

Table of Contents

Introduction	1.1
题目	1.2
0001-0100	1.2.1
0100-0200	1.2.2
0201-0300	1.2.3
0301-0400	1.2.4
0401-0500	1.2.5
0501-0600	1.2.6
0601-0700	1.2.7
0701-0800	1.2.8
mysql	1.2.9
bash	1.2.10
分类	1.3
初级算法	1.3.1
书籍	1.4
剑指offer	1.4.1

go-Leetcode



gitbook

0.参考

- <https://leetcode-cn.com/>

1.完成的题目

Title	Tag	难度	完成情况
第1期			
1.两数之和	数组、哈希表	Easy	完成
7.整数反转	数学	Easy	完成
9.回文数	数学	Easy	完成
13.罗马数字转整数	数学、字符串	Easy	完成
14.最长公共前缀	字符串	Easy	完成
20.有效的括号	栈、字符串	Easy	完成
21.合并两个有序链表	链表	Easy	完成
26.删除排序数组中的重复项	数组、双指针	Easy	完成
27.移除元素	数组、双指针	Easy	完成
28.实现strStr()	双指针、字符串	Easy	完成
第2期			
35.搜索插入位置	数组、二分查找	Easy	完成
38.报数	字符串	Easy	完成
53.最大子序和	数组、分治算法、动态规划	Easy	完成
58.最后一个单词的长度	字符串	Easy	完成

Title	Tag	难度	完成情况
66.加一	数组	Easy	完成
67.二进制求和	数字、字符串	Easy	完成
69.x 的平方根	数学, 二分查找	Easy	完成
70.爬楼梯	动态规划	Easy	完成
83.删除排序链表中的重复元素	链表	Easy	完成
88.合并两个有序数组	数组、双指针	Easy	完成
第3期			
100.相同的树	树、深度优先搜索	Easy	完成
101.对称二叉树	树、深度优先搜索、广度优先搜索	Easy	完成
104.二叉树的最大深度	树、深度优先搜索	Easy	完成
107.二叉树的层次遍历 II	树、广度优先搜索	Easy	完成
108.将有序数组转换为二叉搜索树	树、深度优先搜索	Easy	完成
110.平衡二叉树	树、深度优先搜索	Easy	完成
111.二叉树的最小深度	树、深度优先搜索、广度优先搜索	Easy	完成
112.路径总和	树、深度优先搜索	Easy	完成
118.杨辉三角	数组	Easy	完成

Title	Tag	难度	完成情况
119.杨辉三角 II	数组	Easy	完成
第4期			
121.买卖股票的最佳时机	数组、动态规划	Easy	完成
122.买卖股票的最佳时机 II	贪心算法、数组	Easy	完成
125.验证回文串	双指针、字符串	Easy	完成
136.只出现一次的数字	位运算、哈希表	Easy	完成
141.环形链表	链表、双指针	Easy	完成
155.最小栈	栈、设计	Easy	完成
160.相交链表	链表	Easy	完成
167.两数之和 II - 输入有序数组	数组、双指针、二分查找	Easy	完成
168.Excel表列名称	数学	Easy	完成
169.多数元素	位运算、数组、分治算法	Easy	完成
第5期			
171.Excel表列序号	数学	Easy	完成
172.阶乘后的零	数学	Easy	完成
175.组合两个表	Mysql	Easy	完成

Title	Tag	难度	完成情况
176.第二高的薪水	Mysql	Easy	完成
181.超过经理收入的员工	Mysql	Easy	完成
182.查找重复的电子邮箱	Mysql	Easy	完成
183.从不订购的客户	Mysql	Easy	完成
189.旋转数组	数组	Easy	完成
190.颠倒二进制位	位运算	Easy	完成
191.位1的个数	位运算	Easy	完成
第6期			
193.有效电话号码	Bash	Easy	完成
195.第十行	Bash	Easy	完成
196.删除重复的电子邮箱	Mysql	Easy	完成
197.上升的温度	Mysql	Easy	完成
198.打家劫舍	动态规划	Easy	完成
202.快乐数	哈希表、数学	Easy	完成
203.移除链表元素	链表	Easy	完成
204.计数质数	哈希表、数学	Easy	完成

Title	Tag	难度	完成情况
205.同构字符串	哈希表	Easy	完成
206.反转链表	链表	Easy	完成
第7期			
217.存在重复元素	数组、哈希表	Easy	完成
219.存在重复元素 II	数组、哈希表	Easy	完成
225.用队列实现栈	栈、设计	Easy	完成
226.翻转二叉树	树	Easy	完成
231.2的幂	位运算、数学	Easy	完成
232.用栈实现队列	栈、设计	Easy	完成
234.回文链表	链表、双指针	Easy	完成
235.二叉搜索树的最近公共祖先	树	Easy	完成
237.删除链表中的节点	链表	Easy	完成
242.有效的字母异位词	排序、哈希	Easy	完成
第8期			
257.二叉树的所有路径	树、深度优先搜索	Easy	完成
258.各位相加	数学	Easy	完成

Title	Tag	难度	完成情况
263.丑数	数学	Easy	完成
268.缺失数字	位运算、数组、数学	Easy	完成
278.第一个错误的版本	二分查找	Easy	完成
283.移动零	数组、双指针	Easy	完成
290.单词规律	哈希表	Easy	完成
292.Nim 游戏	脑筋急转弯、极小化极大	Easy	完成
299.猜数字游戏	哈希表	Easy	完成
303.区域和检索 - 数组不可变	动态规划	Easy	完成
第9期			
326.3的幂	数学	Easy	完成
342.4的幂	位运算	Easy	完成
344.反转字符串	双指针、字符串	Easy	完成
345.反转字符串中的元音字母	双指针、字符串	Easy	完成
349.两个数组的交集	排序、哈希表、双指针、二分查找	Easy	完成
350.两个数组的交集 II	排序、哈希表、双指针、二分查找	Easy	完成
367.有效的完全平方数	数学、二分查找	Easy	完成

Title	Tag	难度	完成情况
371.两整数之和	位运算	Easy	完成
374.猜数字大小	二分查找	Easy	完成
383.赎金信	字符串	Easy	完成
第10期			
387.字符串中的第一个唯一字符	哈希表、字符串	Easy	完成
389.找不同	位运算、哈希表	Easy	完成
392.判断子序列	贪心算法、二分查找、动态规划	Easy	完成
401.二进制手表	位运算、回溯算法	Easy	完成
404.左叶子之和	树	Easy	完成
405.数字转换为十六进制数	位运算	Easy	完成
409.最长回文串	哈希表	Easy	完成
412.Fizz Buzz		Easy	完成
414.第三大的数	数组	Easy	完成
415.字符串相加	字符串	Easy	完成
第11期			
434.字符串中的单词数	字符串	Easy	完成

Title	Tag	难度	完成情况
437.路径总和 III	树	Easy	完成
441.排列硬币	数学、二分查找	Easy	完成
443.压缩字符串	字符串	Easy	完成
447.回旋镖的数量	哈希表	Easy	完成
448.找到所有数组中消失的数字	数组	Easy	完成
453.最小移动次数使数组元素相等	数学	Easy	完成
455.分发饼干	贪心算法	Easy	完成
459.重复的子字符串	字符串	Easy	完成
461.汉明距离	位运算	Easy	完成
第12期			
463.岛屿的周长	哈希表	Easy	完成
475.供暖器	二分查找	Easy	完成
476.数字的补数	位运算	Easy	完成
482.密钥格式化		Easy	完成
485.最大连续1的个数	数组	Easy	完成
492.构造矩形		Easy	完成

Title	Tag	难度	完成情况
496.下一个更大元素 I	栈	Easy	完成
500.键盘行	哈希表	Easy	完成
501.二叉搜索树中的众数	树	Easy	完成
504.七进制数		Easy	完成
第13期			
506.相对名次		Easy	完成
507.完美数	数学	Easy	完成
509.斐波那契数	数组	Easy	完成
520.检测大写字母	字符串	Easy	完成
521.最长特殊序列 I	字符串	Easy	完成
530.二叉搜索树的最小绝对差	树	Easy	完成
532.数组中的K-diff数对	数组、双指针	Easy	完成
538.把二叉搜索树转换为累加树	树	Easy	完成
541.反转字符串 II	字符串	Easy	完成
543.二叉树的直径	树	Easy	完成
第14期			

Title	Tag	难度	完成情况
551.学生出勤记录 I	字符串	Easy	完成
557.反转字符串中的单词 III	字符串	Easy	完成
559.N叉树的最大深度	树、深度优先搜索、广度优先搜索	Easy	完成
561.数组拆分 I	数组	Easy	完成
563.二叉树的坡度	树	Easy	完成
566.重塑矩阵	数组	Easy	完成
572.另一个树的子树	树	Easy	完成
575.分糖果	哈希表	Easy	完成
581.最短无序连续子数组	数组	Easy	完成
589.N叉树的前序遍历	树	Easy	完成
第15期			
590.N叉树的后序遍历	树	Easy	完成
594.最长和谐子序列	哈希表	Easy	完成
595.大的国家	Mysql	Easy	完成
596.超过5名学生的课	Mysql	Easy	完成
598.范围求和 II	数学	Easy	完成

Title	Tag	难度	完成情况
599.两个列表的最小索引总和	哈希表	Easy	完成
605.种花问题	数组	Easy	完成
606.根据二叉树创建字符串	树、字符串	Easy	完成
617.合并二叉树	树	Easy	完成
620.有趣的电影	Mysql	Easy	完成
第16期			
627.交换工资	Mysql	Easy	完成
628.三个数的最大乘积	数组、数学	Easy	完成
633.平方数之和	数学	Easy	完成
637.二叉树的层平均值	树	Easy	完成
643.子数组最大平均数 I	数组	Easy	完成
645.错误的集合	哈希表、数学	Easy	完成
653.两数之和 IV - 输入 BST	树	Easy	完成
657.机器人能否返回原点	字符串	Easy	完成
661.图片平滑器	数组	Easy	完成
665.非递减数列	数组	Easy	完成

Title	Tag	难度	完成情况
第17期			
669.修剪二叉搜索树	树	Easy	完成
671.二叉树中第二小的节点	树	Easy	完成
674.最长连续递增序列	数组	Easy	完成
680.验证回文字符串 II	字符串	Easy	完成
682.棒球比赛	栈	Easy	完成
686.重复叠加字符串匹配	字符串	Easy	完成

0001-0100-Easy

- 0001-0100-Easy
 - 1.两数之和(3)
 - 7.整数反转(2)
 - 9.回文数(3)
 - 13.罗马数字转整数(2)
 - 14.最长公共前缀(6)
 - 20.有效的括号(3)
 - 21.合并两个有序链表(2)
 - 26.删除排序数组中的重复项(2)
 - 27.移除元素(3)
 - 28.实现strStr()(4)
 - 35.搜索插入位置(3)
 - 38.报数(2)
 - 53.最大子序和(5)
 - 58.最后一个单词的长度(2)
 - 66.加一(2)
 - 67.二进制求和(2)
 - 69.x的平方根 (5)
 - 70.爬楼梯(3)
 - 83.删除排序链表中的重复元素(3)
 - 88.合并两个有序数组(3)
 - 100.相同的树(2)

