Solution {
    public string LongestPalindrome(string s)
    {
        if (string.IsNullOrEmpty(s))
            return string.Empty;
        int len = s.Length;
        if (len == 1)
            return s;
        int longestPalindromelen = 1;
        string longestPalindromeStr = s.Substring(0, 1);
        bool[,] dp = new bool[len, len];

        for (int r = 1; r < len; r++)
        {
            for (int l = 0; l < r; l++)
            {
                if (s[r] == s[l] && (r - l <= 2 || dp[l + 1, r - 1] == true))
                {
                    dp[l, r] = true;
                    if (longestPalindromelen < r - l + 1)
                    {
                        longestPalindromelen = r - l + 1;
                        longestPalindromeStr = s.Substring(l, r - l + 1);
                    }
                }
            }
        }
        return longestPalindromeStr;
    }
}
```

Python 实现

- 执行结果：通过
- 执行用时：3392 ms，在所有 Python3 提交中击败了 43.87% 的用户
- 内存消耗：21.2 MB，在所有 Python3 提交中击败了 18.81% 的用户

```python
class Solution:
    def longestPalindrome(self, s: str) -> str:
        count = len(s)
        if count == 0 or count == 1:
            return s
        longestPalindromelen = 1
        longestPalindromeStr = s[0:1]
        dp = [[False] * count for i in range(count)]
        for r in range(1, count):
            for l in range(0, r):
                if s[r] == s[l] and (r - l <= 2 or dp[l + 1][r - 1] == True):
                    dp[l][r] = True
                    if longestPalindromelen < r - l + 1:
                        longestPalindromelen = r - l + 1
                        longestPalindromeStr = s[l:l + longestPalindromelen]
        return longestPalindromeStr
```

04 整数反转

- 题号：7
- 难度：简单
- https://leetcode-cn.com/problems/reverse-integer/

给出一个 32 位的有符号整数，你需要将这个整数中每位上的数字进行反转。

示例 1:

```
输入：123
输出：321
```

示例 2:

```
输入：-123
输出：-321
```

示例 3:

```
输入：120
输出：21
```

示例 4:

```
输入：1534236469
输出：0
```

示例 5:

```
输入：-2147483648
输出：0
```

注意:

假设我们的环境只能存储得下 32 位的有符号整数，则其数值范围为`[-2^31，2^31 - 1]`。

请根据这个假设，如果反转后整数溢出那么就返回 0。

第一种：通过队列的方式。

把负数转换为正数，通过“队列”统一处理。

- 状态：通过
- 1032 / 1032 个通过测试用例
- 执行用时: 56 ms, 在所有 C# 提交中击败了 93.28% 的用户
- 内存消耗: 13.9 MB, 在所有 C# 提交中击败了 13.98% 的用户

```
public class Solution {
    public int Reverse(int x) {
        if (x == int.MinValue)
            return 0;

        long result = 0;
        int negative = x < 0 ? -1 : 1;
        x = negative * x;

        Queue<int> q = new Queue<int>();
        while (x != 0)
        {
            q.Enqueue(x % 10);
            x = x/10;
        }

        while (q.Count != 0)
        {
            result += q.Dequeue()*(long) Math.Pow(10, q.Count);
            if (negative == 1 && result > int.MaxValue)
            {
                result = 0;
                break;
            }
            if (negative == -1 && result*-1 < int.MinValue)
            {
                result = 0;
```

```
                break;
            }
        }
        return (int)result*negative;
    }
}
```

05 字符串转换整数（atoi）

- 题号：8
- 难度：中等
- https://leetcode-cn.com/problems/string-to-integer-atoi/

请你来实现一个 `atoi` 函数，使其能将字符串转换成整数。

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。

当我们寻找到的第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字组合起来，作为该整数的正负号；假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。

该字符串除了有效的整数部分之后也可能会存在多余的字符，这些字符可以被忽略，它们对于函数不应该造成影响。

注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0。

说明：

假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 `[-2^31，2^31-1]`。如果数值超过这个范围，请返回 `INT_MAX (2^31- 1)` 或 `INT_MIN (-2^31)` 。

示例 1:

```
输入：“42”
输出：42
```

示例 2:

```
输入：“   -42”
输出：-42
解释：第一个非空白字符为 '-'，它是一个负号。
我们尽可能将负号与后面所有连续出现的数字组合起来，最后得到 -42 。
```

示例 3:

```
输入：“4193 with words”
输出：4193
解释：转换截止于数字 '3' ，因为它的下一个字符不为数字。
```

示例 4:

```
输入：“words and 987”
输出：0
解释：第一个非空字符是 'w'，但它不是数字或正、负号。
因此无法执行有效的转换。
```

示例 5:

```
输入：“-91283472332”
输出：-2147483648
解释：数字 “-91283472332” 超过 32 位有符号整数范围。
因此返回 INT_MIN (−231) 。
```

示例 6:

```
输入：“  0000000000012345678”
输出：12345678
```

示例 7:

```
输入：“20000000000000000000”
输出：2147483647
```

第一种：通过队列的方式

- 状态：通过
- 1079 / 1079 个通过测试用例
- 执行用时: 104 ms, 在所有 C# 提交中击败了 98.32% 的用户
- 内存消耗: 24.3 MB, 在所有 C# 提交中击败了 24.45% 的用户

```
public class Solution {
    public int MyAtoi(string str) {
        str = str.Trim();
        if (string.IsNullOrEmpty(str))
            return 0;

        if (str[0] != '-' && str[0] != '+')
        {
            if (str[0] < '0' || str[0] > '9')
                return 0;
        }
        int negative = 1;
        long result = 0;
        Queue<int> q = new Queue<int>();
        for (int i = 0; i < str.Length; i++)
        {
            if (str[i] == '-' && i == 0)
            {
                negative = -1;
                continue;
            }
            if (str[i] == '+' && i == 0)
            {
                continue;
            }
            if (str[i] < '0' || str[i] > '9')
            {
                break;
            }
            q.Enqueue(str[i] - '0');
        }
```

```
        while (q.Count != 0)
        {
            int i = q.Dequeue();

            //去掉队列前端的零
            if (i == 0 && result == 0)
                continue;

            // 返回超过位数的数字
            if (negative == 1 && q.Count > 10)
            {
                return int.MaxValue;
            }
            if (negative == -1 && q.Count > 10)
            {
                return int.MinValue;
            }

            result += i * (long)Math.Pow(10, q.Count);
            if (negative == 1 && result > int.MaxValue)
            {
                return int.MaxValue;
            }
            if (negative == -1 && result * -1 < int.MinValue)
            {
                return int.MinValue;
            }
        }
        return (int)result * negative;
    }
}
```

06 回文数

- 题号：9
- 难度：简单
- https://leetcode-cn.com/problems/palindrome-number/

判断一个整数是否是回文数。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。

示例 1:

```
输入：121
输出：true
```

示例 2:

```
输入：-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：10
输出：false
解释：从右向左读，为 01 。因此它不是一个回文数。
```

进阶:

你能不将整数转为字符串来解决这个问题吗？

- 状态：通过
- 11509 / 11509 个通过测试用例

- 执行用时: 76 ms, 在所有 C# 提交中击败了 98.90% 的用户
- 内存消耗: 14.9 MB, 在所有 C# 提交中击败了 85.12% 的用户

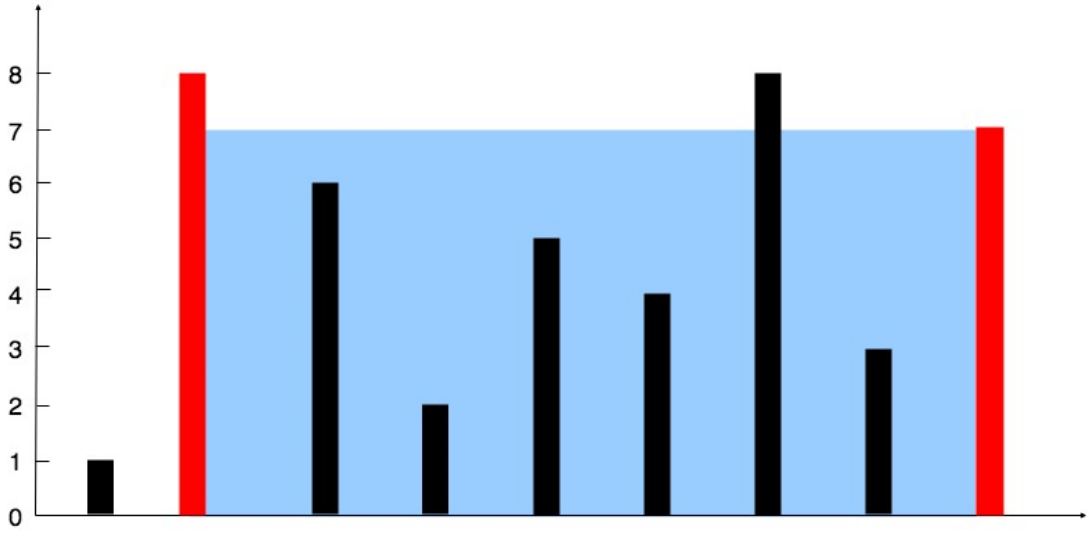
```
public class Solution {
    public bool IsPalindrome(int x) {
        if (x < 0)
            return false;

        int bit = 1;
        while (x / bit >= 10)
        {
            bit = bit * 10;
        }

        while (x > 0)
        {
            int left = x % 10;
            int right = x / bit;
            if (left != right)
            {
                return false;
            }
            x = (x % bit) / 10;
            bit = bit / 100;
        }
        return true;
}
```

07 盛最多水的容器

- 题号：11
- 难度：中等
- https://leetcode-cn.com/problems/container-with-most-water/

给定 n 个非负整数 a1，a2，...，an，每个数代表坐标中的一个点 (i, ai) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, ai) 和 (i, 0)。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。

说明：你不能倾斜容器，且 n 的值至少为 2。



图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

示例:

```
输入：[1,8,6,2,5,4,8,3,7]
输出：49
```

第一种：利用暴力算法

- 状态：超出时间限制
- 49 / 50 个通过测试用例

```
public class Solution {
    public int MaxArea(int[] height) {
        int max = int.MinValue;
        for (int i = 0; i < height.Length - 1; i++)
        {
            for (int j = 1; j < height.Length; j++)
            {
                int temp = (j - i)*Math.Min(height[i], height[j]);
                if (temp > max)
                {
                    max = temp;
                }
            }
        }
        return max;
    }
}
```

第二种：利用双索引的方法

以 0-7 走到 1-7 这一步为例，解释为什么放弃 0-6 这一分支：

```
用 h(i)表示第 i 条线段的高度，S(ij)表示第 i 条线段和第 j 条线段圈起来的面积。

已知 h(0) < h(7)，从而 S(07) = h(0) * 7。

有 S(06) = min(h(0), h(6)) * 6。

当 h(0) <= h(6)，有 S(06) = h(0) * 6;
当 h(0) > h(6)，有 S(06) = h(6) * 6，S(06) < h(0) * 6。
```

由此可知，S(06)必然小于S(07)。

把每一棵子树按照同样的方法分析，很容易可以知道，双索引法走的路径包含了最大面积。

- 状态：通过
- 50 / 50 个通过测试用例
- 执行用时: 144 ms, 在所有 C# 提交中击败了 99.64% 的用户
- 内存消耗: 26.6 MB, 在所有 C# 提交中击败了 5.45% 的用户

```
public class Solution
{
    public int MaxArea(int[] height)
    {
        int i = 0, j = height.Length - 1;
        int max = int.MinValue;
        while (i < j)
        {
            int temp = (j - i) * Math.Min(height[i], height[j]);
            if (temp > max)
            {
                max = temp;
            }
            if (height[i] < height[j])
            {
                i++;
            }
            else
            {
                j--;
            }
        }
        return max;
    }
}
```

08 最长公共前缀

- 题号：14
- 难度：简单
- https://leetcode-cn.com/problems/longest-common-prefix/

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：["flower","flow","flight"]
输出："fl"
```

示例 2:

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

说明:

所有输入只包含小写字母 a-z。

C# 实现

- 状态：通过
- 118 / 118 个通过测试用例
- 执行用时: 144 ms, 在所有 C# 提交中击败了 94.92% 的用户

- 内存消耗: 23.4 MB, 在所有 C# 提交中击败了 11.69% 的用户

```
public class Solution {
    public string LongestCommonPrefix(string[] strs)
    {
        if (strs.Length == 0)
            return string.Empty;

        string result = strs[0];
        for (int i = 1; i < strs.Length; i++)
        {
            result = Prefix(result, strs[i]);
            if (string.IsNullOrEmpty(result))
                break;
        }
        return result;
    }

    public string Prefix(string str1, string str2)
    {
        int len1 = str1.Length;
        int len2 = str2.Length;
        int len = Math.Min(len1, len2);
        int i = 0;
        for (; i < len; i++)
        {
            if (str1[i] != str2[i])
                break;
        }
        return i == 0 ? string.Empty : str1.Substring(0, i);
    }
}
```

Python 实现

- 执行结果：通过
- 执行用时：44 ms, 在所有 Python3 提交中击败了 35.93% 的用户
- 内存消耗：13.6 MB, 在所有 Python3 提交中击败了 5.14% 的用户

```
class Solution:
```

```
    def longestCommonPrefix(self, strs: List[str]) -> str:
        if len(strs) == 0:
            return ""
        result = strs[0]
        for i in range(len(strs)):
            result = self.Prefix(result, strs[i])
            if result == "":
                break
        return result

    def Prefix(self, str1: str, str2: str) -> str:
        len1, len2 = len(str1), len(str2)
        i = 0
        while i < min(len1, len2):
            if str1[i] != str2[i]:
                break
            i += 1
        return "" if i == 0 else str1[0:i]
```

09 三数之和

- 题号：15
- 难度：中等
- https://leetcode-cn.com/problems/3sum/

给定一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c，使得 a + b + c = 0？找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

```
给定数组 nums = [-1, 0, 1, 2, -1, -4],

满足要求的三元组集合为：
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

第一种：三索引法

思路：为了避免三次循环，提升执行效率。首先，对 nums 进行排序。然后，固定 3 个索引 i,l(left),r(right)，i 进行最外层循环，l 指向 nums[i]之后数组的最小值，r 指向 nums[i]之后数组的最大值。模仿快速排序的思路，如果 nums[i] > 0 就不需要继续计算了，否则计算 nums[i] + nums[l] + nums[r]是否等于零并进行相应的处理。如果大于零，向 l 方向移动 r 指针，如果小于零，向 r 方向移动 l 索引，如果等于零，则加入到存储最

后结果的 result 链表中。当然，题目中要求这个三元组不可重复，所以在进行的过程中加入去重就好。

C# 实现

- 执行结果：通过
- 执行用时：348 ms, 在所有 C# 提交中击败了 99.54% 的用户
- 内存消耗：35.8 MB, 在所有 C# 提交中击败了 6.63% 的用户

```
public class Solution
{
    public IList<IList<int>> ThreeSum(int[] nums)
    {
        IList<IList<int>> result = new List<IList<int>>();

        nums = nums.OrderBy(a => a).ToArray();
        int len = nums.Length;

        for (int i = 0; i < len - 2; i++)
        {
            if (nums[i] > 0)
                break; // 如果最小的数字大于0，后面的操作已经没有意义

            if (i > 0 && nums[i - 1] == nums[i])
                continue; // 跳过三元组中第一个元素的重复数据

            int l = i + 1;
            int r = len - 1;

            while (l < r)
            {
                int sum = nums[i] + nums[l] + nums[r];
                if (sum < 0)
                {
                    l++;
                }
                else if (sum > 0)
                {
                    r--;
```

```
                }
                else
                {
                    result.Add(new List<int>() {nums[i], nums[l], nums[r]});
                    // 跳过三元组中第二个元素的重复数据
                    while (l < r && nums[l] == nums[l + 1])
                    {
                        l++;
                    }
                    // 跳过三元组中第三个元素的重复数据
                    while (l < r && nums[r - 1] == nums[r])
                    {
                        r--;
                    }
                    l++;
                    r--;
                }
            }
        }
        return result;
    }
}
```

Python 实现

- 执行结果：通过
- 执行用时：660 ms, 在所有 Python3 提交中击败了 95.64% 的用户
- 内存消耗：16.1 MB, 在所有 Python3 提交中击败了 75.29% 的用户

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums = sorted(nums)

        result = []

        for i in range(0, len(nums) - 2):
            # 如果最小的数字大于0，后面的操作已经没有意义
            if nums[i] > 0:
                break
            # 跳过三元组中第一个元素的重复数据
            if i > 0 and nums[i-1] == nums[i]:
```

```
            continue

        # 限制nums[i]是三元组中最小的元素
        l = i + 1
        r = len(nums) - 1
        while l < r:
            sum = nums[i] + nums[l] + nums[r]
            if sum < 0:
                l += 1
            elif sum > 0:
                r -= 1
            else:
                result.append([nums[i], nums[l], nums[r]])
                # 跳过三元组中第二个元素的重复数据
                while l < r and nums[l] == nums[l+1]:
                    l += 1
                # 跳过三元组中第三个元素的重复数据
                while l < r and nums[r] == nums[r-1]:
                    r -= 1
                l += 1
                r -= 1
    return result
```

10 最接近的三数之和

- 题号：16
- 难度：中等
- https://leetcode-cn.com/problems/3sum-closest/

给定一个包括 n 个整数的数组 nums 和一个目标值 target。找出 nums 中的三个整数，使得它们的和与 target 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

示例：

```
例如，给定数组 nums = [-1, 2, 1, -4]，和 target = 1.

与 target 最接近的三个数的和为 2. (-1 + 2 + 1 = 2).
```

第一种：利用暴力算法

- 状态：通过
- 125 / 125 个通过测试用例
- 执行用时: 680 ms, 在所有 C# 提交中击败了 12.07% 的用户
- 内存消耗: 23.7 MB, 在所有 C# 提交中击败了 7.41% 的用户

```
public class Solution
{
    public int ThreeSumClosest(int[] nums, int target)
    {
        double error = int.MaxValue;
        int sum = 0;
        for (int i = 0; i < nums.Length - 2; i++)
```

```
            for (int j = i + 1; j < nums.Length - 1; j++)
                for (int k = j + 1; k < nums.Length; k++)
                {
                    if (Math.Abs(nums[i] + nums[j] + nums[k] - target) < error)
                    {
                        sum = nums[i] + nums[j] + nums[k];
                        error = Math.Abs(sum - target);
                    }
                }
        return sum;
    }
}
```

第二种：利用双索引算法

C# 实现

- 状态：通过
- 125 / 125 个通过测试用例
- 执行用时: 132 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗: 24 MB, 在所有 C# 提交中击败了 5.55% 的用户

```
public class Solution
{
    public int ThreeSumClosest(int[] nums, int target)
    {
        nums = nums.OrderBy(a => a).ToArray();
        int result = nums[0] + nums[1] + nums[2];
        for (int i = 0; i < nums.Length - 2; i++)
        {
            int start = i + 1, end = nums.Length - 1;
            while (start < end)
            {
                int sum = nums[start] + nums[end] + nums[i];
                if (Math.Abs(target - sum) < Math.Abs(target - result))
                    result = sum;
                if (sum > target)
                    end--;
                else if (sum < target)
```

```
                start++;
            else
                return result;
        }
    }
    return result;
  }
}
```

Pyhton 实现

- 执行结果：通过
- 执行用时：124 ms, 在所有 Python3 提交中击败了 72.19% 的用户
- 内存消耗：13.2 MB, 在所有 Python3 提交中击败了 22.06% 的用户

```
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        nums = sorted(nums)
        result = nums[0] + nums[1] + nums[2]
        for i in range(0, len(nums) - 2):
            start = i + 1
            end = len(nums) - 1
            while start < end:
                sum = nums[start] + nums[end] + nums[i]
                if abs(target - sum) < abs(target - result):
                    result = sum
                if sum > target:
                    end -= 1
                elif sum < target:
                    start += 1
                else:
                    return result
        return result
```

11 有效的括号

- 题号：20
- 难度：简单
- https://leetcode-cn.com/problems/valid-parentheses/

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

有效字符串需满足：

- 左括号必须用相同类型的右括号闭合。
- 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1:

```
输入: "()"
输出: true
```

示例 2:

```
输入: "()[]{}"
输出: true
```

示例 3:

```
输入: "(]"
输出: false
```

示例 4:

```
输入: "([)]"
输出: false
```

示例 5:

```
输入："{[]}"
输出：true
```

示例 6:

```
输入：""
输出：true
```

示例 7:

```
输入："(("
输出：false
```

第一种：利用栈

思路：首先判断该字符串长度的奇偶性，如果是奇数，则返回 false。否则，利用栈先进后出的特点，遇到“{”，“[”，“(”进行入栈操作，遇到“}”，“]”，“)”就与栈顶元素进行比较，如果是对应括号则出栈，否则返回 false。

C# 语言

- 执行结果：通过
- 执行用时：88 ms, 在所有 C# 提交中击败了 70.31% 的用户
- 内存消耗：22 MB, 在所有 C# 提交中击败了 17.91% 的用户

```
public class Solution {
    public bool IsValid(string s)
    {
        if (string.IsNullOrEmpty(s))
            return true;

        int count = s.Length;
```

```
        if(count%2 == 1)
            return false;

        Stack<char> stack = new Stack<char>();
        for (int i = 0; i < count; i++)
        {
            char c = s[i];
            if (stack.Count == 0)
            {
                stack.Push(c);
            }
            else if(c == '[' || c == '{' || c == '(')
            {
                stack.Push(c);
            }
            else if (stack.Peek() == GetPair(c))
            {
                stack.Pop();
            }
            else
            {
                return false;
            }
        }
        return stack.Count == 0;
    }

    public static char GetPair(char c)
    {
        if (c == ')')
            return '(';
        if (c == '}')
            return '{';
        if (c == ']')
            return '[';
        return '\0';
    }
}
```

Python 语言

- 执行结果：通过

- 执行用时：40 ms, 在所有 Python3 提交中击败了 47.30% 的用户
- 内存消耗：13.5 MB, 在所有 Python3 提交中击败了 5.16% 的用户

```python
class Solution:
    def isValid(self, s: str) -> bool:
        dic = {")": "(", "}": "{", "]": "["}
        if s is None:
            return True

        count = len(s)
        if count % 2 == 1:
            return False

        lst = list()
        for i in range(count):
            c = s[i]
            if len(lst) == 0:
                lst.append(c)
            elif c == "[" or c == "{" or c == "(":
                lst.append(c)
            elif lst[-1] == dic[c]:
                lst.pop()
            else:
                return False
        return len(lst) == 0
```

12 合并两个有序链表

- 题号：21
- 难度：简单
- https://leetcode-cn.com/problems/merge-two-sorted-lists/

将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例：

```
输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
```

C# 实现

- 执行结果：通过
- 执行用时：108 ms, 在所有 C# 提交中击败了 83.80% 的用户
- 内存消耗：25.9 MB, 在所有 C# 提交中击败了 5.85% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */
public class Solution
{
```

```
    public ListNode MergeTwoLists(ListNode l1, ListNode l2)
    {
        ListNode pHead = new ListNode(int.MaxValue);
        ListNode temp = pHead;

        while (l1 != null && l2 != null)
        {
            if (l1.val < l2.val)
            {
                temp.next = l1;
                l1 = l1.next;
            }
            else
            {
                temp.next = l2;
                l2 = l2.next;
            }
            temp = temp.next;
        }

        if (l1 != null)
            temp.next = l1;

        if (l2 != null)
            temp.next = l2;

        return pHead.next;
    }
}
```

Python 实现

- 执行结果：通过
- 执行用时：40 ms, 在所有 Python3 提交中击败了 68.12% 的用户
- 内存消耗：13.4 MB, 在所有 Python3 提交中击败了 17.11% 的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
```

```python
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        pHead = ListNode(None)
        temp = pHead
        while(l1 and l2):
            if (l1.val <= l2.val):
                temp.next = l1
                l1 = l1.next
            else :
                temp.next = l2
                l2 = l2.next
            temp = temp.next

        if l1 is not None:
            temp.next = l1
        else :
            temp.next = l2

        return pHead.next
```

13 合并 K 个排序链表

- 题号：23
- 难度：困难
- https://leetcode-cn.com/problems/merge-k-sorted-lists/

合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

示例:

```
输入:
[
  1->4->5,
  1->3->4,
  2->6
]
输出: 1->1->2->3->4->4->5->6
```

第一种：两两合并的方式

构造合并两个有序链表得到一个新的有序链表的方法：`ListNode MergeTwoLists(ListNode l1, ListNode l2)`。可以使用该方法合并前两个有序链表得到一个新的有序链表，之后把这个新链表与第三个有序链表合并，依次类推，最后得到合并 `k` 个有序列表的新列表。

- 执行结果：通过
- 执行用时：256 ms, 在所有 C# 提交中击败了 36.69% 的用户
- 内存消耗：29.3 MB, 在所有 C# 提交中击败了 18.37% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public ListNode MergeTwoLists(ListNode l1, ListNode l2)
    {
        ListNode pHead = new ListNode(int.MaxValue);
        ListNode temp = pHead;

        while (l1 != null && l2 != null)
        {
            if (l1.val < l2.val)
            {
                temp.next = l1;
                l1 = l1.next;
            }
            else
            {
                temp.next = l2;
                l2 = l2.next;
            }
            temp = temp.next;
        }

        if (l1 != null)
            temp.next = l1;

        if (l2 != null)
            temp.next = l2;

        return pHead.next;
    }

    public ListNode MergeKLists(ListNode[] lists) {
        if (lists.Length == 0)
            return null;

        ListNode result = lists[0];
```

```
        for (int i = 1; i < lists.Length; i++)
        {
            result = MergeTwoLists(result, lists[i]);
        }
        return result;
    }
}
```

第二种：选择值最小结点的方式

C# 语言

- 执行结果：通过
- 执行用时：776 ms, 在所有 C# 提交中击败了 7.10% 的用户
- 内存消耗：29.2 MB, 在所有 C# 提交中击败了 22.45% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public ListNode MergeKLists(ListNode[] lists)
    {
        int len = lists.Length;
        if (len == 0)
            return null;

        ListNode pHead = new ListNode(-1);
        ListNode temp = pHead;
        while (true)
        {
            int min = int.MaxValue;
            int minIndex = -1;
            for (int i = 0; i < len; i++)
            {
```

```
                if (lists[i] != null)
                {
                    if (lists[i].val < min)
                    {
                        minIndex = i;
                        min = lists[i].val;
                    }
                }
            }
            if (minIndex == -1)
            {
                break;
            }
            temp.next = lists[minIndex];
            temp = temp.next;
            lists[minIndex] = lists[minIndex].next;
        }
        return pHead.next;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：7772 ms, 在所有 Python3 提交中击败了 5.02% 的用户
- 内存消耗：16.5 MB, 在所有 Python3 提交中击败了 44.31% 的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        import sys
        count = len(lists)
        if count == 0:
            return None
        pHead = ListNode(-1)
        temp = pHead
```

```
        while True:
            minValue = sys.maxsize
            minIndex = -1
            for i in range(count):
                if lists[i] is not None:
                    if lists[i].val<minValue :
                        minIndex = i
                        minValue = lists[i].val
            if minIndex == -1:
                break
            temp.next = lists[minIndex]
            temp = temp.next
            lists[minIndex] = lists[minIndex].next
        return pHead.next
```

14 删除排序数组中的重复项

- 题号：26
- 难度：简单
- https://leetcode-cn.com/problems/remove-duplicates-from-sorted-array/

给定一个 **排序数组**，你需要在 **原地** 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在 **原地修改输入数组** 并在使用 O(1) 额外空间的条件下完成。

示例 1:

```
给定数组 nums = [1,1,2],

函数应该返回新的长度 2，并且原数组 nums 的前两个元素被修改为 1，2。

你不需要考虑数组中超出新长度后面的元素。
```

示例 2:

```
给定 nums = [0,0,1,1,1,2,2,3,3,4],

函数应该返回新的长度 5，并且原数组 nums 的前五个元素被修改为 0，1，2，3，4。

你不需要考虑数组中超出新长度后面的元素。
```