1.两数之和(3)

- 题目

给定一个整数数组 `nums` 和一个目标值 `target`，
请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。
你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样|

示例：

给定 `nums = [2, 7, 11, 15]`, `target = 9`

因为 `nums[0] + nums[1] = 2 + 7 = 9`
所以返回 `[0, 1]`

- 解答思路

No.	思路	时间复杂度	空间复杂度
01	暴力法: 2层循环遍历	$O(n^2)$	$O(1)$
02	两遍哈希遍历	$O(n)$	$O(n)$
03(最优)	一遍哈希遍历	$O(n)$	$O(n)$

```

# 暴力法: 2层循环遍历
func twoSum(nums []int, target int) []int {
    for i := 0; i < len(nums); i++ {
        for j := i + 1; j < len(nums); j++ {
            if nums[i]+nums[j] == target {
                return []int{i, j}
            }
        }
    }
    return []int{}
}

# 两遍哈希遍历
func twoSum(nums []int, target int) []int {
    m := make(map[int]int, len(nums))
    for k, v := range nums{
        m[v] = k
    }

    for i := 0; i < len(nums); i++{
        b := target - nums[i]
        if num, ok := m[b]; ok && num != i{
            return []int{i,m[b]}
        }
    }
    return []int{}
}

# 一遍哈希遍历
func twoSum(nums []int, target int) []int {
    m := make(map[int]int, len(nums))
    for i, b := range nums {
        if j, ok := m[target-b]; ok {
            return []int{j, i}
        }
        m[b] = i
    }
    return nil
}

```

7. 整数反转(2)

- 题目

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

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。

- 解答思路

No.	思路	时间复杂度	空间复杂度
01	使用符号标记，转成正数，循环得到%10的余数，再加上符号	$O(\log(x))$	$O(1)$
02(最优)	对x进行逐个%10取个位，一旦溢出，直接跳出循环	$O(\log(x))$	$O(1)$

```
// 使用符号标记，转成正数，循环得到%10的余数，再加上符号
func reverse(x int) int {
    flag := 1
    if x < 0 {
        flag = -1
        x = -1 * x
    }

    result := 0
    for x > 0 {
        temp := x % 10
        x = x / 10

        result = result*10 + temp
    }

    result = flag * result
    if result > math.MaxInt32 || result < math.MinInt32 {
        result = 0
    }
    return result
}

// 对x进行逐个%10取个位，一旦溢出，直接跳出循环
func reverse(x int) int {
    result := 0
    for x != 0 {
        temp := x % 10
        result = result*10 + temp
        if result > math.MaxInt32 || result < math.MinInt32 {
            return 0
        }
        x = x / 10
    }
    return result
}
```

9.回文数(3)

- 题目

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）。

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:

你能不能将整数转为字符串来解决这个问题吗?

- 解答思路

No.	思路	时间复杂度	空间复杂度
01(最优)	数学解法, 取出后半段数字进行翻转, 然后判断是否相等	$O(\log(x))$	$O(1)$
02	转成字符串, 依次判断	$O(\log(x))$	$O(\log(x))$
03	转成byte数组, 依次判断, 同2	$O(\log(x))$	$O(\log(x))$

```

// 数学解法，取出后半段数字进行翻转，然后判断是否相等
func isPalindrome(x int) bool {
    if x < 0 || (x%10 == 0 && x != 0) {
        return false
    }

    revertedNumber := 0
    for x > revertedNumber {
        temp := x % 10
        revertedNumber = revertedNumber*10 + temp
        x = x / 10
    }
    // for example:
    // x = 1221 => x = 12 revertedNumber = 12
    // x = 12321 => x = 12 revertedNumber = 123
    return x == revertedNumber || x == revertedNumber/10
}

// 转成字符串，依次判断
func isPalindrome(x int) bool {
    if x < 0 {
        return false
    }

    s := strconv.Itoa(x)
    for i, j := 0, len(s)-1; i < j; i, j = i+1, j-1 {
        if s[i] != s[j] {
            return false
        }
    }
    return true
}

// 转成byte数组，依次判断，同2
func isPalindrome(x int) bool {
    if x < 0 {
        return false
    }
    arrs := []byte(strconv.Itoa(x))
    Len := len(arrs)
    for i := 0; i < Len/2; i++ {
        if arrs[i] != arrs[Len-i-1] {
            return false
        }
    }
    return true
}

```

13. 罗马数字转整数(2)

- 题目

罗马数字包含以下七种字符： I, V, X, L, C, D 和 M。

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

例如， 罗马数字 2 写做 II ，即为两个并列的 1。12 写做 XII ，即为 X 通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 :

I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。

X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。

C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。输入确保在 1 到 3999 的范围内。

示例 1: 输入: "III" 输出: 3

示例 2: 输入: "IV" 输出: 4

示例 3: 输入: "IX" 输出: 9

示例 4: 输入: "LVIII" 输出: 58 解释: L = 50, V= 5, III :

示例 5: 输入: "MCMXCIV" 输出: 1994 解释: M = 1000, CM = 900,

- 解答思路

No.	思路	时间复杂度	空间复杂度
01	本质上其实就是全部累加，然后遇到特殊的就做判断。使用一个字段记录递增	O(n)	O(1)
02(最优)	从右到左遍历字符串，如果当前字符代表的值不小于其右边，就加上该值；否则就减去该值。	O(n)	O(1)

```

// 带标记位
func romanToInt(s string) int {
    m := map[byte]int{
        'I': 1,
        'V': 5,
        'X': 10,
        'L': 50,
        'C': 100,
        'D': 500,
        'M': 1000,
    }
    result := 0
    last := 0

    for i := len(s) - 1; i >= 0; i-- {
        current := m[s[i]]
        flag := 1
        if current < last {
            flag = -1
        }
        result = result + flag*current
        last = current
    }
    return result
}

// 不带标记位，小于则减去2倍数
func romanToInt(s string) int {
    m := map[byte]int{
        'I': 1,
        'V': 5,
        'X': 10,
        'L': 50,
        'C': 100,
        'D': 500,
        'M': 1000,
    }
    result := 0
    last := 0

    for i := len(s) - 1; i >= 0; i-- {
        current := m[s[i]]
        if current < last {
            result = result - current
        }else {
            result = result + current
        }
        last = current
    }
}

```

```
    }
    return result
}
```

14.最长公共前缀(6)

- 题目

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 。

- 解答思路

No.	思路	时间复杂度	空间复杂度
01	先找最短的一个字符串，依次比较最短字符串子串是否是其他字符串子串	$O(n^2)/O(n*m)$	$O(1)$
02	纵向扫描(暴力法):直接取第一个字符串作为最长公共前缀，将其每个字符遍历过一次	$O(n^2)/O(n*m)$	$O(1)$
03(最优)	排序后，然后计算第一个，和最后一个字符串的最长前缀	$O(n\log(n))$	$O(1)$
04	trie树	$O(n^2)$	$O(n^2)$
05	水平扫描法:比较前2个字符串得到最长前缀，然后跟第3个比较得到一个新的最长前缀，继续比较，直到最后	$O(n^2)/O(n*m)$	$O(1)$
06	分治法	$O(n^2)$	$O(1)$

```

// 先找最短的一个字符串，依次比较最短字符串子串是否是其他字符串子串
func longestCommonPrefix(strs []string) string {
    if len(strs) == 0{
        return ""
    }
    if len(strs) == 1{
        return strs[0]
    }

    short := strs[0]
    for _, s := range strs{
        if len(short) > len(s){
            short = s
        }
    }

    for i := range short{
        shortest := short[:i+1]
        for _,str := range strs{
            if strings.Index(str,shortest) != 0{
                return short[:i]
            }
        }
    }
    return short
}

// 暴力法：直接依次遍历
func longestCommonPrefix(strs []string) string {
    if len(strs) == 0 {
        return ""
    }
    if len(strs) == 1 {
        return strs[0]
    }

    length := 0

    for i := 0; i < len(strs[0]); i++ {
        char := strs[0][i]
        for j := 1; j < len(strs); j++ {
            if i >= len(strs[j]) || char != strs[j][i] {
                return strs[0][:length]
            }
        }
        length++
    }
    return strs[0][:length]
}

```

```

}

// 排序后，遍历比较第一个，和最后一个字符串
func longestCommonPrefix(strs []string) string {
    if len(strs) == 0{
        return ""
    }
    if len(strs) == 1{
        return strs[0]
    }

    sort.Strings(strs)
    first := strs[0]
    last := strs[len(strs)-1]
    i := 0
    length := len(first)
    if len(last) < length{
        length = len(last)
    }
    for i < length{
        if first[i] != last[i]{
            return first[:i]
        }
        i++
    }

    return first[:i]
}

// trie树
var trie [][]int
var index int

func longestCommonPrefix(strs []string) string {
    if len(strs) == 0 {
        return ""
    }
    if len(strs) == 1 {
        return strs[0]
    }

    trie = make([][]int, 2000)
    for k := range trie {
        value := make([]int, 26)
        trie[k] = value
    }
    insert(strs[0])
}

```

```

minValue := math.MaxInt32
for i := 1; i < len(strs); i++ {
    retValue := insert(strs[i])
    if minValue > retValue {
        minValue = retValue
    }
}
return strs[0][:minValue]
}

func insert(str string) int {
    p := 0
    count := 0
    for i := 0; i < len(str); i++ {
        ch := str[i] - 'a'
        // fmt.Println(string(str[i]), p, ch, trie[p][ch])
        if value := trie[p][ch]; value == 0 {
            index++
            trie[p][ch] = index
        } else {
            count++
        }
        p = trie[p][ch]
    }
    return count
}

// 水平扫描法:比较前2个字符串得到最长前缀, 然后跟第3个比较得到一个新的
func longestCommonPrefix(strs []string) string {
    if len(strs) == 0 {
        return ""
    }
    if len(strs) == 1 {
        return strs[0]
    }

    commonStr := common(strs[0], strs[1])
    if commonStr == "" {
        return ""
    }
    for i := 2; i < len(strs); i++ {
        if commonStr == "" {
            return ""
        }
        commonStr = common(commonStr, strs[i])
    }
    return commonStr
}

```

```

func common(str1, str2 string) string {
    length := 0
    for i := 0; i < len(str1); i++ {
        char := str1[i]
        if i >= len(str2) || char != str2[i] {
            return str1[:length]
        }
        length++
    }
    return str1[:length]
}

// 分治法
func longestCommonPrefix(strs []string) string {
    if len(strs) == 0 {
        return ""
    }
    if len(strs) == 1 {
        return strs[0]
    }

    return commonPrefix(strs, 0, len(strs)-1)
}

func commonPrefix(strs []string, left, right int) string {
    if left == right {
        return strs[left]
    }

    middle := (left + right) / 2
    leftStr := commonPrefix(strs, left, middle)
    rightStr := commonPrefix(strs, middle+1, right)
    return commonPrefixWord(leftStr, rightStr)
}

func commonPrefixWord(leftStr, rightStr string) string {
    if len(leftStr) > len(rightStr) {
        leftStr = leftStr[:len(rightStr)]
    }

    if len(leftStr) < 1 {
        return leftStr
    }

    for i := 0; i < len(leftStr); i++ {
        if leftStr[i] != rightStr[i] {
            return leftStr[:i]
        }
    }
}

```

```

        }
    }
    return leftStr
}

```

20.有效的括号(3)

- 题目

给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串，判断字符串是否为有效字符串需满足：

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

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	使用栈结构实现栈	O(n)	O(n)
02	借助数组实现栈	O(n)	O(n)
03	借助数组实现栈，使用数字表示来匹配	O(n)	O(n)

```

// 使用栈结构实现
func isValid(s string) bool {
    st := new(stack)
    for _, char := range s {
        switch char {
        case '(', '[', '{':
            st.push(char)
        case ')', ')', '}':
            ret, ok := st.pop()
            if !ok || ret != match[char] {
                return false
            }
        }
    }

    if len(*st) > 0 {
        return false
    }
    return true
}

var match = map[rune]rune{
    ')': '(',
    ']': '[',
    '}': '{',
}

type stack []rune

func (s *stack) push(b rune) {
    *s = append(*s, b)
}
func (s *stack) pop() (rune, bool) {
    if len(*s) > 0 {
        res := (*s)[len(*s)-1]
        *s = (*s)[:len(*s)-1]
        return res, true
    }
    return 0, false
}

// 借助数组实现栈
func isValid(s string) bool {
    if s == "" {
        return true
    }

    stack := make([]rune, len(s))

```

```

length := 0
var match = map[rune]rune{
    ')' : '(',
    ']' : '[',
    '}' : '{',
}

for _, char := range s {
    switch char {
    case '(', '[', '{':
        stack[length] = char
        length++
    case ')', ']', '}':
        if length == 0 {
            return false
        }
        if stack[length-1] != match[char]{
            return false
        } else {
            length--
        }
    }
}
return length == 0
}

// 借助数组实现栈，使用数字表示来匹配
func isValid(s string) bool {
    if s == "" {
        return true
    }

    stack := make([]int, len(s))
    length := 0
    var match = map[rune]int{
        ')': 1,
        '(': -1,
        ']': 2,
        '[': -2,
        '}': 3,
        '{': -3,
    }

    for _, char := range s {
        switch char {
        case '(', '[', '{':
            stack[length] = match[char]
            length++
        }
    }
}

```

```

        case ')', '[', '}':
            if length == 0 {
                return false
            }
            if stack[length-1]+match[char] != 0 {
                return false
            } else {
                length--
            }
        }
    }
    return length == 0
}

```

21. 合并两个有序链表(2)

- 题目

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

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	迭代遍历	O(n)	O(1)
02	递归实现	O(n)	O(n)

```

// 迭代遍历
func mergeTwoLists(l1 *ListNode, l2 *ListNode) *ListNode {
    if l1 == nil {
        return l2
    }
    if l2 == nil {
        return l1
    }

    var head, node *ListNode
    if l1.Val < l2.Val {
        head = l1
        node = l1
        l1 = l1.Next
    } else {
        head = l2
        node = l2
        l2 = l2.Next
    }

    for l1 != nil && l2 != nil {
        if l1.Val < l2.Val {
            node.Next = l1
            l1 = l1.Next
        } else {
            node.Next = l2
            l2 = l2.Next
        }
        node = node.Next
    }
    if l1 != nil {
        node.Next = l1
    }
    if l2 != nil {
        node.Next = l2
    }
    return head
}

// 递归遍历
func mergeTwoLists(l1 *ListNode, l2 *ListNode) *ListNode {
    if l1 == nil {
        return l2
    }
    if l2 == nil {
        return l1
    }
}

```

```

if l1.Val < l2.Val{
    l1.Next = mergeTwoLists(l1.Next, l2)
    return l1
} else {
    l2.Next = mergeTwoLists(l1, l2.Next)
    return l2
}
}

```

26. 删除排序数组中的重复项(2)

- 题目

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。
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。

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]);
}
```

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	双指针法	O(n)	O(1)
02(最优)	计数法	O(n)	O(1)

```
// 双指针法
func removeDuplicates(nums []int) int {
    i, j, length := 0, 1, len(nums)
    for ; j < length; j++{
        if nums[i] == nums[j]{
            continue
        }
        i++
        nums[i] = nums[j]
    }
    return i+1
}

// 计数法
func removeDuplicates(nums []int) int {
    count := 1
    for i := 0; i < len(nums)-1; i++ {
        if nums[i] != nums[i+1] {
            nums[count] = nums[i+1]
            count++
        }
    }
    return count
}
```

27. 移除元素(3)

- 题目

给定一个数组 `nums` 和一个值 `val`, 你需要原地移除所有数值等于 `val` 的元素。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 $O(1)$ 额外空间的情况下完成操作。元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

示例 1:

给定 `nums = [3, 2, 2, 3]`, `val = 3`,
函数应该返回新的长度 2, 并且 `nums` 中的前两个元素均为 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, 1, 2, 2, 3, 0, 4, 2]`, `val = 2`,
函数应该返回新的长度 5, 并且 `nums` 中的前五个元素为 0, 1, 3, 0, 4。
注意这五个元素可为任意顺序。
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。

```
// nums 是以“引用”方式传递的。也就是说, 不对实参作任何拷贝
int len = removeElement(nums, val);
// 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]);
}
```

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	双指针, 数字前移	$O(n)$	$O(1)$
02	双指针, 出现重复最后数字前移	$O(n)$	$O(1)$
03	首位指针法	$O(n)$	$O(1)$

```

// 双指针，数字前移
func removeElement(nums []int, val int) int {
    i := 0
    for j := 0; j < len(nums); j++{
        if nums[j] != val{
            nums[i] = nums[j]
            i++
        }
    }
    return i
}

// 双指针，出现重复最后数字前移
func removeElement(nums []int, val int) int {
    i := 0
    n := len(nums)
    for i < n{
        if nums[i] == val{
            nums[i] = nums[n-1]
            n--
        }else {
            i++
        }
    }
    return n
}

// 首位指针法
func removeElement(nums []int, val int) int {
    i, j := 0, len(nums)-1
    for {
        // 从左向右找到等于 val 的位置
        for i < len(nums) && nums[i] != val {
            i++
        }
        // 从右向左找到不等于 val 的位置
        for j >= 0 && nums[j] == val {
            j--
        }
        if i >= j {
            break
        }
        // fmt.Println(i,j)
        nums[i], nums[j] = nums[j], nums[i]
    }
    return i
}

```

28. 实现strStr()(4)

- 题目

实现 strStr() 函数。

给定一个 haystack 字符串和一个 needle 字符串，

在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。

如果不存在，则返回 -1。

示例 1：

输入: haystack = "hello", needle = "ll"

输出: 2

示例 2：

输入: haystack = "aaaaa", needle = "bba"

输出: -1

说明：

当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。

对于本题而言，当 needle 是空字符串时我们应当返回 0。

这与C语言的 strstr() 以及 Java的 indexOf() 定义相符。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	Sunday算法	O(n)	O(1)
02	直接匹配	O(n)	O(1)
03	系统函数	O(n)	O(1)
04	kmp算法	O(n)	O(n)

```

// Sunday算法
func strStr(haystack string, needle string) int {
    if needle == ""{
        return 0
    }
    if len(needle) > len(haystack){
        return -1
    }
    // 计算模式串needle的偏移量
    m := make(map[int32]int)
    for k,v := range needle{
        m[v] = len(needle)-k
    }

    index := 0
    for index+len(needle) <= len(haystack){
        // 匹配字符串
        str := haystack[index:index+len(needle)]
        if str == needle{
            return index
        }else {
            if index + len(needle) >= len(haystack){
                return -1
            }
            // 后一位字符串
            next := haystack[index+len(needle)]
            if nextStep,ok := m[int32(next)];ok{
                index = index+nextStep
            }else {
                index = index+len(needle)+1
            }
        }
    }
    if index + len(needle) >= len(haystack){
        return -1
    }else {
        return index
    }
}

// 
func strStr(haystack string, needle string) int {
    hlen, nlen := len(haystack), len(needle)
    for i := 0; i <= hlen-nlen; i++ {
        if haystack[i:i+nlen] == needle {
            return i
        }
    }
}

```

```

        return -1
    }

    //

func strStr(haystack string, needle string) int {
    return strings.Index(haystack, needle)
}

//

func strStr(haystack string, needle string) int {
    if len(needle) == 0 {
        return 0
    }

    next := getNext(needle)

    i := 0
    j := 0
    for i < len(haystack) && j < len(needle) {
        if j == -1 || haystack[i] == needle[j] {
            i++
            j++
        } else {
            j = next[j]
        }
    }

    if j == len(needle) {
        return i - j
    }
    return -1
}

// 求next数组
func getNext(str string) []int {
    var next = make([]int, len(str))
    next[0] = -1

    i := 0
    j := -1

    for i < len(str)-1 {
        if j == -1 || str[i] == str[j] {
            i++
            j++
            next[i] = j
        } else {
            j = next[j]
        }
    }
}

```

```

        }
    }
    return next
}

```

35. 搜索插入位置(3)

- 题目

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。
如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

你可以假设数组中无重复元素。

示例 1：输入：[1, 3, 5, 6]，5 输出：2

示例 2：输入：[1, 3, 5, 6]，2 输出：1

示例 3：输入：[1, 3, 5, 6]，7 输出：4

示例 4：输入：[1, 3, 5, 6]，0 输出：0

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	二分查找	$O(\log(n))$	$O(1)$
02	顺序查找	$O(n)$	$O(1)$
03	顺序查找	$O(n)$	$O(1)$

```

// 二分查找
func searchInsert(nums []int, target int) int {
    low, high := 0, len(nums)-1
    for low <= high {
        mid := (low + high) / 2
        switch {
        case nums[mid] < target:
            low = mid + 1
        case nums[mid] > target:
            high = mid - 1
        default:
            return mid
        }
    }
    return low
}

// 顺序查找
func searchInsert(nums []int, target int) int {
    i := 0
    for i < len(nums) && nums[i] < target {
        if nums[i] == target {
            return i
        }
        i++
    }
    return i
}