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以“**引用**”方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

第一种：双索引法

思路 就是一个快索引一个慢索引，j 快 i 慢，当 nums[j] == nums[i]时，j++就可以跳过重复项，不相等时，让 i++并让 nums[i] = nums[j]，把值复制过来继续执行到末尾即可，时间复杂度为 O(n)。

C# 实现

- 执行结果：通过
- 执行用时：300 ms, 在所有 C# 提交中击败了 64.43% 的用户
- 内存消耗：33.5 MB, 在所有 C# 提交中击败了 5.48% 的用户

```
public class Solution
{
    public int RemoveDuplicates(int[] nums)
    {
        if (nums.Length < 2)
            return nums.Length;
```

```
        int i = 0;
        for (int j = 1; j < nums.Length; j++)
        {
            if (nums[j] != nums[i])
            {
                i++;
                nums[i] = nums[j];
            }
        }
        return i + 1;
    }
}
```

Python 实现

- 执行结果：通过
- 执行用时：56 ms, 在所有 Python3 提交中击败了 95.28% 的用户
- 内存消耗：14.4 MB, 在所有 Python3 提交中击败了 57.13% 的用户

```
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        if len(nums) < 2:
            return 1
        i, j = 0, 1
        while j < len(nums):
            if nums[i] != nums[j]:
                i += 1
                nums[i] = nums[j]
            else:
                j += 1
        return i + 1
```

15 搜索旋转排序数组

- 题号：33
- 难度：中等
- https://leetcode-cn.com/problems/search-in-rotated-sorted-array/

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 O(log n) 级别。

示例 1:

```
输入: nums = [4,5,6,7,0,1,2], target = 0
输出: 4
```

示例 2:

```
输入: nums = [4,5,6,7,0,1,2], target = 3
输出: -1
```

示例 3:

```
输入: nums = [5,1,3], target = 5
输出: 0
```

示例 4:

```
输入: nums = [4,5,6,7,8,1,2,3], target = 8
输出: 0
```

示例 5:

```
输入：nums = [3,1], target = 1
输出：1
```

第一种：利用二分法

- 状态：通过
- 196 / 196 个通过测试用例
- 执行用时: 128 ms, 在所有 C# 提交中击败了 97.17% 的用户
- 内存消耗: 23.8 MB, 在所有 C# 提交中击败了 12.00% 的用户

```
public class Solution
{
    public int Search(int[] nums, int target)
    {
        int i = 0, j = nums.Length - 1;
        while (i <= j)
        {
            int mid = (i + j) / 2;
            if (nums[mid] == target)
                return mid;

            if (nums[mid] >= nums[i])
            {
                //左半部分有序
                if (target > nums[mid])
                {
                    i = mid + 1;
                }
                else
                {
                    if (target == nums[i])
                        return i;

                    if (target > nums[i])
```

```
                    j = mid - 1;
                else
                    i = mid + 1;
            }
        }
        else
        {
            if (target < nums[mid])
            {
                j = mid - 1;
            }
            else
            {
                if (target == nums[j])
                    return j;
                if (target < nums[j])
                    i = mid + 1;
                else
                    j = mid - 1;
            }
        }
    }
    return -1;
  }
}
```

16 字符串相乘

- 题号：43
- 难度：中等
- https://leetcode-cn.com/problems/multiply-strings/

给定两个以字符串形式表示的非负整数 num1 和 num2，返回 num1 和 num2 的乘积，它们的乘积也表示为字符串形式。

示例 1:

```
输入: num1 = "2", num2 = "3"
输出: "6"
```

示例 2:

```
输入: num1 = "123", num2 = "456"
输出: "56088"
```

示例 3:

```
输入: num1 = "498828660196", num2 = "840477629533"
输出: "419254329864656431168468"
```

说明:

- num1 和 num2 的长度小于 110。
- num1 和 num2 只包含数字 0-9。
- num1 和 num2 均不以零开头，除非是数字 0 本身。
- 不能使用任何标准库的大数类型（比如 BigInteger）或直接将输入转换为整数来处理。

- 状态：通过
- 311 / 311 个通过测试用例
- 执行用时：132 ms, 在所有 C# 提交中击败了 94.74% 的用户
- 内存消耗：24.1 MB, 在所有 C# 提交中击败了 31.82% 的用户

```
public class Solution {
    public string Multiply(string num1, string num2) {
        if (num1 == "0" || num2 == "0")
            return "0";
        int len1 = num1.Length;
        int len2 = num2.Length;
        int len = len1 + len2;
        int[] temp = new int[len];

        for (int i = len2 - 1; i >= 0; i--)
        {
            int k = len2 - i;
            int b = num2[i] - '0';
            for (int j = len1 - 1; j >= 0; j--)
            {
                int a = num1[j] - '0';
                int c = a*b;

                temp[len - k] += c%10;
                if (temp[len - k] >= 10)
                {
                    temp[len - k] = temp[len - k]%10;
                    temp[len - k - 1]++;
                }

                temp[len - k - 1] += c/10;
                if (temp[len - k - 1] >= 10)
                {
                    temp[len - k - 1] = temp[len - k - 1]%10;
                    temp[len - k - 2]++;
                }
                k++;
            }
```

```
        }

        StringBuilder sb = new StringBuilder();
        int s = temp[0] == 0 ? 1 : 0;
        while (s < len)
        {
            sb.Append(temp[s]);
            s++;
        }
        return sb.ToString();
    }
}
```

17 全排列

- 题号：46
- 难度：中等
- https://leetcode-cn.com/problems/permutations/

给定一个没有重复数字的序列，返回其所有可能的全排列。

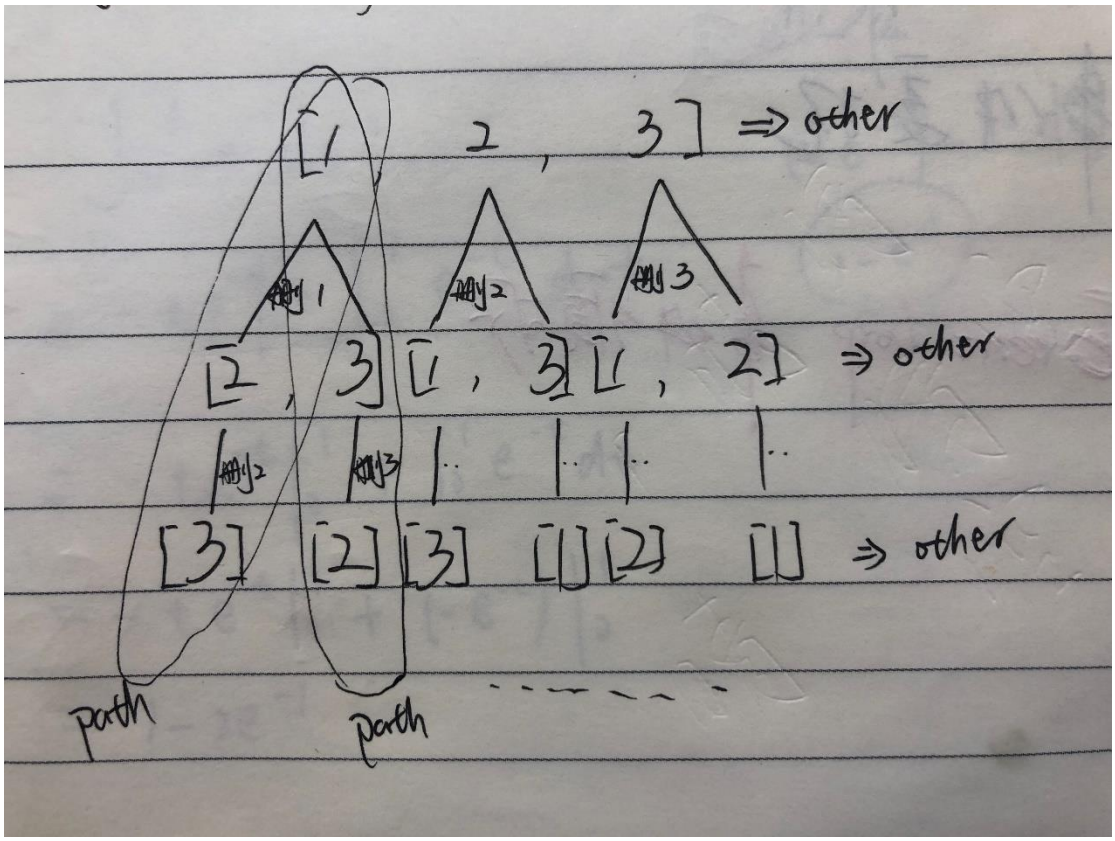
示例:

```
输入：[1,2,3]
输出：
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

第一种：回溯法（back tracking） 是一种选优搜索法，又称为试探法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

白话：回溯法可以理解为通过选择不同的岔路口寻找目的地，一个岔路口一个岔路口的去尝试找到目的地。如果走错了路，继续返回来找到岔路口的另一条路，直到找到目的地。

本练习的回溯过程如下所示：



- 状态：通过
- 25 / 25 个通过测试用例
- 执行用时: 364 ms, 在所有 C# 提交中击败了 80.00% 的用户
- 内存消耗: 30.6 MB, 在所有 C# 提交中击败了 7.14% 的用户

```
public class Solution
{
    private IList<IList<int>> _result;
    private bool[] _used;

    public IList<IList<int>> Permute(int[] nums)
    {
        _result = new List<IList<int>>();
        if (nums == null || nums.Length == 0)
            return _result;
```

```
        _used = new bool[nums.Length];

        FindPath(nums, 0, new List<int>());
        return _result;
    }

    public void FindPath(int[] nums, int count, List<int> path)
    {
        if (count == nums.Length)
        {
            //递归终止条件
            List<int> item = new List<int>();
            item.AddRange(path);
            //加入拷贝
            _result.Add(item);
            return;
        }
        for (int i = 0; i < nums.Length; i++)
        {
            if (_used[i] == false)
            {
                path.Add(nums[i]);
                _used[i] = true;
                FindPath(nums, count + 1, path);
                path.RemoveAt(path.Count - 1);
                _used[i] = false;
            }
        }
    }
}
```

18 最大子序和

- 题号：53
- 难度：简单
- https://leetcode-cn.com/problems/maximum-subarray/

给定一个整数数组 nums，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例 1:

```
输入：[-2,1,-3,4,-1,2,1,-5,4],
输出：6
解释：连续子数组 [4,-1,2,1] 的和最大，为 6。
```

示例 2:

```
输入：[-2,1],
输出：1
```

进阶:

如果你已经实现复杂度为 O(n) 的解法，尝试使用更为精妙的分治法求解。

第一种：利用暴力算法

- 状态：通过
- 202 / 202 个通过测试用例
- 执行用时: 596 ms, 在所有 C# 提交中击败了 14.18% 的用户

- 内存消耗: 24.5 MB, 在所有 C# 提交中击败了 5.88% 的用户

```
public class Solution {
    public int MaxSubArray(int[] nums) {
        int len = nums.Length;
        if (len == 0)
            return 0;
        if (len == 1)
            return nums[0];
        int max = int.MinValue;

        for (int i = 0; i < len; i++)
        {
            int sum = nums[i];
            if (sum > max)
            {
                max = sum;
            }
            for (int j = i + 1; j < len; j++)
            {
                sum += nums[j];
                if (sum > max)
                {
                    max = sum;
                }
            }
        }
        return max;
    }
}
```

第二种：利用动态规划

动态规划的最优子结构如下：

```
max[i] = Max(max[i-1] + nums[i], nums[i])
```

- 状态：通过
- 202 / 202 个通过测试用例
- 执行用时: 136 ms, 在所有 C# 提交中击败了 91.85% 的用户

- 内存消耗: 24.4 MB, 在所有 C# 提交中击败了 5.88% 的用户

```
public class Solution {
    public int MaxSubArray(int[] nums) {
        int len = nums.Length;
        if (len == 0)
            return 0;
        if (len == 1)
            return nums[0];
        int[] max = new int[len];
        max[0] = nums[0];
        int result = max[0];
        for (int i = 1; i < len; i++)
        {
            if (max[i - 1] + nums[i] > nums[i])
            {
                max[i] = max[i - 1] + nums[i];
            }
            else
            {
                max[i] = nums[i];
            }

            if (max[i] > result)
            {
                result = max[i];
            }
        }
        return result;
    }
}
```

19 螺旋矩阵

- 题号：54
- 难度：中等
- https://leetcode-cn.com/problems/spiral-matrix/

给定一个包含 m x n 个元素的矩阵（m 行, n 列），请按照顺时针螺旋顺序，返回矩阵中的所有元素。

示例 1:

```
输入:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
输出: [1,2,3,6,9,8,7,4,5]
```

示例 2:

```
输入:
[
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9,10,11,12]
]
输出: [1,2,3,4,8,12,11,10,9,5,6,7]
```

示例 3:

```
输入:
[
  [1]
]
输出: [1]
```

示例 4:

```
输入:
[
  [2, 3, 4],
  [5, 6, 7],
  [8, 9, 10],
  [11, 12, 13]
]
输出: [2,3,4,7,10,13,12,11,8,5,6,9]
```

- 状态：通过
- 22 / 22 个通过测试用例
- 执行用时: 348 ms, 在所有 C# 提交中击败了 85.53% 的用户
- 内存消耗: 28.9 MB, 在所有 C# 提交中击败了 5.26% 的用户

```
public class Solution
{
    public IList<int> SpiralOrder(int[][] matrix)
    {
        IList<int> result = new List<int>();
        if (matrix == null || matrix.Length == 0)
            return result;

        int start = 0;
        int end1 = matrix[start].Length - 1 - start;
        int end2 = matrix.Length - 1 - start;

        // 只有横着的情况
        if (start == end2)
        {
            LeftToRight(start, end1, start, matrix, result);
            return result;
        }
        //只有竖着的情况
        if (start == end1)
        {
```