// 顺序查找
func searchInsert(nums []int, target int) int {
    for i := 0; i < len(nums); i++ {
        if nums[i] >= target {
            return i
        }
    }
    return len(nums)
}

```

38.报数(2)

- 题目

报数序列是一个整数序列，按照其中的整数的顺序进行报数，得到下一个数。其：

```

1.      1
2.      11
3.      21
4.      1211
5.      111221

```

1 被读作 "one 1" ("一个一")，即 11。

11 被读作 "two 1s" ("两个一")，即 21。

21 被读作 "one 2", "one 1" ("一个二", "一个一")，即 1211。

给定一个正整数 n ($1 \leq n \leq 30$)，输出报数序列的第 n 项。

注意：整数顺序将表示为一个字符串。

示例 1: 输入: 1 输出: "1"

示例 2: 输入: 4 输出: "1211"

- 解题思路

No.	思路	时间复杂度	空间复杂度
01 (最优)	递推+双指针计数	$O(n^2)$	$O(1)$
02	递归+双指针计数	$O(n^2)$	$O(n)$

```

// 递推+双指针计数
func countAndSay(n int) string {
    strs := []byte{'1'}
    for i := 1; i < n; i++ {
        strs = say(strs)
    }
    return string(strs)
}

func say(strs []byte) []byte {
    result := make([]byte, 0, len(strs)*2)

    i, j := 0, 1
    for i < len(strs) {
        for j < len(strs) && strs[i] == strs[j] {
            j++
        }
        // 几个几
        result = append(result, byte(j-i+'0'))
        result = append(result, strs[i])
        i = j
    }
    return result
}

// 递归+双指针计数
func countAndSay(n int) string {
    if n == 1 {
        return "1"
    }
    strs := countAndSay(n - 1)

    result := make([]byte, 0, len(strs)*2)

    i, j := 0, 1
    for i < len(strs) {
        for j < len(strs) && strs[i] == strs[j] {
            j++
        }
        // 几个几
        result = append(result, byte(j-i+'0'))
        result = append(result, strs[i])
        i = j
    }
    return string(result)
}

```

53. 最大子序和(5)

- 题目

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）。

示例：

输入： [-2, 1, -3, 4, -1, 2, 1, -5, 4]，输出： 6 解释： 连续子数组 [4, -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。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	贪心法	$O(n)$	$O(1)$
02	暴力法	$O(n^2)$	$O(1)$
03	动态规划	$O(n)$	$O(n)$
04	动态规划	$O(n)$	$O(1)$
05	分治	$O(n \log(n))$	$O(\log(n))$

```

// 贪心法
func maxSubArray(nums []int) int {
    result := nums[0]
    sum := 0
    for i := 0; i < len(nums); i++ {
        if sum > 0 {
            sum += nums[i]
        } else {
            sum = nums[i]
        }
        if sum > result {
            result = sum
        }
    }
    return result
}

// 暴力法
func maxSubArray(nums []int) int {
    result := math.MinInt32

    for i := 0; i < len(nums); i++ {
        sum := 0
        for j := i; j < len(nums); j++ {
            sum += nums[j]
            if sum > result {
                result = sum
            }
        }
    }
    return result
}

//
func maxSubArray(nums []int) int {
    dp := make([]int, len(nums))
    dp[0] = nums[0]
    result := nums[0]

    for i := 1; i < len(nums); i++ {
        if dp[i-1]+nums[i] > nums[i] {
            dp[i] = dp[i-1] + nums[i]
        } else {
            dp[i] = nums[i]
        }

        if dp[i] > result {
            result = dp[i]
        }
    }
}

```

```

        }
    }
    return result
}

// 动态规划
func maxSubArray(nums []int) int {
    dp := nums[0]
    result := dp

    for i := 1; i < len(nums); i++ {
        if dp+nums[i] > nums[i] {
            dp = dp + nums[i]
        } else {
            dp = nums[i]
        }

        if dp > result {
            result = dp
        }
    }
    return result
}

// 分治法
func maxSubArray(nums []int) int {
    result := maxSubArr(nums, 0, len(nums)-1)
    return result
}

func maxSubArr(nums []int, left, right int) int {
    if left == right {
        return nums[left]
    }

    mid := (left + right) / 2
    leftSum := maxSubArr(nums, left, mid)           // 最大子序
    rightSum := maxSubArr(nums, mid+1, right)       // 最大子序
    midSum := findMaxArr(nums, left, mid, right)   // 跨中心
    result := max(leftSum, rightSum)
    result = max(result, midSum)
    return result
}

func findMaxArr(nums []int, left, mid, right int) int {
    leftSum := math.MinInt32
    sum := 0
    // 从右到左
}

```

```

    for i := mid; i >= left; i-- {
        sum += nums[i]
        leftSum = max(leftSum, sum)
    }

    rightSum := math.MinInt32
    sum = 0
    // 从左到右
    for i := mid + 1; i <= right; i++ {
        sum += nums[i]
        rightSum = max(rightSum, sum)
    }
    return leftSum + rightSum
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

```

58.最后一个单词的长度(2)

- 题目

给定一个仅包含大小写字母和空格 ' ' 的字符串，返回其最后一个单词的长度。
如果不存在最后一个单词，请返回 0。
说明：一个单词是指由字母组成，但不包含任何空格的字符串。

示例：输入： "Hello World" 输出： 5

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	调用系统函数，切割为数组取最后一个值	O(n)	O(1)
02	遍历统计	O(n)	O(1)

```

// 调用系统函数，切割为数组取最后一个值
func lengthOfLastWord(s string) int {
    arr := strings.Split(strings.TrimSpace(s), " ")
    return len(arr[len(arr)-1])
}

// 遍历统计
func lengthOfLastWord(s string) int {
    length := len(s)
    if length == 0 {
        return 0
    }

    result := 0
    for i := length - 1; i >= 0; i-- {
        if s[i] == ' ' {
            if result > 0 {
                return result
            }
            continue
        }
        result++
    }
    return result
}

```

66. 加一(2)

- 题目

给定一个由整数组成的非空数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位， 数组中每个元素只存储单个数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

示例 1：输入：[1,2,3] 输出：[1,2,4] 解释：输入数组表示数字 123。

示例 2：输入：[4,3,2,1] 输出：[4,3,2,2] 解释：输入数组表示数字 4

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	直接模拟	O(n)	O(1)
02(最优)	直接模拟	O(n)	O(1)

```
// 模拟进位
func plusOne(digits []int) []int {
    length := len(digits)
    if length == 0 {
        return []int{1}
    }

    digits[length-1]++
    for i := length - 1; i > 0; i-- {
        if digits[i] < 10 {
            break
        }
        digits[i] = digits[i] - 10
        digits[i-1]++
    }

    if digits[0] > 9 {
        digits[0] = digits[0] - 10
        digits = append([]int{1}, digits...)
    }

    return digits
}

// 模拟进位
func plusOne(digits []int) []int {
    for i := len(digits) - 1; i >= 0; i-- {
        if digits[i] < 9 {
            digits[i]++
            return digits
        } else {
            digits[i] = 0
        }
    }
    return append([]int{1}, digits...)
}
```

67.二进制求和(2)

- 题目

给定两个二进制字符串，返回他们的和（用二进制表示）。
输入为非空字符串且只包含数字 1 和 0。

示例 1: 输入: a = "11", b = "1" 输出: "100"
示例 2: 输入: a = "1010", b = "1011" 输出: "10101"

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	转换成数组模拟	$O(n)$	$O(n)$
02(最优)	直接模拟	$O(n)$	$O(1)$

```

// 转换成数组模拟
func addBinary(a string, b string) string {
    if len(a) < len(b) {
        a, b = b, a
    }
    length := len(a)

    A := transToInt(a, length)
    B := transToInt(b, length)

    return makeString(add(A, B))
}

func transToInt(s string, length int) []int {
    result := make([]int, length)
    ls := len(s)
    for i, b := range s {
        result[length-ls+i] = int(b - '0')
    }
    return result
}

func add(a, b []int) []int {
    length := len(a) + 1
    result := make([]int, length)
    for i := length - 1; i >= 1; i-- {
        temp := result[i] + a[i-1] + b[i-1]
        result[i] = temp % 2
        result[i-1] = temp / 2
    }
    i := 0
    for i < length-1 && result[i] == 0 {
        i++
    }
    return result[i:]
}

func makeString(nums []int) string {
    bytes := make([]byte, len(nums))
    for i := range bytes {
        bytes[i] = byte(nums[i]) + '0'
    }
    return string(bytes)
}

// 直接模拟
func addBinary(a string, b string) string {
    i := len(a) - 1

```

```

j := len(b) - 1
result := ""
flag := 0
current := 0

for i >= 0 || j >= 0 {
    intA, intB := 0, 0
    if i >= 0 {
        intA = int(a[i] - '0')
    }
    if j >= 0 {
        intB = int(b[j] - '0')
    }
    current = intA + intB + flag
    flag = 0
    if current >= 2 {
        flag = 1
        current = current - 2
    }
    cur := strconv.Itoa(current)
    result = cur + result
    i--
    j--
}
if flag == 1 {
    result = "1" + result
}
return result
}

```

69.x的平方根 (5)

- 题目

实现 `int sqrt(int x)` 函数。
 计算并返回 x 的平方根，其中 x 是非负整数。
 由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例 1：

输入： 4

输出： 2

示例 2：

输入： 8

输出： 2

说明： 8 的平方根是 $2.82842\dots$,

由于返回类型是整数，小数部分将被舍去。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	系统函数	$O(\log(n))$	$O(1)$
02	系统函数	$O(\log(n))$	$O(1)$
03(最优)	牛顿迭代法	$O(\log(n))$	$O(1)$
04	二分查找法	$O(\log(n))$	$O(1)$
05	暴力法:遍历	$O(n)$	$O(1)$

```

// 系统函数
func mySqrt(x int) int {
    result := int(math.Sqrt(float64(x)))
    return result
}

// 系统函数
func mySqrt(x int) int {
    result := math.Floor(math.Sqrt(float64(x)))
    return int(result)
}

// 牛顿迭代法
func mySqrt(x int) int {
    result := x
    for result*result > x {
        result = (result + x/result) / 2
    }
    return result
}

// 二分查找法
func mySqrt(x int) int {
    left := 1
    right := x
    for left <= right {
        mid := (left + right) / 2
        if mid == x/mid {
            return mid
        } else if mid < x/mid {
            left = mid + 1
        } else {
            right = mid - 1
        }
    }
    if left * left <= x{
        return left
    }else {
        return left-1
    }
}

// 暴力法:遍历
func mySqrt(x int) int {
    result := 0
    for i := 1; i <= x/i; i++ {
        if i*i == x {
            return i
        }
    }
}

```

```

    }
    result = i
}
return result
}

```

70.爬楼梯(3)

- 题目

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

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(n)$
02	动态规划	$O(n)$	$O(n)$
03(最优)	斐波那契	$O(n)$	$O(1)$

```

// 递归
func climbStart(i, n int) int {
    if i > n {
        return 0
    }
    if i == n {
        return 1
    }
    if arr[i] > 0 {
        return arr[i]
    }

    arr[i] = climbStart(i+1, n) + climbStart(i+2, n)
    return arr[i]
}

// 动态规划
func climbStairs(n int) int {
    if n == 1 {
        return 1
    }
    if n == 2 {
        return 2
    }
    dp := make([]int, n+1)
    dp[1] = 1
    dp[2] = 2
    for i := 3; i <= n; i++ {
        dp[i] = dp[i-1] + dp[i-2]
    }
    return dp[n]
}

// 斐波那契
func climbStairs(n int) int {
    if n == 1 {
        return 1
    }
    first := 1
    second := 2
    for i := 3; i <= n; i++ {
        third := first + second
        first = second
        second = third
    }
    return second
}

```

83. 删除排序链表中的重复元素(3)

- 题目

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

示例 1：

输入： 1->1->2

输出： 1->2

示例 2：

输入： 1->1->2->3->3

输出： 1->2->3

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	直接法	$O(n)$	$O(1)$
02	递归法	$O(n)$	$O(1)$
03	双指针法	$O(n)$	$O(1)$

```

// 直接法
func deleteDuplicates(head *ListNode) *ListNode {
    if head == nil {
        return nil
    }
    temp := head
    for temp.Next != nil {
        if temp.Val == temp.Next.Val {
            temp.Next = temp.Next.Next
        } else {
            temp = temp.Next
        }
    }
    return head
}

// 递归法
func deleteDuplicates(head *ListNode) *ListNode {
    if head == nil || head.Next == nil{
        return head
    }
    head.Next = deleteDuplicates(head.Next)
    if head.Val == head.Next.Val{
        head = head.Next
    }
    return head
}

// 双指针法
func deleteDuplicates(head *ListNode) *ListNode {
    if head == nil || head.Next == nil{
        return head
    }
    p := head
    q := head.Next
    for p.Next != nil{
        if p.Val == q.Val{
            if q.Next == nil{
                p.Next = nil
            }else {
                p.Next = q.Next
                q = q.Next
            }
        }else {
            p = p.Next
            q = q.Next
        }
    }
}

```

```

    return head
}

```

88. 合并两个有序数组(3)

- 题目

给定两个有序整数数组 `nums1` 和 `nums2`, 将 `nums2` 合并到 `nums1` 中, 使说明:

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

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]`

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	合并后排序	$O(n \log(n))$	$O(1)$
02(最优)	双指针法	$O(n)$	$O(1)$
03	拷贝后插入	$O(n)$	$O(n)$

```

// 合并后排序
func merge(n int, m int, nums1 []int, n int) {
    nums1 = nums1[:m]
    nums1 = append(nums1, nums2[:n]...)
    sort.Ints(nums1)
}

// 双指针法
func merge(nums1 []int, m int, nums2 []int, n int) {
    for m > 0 && n > 0 {
        if nums1[m-1] < nums2[n-1] {
            nums1[m+n-1] = nums2[n-1]
            n--
        } else {
            nums1[m+n-1] = nums1[m-1]
            m--
        }
    }
    if m == 0 && n > 0 {
        for n > 0 {
            nums1[n-1] = nums2[n-1]
            n--
        }
    }
}

// 拷贝后插入
func merge(nums1 []int, m int, nums2 []int, n int) {
    temp := make([]int, m)
    copy(temp, nums1)

    if n == 0 {
        return
    }
    first, second := 0, 0
    for i := 0; i < len(nums1); i++ {
        if second >= n {
            nums1[i] = temp[first]
            first++
            continue
        }
        if first >= m {
            nums1[i] = nums2[second]
            second++
            continue
        }
        if temp[first] < nums2[second] {
            nums1[i] = temp[first]
        }
    }
}

```

```

        first++
    } else {
        nums1[i] = nums2[second]
        second++
    }
}
}

```

100. 相同的树(2)

- 题目

给定两个二叉树，编写一个函数来检验它们是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

示例 1:

输入: 1 1
 / \ / \\\
 2 3 2 3
 [1, 2, 3], [1, 2, 3]

输出: true

示例 2:

输入: 1 1
 / \\\
 2 2
 [1, 2], [1, null, 2]

输出: false

示例 3:

输入: 1 1
 / \ / \\\
 2 1 1 2
 [1, 2, 1], [1, 1, 2]

输出: false

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归(深度优先)	O(n)	O(log(n))
02	层序遍历(宽度优先)	O(n)	O(log(n))

```

// 递归(深度优先)
func isSameTree(p *TreeNode, q *TreeNode) bool {
    if p == nil && q == nil {
        return true
    }

    if p == nil || q == nil {
        return false
    }

    return p.Val == q.Val && isSameTree(p.Left, q.Left) &&
           isSameTree(p.Right, q.Right)
}

// 层序遍历(宽度优先)
func isSameTree(p *TreeNode, q *TreeNode) bool {
    if p == nil && q == nil {
        return true
    }

    if p == nil || q == nil {
        return false
    }

    var queueP, queueQ []*TreeNode
    if p != nil {
        queueP = append(queueP, p)
        queueQ = append(queueQ, q)
    }

    for len(queueP) > 0 && len(queueQ) > 0 {
        tempP := queueP[0]
        queueP = queueP[1:]

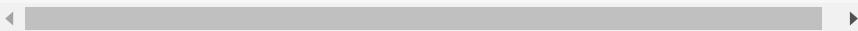
        tempQ := queueQ[0]
        queueQ = queueQ[1:]

        if tempP.Val != tempQ.Val {
            return false
        }

        if (tempP.Left != nil && tempQ.Left == nil) ||
           (tempP.Left == nil && tempQ.Left != nil) {
            return false
        }
        if tempP.Left != nil {
            queueP = append(queueP, tempP.Left)
            queueQ = append(queueQ, tempQ.Left)
        }
    }
}

```

```
if (tempP.Right != nil && tempQ.Right == nil) ||
    (tempP.Right == nil && tempQ.Right != nil) {
    return false
}
if tempP.Right != nil {
    queueP = append(queueP, tempP.Right)
    queueQ = append(queueQ, tempQ.Right)
}
return true
}
```



0101-0200-Easy

- 0101-0200-Easy
 - 101. 对称二叉树(2)
 - 104.二叉树的最大深度(2)
 - 107.二叉树的层次遍历II(2)
 - 108.将有序数组转换为二叉搜索树(2)
 - 110.平衡二叉树(1)
 - 111.二叉树的最小深度(2)
 - 112.路径总和(2)
 - 118.杨辉三角(2)
 - 119.杨辉三角II(3)
 - 121.买卖股票的最佳时机(3)
 - 122.买卖股票的最佳时机II(2)
 - 125.验证回文串(2)
 - 136.只出现一次的数字(4)
 - 141.环形链表(2)
 - 155.最小栈(2)
 - 160.相交链表(4)
 - 167.两数之和 II - 输入有序数组(4)
 - 168.Excel表列名称(2)
 - 169.多数元素(5)
 - 171.Excel表列序号(1)
 - 172.阶乘后的零(1)
 - 189.旋转数组(4)
 - 190.颠倒二进制位(3)
 - 191.位1的个数(2)
 - 198.打家劫舍(4)

101. 对称二叉树(2)

- 题目

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 `[1, 2, 2, 3, 4, 4, 3]` 是对称的。

```

      1
     / \
    2   2
   / \ / \
  3  4 4  3
  
```

但是下面这个 `[1, 2, 2, null, 3, null, 3]` 则不是镜像对称的：

```

      1
     / \
    2   2
   \   \
  3   3
  
```

说明：

如果你可以运用递归和迭代两种方法解决这个问题，会很加分。

- 解答思路

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	$O(n)$	$O(n)$
02	迭代	$O(n)$	$O(n)$

```

// 递归
func isSymmetric(root *TreeNode) bool {
    if root == nil {
        return true
    }
    return recur(root.Left, root.Right)
}

func recur(left, right *TreeNode) bool {
    if left == nil && right == nil {
        return true
    }
    if left == nil || right == nil {
        return false
    }

    return left.Val == right.Val &&
        recur(left.Left, right.Right) &&
        recur(left.Right, right.Left)
}

// 迭代
func isSymmetric(root *TreeNode) bool {
    leftQ := make([]*TreeNode, 0)
    rightQ := make([]*TreeNode, 0)
    leftQ = append(leftQ, root)
    rightQ = append(rightQ, root)

    for len(leftQ) != 0 && len(rightQ) != 0 {
        leftCur, rightCur := leftQ[0], rightQ[0]
        leftQ, rightQ = leftQ[1:], rightQ[1:]

        if leftCur == nil && rightCur == nil {
            continue
        } else if leftCur != nil && rightCur != nil && leftCur.Val == rightCur.Val {
            leftQ = append(leftQ, leftCur.Left, leftCur.Right)
            rightQ = append(rightQ, rightCur.Right, rightCur.Left)
        } else {
            return false
        }
    }

    if len(leftQ) == 0 && len(rightQ) == 0 {
        return true
    } else {
        return false
    }
}

```



104. 二叉树的最大深度(2)

- 题目

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	$O(n)$	$O(\log(n))$
02	迭代	$O(n)$	$O(n)$

- 解答思路

```

// 递归
func maxDepth(root *TreeNode) int {
    if root == nil {
        return 0
    }
    left := maxDepth(root.Left)
    right := maxDepth(root.Right)

    return max(left, right) + 1
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

// 迭代
func maxDepth(root *TreeNode) int {
    if root == nil {
        return 0
    }
    queue := make([]*TreeNode, 0)
    queue = append(queue, root)
    depth := 0

    for len(queue) > 0{
        length := len(queue)

        for i := 0; i < length; i++{
            node := queue[0]
            queue = queue[1:]
            if node.Left != nil{
                queue = append(queue, node.Left)
            }
            if node.Right != nil{
                queue = append(queue, node.Right)
            }
        }
        depth++
    }
    return depth
}

```

107.二叉树的层次遍历II(2)

- 题目

给定一个二叉树，返回其节点值自底向上的层次遍历。 (即按从叶子节点所在层)

例如：

给定二叉树 [3, 9, 20, null, null, 15, 7],



返回其自底向上的层次遍历为：

```
[
  [15, 7],
  [9, 20],
  [3]
]
```



- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	$O(n)$	$O(n)$
02	迭代	$O(n)$	$O(n)$