```
            TopToBottom(start, end2, start, matrix, result);
            return result;
        }

        while (start < end1 && start < end2)
        {
            LeftToRight(start, end1, start, matrix, result);
            TopToBottom(start + 1, end2, end1, matrix, result);
            RightToLeft(end1 - 1, start, end2, matrix, result);
            BottomToTop(end2 - 1, start + 1, start, matrix, result);
            start++;
            end1 = matrix[start].Length - 1 - start;
            end2 = matrix.Length - 1 - start;
        }
        // 只剩下横着的情况
        if (start == end2)
        {
            LeftToRight(start, end1, start, matrix, result);
        }
        else if (start == end1)
        {
            //只剩下竖着的情况
            TopToBottom(start, end2, start, matrix, result);
        }
        return result;
    }

    private void LeftToRight(int start, int end, int rowIndex, int[][] matrix,
IList<int> lst)
    {
        for (int i = start; i <= end; i++)
        {
            lst.Add(matrix[rowIndex][i]);
        }
    }

    private void TopToBottom(int start, int end, int colIndex, int[][] matrix,
IList<int> lst)
    {
        for (int i = start; i <= end; i++)
        {
            lst.Add(matrix[i][colIndex]);
        }
    }
```

```
    private void RightToLeft(int start, int end, int rowIndex, int[][] matrix,
IList<int> lst)
    {
        for (int i = start; i >= end; i--)
        {
            lst.Add(matrix[rowIndex][i]);
        }
    }

    private void BottomToTop(int start, int end, int colIndex, int[][] matrix,
IList<int> lst)
    {
        for (int i = start; i >= end; i--)
        {
            lst.Add(matrix[i][colIndex]);
        }
    }
}
```

20 螺旋矩阵 II

- 题号：59
- 难度：中等
- https://leetcode-cn.com/problems/spiral-matrix-ii/

给定一个正整数 n，生成一个包含 1 到 n^2 所有元素，且元素按顺时针顺序螺旋排列的正方形矩阵。

示例:

```
输入：3
输出：
[
 [ 1, 2, 3 ],
 [ 8, 9, 4 ],
 [ 7, 6, 5 ]
]
```

- 状态：通过
- 20 / 20 个通过测试用例
- 执行用时: 296 ms, 在所有 C# 提交中击败了 97.67% 的用户
- 内存消耗: 25 MB, 在所有 C# 提交中击败了 11.11% 的用户

```
public class Solution
{
    public int[][] GenerateMatrix(int n)
    {
        int[][] matrix = new int[n][];
```

```
        for (int i = 0; i < n; i++)
        {
            matrix[i] = new int[n];
        }

        int start = 0;//起始位置
        int end1 = n - 1;//最左边位置
        int end2 = n - 1;//最下边位置
        int count = 1;

        while (start < end1 && start < end2)
        {
            LeftToRight(start, end1, start, matrix, ref count);
            TopToBottom(start + 1, end2, end1, matrix, ref count);
            RightToLeft(end1 - 1, start, end2, matrix, ref count);
            BottomToTop(end2 - 1, start + 1, start, matrix, ref count);
            start++;
            end1 = n - 1 - start;
            end2 = n - 1 - start;
        }
        if (n%2 == 1)
        {
            matrix[start][start] = count;
        }
        return matrix;
    }

    private void LeftToRight(int start, int end, int rowIndex, int[][] matrix, ref
int from)
    {
        for (int i = start; i <= end; i++)
        {
            matrix[rowIndex][i] = from;
            from++;
        }
    }

    private void TopToBottom(int start, int end, int colIndex, int[][] matrix, ref
int from)
    {
        for (int i = start; i <= end; i++)
        {
            matrix[i][colIndex] = from;
            from++;
```

```
        }
    }

    private void RightToLeft(int start, int end, int rowIndex, int[][] matrix, ref
int from)
    {
        for (int i = start; i >= end; i--)
        {
            matrix[rowIndex][i] = from;
            from++;
        }
    }

    private void BottomToTop(int start, int end, int colIndex, int[][] matrix, ref
int from)
    {
        for (int i = start; i >= end; i--)
        {
            matrix[i][colIndex] = from;
            from++;
        }
    }
}
```

21 旋转链表

- 题号：61
- 难度：中等
- https://leetcode-cn.com/problems/rotate-list/

给定一个链表，旋转链表，将链表每个节点向右移动 k 个位置，其中 k 是非负数。

示例 1:

```
输入: 1->2->3->4->5->NULL, k = 2
输出: 4->5->1->2->3->NULL

解释:
向右旋转 1 步: 5->1->2->3->4->NULL
向右旋转 2 步: 4->5->1->2->3->NULL
```

示例 2:

```
输入: 0->1->2->NULL, k = 4
输出: 2->0->1->NULL

解释:
向右旋转 1 步: 2->0->1->NULL
向右旋转 2 步: 1->2->0->NULL
向右旋转 3 步: 0->1->2->NULL
向右旋转 4 步: 2->0->1->NULL
```

- 执行结果：通过
- 执行用时：100 ms, 在所有 C# 提交中击败了 98.13% 的用户
- 内存消耗：25.1 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public ListNode RotateRight(ListNode head, int k)
    {
        if (head == null || k == 0)
            return head;

        int len = GetLength(head);
        int index = len - k%len;

        if (index == len)
            return head;

        ListNode temp1 = head;
        ListNode temp2 = head;
        for (int i = 0; i < index - 1; i++)
        {
            temp1 = temp1.next;
        }
        head = temp1.next;
        temp1.next = null;

        temp1 = head;
        while (temp1.next != null)
        {
            temp1 = temp1.next;
        }
        temp1.next = temp2;
        return head;
    }

    public int GetLength(ListNode head)
    {
        ListNode temp = head;
        int i = 0;
```

```
        while (temp != null)
        {
            i++;
            temp = temp.next;
        }
        return i;
    }
}
```

22 不同路径

- 题号：62
- 难度：中等
- https://leetcode-cn.com/problems/unique-paths/

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径？

例如，上图是一个 7 x 3 的网格。有多少可能的路径？

说明：m 和 n 的值均不超过 100。

示例 1：

```
输入：m = 3, n = 2
输出：3
解释：
从左上角开始，总共有 3 条路径可以到达右下角。
```

```
1. 向右 -> 向右 -> 向下
2. 向右 -> 向下 -> 向右
3. 向下 -> 向右 -> 向右
```

示例 2:

```
输入: m = 7, n = 3
输出: 28
```

示例 3:

```
输入: m = 23, n = 12
输出: 193536720
```

第一种：利用递归

```
public class Solution
{
    private int _m;
    private int _n;
    public int UniquePaths(int m, int n)
    {
        _m = m;
        _n = n;
        int count = 0;
        RecordPaths(0, 0, ref count);
        return count;
    }
    private void RecordPaths(int i, int j, ref int count)
    {
        if (i == _m - 1 && j == _n - 1)
        {
            count++;
            return;
        }
        if (i < _m)
        {
            RecordPaths(i + 1, j, ref count);
        }
        if (j < _n)
        {
            RecordPaths(i, j + 1, ref count);
```

```
        }
    }
}
```

使用递归的方式，容易理解但会耗费大量的时间，所以在运行 示例 3 的时候，超时了。

第二种：利用动态规划

动态规划表格 01：

1	1	1	1	1	1	1
1						
1						

动态规划表格 02：

1	1	1	1	1	1	1
1	2	3	4	5	6	7
1	3	6	10	15	21	28

动态规划的最优子结构为：`d[i,j] = d[i-1,j] + d[i,j-1]`

- 状态：通过
- 62 / 62 个通过测试用例
- 执行用时: 52 ms, 在所有 C# 提交中击败了 93.18% 的用户
- 内存消耗: 13.6 MB, 在所有 C# 提交中击败了 17.65% 的用户

```
public class Solution
{
    public int UniquePaths(int m, int n)
    {
        int[,] memo = new int[m, n];
        for (int i = 0; i < m; i++)
```

```
        {
            for (int j = 0; j < n; j++)
            {
                if (i == 0)
                {
                    memo[i, j] = 1;
                }
                else if (j == 0)
                {
                    memo[i, j] = 1;
                }
                else
                {
                    memo[i, j] = memo[i - 1, j] + memo[i, j - 1];
                }
            }
        }
        return memo[m - 1, n - 1];
    }
}
```

23 爬楼梯

- 题号：70
- 难度：简单
- https://leetcode-cn.com/problems/climbing-stairs/

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

注意：给定 n 是一个正整数。

示例 1：

```
输入： 2
输出： 2
解释： 有两种方法可以爬到楼顶。
1. 1 阶 + 1 阶
2. 2 阶
```

示例 2：

```
输入： 3
输出： 3
解释： 有三种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶
```

示例 3：

```
输入： 44
输出： 1134903170
```

分析这个题目：

- 1 阶，f(1) = 1 种方案
- 2 阶，f(2) = 2 种方案
- 3 阶，f(3) = 3 种方案
- 4 阶，f(4) = 5 种方案
- ……
- n 阶，f(n) = f(n-1) + f(n-2) 种方案

即，该问题可以转换为斐波那契数列问题。

第一种：利用递归

```
public class Solution {
    public int ClimbStairs(int n) {
        if (n <= 2)
            return n;

        return ClimbStairs(n - 1) + ClimbStairs(n - 2);
    }
}
```

由于递归的执行速度，远远小于循环，导致“超出时间限制”。

第二种：利用循环

- 状态：通过
- 45 / 45 个通过测试用例
- 执行用时: 52 ms, 在所有 C# 提交中击败了 97.87% 的用户
- 内存消耗: 13.7 MB, 在所有 C# 提交中击败了 5.98% 的用户

```
public class Solution {
    public int ClimbStairs(int n) {
        if (n <= 2)
            return n;

        int first = 1;
        int second = 2;
        int result = 0;

        for (int i = 3; i <= n; i++)
        {
            result = first + second;
            first = second;
            second = result;
        }
        return result;
    }
}
```

24 子集

- 题号：78
- 难度：中等
- https://leetcode-cn.com/problems/subsets/

给定一组**不含重复元素**的整数数组 nums，返回该数组所有可能的子集（幂集）。

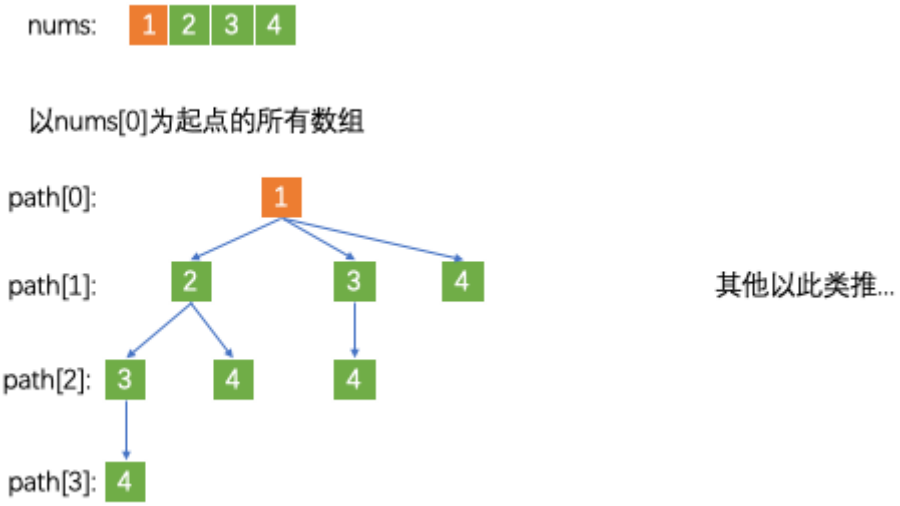
说明：解集不能包含重复的子集。

示例:

```
输入: nums = [1,2,3]
输出:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

第一种：回溯法

依次以 nums[i]为启始点进行搜索，且后续搜索数值都要大于前一个数值，这样会避免重复搜索。



- 状态：通过
- 10 / 10 个通过测试用例
- 行用时: 356 ms, 在所有 C# 提交中击败了 92.31% 的用户
- 内存消耗: 29.2 MB, 在所有 C# 提交中击败了 6.67% 的用户

```
public class Solution
{
    private IList<IList<int>> _result;

    public IList<IList<int>> Subsets(int[] nums)
    {
        _result = new List<IList<int>>();
        int len = nums.Length;

        if (len == 0)
        {
            return _result;
        }
        IList<int> item = new List<int>();
        Find(nums, 0, item);
        return _result;
    }

    private void Find(int[] nums, int begin, IList<int> item)
    {
```

```
        // 注意：这里要 new 一下
        _result.Add(new List<int>(item));

        if (begin == nums.Length)
            return;

        for (int i = begin; i < nums.Length; i++)
        {
            item.Add(nums[i]);
            Find(nums, i + 1, item);

            // 组合问题，状态在递归完成后要重置
            item.RemoveAt(item.Count - 1);
        }
    }
}
```

第二种：子集扩展法

- 状态：通过
- 10 / 10 个通过测试用例
- 执行用时: 352 ms, 在所有 C# 提交中击败了 94.51% 的用户
- 内存消耗: 29.2 MB, 在所有 C# 提交中击败了 6.67% 的用户

```
public class Solution
{
    public IList<IList<int>> Subsets(int[] nums)
    {
        IList<IList<int>> result = new List<IList<int>>();
        IList<int> item = new List<int>();
        result.Add(item);
        for (int i = 0; i < nums.Length; i++)
        {
            for (int j = 0, len = result.Count; j < len; j++)
            {
                item = new List<int>(result[j]);
                item.Add(nums[i]);
                result.Add(item);
            }
        }
```

```
        return result;
    }
}
```

第三种：位运算

通过二进制位指示子序列中保存哪些数。

以{1,2,3}为例，三个数，共 2^3 个子集，for 循环从 1 到 7，bit 表示就是 000 到 111，那么直接通过比特为指示哪些数保存到子序列中即可， 比如 101 就是保存{1,3}， 110 就是{1,2}……

- 状态：通过
- 10 / 10 个通过测试用例
- 执行用时: 348 ms, 在所有 C# 提交中击败了 97.80% 的用户
- 内存消耗: 29.5 MB, 在所有 C# 提交中击败了 6.67% 的用户

```
public class Solution
{
    public IList<IList<int>> Subsets(int[] nums)
    {
        IList<IList<int>> result = new List<IList<int>>();
        int len = nums.Length;

        for (int i = 0; i < 1 << len; i++)
        {
            IList<int> item = new List<int>();
            for (int j = 0; j < len; j++)
            {
                if (((1 << j) & i) != 0)
                    item.Add(nums[j]);
            }
            result.Add(item);
        }
        return result;
    }
```