```

// 迭代
func levelOrderBottom(root *TreeNode) [][]int {
    if root == nil {
        return nil
    }
    queue := make([]*TreeNode, 0)
    out := make([][]int, 0)
    queue = append(queue, root)

    for len(queue) != 0 {
        l := len(queue)
        arr := make([]int, 0)
        for i := 0; i < l; i++ {
            pop := queue[i]
            arr = append(arr, pop.Val)
            if pop.Left != nil {
                queue = append(queue, pop.Left)
            }
            if pop.Right != nil {
                queue = append(queue, pop.Right)
            }
        }
        out = append(out, arr)
        queue = queue[l:]
    }

    out2 := make([][]int, len(out))
    for i := 0; i < len(out); i++ {
        out2[len(out)-1-i] = out[i]
    }

    return out2
}

// 递归
func levelOrderBottom(root *TreeNode) [][]int {
    result := make([][]int, 0)
    level := 0
    if root == nil {
        return result
    }

    orderBottom(root, &result, level)

    left, right := 0, len(result)-1
    for left < right {
        result[left], result[right] = result[right], result
        left++
    }
}

```

```

        right--
    }
    return result
}

func orderBottom(root *TreeNode, result *[][]int, level int
    if root == nil {
        return
    }
    if len(*result) > level {
        fmt.Println(level, result, root.Val)
        (*result)[level] = append((*result)[level], root.Val)
    } else {
        *result = append(*result, []int{root.Val})
    }
    orderBottom(root.Left, result, level+1)
    orderBottom(root.Right, result, level+1)
}

```

108. 将有序数组转换为二叉搜索树(2)

- 题目

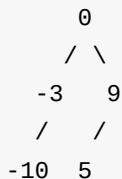
将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差

示例：

给定有序数组：[-10, -3, 0, 5, 9]，

一个可能的答案是：[0, -3, 9, -10, null, 5]，它可以表示下面这个高度平衡二



- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	$O(n)$	$O(\log(n))$
02	迭代	$O(n)$	$O(n)$

```

// 递归
func sortedArrayToBST(nums []int) *TreeNode {
    if len(nums) == 0 {
        return nil
    }

    mid := len(nums) / 2

    return &TreeNode{
        Val:   nums[mid],
        Left:  sortedArrayToBST(nums[:mid]),
        Right: sortedArrayToBST(nums[mid+1:]),
    }
}

// 迭代
type MyTreeNode struct {
    root  *TreeNode
    start int
    end   int
}

func sortedArrayToBST(nums []int) *TreeNode {
    if len(nums) == 0 {
        return nil
    }

    queue := make([]MyTreeNode, 0)
    root := &TreeNode{Val: 0}
    queue = append(queue, MyTreeNode{root, 0, len(nums)})
    for len(queue) > 0 {
        myRoot := queue[0]
        queue = queue[1:]
        start := myRoot.start
        end := myRoot.end
        mid := (start + end) / 2
        curRoot := myRoot.root
        curRoot.Val = nums[mid]
        if start < mid {
            curRoot.Left = &TreeNode{Val: 0}
            queue = append(queue, MyTreeNode{curRoot.Left,
        })
        if mid+1 < end {
            curRoot.Right = &TreeNode{Val: 0}
            queue = append(queue, MyTreeNode{curRoot.Right,
        })
    }
}

```

```

        return root
    }

```

110. 平衡二叉树(1)

- 题目

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过1。

示例 1：

给定二叉树 [3, 9, 20, null, null, 15, 7]

```

      3
     / \
    9   20
       / \
      15   7

```

返回 true。

示例 2：

给定二叉树 [1, 2, 2, 3, 3, null, null, 4, 4]

```

      1
     / \
    2   2
   / \
  3   3
 / \
4   4

```

返回 false。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	O(n)	O(log(n))

```

func isBalanced(root *TreeNode) bool {
    _, isBalanced := recur(root)
    return isBalanced
}

func recur(root *TreeNode) (int, bool) {
    if root == nil {
        return 0, true
    }

    leftDepth, leftIsBalanced := recur(root.Left)
    if leftIsBalanced == false{
        return 0, false
    }
    rightDepth, rightIsBalanced := recur(root.Right)
    if rightIsBalanced == false{
        return 0, false
    }

    if -1 <= leftDepth-rightDepth &&
        leftDepth-rightDepth <= 1 {
        return max(leftDepth, rightDepth) + 1, true
    }
    return 0, false
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

```

111.二叉树的最小深度(2)

- 题目

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

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],



返回它的最小深度 2.

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	O(n)	O(log(n))
02	广度优先	O(n)	O(n)

```

// 递归
func minDepth(root *TreeNode) int {
    if root == nil {
        return 0
    } else if root.Left == nil {
        return 1 + minDepth(root.Right)
    } else if root.Right == nil {
        return 1 + minDepth(root.Left)
    } else {
        return 1 + min(minDepth(root.Left), minDepth(root.Right))
    }
}

func min(a, b int) int {
    if a < b {
        return a
    }
    return b
}

// 广度优先搜索
func minDepth(root *TreeNode) int {
    if root == nil {
        return 0
    }

    list := make([]*TreeNode, 0)
    list = append(list, root)
    depth := 1

    for len(list) > 0 {
        length := len(list)
        for i := 0; i < length; i++ {
            node := list[0]
            list = list[1:]
            if node.Left == nil && node.Right == nil {
                return depth
            }
            if node.Left != nil {
                list = append(list, node.Left)
            }
            if node.Right != nil {
                list = append(list, node.Right)
            }
        }
        depth++
    }
}

```

```

        return depth
    }

```

112.路径总和(2)

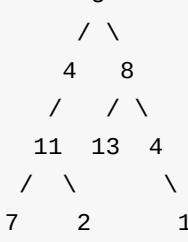
- 题目

给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，使得所有节点值之和等于目标和。

说明：叶子节点是指没有子节点的节点。

示例：

给定如下二叉树，以及目标和 $sum = 22$,



返回 `true`, 因为存在目标和为 22 的根节点到叶子节点的路径 $5->4->11->$

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	$O(n)$	$O(\log(n))$
02	迭代	$O(n)$	$O(n)$

```

// 递归
func hasPathSum(root *TreeNode, sum int) bool {
    if root == nil {
        return false
    }
    sum = sum - root.Val
    if root.Left == nil && root.Right == nil {
        return sum == 0
    }
    return hasPathSum(root.Left, sum) || hasPathSum(root.Ri
}

// 迭代
func hasPathSum(root *TreeNode, sum int) bool {
    if root == nil {
        return false
    }
    list1 := list.New()
    list2 := list.New()

    list1.PushFront(root)
    list2.PushFront(sum - root.Val)
    for list1.Len() > 0 {
        length := list1.Len()

        for i := 0; i < length; i++ {
            node := list1.Remove(list1.Back()).(*TreeNode)
            currentSum := list2.Remove(list2.Back()).(int)
            if node.Left == nil && node.Right == nil && cur
                return true
            }
            if node.Left != nil {
                list1.PushFront(node.Left)
                list2.PushFront(currentSum - node.Left.Val)
            }
            if node.Right != nil {
                list1.PushFront(node.Right)
                list2.PushFront(currentSum - node.Right.Va]
            }
        }
    }
    return false
}

```

118.杨辉三角(2)

- 题目

给定一个非负整数 numRows，生成杨辉三角的前 numRows 行。
在杨辉三角中，每个数是它左上方和右上方的数的和。

示例：

输入： 5

输出：

```
[  
    [1],  
    [1,1],  
    [1,2,1],  
    [1,3,3,1],  
    [1,4,6,4,1]  
]
```

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	动态规划	$O(n^2)$	$O(n^2)$
02(最优)	递推	$O(n^2)$	$O(n^2)$

```

// 动态规划
func generate(numRows int) [][]int {
    var result [][]int
    for i := 0; i < numRows; i++ {
        var row []int
        for j := 0; j <= i; j++ {
            tmp := 1
            if j == 0 || j == i {
                } else {
                    tmp = result[i-1][j-1] + result[i-1][j]
                }
            row = append(row, tmp)
        }
        result = append(result, row)
    }
    return result
}

// 递推
func generate(numRows int) [][]int {
    res := make([][]int, 0)
    if numRows == 0 {
        return res
    }

    res = append(res, []int{1})
    if numRows == 1 {
        return res
    }

    for i := 1; i < numRows; i++ {
        res = append(res, genNext(res[i-1]))
    }
    return res
}

func genNext(p []int) []int {
    res := make([]int, 1, len(p)+1)
    res = append(res, p...)

    for i := 0; i < len(res)-1; i++ {
        res[i] = res[i] + res[i+1]
    }
    return res
}

```

119. 杨辉三角II(3)

- 题目

给定一个非负索引 k ，其中 $k \leq 33$ ，返回杨辉三角的第 k 行。
在杨辉三角中，每个数是它左上方和右上方的数的和。

示例：

输入： 3

输出： [1, 3, 3, 1]

进阶：

你可以优化你的算法到 $O(k)$ 空间复杂度吗？

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	动态规划	$O(n^2)$	$O(n^2)$
02	递推	$O(n^2)$	$O(n)$
03(最优)	二项式定理	$O(n)$	$O(n)$

```

// 动态规划
func.getRow(rowIndex int) []int {
    var result [][]int
    for i := 0; i < rowIndex+1; i++ {
        var row []int
        for j := 0; j <= i; j++ {
            tmp := 1
            if j == 0 || j == i {

            } else {
                tmp = result[i-1][j-1] + result[i-1][j]
            }
            row = append(row, tmp)
        }
        result = append(result, row)
    }
    return result[rowIndex]
}

// 递推
func.getRow(rowIndex int) []int {
    res := make([]int, 1, rowIndex+1)
    res[0] = 1
    if rowIndex == 0{
        return res
    }

    for i := 0; i < rowIndex; i++{
        res = append(res, 1)
        for j := len(res) -2 ; j > 0; j--{
            res[j] = res[j] + res[j-1]
        }
    }

    return res
}

// 二项式定理
func.getRow(rowIndex int) []int {
    res := make([]int, rowIndex+1)
    res[0] = 1
    if rowIndex == 0{
        return res
    }

    // 公式
    // C(n, k) = n! / (k! * (n-k)!)
    // C(n, k) = (n-k+1)/k * C(n, k-1)
}

```

```

for i := 1; i <= rowIndex; i++{
    res[i] = res[i-1] * (rowIndex-i+1)/i
}
return res
}

```

121. 买卖股票的最佳时机(3)

- 题目

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）卖出。

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。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	暴力法	$O(n^2)$	$O(1)$
02(最优)	动态规划(从前到后) 最大利润= $\max\{\text{前一天最大利润}, \text{今天的价格} - \text{之前最低价格}\}$	$O(n)$	$O(1)$
03	动态规划(从后到前)	$O(n)$	$O(1)$

```

// 暴力法
func maxProfit(prices []int) int {
    max := 0
    length := len(prices)

    for i := 0; i < length-1 ; i++{
        for j := i+1; j <= length-1; j++{
            if prices[j] - prices[i] > max{
                max = prices[j] - prices[i]
            }
        }
    }
    return max
}

// 动态规划(从前到后)
func maxProfit(prices []int) int {
    if len(prices) < 2 {
        return 0
    }

    min := prices[0]
    profit := 0

    for i := 1; i < len(prices); i++ {
        if prices[i] < min {
            min = prices[i]
        }
        if profit < prices[i]-min {
            profit = prices[i] - min
        }
    }
    return profit
}

// 动态规划(从后到前)
func maxProfit(prices []int) int {
    if len(prices) < 2 {
        return 0
    }

    max := 0
    profit := 0

    for i := len(prices) - 1; i >= 0; i-- {
        if max < prices[i] {
            max = prices[i]
        }
    }
}

```

```

    }
    if profit < max-prices[i] {
        profit = max - prices[i]
    }
}

return profit
}

```

122. 买卖股票的最佳时机II(2)

- 题目

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（ ≥ 1 次）。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）随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格

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）

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

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。



- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	贪心法	$O(n)$	$O(1)$
02	峰谷峰顶法	$O(n)$	$O(1)$

```

func maxProfit(prices []int) int {
    max := 0
    for i := 1; i < len(prices); i++ {
        if prices[i] > prices[i-1] {
            max = max + prices[i] - prices[i-1]
        }
    }
    return max
}

func maxProfit(prices []int) int {
    if len(prices) == 0 {
        return 0
    }
    i := 0
    valley := prices[0]
    peak := prices[0]
    profit := 0
    for i < len(prices)-1 {
        for i < len(prices)-1 && prices[i] >= prices[i+1] {
            i++
        }
        valley = prices[i]
        for i < len(prices)-1 && prices[i] <= prices[i+1] {
            i++
        }
        peak = prices[i]
        profit = profit + peak - valley
    }
    return profit
}

```

125. 验证回文串(2)

- 题目

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母大小写。
说明：本题中，我们将空字符串定义为有效的回文串。

示例 1：

输入: "A man, a plan, a canal: Panama"

输出: true

示例 2：

输入: "race a car"

输出: false

• 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	双指针法	$O(n)$	$O(1)$
02	双指针法	$O(n)$	$O(n)$

```

func isPalindrome(s string) bool {
    s = strings.ToLower(s)
    i, j := 0, len(s)-1

    for i < j {
        for i < j && !isChar(s[i]) {
            i++
        }
        for i < j && !isChar(s[j]) {
            j--
        }
        if s[i] != s[j] {
            return false
        }
        i++
        j--
    }
    return true
}

func isChar(c byte) bool {
    if ('a' <= c && c <= 'z') || ('0' <= c && c <= '9') {
        return true
    }
    return false
}

// 
func isPalindrome(s string) bool {
    str := ""
    s = strings.ToLower(s)
    for _, value := range s {
        if (value >= '0' && value <= '9') || (value >= 'a'
            str += string(value)
        }
    }
    if len(str) == 0 {
        return true
    }
    i := 0
    j := len(str) - 1
    for i <= j {
        if str[i] != str[j] {
            return false
        }
        i++
        j--
    }
}

```

```

        return true
    }

```

136. 只出现一次的数字(4)

- 题目

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。
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

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	异或	O(n)	O(1)
02	哈希	O(n)	O(n)
03	暴力法	O(n^2)	O(1)
04	排序遍历	O(nlog(n))	O(1)

```

// 异或
func singleNumber(nums []int) int {
    res := 0
    for _, n := range nums {
        res = res ^ n
    }
    return res
}

// 哈希
func singleNumber(nums []int) int {
    m := make(map[int]int)

    for _, v := range nums{
        m[v]++
    }

    for k,v := range m{
        if v == 1{
            return k
        }
    }
    return -1
}

// 暴力法
func singleNumber(nums []int) int {
    for i := 0; i < len(nums); i++ {
        flag := false
        for j := 0; j < len(nums); j++ {
            if nums[i] == nums[j] && i != j {
                flag = true
                break
            }
        }
        if flag == false {
            return nums[i]
        }
    }
    return -1
}

// 排序遍历
func singleNumber(nums []int) int {
    sort.Ints(nums)
    for i := 0; i < len(nums); i = i+2{
        if i+1 == len(nums){

```

```

        return nums[i]
    }
    if nums[i] != nums[i+1]{
        return nums[i]
    }
}
return -1
}

```

141. 环形链表(2)

- 题目

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。
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。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	哈希法	$O(n)$	$O(n)$
02(最优)	双指针(快慢指针)	$O(n)$	$O(1)$

```

func hasCycle(head *ListNode) bool {
    m := make(map[*ListNode]bool)
    for head != nil {
        if m[head] {
            return true
        }
        m[head] = true
        head = head.Next
    }
    return false
}

// 双指针(快慢指针)
func hasCycle(head *ListNode) bool {
    if head == nil {
        return false
    }
    fast := head.Next
    for fast != nil && head != nil && fast.Next != nil {
        if fast == head {
            return true
        }
        fast = fast.Next.Next
        head = head.Next
    }
    return false
}

```

155. 最小栈(2)

- 题目

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.
```

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	使用数组模拟栈, 保存数据的时候同时保存当前的最小值	$O(n)$	$O(n)$
02	使用双栈	$O(n)$	$O(n)$

```

type item struct {
    min, x int
}
type MinStack struct {
    stack []item
}

func Constructor() MinStack {
    return MinStack{}
}

func (this *MinStack) Push(x int) {
    min := x
    if len(this.stack) > 0 && this.GetMin() < x {
        min = this.GetMin()
    }
    this.stack = append(this.stack, item{
        min: min,
        x:   x,
    })
}

func (this *MinStack) Pop() {
    this.stack = this.stack[:len(this.stack)-1]
}

func (this *MinStack) Top() int {
    if len(this.stack) == 0 {
        return 0
    }
    return this.stack[len(this.stack)-1].x
}

func (this *MinStack) GetMin() int {
    if len(this.stack) == 0 {
        return 0
    }
    return this.stack[len(this.stack)-1].min
}

// 
type MinStack struct {
    data []int
    min  []int
}

func Constructor() MinStack {
    return MinStack{[]int{}, []int{}}
}

```

```

}

func (this *MinStack) Push(x int) {
    if len(this.data) == 0 || x <= this.GetMin() {
        this.min = append(this.min, x)
    }
    this.data = append(this.data, x)
}

func (this *MinStack) Pop() {
    x := this.data[len(this.data)-1]
    this.data = this.data[:len(this.data)-1]
    if x == this.GetMin() {
        this.min = this.min[:len(this.min)-1]
    }
}

func (this *MinStack) Top() int {
    if len(this.data) == 0 {
        return 0
    }
    return this.data[len(this.data)-1]
}

func (this *MinStack) GetMin() int {
    return this.min[len(this.min)-1]
}

```

160.相交链表(4)

- 题目

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,`
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]。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]`
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 = 2, skipB = 1`
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`

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。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	计算长度后, 对齐长度再比较	$O(n)$	$O(1)$
02(最优)	交换后相连, 再比较	$O(n)$	$O(1)$
03	暴力法	$O(n^2)$	$O(1)$
04	哈希法	$O(n)$	$O(n)$

```

func getIntersectionNode(headA, headB *ListNode) *ListNode
    ALength := 0
    A := headA
    for A != nil {
        ALength++
        A = A.Next
    }
    BLength := 0
    B := headB
    for B != nil {
        BLength++
        B = B.Next
    }

    pA := headA
    pB := headB
    if ALength > BLength {
        n := ALength - BLength
        for n > 0 {
            pA = pA.Next
            n--
        }
    } else {
        n := BLength - ALength
        for n > 0 {
            pB = pB.Next
            n--
        }
    }

    for pA != pB {
        pA = pA.Next
        pB = pB.Next
    }
    return pA
}

// 
func getIntersectionNode(headA, headB *ListNode) *ListNode
    A, B := headA, headB
    for A != B {
        if A != nil {
            A = A.Next
        } else {
            A = headB
        }
        if B != nil {
            B = B.Next
        }
    }
    return A
}

```

```

        } else {
            B = headA
        }
    }
    return A
}

// 暴力法
func getIntersectionNode(headA, headB *ListNode) *ListNode
A, B := headA, headB
for A != nil {
    for B != nil {
        if A == B {
            return A
        }
        B = B.Next
    }
    A = A.Next
    B = headB
}
return nil
}

// 哈希表法
func getIntersectionNode(headA, headB *ListNode) *ListNode
m := make(map[*ListNode]bool)
for headA != nil {
    m[headA] = true
    headA = headA.Next
}

for headB != nil {
    if _, ok := m[headB]; ok {
        return headB
    }
    headB = headB.Next
}
return nil
}

```

167.两数之和 II - 输入有序数组(4)

- 题目

给定一个已按照升序排列 的有序数组，找到两个数使得它们相加之和等于目标数。函数应该返回这两个下标值 `index1` 和 `index2`，其中 `index1` 必须小于 `index2`。

说明：

返回的下标值（`index1` 和 `index2`）不是从零开始的。

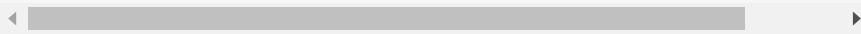
你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

示例：

输入: `numbers = [2, 7, 11, 15]`, `target = 9`

输出: `[1, 2]`

解释: 2 与 7 之和等于目标数 9 。因此 `index1 = 1`, `index2 = 2` 。



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	暴力法: 2层循环遍历	$O(n^2)$	$O(1)$
02	两遍哈希遍历	$O(n)$	$O(n)$
03	一遍哈希遍历	$O(n)$	$O(n)$
04(最优)	一遍哈希遍历	$O(n)$	$O(1)$

```

// 暴力法：2层循环遍历
func twoSum(nums []int, target int) []int {
    for i := 0; i < len(nums); i++ {
        for j := i + 1; j < len(nums); j++ {
            if nums[i]+nums[j] == target {
                return []int{i + 1, j + 1}
            }
        }
    }
    return []int{}
}

// 两遍哈希遍历
func twoSum(nums []int, target int) []int {
    m := make(map[int]int, len(nums))
    for k, v := range nums {
        m[v] = k
    }

    for i := 0; i < len(nums); i++ {
        b := target - nums[i]
        if num, ok := m[b]; ok && num != i {
            return []int{i + 1, m[b] + 1}
        }
    }
    return []int{}
}

// 一遍哈希遍历
func twoSum(numbers []int, target int) []int {
    m := make(map[int]int, len(numbers))

    for i, n := range numbers {
        if m[target-n] != 0 {
            return []int{m[target-n], i + 1}
        }
        m[n] = i + 1
    }
    return nil
}

// 双指针法
func twoSum(numbers []int, target int) []int {
    first := 0
    last := len(numbers) - 1

    result := make([]int, 2)

```

```

for {
    if numbers[first]+numbers[last] == target {
        result[0] = first + 1
        result[1] = last + 1
        return result
    } else if numbers[first]+numbers[last] > target {
        last--
    } else {
        first++
    }
}

```

168.Excel表列名称(2)

- 题目

给定一个正整数，返回它在 Excel 表中相对应的列名称。
例如，

1 -> A
 2 -> B
 3 -> C
 ...
 26 -> Z
 27 -> AA
 28 -> AB
 ...