```
}
```

25 合并两个有序数组

- 题号：88
- 难度：简单
- https://leetcode-cn.com/problems/merge-sorted-array/

给定两个有序整数数组 nums1 和 nums2，将 nums2 合并到 nums1 中，使得 num1 成为一个有序数组。

说明:

- 初始化 nums1 和 nums2 的元素数量分别为 m 和 n。
- 你可以假设 nums1 有足够的空间（空间大小大于或等于 m + n）来保存 nums2 中的元素。

示例:

```
输入:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],       n = 3

输出: [1,2,2,3,5,6]
```

- 状态：通过
- 59 / 59 个通过测试用例
- 执行用时: 348 ms, 在所有 C# 提交中击败了 93.33% 的用户

- 内存消耗: 29.2 MB, 在所有 C# 提交中击败了 6.43% 的用户

```
public class Solution
{
    public void Merge(int[] nums1, int m, int[] nums2, int n)
    {
        int j = 0;
        for (int i = 0; i < n; i++)
        {
            int num2 = nums2[i];
            while (num2 > nums1[j] && j < m)
            {
                j++;
            }
            int k = m;
            while (k > j)
            {
                nums1[k] = nums1[k - 1];
                k--;
            }
            m++;
            nums1[j] = num2;
        }
    }
}
```

26 格雷编码

- 题号：89
- 难度：中等
- https://leetcode-cn.com/problems/gray-code/

格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有一个位数的差异。

给定一个代表编码总位数的非负整数 n，打印其格雷编码序列。格雷编码序列必须以 0 开头。

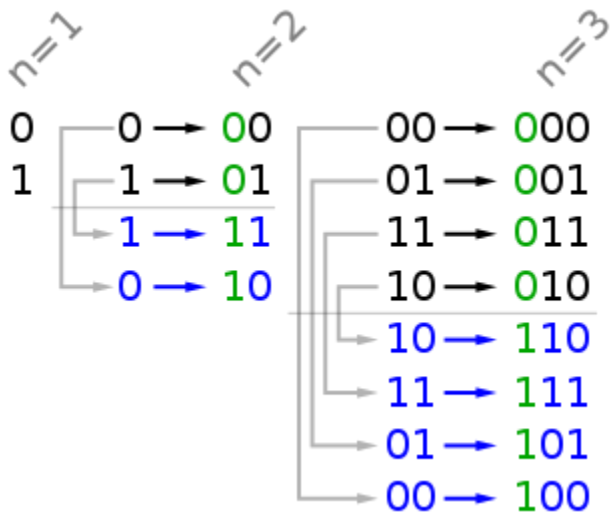
示例 1:

```
输入: 2
输出: [0,1,3,2]
解释:
00 - 0
01 - 1
11 - 3
10 - 2

对于给定的 n，其格雷编码序列并不唯一。
例如，[0,2,3,1] 也是一个有效的格雷编码序列。

00 - 0
10 - 2
11 - 3
01 - 1
```

示例 2:

```
输入: 0
输出: [0]
解释: 我们定义格雷编码序列必须以 0 开头。
    给定编码总位数为 n 的格雷编码序列，其长度为 2^n。
    当 n = 0 时，长度为 2^0 = 1。
```

因此，当 n = 0 时，其格雷编码序列为 [0]。



由 n 位推导 n+1 位结果时，n+1 位结果包含 n 位结果，同时包含 n 位结果中在高位再增加一个位 1 所形成的令一半结果，但是这一半结果需要与前一半结果镜像排列。

- 状态：通过
- 12 / 12 个通过测试用例
- 执行用时: 296 ms, 在所有 C# 提交中击败了 95.83% 的用户
- 内存消耗: 24.8 MB, 在所有 C# 提交中击败了 16.67% 的用户

```
public class Solution
{
    public IList<int> GrayCode(int n)
    {
        IList<int> lst = new List<int>();
        lst.Add(0);
        for (int i = 1; i <= n; i++)
        {
            for (int j = lst.Count - 1; j >= 0; j--)
            {
                int item = lst[j] + (1 << i - 1);
                lst.Add(item);
```

```
            }
        }
        return lst;
    }
}
```

27 二叉树的最大深度

- 题号：104
- 难度：简单
- https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

示例:

给定二叉树 [3,9,20,null,null,15,7]

```
  3
 / \
9  20
  /  \
 15   7
```

返回它的最大深度 3 。

第一种：利用队列实现层次遍历的思路

- 执行结果：通过
- 执行用时：108 ms, 在所有 C# 提交中击败了 88.13% 的用户
- 内存消耗：25.5 MB, 在所有 C# 提交中击败了 5.97% 的用户

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public int MaxDepth(TreeNode root)
    {
        if (root == null)
            return 0;

        Queue<TreeNode> q = new Queue<TreeNode>();
        int deep = 0;
        q.Enqueue(root);

        while (q.Count != 0)
        {
            deep++;
            int count = 0;
            int size = q.Count;

            while (count < size)
            {
                TreeNode node = q.Dequeue();
                count++;

                if (node.left != null)
                    q.Enqueue(node.left);
                if (node.right != null)
                    q.Enqueue(node.right);
            }
        }
        return deep;
    }
}
```

第二种：利用递归

思路：递归分别求左右子树的最大深度，并加到原有层数上，最后返回两者中的最大值。

C# 实现

- 执行结果：通过
- 执行用时：132 ms, 在所有 C# 提交中击败了 16.62% 的用户
- 内存消耗：25.5 MB, 在所有 C# 提交中击败了 6.06% 的用户

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public int MaxDepth(TreeNode root)
    {
        if (root == null)
            return 0;
        int llen = 1;
        int rlen = 1;
        if (root.left != null)
        {
            llen += MaxDepth(root.left);
        }
        if (root.right != null)
        {
            rlen += MaxDepth(root.right);
        }
        return llen > rlen ? llen : rlen;
    }
}
```

Python 实现

- 执行结果：通过

- 执行用时：40 ms, 在所有 Python3 提交中击败了 93.87% 的用户
- 内存消耗：14.9 MB, 在所有 Python3 提交中击败了 10.18% 的用户

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if root is None:
            return 0
        llen, rlen = 1, 1
        if root.left is not None:
            llen += self.maxDepth(root.left)
        if root.right is not None:
            rlen += self.maxDepth(root.right)
        return max(llen, rlen)
```

28 买卖股票的最佳时机

- 题号：121
- 难度：简单
- https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。

注意你不能在买入股票前卖出股票。

示例 1:

```
输入：[7,1,5,3,6,4]
输出：5
解释：在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6，因为卖出价格需要大于买入价格。
```

示例 2:

```
输入：[7,6,4,3,1]
输出：0
解释：在这种情况下，没有交易完成，所以最大利润为 0。
```

C# 实现

- 状态：通过

- 200 / 200 个通过测试用例
- 执行用时: 132 ms, 在所有 C# 提交中击败了 97.33% 的用户
- 内存消耗: 24 MB, 在所有 C# 提交中击败了 5.62% 的用户

```
public class Solution
{
    public int MaxProfit(int[] prices)
    {
        if (prices.Length <= 1)
            return 0;

        int min = prices[0];
        int max = 0;
        for (int i = 1; i < prices.Length; i++)
        {
            if (prices[i] < min)
            {
                min = prices[i];
            }
            int earn = prices[i] - min;
            if (earn > max)
            {
                max = earn;
            }
        }
        return max;
    }
}
```

Python 实现

- 执行结果：通过
- 执行用时：56 ms, 在所有 Python3 提交中击败了 64.24% 的用户
- 内存消耗：14.7 MB, 在所有 Python3 提交中击败了 11.35% 的用户

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if len(prices) <= 1:
```

```
        return 0
    min = prices[0]
    max = 0
    for i in range(1, len(prices)):
        if prices[i] < min:
            min = prices[i]
        earn = prices[i] - min
        if earn > max:
            max = earn
    return max
```

29 买卖股票的最佳时机 II

- 题号：122
- 难度：简单
- https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 7
解释:
    在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
    随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = 6-3 = 3 。
```

示例 2:

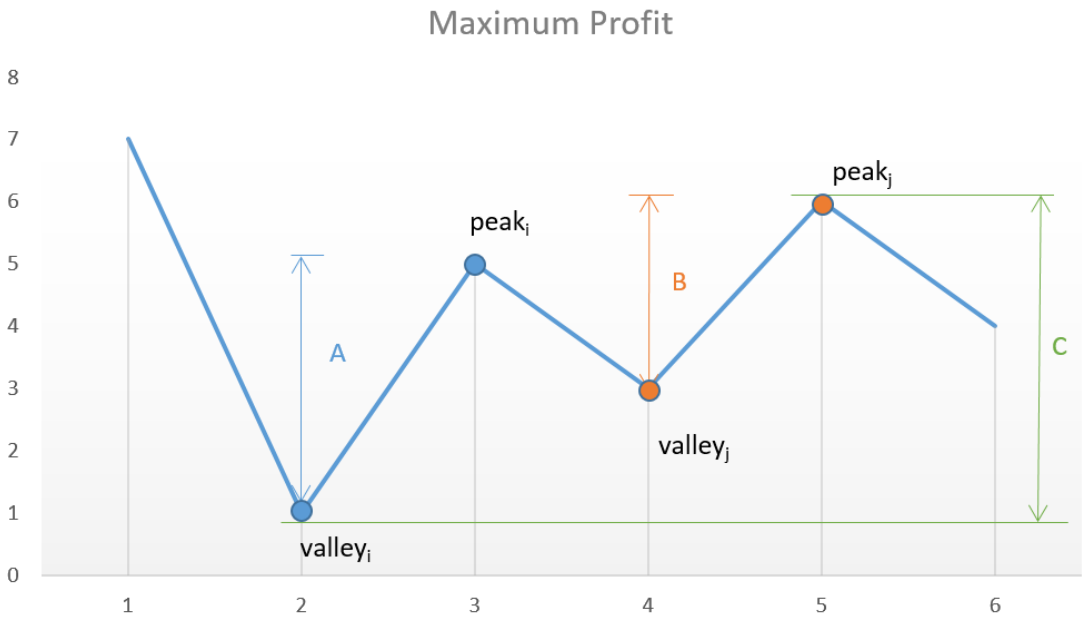
```
输入: [1,2,3,4,5]
输出: 4
解释:
    在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
    注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
    因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。
```

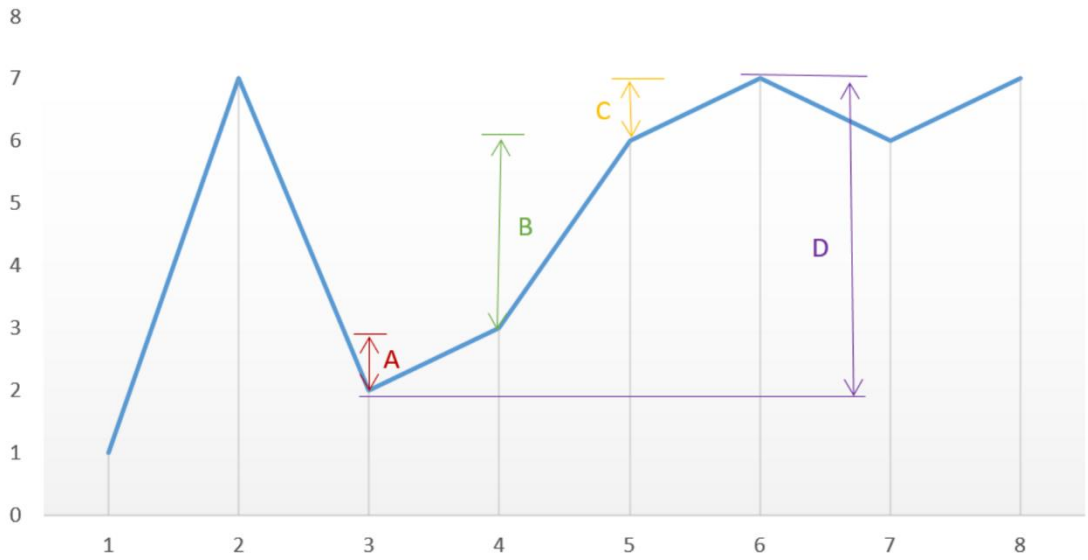
示例 3:

```
输入：[7,6,4,3,1]
输出：0
解释：在这种情况下，没有交易完成，所以最大利润为 0。
```

第一种：贪心算法

贪心策略：只要后一天价格比前一天高，就在前一天买进后一天卖出。





C# 实现

- 状态：通过
- 201 / 201 个通过测试用例
- 执行用时: 140 ms, 在所有 C# 提交中击败了 72.02% 的用户
- 内存消耗: 24.2 MB, 在所有 C# 提交中击败了 5.36% 的用户

```
public class Solution
{
    public int MaxProfit(int[] prices)
    {
        int earn = 0;
        for (int i = 0; i < prices.Length-1; i++)
        {
            earn += Math.Max(prices[i + 1] - prices[i], 0);
        }
        return earn;
    }
}
```

Python 实现

- 执行结果：通过
- 执行用时：40 ms, 在所有 Python3 提交中击败了 92.58% 的用户
- 内存消耗：14.7 MB, 在所有 Python3 提交中击败了 9.98% 的用户

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        earn = 0
        for i in range(0, len(prices) - 1):
            earn += max(prices[i + 1] - prices[i], 0)
        return earn
```

30 二叉树中的最大路径和

- 题号：124
- 难度：困难
- https://leetcode-cn.com/problems/binary-tree-maximum-path-sum/

给定一个非空二叉树，返回其最大路径和。

本题中，路径被定义为一条从树中任意节点出发，达到任意节点的序列。该路径**至少包含一个节点**，且不一定经过根节点。

示例 1:

```
输入: [1,2,3]

      1
     / \
    2   3

输出: 6
```

示例 2:

```
输入: [-10,9,20,null,null,15,7]

   -10
   / \
  9  20
    /  \
   15   7

输出: 42
```

对于任意一个节点，如果最大和路径包含该节点，那么只可能是两种情况：

- 其左右子树中所构成的和路径值较大的那个加上该节点的值后向父节点回溯构成最大路径。
- 左右子树都在最大路径中，加上该节点的值构成了最终的最大路径。

```
    8
   / \
  -3   7
 /  \
1    4
```

考虑左子树 -3 的路径的时候，我们有左子树 1 和右子树 4 的选择，但我们不能同时选择。

如果同时选了，路径就是 ... -> 1 -> -3 -> 4 就无法通过根节点 8 了。

所以我们只能去求左子树能返回的最大值，右子树能返回的最大值，选一个较大的。

- 状态：通过
- 93 / 93 个通过测试用例
- 执行用时: 152 ms，在所有 C# 提交中击败了 86.96% 的用户
- 内存消耗: 29.7 MB，在所有 C# 提交中击败了 37.50% 的用户

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */
```

```
public class Solution
{
    private int _max = int.MinValue;
    public int MaxPathSum(TreeNode root)
    {
        MaxPath(root);
        return _max;
    }

    private int MaxPath(TreeNode current)
    {
        if (current == null)
            return 0;
        // 如果子树路径和为负则应当置 0 表示最大路径不包含子树
        int left = Math.Max(MaxPath(current.left), 0);
        int right = Math.Max(MaxPath(current.right), 0);

        // 判断在该节点包含左右子树的路径和是否大于当前最大路径和
        _max = Math.Max(_max, current.val + left + right);
        return current.val + Math.Max(left, right);
    }
}
```

31 只出现一次的数字

- 题号：136
- 难度：简单
- https://leetcode-cn.com/problems/single-number/

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

示例 1：

```
输入：[2,2,1]
输出：1
```

示例 2：

```
输入：[4,1,2,1,2]
输出：4
```

第一种：利用“哈希”的方法。

- 状态：通过
- 16 / 16 个通过测试用例
- 执行用时: 136 ms，在所有 C# 提交中击败了 98.86% 的用户

- 内存消耗: 26.4 MB, 在所有 C# 提交中击败了 5.34% 的用户

```
public class Solution
{
    public int SingleNumber(int[] nums)
    {
        HashSet<int> h = new HashSet<int>();
        for (int i = 0; i < nums.Length; i++)
        {
            if (h.Contains(nums[i]))
            {
                h.Remove(nums[i]);
            }
            else
            {
                h.Add(nums[i]);
            }
        }
        return h.ElementAt(0);
    }
}
```

第二种：利用位运算的方法。

- 状态：通过
- 16 / 16 个通过测试用例
- 执行用时: 144 ms, 在所有 C# 提交中击败了 91.76% 的用户
- 内存消耗: 25.4 MB, 在所有 C# 提交中击败了 11.39% 的用户

```
A: 0 0 0 0 1 1 0 0
B: 0 0 0 0 0 1 1 1

A^B: 0 0 0 0 1 0 1 1
B^A: 0 0 0 0 1 0 1 1

A^A: 0 0 0 0 0 0 0 0
A^0: 0 0 0 0 1 1 0 0

A^B^A: = A^A^B = B = 0 0 0 0 0 1 1 1
```

"异或"操作满足交换律和结合律。

```
public class Solution
{
    public int SingleNumber(int[] nums)
    {
        int result = 0;

        for (int i = 0; i < nums.Length; i++)
        {
            result ^= nums[i];
        }
        return result;
    }
}
```

32 环形链表

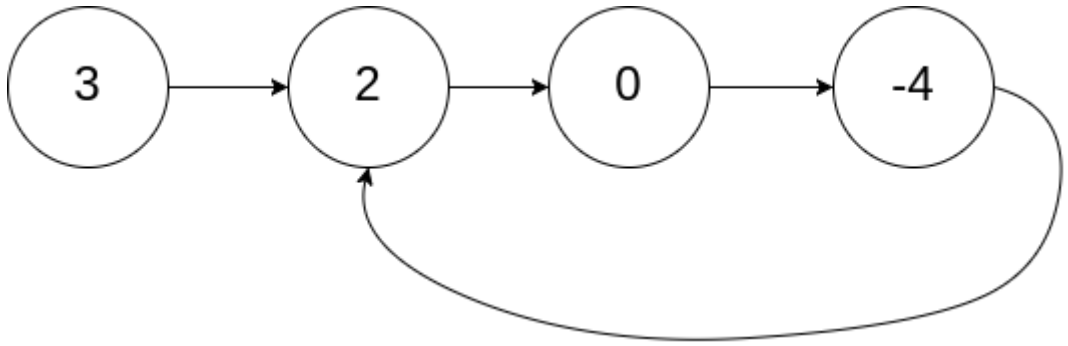
- 题号：141
- 难度：简单
- https://leetcode-cn.com/problems/linked-list-cycle/

给定一个链表，判断链表中是否有环。

为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。

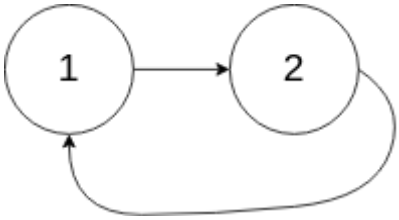
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：true
解释：链表中有一个环，其尾部连接到第一个节点。
```



示例 3：

```
输入：head = [1], pos = -1
输出：false
解释：链表中没有环。
```

进阶：

你能用 O(1)（即，常量）内存解决此问题吗？

第一种：利用 Hash 的方式

通过检查一个结点此前是否被访问过来判断链表是否为环形链表。

C# 语言

- 状态：通过
- 执行用时：172 ms, 在所有 C# 提交中击败了 8.84% 的用户
- 内存消耗：26.9 MB, 在所有 C# 提交中击败了 5.17% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
```

```
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */

public class Solution
{
    public bool HasCycle(ListNode head)
    {
        HashSet<ListNode> hashSet = new HashSet<ListNode>();
        hashSet.Add(head);

        while (head != null)
        {
            head = head.next;
            if (head == null)
                return false;
            if (hashSet.Contains(head))
                return true;
            hashSet.Add(head);
        }
        return false;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：88 ms, 在所有 Python3 提交中击败了 16.80% 的用户
- 内存消耗：16.9 MB, 在所有 Python3 提交中击败了 7.04% 的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
```

```python
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        hashset = set()
        hashset.add(head)
        while head is not None:
            head = head.next
            if head is None:
                return False
            if head in hashset:
                return True
            hashset.add(head)
        return False
```

第二种：利用双指针的方式

C# 语言

- 状态：通过
- 执行用时: 112 ms, 在所有 C# 提交中击败了 98.43% 的用户
- 内存消耗: 24.9 MB, 在所有 C# 提交中击败了 5.13% 的用户

```csharp
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public bool HasCycle(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;

        while (fast != null && fast.next != null)
        {
            fast = fast.next.next;
```

```
            slow = slow.next;
            if (fast == slow)
                return true;
        }
        return false;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：56 ms, 在所有 Python3 提交中击败了 60.97% 的用户
- 内存消耗：16.6 MB, 在所有 Python3 提交中击败了 11.81% 的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        fast = head
        slow = head
        while fast is not None and fast.next is not None:
            fast = fast.next.next
            slow = slow.next
            if fast == slow:
                return True
        return False
```

33 环形链表 II

- 题号：142
- 难度：中等
- https://leetcode-cn.com/problems/linked-list-cycle-ii/

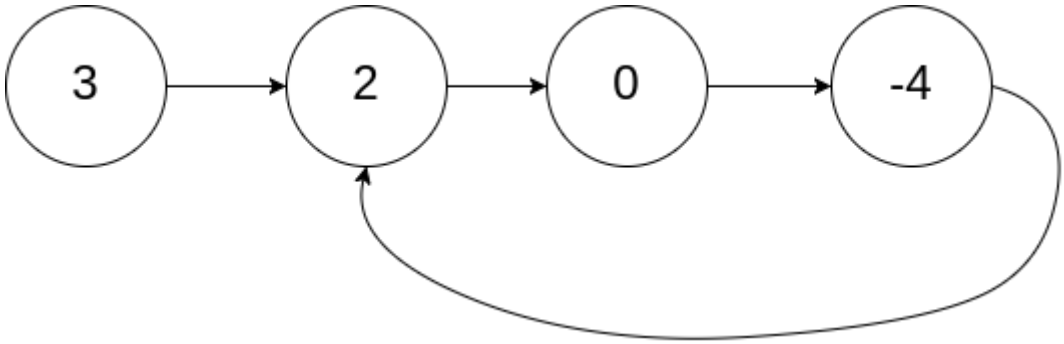
给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。

为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。

说明：不允许修改给定的链表。

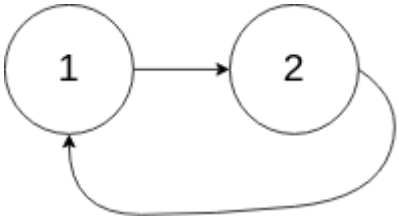
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：tail connects to node index 1
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：tail connects to node index 0
解释：链表中有一个环，其尾部连接到第一个节点。
```



示例 3:

```
输入：head = [1], pos = -1
输出：no cycle
解释：链表中没有环。
```

进阶：

你是否可以不用额外空间解决此题？

第一种：利用暴力匹配的方式

- 状态：通过
- 16 / 16 个通过测试用例
- 执行用时: 412 ms, 在所有 C# 提交中击败了 17.07% 的用户
- 内存消耗: 24.8 MB, 在所有 C# 提交中击败了 10.00% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) {
 *         val = x;
```

```
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode DetectCycle(ListNode head)
    {
        if (head == null)
            return null;

        ListNode p1 = head;
        int i = 0;
        while (p1 != null)
        {
            p1 = p1.next;
            i++;

            ListNode p2 = head;
            int j = 0;
            while (j < i)
            {
                if (p2 == p1)
                {
                    return p2;
                }
                p2 = p2.next;
                j++;
            }
        }
        return null;
    }
}
```

第二种：利用 Hash 的方式

- 状态：通过
- 16 / 16 个通过测试用例
- 执行用时: 140 ms, 在所有 C# 提交中击败了 82.93% 的用户
- 内存消耗: 26 MB, 在所有 C# 提交中击败了 5.00% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode DetectCycle(ListNode head)
    {
        if (head == null)
            return null;

        HashSet<ListNode> hash = new HashSet<ListNode>();
        hash.Add(head);

        ListNode temp = head.next;
        while (temp != null)
        {
            if (hash.Contains(temp))
                return temp;

            hash.Add(temp);
            temp = temp.next;
        }
        return null;
    }
}
```

34 LRU 缓存机制

- 题号：146
- 难度：中等
- https://leetcode-cn.com/problems/lru-cache/

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 `get(key)` - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。

写入数据 `put(key, value)` - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

进阶:

你是否可以在 O(1) 时间复杂度内完成这两种操作?

示例:

```
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回  1
cache.put(3, 3);    // 该操作会使得密钥 2 作废
cache.get(2);       // 返回 -1 (未找到)
cache.put(4, 4);    // 该操作会使得密钥 1 作废
cache.get(1);       // 返回 -1 (未找到)
cache.get(3);       // 返回  3
cache.get(4);       // 返回  4
```

计算机的缓存容量有限，如果缓存满了就要删除一些内容，给新内容腾位置。但问题是，删除哪些内容呢？我们肯定希望删掉哪些没什么用的缓存，而把有用的数据继续留在缓存里，方便之后继续使用。那么，什么样的数据，我们判定为「有用的」的数据呢？

LRU 缓存淘汰算法就是一种常用策略。LRU 的全称是 Least Recently Used，也就是说我们认为最近使用过的数据应该是是「有用的」，很久都没用过的数据应该是无用的，内存满了就优先删那些很久没用过的数据。

第一种：利用单链表的方式：

- 状态：通过
- 18 / 18 个通过测试用例
- 执行用时: 868 ms, 在所有 C# 提交中击败了 6.25% 的用户
- 内存消耗: 47.8 MB, 在所有 C# 提交中击败了 26.67% 的用户

```
public class LRUCache
{
    private readonly int _length;
    private readonly List<KeyValuePair<int, int>> _lst;

    public LRUCache(int capacity)
    {
        _length = capacity;
        _lst = new List<KeyValuePair<int, int>>();
    }

    private int GetIndex(int key)
    {
        for (int i=0,len=_lst.Count;i<len;i++)
        {
            if (_lst[i].Key == key)
```

```
            {
                return i;
            }
        }
        return -1;
    }

    public int Get(int key)
    {
        int index = GetIndex(key);
        if (index!=-1)
        {
            int val = _lst[index].Value;
            _lst.RemoveAt(index);
            _lst.Add(new KeyValuePair<int, int>(key, val));
            return val;
        }
        return -1;
    }

    public void Put(int key, int value)
    {
        int index = GetIndex(key);
        if (index!=-1)
        {
            _lst.RemoveAt(index);
        }
        else if (_lst.Count == _length)
        {
            _lst.RemoveAt(0);
        }
        _lst.Add(new KeyValuePair<int, int>(key, value));
    }
}

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.Get(key);
 * obj.Put(key,value);
 */
```

第二种：利用 字典 + 单链表 的方式：

- 状态：通过
- 18 / 18 个通过测试用例
- 执行用时: 392 ms, 在所有 C# 提交中击败了 76.56% 的用户
- 内存消耗: 47.9 MB, 在所有 C# 提交中击败了 20.00% 的用户

```
public class LRUCache
{
    private readonly List<int> _keys;
    private readonly Dictionary<int, int> _dict;


    public LRUCache(int capacity)
    {
        _keys = new List<int>(capacity);
        _dict = new Dictionary<int, int>(capacity);
    }

    public int Get(int key)
    {
        if (_dict.ContainsKey(key))
        {
            _keys.Remove(key);
            _keys.Add(key);
            return _dict[key];
        }
        return -1;
    }

    public void Put(int key, int value)
    {
        if (_dict.ContainsKey(key))
        {
            _dict.Remove(key);
            _keys.Remove(key);
        }
        else if (_keys.Count == _keys.Capacity)
        {
            _dict.Remove(_keys[0]);
            _keys.RemoveAt(0);
        }
```

```
        _keys.Add(key);
        _dict.Add(key, value);
    }
}

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.Get(key);
 * obj.Put(key,value);
 */
```

35 排序链表

- 题号：148
- 难度：中等
- https://leetcode-cn.com/problems/sort-list/

在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序。

示例 1:

```
输入：4->2->1->3
输出：1->2->3->4
```

示例 2:

```
输入：-1->5->3->4->0
输出：-1->0->3->4->5
```

思路： 利用归并的思想，递归地将当前链表分为两段，然后 merge，分两段的方法是使用 fast-slow 法，用两个指针，一个每次走两步，一个走一步，直到快的走到了末尾，然后慢的所在位置就是中间位置，这样就分成了两段。

- 状态：通过
- 16 / 16 个通过测试用例
- 执行用时: 124 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗: 29 MB, 在所有 C# 提交中击败了 25.00% 的用户

```
/**
```

```
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public ListNode SortList(ListNode head)
    {
        if (head == null)
            return null;
        return MergeSort(head);

    }

    private ListNode MergeSort(ListNode node)
    {
        if (node.next == null)
        {
            return node;
        }
        ListNode fast = node;
        ListNode slow = node;
        ListNode cut = null;
        while (fast != null && fast.next != null)
        {
            cut = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        cut.next = null;
        ListNode l1 = MergeSort(node);
        ListNode l2 = MergeSort(slow);
        return MergeTwoLists(l1, l2);
    }

    private ListNode MergeTwoLists(ListNode l1, ListNode l2)
    {
        ListNode pHead = new ListNode(int.MaxValue);
        ListNode temp = pHead;
```

```
        while (l1 != null && l2 != null)
        {
            if (l1.val < l2.val)
            {
                temp.next = l1;
                l1 = l1.next;
            }
            else
            {
                temp.next = l2;
                l2 = l2.next;
            }
            temp = temp.next;
        }

        if (l1 != null)
            temp.next = l1;

        if (l2 != null)
            temp.next = l2;

        return pHead.next;
    }
}
```

36 最小栈

- 题号：155
- 难度：简单
- https://leetcode-cn.com/problems/min-stack/

设计一个支持 push，pop，top 操作，并能在常数时间内检索到最小元素的栈。

- push(x) -- 将元素 x 推入栈中。
- pop() -- 删除栈顶的元素。
- top() -- 获取栈顶元素。
- getMin() -- 检索栈中的最小元素。

示例:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

第一种：利用单链表的方式

- 状态：通过
- 18 / 18 个通过测试用例

- 执行用时: 776 ms, 在所有 C# 提交中击败了 22.32% 的用户
- 内存消耗: 33.8 MB, 在所有 C# 提交中击败了 10.60% 的用户

```
public class MinStack
{
    /** initialize your data structure here. */
    private readonly IList<int> _lst;
    public MinStack()
    {
        _lst = new List<int>();
    }

    public void Push(int x)
    {
        _lst.Add(x);
    }

    public void Pop()
    {
        _lst.RemoveAt(_lst.Count - 1);
    }

    public int Top()
    {
        return _lst[_lst.Count - 1];
    }

    public int GetMin()
    {
        return _lst.Min();
    }
}

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.Push(x);
 * obj.Pop();
 * int param_3 = obj.Top();
 * int param_4 = obj.GetMin();
 */
```

第二种：利用辅助栈的方式

- 状态：通过
- 18 / 18 个通过测试用例
- 执行用时: 192 ms, 在所有 C# 提交中击败了 96.43% 的用户
- 内存消耗: 33.5 MB, 在所有 C# 提交中击败了 13.63% 的用户

```
public class MinStack
{

    /** initialize your data structure here. */
    private readonly Stack<int> _stack;
    private readonly Stack<int> _stackMin;

    public MinStack()
    {
        _stack = new Stack<int>();
        _stackMin = new Stack<int>();
    }

    public void Push(int x)
    {
        _stack.Push(x);
        if (_stackMin.Count == 0 || _stackMin.Peek() >= x)
        {
            _stackMin.Push(x);
        }
    }

    public void Pop()
    {
        int x = _stack.Pop();
        if (_stackMin.Peek() == x)
        {
            _stackMin.Pop();
        }
    }

    public int Top()
```

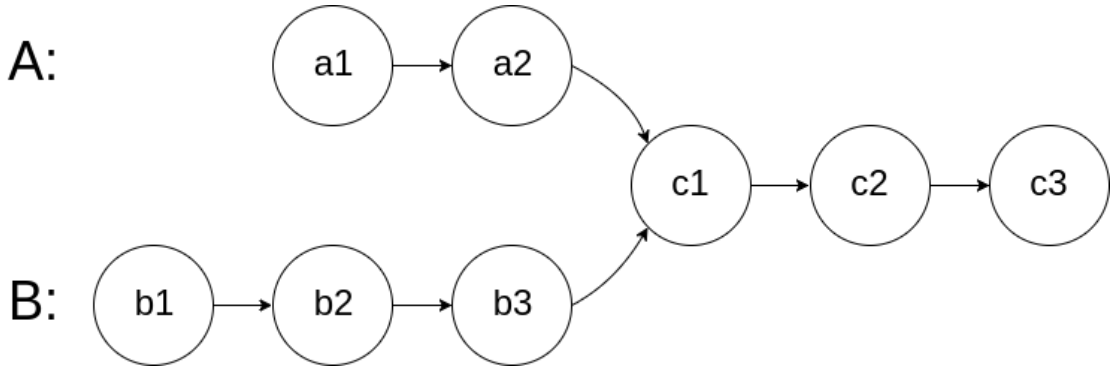
```
    {
        return _stack.Peek();
    }

    public int GetMin()
    {
        return _stackMin.Peek();
    }
}

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.Push(x);
 * obj.Pop();
 * int param_3 = obj.Top();
 * int param_4 = obj.GetMin();
 */
```

37 相交链表

- 题号：160
- 难度：简单
- https://leetcode-cn.com/problems/intersection-of-two-linked-lists/

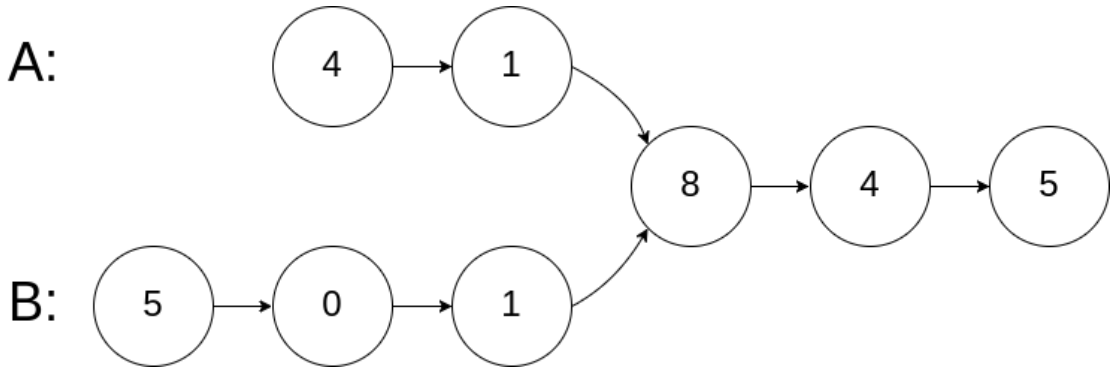
编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：



在节点 c1 开始相交。

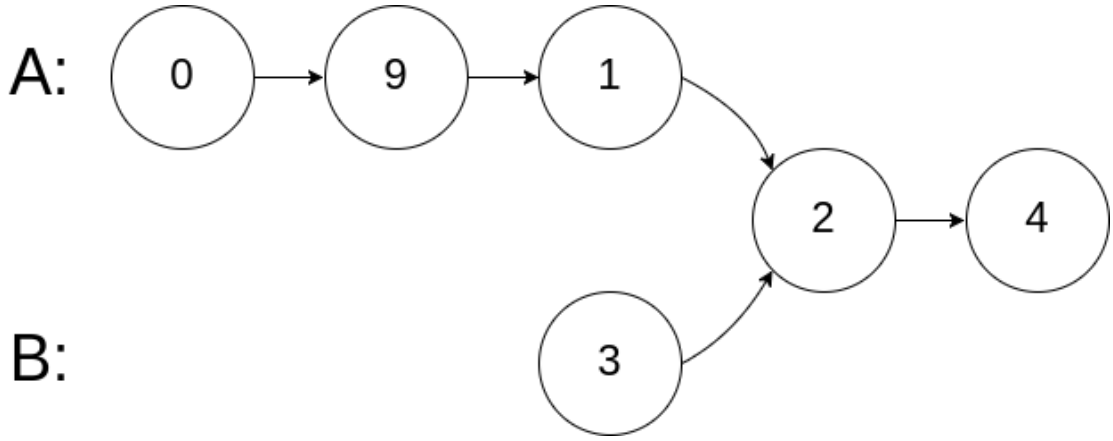
示例 1：




```
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
```

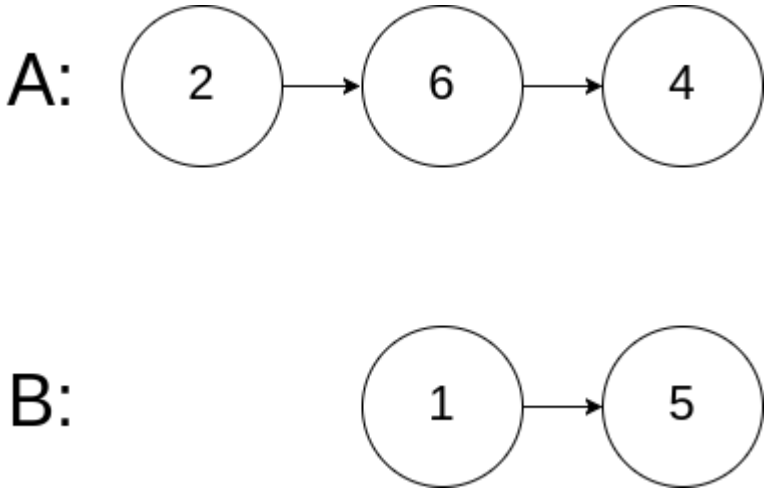
```
输出：Reference of the node with value = 8
输入解释：相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。
从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。
在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。
```

示例 2：




```
输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出：Reference of the node with value = 2
输入解释：相交节点的值为 2 （注意，如果两个列表相交则不能为 0）。
从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。
在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。
```

示例 3：




```
输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出：null
输入解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。
由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
解释：这两个链表不相交，因此返回 null。
```

注意：

- 如果两个链表没有交点，返回 null.
- 在返回结果后，两个链表仍须保持原有的结构。
- 可假定整个链表结构中没有循环。
- 程序尽量满足 O(n) 时间复杂度，且仅用 O(1) 内存。

第一种：利用哈希的方式

- 状态：通过
- 45 / 45 个通过测试用例
- 执行用时: 172 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗: 37.6 MB, 在所有 C# 提交中击败了 5.88% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public ListNode GetIntersectionNode(ListNode headA, ListNode headB)
    {
        HashSet<ListNode> hash = new HashSet<ListNode>();
        ListNode temp = headA;
        while (temp != null)
        {
```

```
            hash.Add(temp);
            temp = temp.next;
        }
        temp = headB;
        while (temp != null)
        {
            if (hash.Contains(temp))
                return temp;
            temp = temp.next;
        }
        return null;
    }
}
```

38 求众数

- 题号：169
- 难度：简单
- https://leetcode-cn.com/problems/majority-element/

给定一个大小为 n 的数组，找到其中的众数。众数是指在数组中出现次数大于 ⌊ n/2 ⌋ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在众数。

示例 1:

```
输入: [3,2,3]
输出: 3
```

示例 2:

```
输入: [2,2,1,1,1,2,2]
输出: 2
```

第一种：利用排序的方式

- 状态：通过
- 44 / 44 个通过测试用例
- 执行用时：192 ms

```
public class Solution {
    public int MajorityElement(int[] nums) {
```

```
        nums = nums.OrderBy(a => a).ToArray();
        return nums[nums.Length / 2];
    }
}
```

第二种：利用 Boyer-Moore 投票算法

寻找数组中超过一半的数字，这意味着数组中其他数字出现次数的总和都是比不上这个数字出现的次数。即如果把 该众数记为 +1 ，把其他数记为 −1 ，将它们全部加起来，和是大于 0 的。

- 状态：通过
- 44 / 44 个通过测试用例
- 执行用时：148 ms

```
public class Solution
{
    public int MajorityElement(int[] nums)
    {
        int candidate = nums[0];
        int count = 1;
        for (int i = 1; i < nums.Length; i++)
        {
            if (count == 0)
                candidate = nums[i];

            count += (nums[i] == candidate) ? 1 : -1;
        }
        return candidate;
    }
}
```

39 反转链表

- 题号：206
- 难度：简单
- https://leetcode-cn.com/problems/reverse-linked-list/

反转一个单链表。

示例:

```
输入：1->2->3->4->5->NULL
输出：5->4->3->2->1->NULL
```

进阶:

你可以迭代或递归地反转链表。你能否用两种方法解决这道题?

第一种：利用双指针的方式

- 状态：通过
- 27 / 27 个通过测试用例
- 执行用时: 116 ms, 在所有 C# 提交中击败了 97.50% 的用户
- 内存消耗: 23.3 MB, 在所有 C# 提交中击败了 5.26% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
```

```
 *     public ListNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public ListNode ReverseList(ListNode head)
    {
        if (head == null || head.next == null)
            return head;
        ListNode currentNode = head;
        ListNode newNode = null;
        while (currentNode != null)
        {
            ListNode tempNode = currentNode.next;
            currentNode.next = newNode;
            newNode = currentNode;
            currentNode = tempNode;
        }
        return newNode;
    }
}
```

40 数组中的第 K 个最大元素

- 题号：215
- 难度：中等
- https://leetcode-cn.com/problems/kth-largest-element-in-an-array/

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

```
输入：[3,2,1,5,6,4] 和 k = 2
输出：5
```

示例 2:

```
输入：[3,2,3,1,2,4,5,5,6] 和 k = 4
输出：4
```

说明:

你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

第一种：利用排序的方式

- 状态：通过
- 32 / 32 个通过测试用例
- 执行用时: 152 ms, 在所有 C# 提交中击败了 76.47% 的用户

- 内存消耗: 24.6 MB, 在所有 C# 提交中击败了 5.55% 的用户

```
public class Solution
{
    public int FindKthLargest(int[] nums, int k)
    {
        nums = nums.OrderBy(a => a).ToArray();
        return nums[nums.Length - k];
    }
}
```

41 存在重复元素

- 题号：217
- 难度：简单
- https://leetcode-cn.com/problems/contains-duplicate/

给定一个整数数组，判断是否存在重复元素。

如果任何值在数组中出现至少两次，函数返回 true。如果数组中每个元素都不相同，则返回 false。

示例 1:

```
输入：[1,2,3,1]
输出：true
```

示例 2:

```
输入：[1,2,3,4]
输出：false
```

示例 3:

```
输入：[1,1,1,3,3,4,3,2,4,2]
输出：true
```

第一种：通过哈希的方式

- 状态：通过
- 18 / 18 个通过测试用例

- 执行用时: 156 ms, 在所有 C# 提交中击败了 93.33% 的用户
- 内存消耗: 30.3 MB, 在所有 C# 提交中击败了 5.31% 的用户

```
public class Solution
{
    public bool ContainsDuplicate(int[] nums)
    {
        if (nums.Length < 2)
            return false;

        HashSet<int> h = new HashSet<int>();
        for (int i = 0; i < nums.Length; i++)
        {
            if (h.Contains(nums[i]))
                return true;
            h.Add(nums[i]);
        }
        return false;
    }
}
```

42 二叉搜索树中第 K 小的元素

- 题号：230
- 难度：中等
- https://leetcode-cn.com/problems/kth-smallest-element-in-a-bst/

给定一个二叉搜索树，编写一个函数 kthSmallest 来查找其中第 k 个最小的元素。

说明：

你可以假设 k 总是有效的，1 ≤ k ≤ 二叉搜索树元素个数。

示例 1：

```
输入: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
输出: 1
```

示例 2：

```
输入: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
输出: 3
```

进阶：

如果二叉搜索树经常被修改（插入/删除操作）并且你需要频繁地查找第 k 小的值，你将如何优化 kthSmallest 函数?

思路： 对二叉搜索树进行中序遍历，即可得到由小到大的序列。

- 状态：通过
- 91 / 91 个通过测试用例
- 执行用时: 128 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗: 27.2 MB, 在所有 C# 提交中击败了 9.09% 的用户

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */

public class Solution
{
    List<int> _lst;
    public int KthSmallest(TreeNode root, int k)
    {
        _lst=new List<int>();
        MidTraver(root);
        return _lst[k-1];
    }

    private void MidTraver(TreeNode current)
    {
        if(current==null)
            return;
        MidTraver(current.left);
```

```
        _lst.Add(current.val);
        MidTraver(current.right);
    }
}
```

43 2 的幂

- 题号：231
- 难度：简单
- https://leetcode-cn.com/problems/power-of-two/

给定一个整数，编写一个函数来判断它是否是 2 的幂次方。

示例 1:

```
输入：1
输出：true
解释：2^0 = 1
```

示例 2:

```
输入：16
输出：true
解释：2^4 = 16
```

示例 3:

```
输入：218
输出：false
```

第一种：利用二进制

思路： 异或具有该性质 `a^a=0`。

- 状态：通过
- 1108 / 1108 个通过测试用例

- 执行用时: 36 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗: 14.7 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
public class Solution
{
    public bool IsPowerOfTwo(int n)
    {
        if (n < 0)
            return false;
        for (int i = 0; i < 32; i++)
        {
            int k = 1 << i;
            if ((n ^ k) == 0)
                return true;
        }
        return false;
    }
}
```

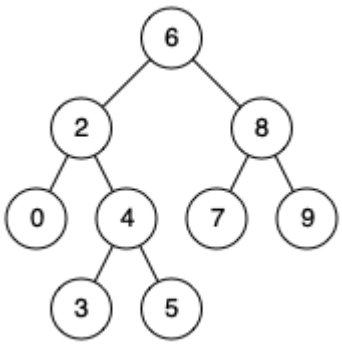
44 二叉搜索树的最近公共祖先

- 题号：235
- 难度：简单
- https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-search-tree/

给定一个二叉搜索树，找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1:

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。
```

示例 2:

```
输入：root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出：2
解释：节点 2 和节点 4 的最近公共祖先是 2，因为根据定义最近公共祖先节点可以为节点本身。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉搜索树中。

第一种：利用递归

- 状态：通过
- 27 / 27 个通过测试用例
- 执行用时: 128 ms, 在所有 C# 提交中击败了 95.71% 的用户
- 内存消耗: 30.8 MB, 在所有 C# 提交中击败了 7.69% 的用户

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public TreeNode LowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
    {
        if (root.val == p.val || root.val == q.val)
            return root;
        if (root.val > p.val && root.val < q.val)
            return root;
        if (root.val < p.val && root.val > q.val)
```

```
            return root;

        if (root.val < p.val && root.val < q.val)
        {
            return LowestCommonAncestor(root.right, p, q);
        }
        return LowestCommonAncestor(root.left, p, q);
    }
}
```

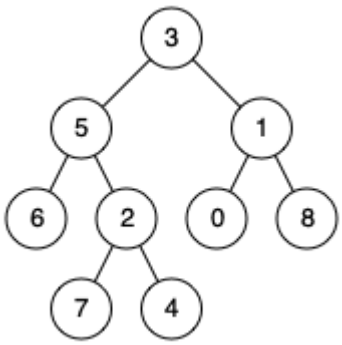
45 二叉树的最近公共祖先

- 题号：236
- 难度：中等
- https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/

给定一个二叉树，找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]



示例 1:

```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出：3
解释：节点 5 和节点 1 的最近公共祖先是节点 3。
```

示例 2:

```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出：5
解释：节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

第一种：利用递归

- 状态：通过
- 31 / 31 个通过测试用例
- 执行用时: 132 ms, 在所有 C# 提交中击败了 96.10% 的用户
- 内存消耗: 27.5 MB, 在所有 C# 提交中击败了 20.00% 的用户

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left;
 *     public TreeNode right;
 *     public TreeNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public TreeNode LowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
    {
        return Find(root, p, q);
    }

    private TreeNode Find(TreeNode current, TreeNode p, TreeNode q)
    {
```

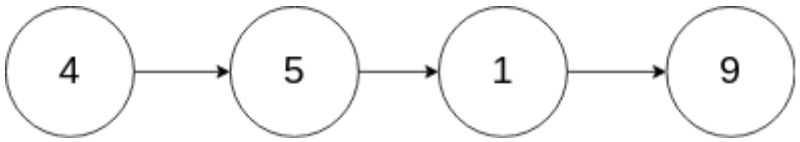
```
        if (current == null || current == p || current == q)
            return current;
        TreeNode left = Find(current.left, p, q);
        TreeNode right = Find(current.right, p, q);
        if (left != null && right != null)
            return current;
        return left != null ? left : right;
    }
}
```

46 删除链表中的节点

- 题号：237
- 难度：简单
- https://leetcode-cn.com/problems/delete-node-in-a-linked-list/

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点，你将只被给定要求被删除的节点。

现有一个链表 -- head = [4,5,1,9]，它可以表示为:



示例 1:

```
输入: head = [4,5,1,9], node = 5
输出: [4,1,9]
解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

示例 2:

```
输入: head = [4,5,1,9], node = 1
输出: [4,5,9]
解释: 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.
```

说明:

- 链表至少包含两个节点。
- 链表中所有节点的值都是唯一的。
- 给定的节点为非末尾节点并且一定是链表中的一个有效节点。

- 不要从你的函数中返回任何结果。

思路： 这道题没有给出链表的头节点，而是直接给出要删除的节点，让我们进行原地删除。我们对于该节点的前一个节点一无所知，所以无法直接执行删除操作。因此，我们将要删除节点的 next 节点的值赋值给要删除的节点，转而去删除 next 节点，从而达成目的。

- 状态：通过
- 41 / 41 个通过测试用例
- 执行用时: 120 ms, 在所有 C# 提交中击败了 99.55% 的用户
- 内存消耗: 24.4 MB, 在所有 C# 提交中击败了 5.88% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public void DeleteNode(ListNode node)
    {
        ListNode temp = node.next;
        while (temp != null)
        {
            node.val = temp.val;
            temp = temp.next;
            if (temp != null)
            {
                node = node.next;
```

```
            }
        }
        node.next = null;
    }
}
```

47 除自身以外数组的乘积

- 题号：238
- 难度：中等
- https://leetcode-cn.com/problems/product-of-array-except-self/

给定长度为 n 的整数数组 nums，其中 n > 1，返回输出数组 output，其中 output[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积。

示例:

```
输入：[1,2,3,4]
输出：[24,12,8,6]
```

说明: 请不要使用除法，且在 O(n) 时间复杂度内完成此题。

进阶:

你可以在常数空间复杂度内完成这个题目吗？（出于对空间复杂度分析的目的，输出数组不被视为额外空间。）

思路：乘积 = 当前数左边的乘积 * 当前数右边的乘积

```
              [1, 2, 3, 4]
   左边的乘积    [1, 1, 2, 6]
   右边的乘积    [24,12,4, 1]
结果 = 左*右     [24,12,8, 6]
```

- 状态：通过

- 17 / 17 个通过测试用例
- 执行用时: 304 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗: 34.6 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
public class Solution
{
    public int[] ProductExceptSelf(int[] nums)
    {
        int len = nums.Length;
        int[] output1 = new int[len];//正向乘积
        int[] output2 = new int[len];//反向乘积
        output1[0] = 1;
        output2[len - 1] = 1;
        for (int i = 1; i < len; i++)
        {
            output1[i] = output1[i - 1]*nums[i - 1];
            output2[len - i - 1] = output2[len - i]*nums[len - i];
        }
        for (int i = 0; i < len; i++)
        {
            output1[i] *= output2[i];
        }
        return output1;
    }
}
```

48 Nim 游戏

- 题号：292
- 难度：简单
- https://leetcode-cn.com/problems/nim-game/

你和你的朋友，两个人一起玩 Nim 游戏：桌子上有一堆石头，每次你们轮流拿掉 1 - 3 块石头。 拿掉最后一块石头的人就是获胜者。你作为先手。

你们是聪明人，每一步都是最优解。 编写一个函数，来判断你是否可以在给定石头数量的情况下赢得游戏。

示例:

```
输入：4
输出：false
解释：如果堆中有 4 块石头，那么你永远不会赢得比赛；
    因为无论你拿走 1 块、2 块 还是 3 块石头，最后一块石头总是会被你的朋友拿走。
```

思路：每一回合都必须拿石子，所以当到谁的回合还剩下 4 个石子，那么谁就输了。

- [1,3]先手赢
- [4]后手赢
- [5,7]先手赢，因为你可以使到对方回合时是剩下 4 个石子
- [8]后手赢，此时对方可以使在你的回合时剩下 4 个石子
- 以次类推可以发现当 n 为 4 的倍数时先手总会输

C# 实现

- 状态：通过
- 60 / 60 个通过测试用例
- 执行用时: 56 ms, 在所有 C# 提交中击败了 82.41% 的用户
- 内存消耗: 13.6 MB, 在所有 C# 提交中击败了 5.17% 的用户

```
public class Solution
{
    public bool CanWinNim(int n)
    {
        return (n % 4 != 0);
    }
}
```

Python 实现

- 执行结果：通过
- 执行用时：36 ms, 在所有 Python3 提交中击败了 68.15% 的用户
- 内存消耗：13.7 MB, 在所有 Python3 提交中击败了 10.00% 的用户

```
class Solution:
    def canWinNim(self, n: int) -> bool:
        return n%4 != 0
```

49 反转字符串

- 题号：344
- 难度：简单
- https://leetcode-cn.com/problems/reverse-string/

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 char[] 的形式给出。

不要给另外的数组分配额外的空间，你必须原地 **修改输入数组**、使用 O(1) 的额外空间解决这一问题。

你可以假设数组中的所有字符都是 ASCII 码表中的可打印字符。

示例 1：

```
输入：["h","e","l","l","o"]
输出：["o","l","l","e","h"]
```

示例 2：

```
输入：["H","a","n","n","a","h"]
输出：["h","a","n","n","a","H"]
```

第一种：利用双索引的方式

- 状态：通过
- 478 / 478 个通过测试用例

- 执行用时: 572 ms, 在所有 C# 提交中击败了 94.94% 的用户
- 内存消耗: 33.6 MB, 在所有 C# 提交中击败了 5.05% 的用户

```
public class Solution {
    public void ReverseString(char[] s) {
        int i = 0;
        int j = s.Length-1;
        while (i < j)
        {
            char c = s[i];
            s[i] = s[j];
            s[j] = c;
            i++;
            j--;
        }
    }
}
```

50 反转字符串中的单词 III

- 题号：557
- 难度：简单
- https://leetcode-cn.com/problems/reverse-words-in-a-string-iii/

给定一个字符串，你需要反转字符串中每个单词的字符顺序，同时仍保留空格和单词的初始顺序。

示例 1:

```
输入："Let's take LeetCode contest"
输出："s'teL ekat edoCteeL tsetnoc"
注意：在字符串中，每个单词由单个空格分隔，并且字符串中不会有任何额外的空格。
```

第一种：利用双索引的方式

- 状态：通过
- 30 / 30 个通过测试用例
- 执行用时: 128 ms, 在所有 C# 提交中击败了 98.82% 的用户
- 内存消耗: 33.9 MB, 在所有 C# 提交中击败了 20.00% 的用户

```
public class Solution
{
    public string ReverseWords(string s)
    {
        string[] words = s.Split(new char[] {' '});
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < words.Length; i++)
```

```
        {
            char[] w = words[i].ToArray();
            ReverseString(w);
            sb.Append(w);
            if (i != words.Length - 1)
            {
                sb.Append(' ');
            }
        }
        return sb.ToString();
    }

    public void ReverseString(char[] s)
    {
        int i = 0;
        int j = s.Length - 1;
        while (i < j)
        {
            char c = s[i];
            s[i] = s[j];
            s[j] = c;
            i++;
            j--;
        }
    }
}
```