示例 1：
 输入：1
 输出："A"

示例 2：
 输入：28
 输出："AB"

示例 3：
 输入：701
 输出："ZY"

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	求余模拟进制	O(log(n))	O(1)
02	递归计算	O(log(n))	O(log(n))

```
// 求余模拟进制
func convertToTitle(n int) string {
    str := ""

    for n > 0 {
        n--
        str = string(byte(n%26)+'A') + str
        n /= 26
    }
    return str
}

// 递归计算
func convertToTitle(n int) string {
    if n <= 26{
        return string('A'+n-1)
    }
    y := n % 26
    if y == 0{
        // 26的倍数 如52%26=0 => AZ
        return convertToTitle((n-y-1)/26)+convertToTitle(26)
    }
    return convertToTitle((n-y)/26)+convertToTitle(y)
}
```

169. 多数元素(5)

- 题目

给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数大于 $\lceil \frac{n}{2} \rceil$ 次的元素。你可以假设数组是非空的，并且给定的数组总是存在多数元素。

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

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	排序取半	$O(\log(n))$	$O(1)$
02	哈希法	$O(n)$	$O(n)$
03(最优)	Boyer-Moore投票算法	$O(n)$	$O(1)$
04	位运算	$O(n)$	$O(1)$
05	分治法	$O(n\log(n))$	$O(\log(n))$

```

// 排序取半
func majorityElement(nums []int) int {
    sort.Ints(nums)
    return nums[len(nums)/2]
}

// 哈希法
func majorityElement(nums []int) int {
    m := make(map[int]int)
    result := 0
    for _, v := range nums{
        if _,ok := m[v];ok{
            m[v]++
        }else {
            m[v]=1
        }
        if m[v] > (len(nums)/2){
            result = v
        }
    }
    return result
}

// Boyer-Moore投票算法
func majorityElement(nums []int) int {
    result, count := 0, 0
    for i := 0; i < len(nums); i++ {
        if count == 0 {
            result = nums[i]
            count++
        } else if result == nums[i] {
            count++
        } else {
            count--
        }
    }
    return result
}

// 位运算
func majorityElement(nums []int) int {
    if len(nums) == 1 {
        return nums[0]
    }
    result := int32(0)
    // 64位有坑
    mask := int32(1)
    for i := 0; i < 32; i++ {

```

```

        count := 0
        for j := 0; j < len(nums); j++ {
            if mask&int32(nums[j]) == mask {
                count++
            }
        }
        if count > len(nums)/2 {
            result = result | mask
        }
        mask = mask << 1
    }
    return int(result)
}

// 分治法
func majorityElement(nums []int) int {
    return majority(nums, 0, len(nums)-1)
}

func count(nums []int, target int, start int, end int) int
    countNum := 0
    for i := start; i <= end; i++ {
        if nums[i] == target {
            countNum++
        }
    }
    return countNum
}

func majority(nums []int, start, end int) int {
    if start == end {
        return nums[start]
    }

    mid := (start + end) / 2

    left := majority(nums, start, mid)
    right := majority(nums, mid+1, end)
    if left == right {
        return left
    }

    leftCount := count(nums, left, start, end)
    rightCount := count(nums, right, start, end)
    if leftCount > rightCount {
        return left
    }
}

```

```
        return right
    }
```



171.Excel表列序号(1)

- 题目

给定一个Excel表格中的列名称，返回其相应的列序号。

例如，

```
A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28
...
```

示例 1：

输入： "A"

输出： 1

示例 2：

输入： "AB"

输出： 28

示例 3：

输入： "ZY"

输出： 701

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	26进制计算	O(log(n))	O(1)

```
func titleToNumber(s string) int {
    result := 0
    for i := 0; i < len(s); i++ {
        temp := int(s[i] - 'A' + 1)
        result = result*26 + temp
    }
    return result
}
```

172. 阶乘后的零(1)

- 题目

给定一个整数 n , 返回 $n!$ 结果尾数中零的数量。

示例 1:

输入: 3

输出: 0

解释: $3! = 6$, 尾数中没有零。

示例 2:

输入: 5

输出: 1

解释: $5! = 120$, 尾数中有 1 个零.

说明: 你算法的时间复杂度应为 $O(\log n)$ 。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	数学, 找规律	$O(\log(n))$	$O(1)$

```
func trailingZeroes(n int) int {
    result := 0
    for n >= 5 {
        n = n / 5
        result = result + n
    }
    return result
}
```

189. 旋转数组(4)

- 题目

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

示例 1：

输入：[1, 2, 3, 4, 5, 6, 7] 和 $k = 3$

输出：[5, 6, 7, 1, 2, 3, 4]

解释：

向右旋转 1 步：[7, 1, 2, 3, 4, 5, 6]

向右旋转 2 步：[6, 7, 1, 2, 3, 4, 5]

向右旋转 3 步：[5, 6, 7, 1, 2, 3, 4]

示例 2：

输入：[-1, -100, 3, 99] 和 $k = 2$

输出：[3, 99, -1, -100]

解释：

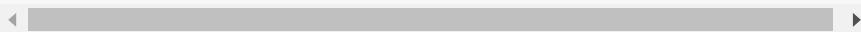
向右旋转 1 步：[99, -1, -100, 3]

向右旋转 2 步：[3, 99, -1, -100]

说明：

尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。

要求使用空间复杂度为 $O(1)$ 的 原地 算法。



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	暴力法	$O(n^2)$	$O(1)$
02	三次反转法	$O(n)$	$O(1)$
03	使用额外的数组	$O(n)$	$O(n)$
04(最优)	环形替换	$O(n)$	$O(1)$

```

// 暴力法
func rotate(nums []int, k int) {
    n := len(nums)

    if k > n {
        k = k % n
    }
    if k == 0 || k == n {
        return
    }
    for i := 0; i < k; i++ {
        last := nums[len(nums)-1]
        for j := 0; j < len(nums); j++ {
            nums[j], last = last, nums[j]
        }
    }
}

// 三次反转法
func rotate(nums []int, k int) {
    n := len(nums)

    if k > n {
        k = k % n
    }
    if k == 0 || k == n {
        return
    }
    reverse(nums, 0, n-1)
    reverse(nums, 0, k-1)
    reverse(nums, k, n-1)
}

func reverse(nums []int, i, j int) {
    for i < j {
        nums[i], nums[j] = nums[j], nums[i]
        i++
        j--
    }
}

// 使用额外的数组
func rotate(nums []int, k int) {
    n := len(nums)

    if k > n {
        k = k % n
    }
}

```

```

if k == 0 || k == n {
    return
}

arr := make([]int, len(nums))
for i := 0; i < len(nums); i++ {
    arr[(i+k)%len(nums)] = nums[i]
}

for i := 0; i < len(nums); i++ {
    nums[i] = arr[i]
}
}

// 环形替换
func rotate(nums []int, k int) {
    n := len(nums)

    if k > n {
        k = k % n
    }
    if k == 0 || k == n {
        return
    }
    count := 0

    for i := 0; count < len(nums); i++ {
        current := i
        prev := nums[i]
        for {
            next := (current + k) % len(nums)
            nums[next], prev = prev, nums[next]
            current = next
            // fmt.Println(nums, prev)
            count++
            if i == current {
                break
            }
        }
    }
}

```

190. 颠倒二进制位(3)

- 题目

颠倒给定的 32 位无符号整数的二进制位。

示例 1：

输入：0000001010010100000111010011100

输出：0011100101110000010100101000000

解释：输入的二进制串 0000001010010100000111010011100 表示无符号整数 964176192，其二进制表示形式为 001110010111100000：

示例 2：

输入：111111111111111111111111111111101

输出：101111111111111111111111111111111

解释：输入的二进制串 111111111111111111111111111111101 表示无符号整数 3221225471，其二进制表示形式为 10101111100101100：

提示：

请注意，在某些语言（如 Java）中，没有无符号整数类型。

在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您

因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同

在 Java 中，编译器使用二进制补码记法来表示有符号整数。

因此，在上面的示例 2 中，输入表示有符号整数 -3，输出表示有符号整数 -1。进阶：

如果多次调用这个函数，你将如何优化你的算法？

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	位操作	O(1)	O(1)
02	转字符串	O(n)	O(1)
03	二进制交换	O(1)	O(1)

```

func reverseBits(num uint32) uint32 {
    result := uint32(0)
    for i := 0; i < 32; i++ {
        last := num & 1 // 取最后一位
        result = (result << 1) + last // 前移
        num = num >> 1
    }
    return result
}

// 将整数转换为二进制字符串并反转
func reverseBits(num uint32) uint32 {
    str := strconv.FormatUint(uint64(num), 2)
    rev := ""
    for i := len(str) - 1; i >= 0; i-- {
        rev = rev + str[i:i+1]
    }
    if len(rev) < 32 {
        rev = rev + strings.Repeat("0", 32-len(rev))
    }
    n, _ := strconv.ParseUint(rev, 2, 64)
    return uint32(n)
}

// 二进制交换
import (
    "github.com/imroc/biu"
)

func reverseBits(num uint32) uint32 {
    fmt.Println(biu.Uint32ToBinaryString(num))
    num = ((num & 0xffff0000) >> 16) | ((num & 0x0000ffff) <
    num = ((num & 0xff00ff00) >> 8) | ((num & 0x00ff00ff) <
    num = ((num & 0xf0f0f0f0) >> 4) | ((num & 0x0f0f0f0f) <
    num = ((num & 0xcccccccc) >> 2) | ((num & 0x33333333) <
    num = ((num & 0xaaaaaaaaaa) >> 1) | ((num & 0x55555555) <
    return num
}

```

191. 位1的个数(2)

- 题目

编写一个函数，输入是一个无符号整数，返回其二进制表达式中数字位数为 ‘1’（也被称为汉明重量）。

示例 1：

输入：00000000000000000000000000001011

输出：3

解释：输入的二进制串 00000000000000000000000000001011 中，共有三

示例 2：

输入：000000000000000000000000000010000000

输出：1

解释：输入的二进制串 000000000000000000000000000010000000 中，共有一

示例 3：

输入：11111111111111111111111111111101

输出：31

解释：输入的二进制串 111111111111111111111111111101 中，共有：

提示：

请注意，在某些语言（如 Java）中，没有无符号整数类型。

在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您
因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同
在 Java 中，编译器使用二进制补码记法来表示有符号整数。

因此，在上面的示例 3 中，输入表示有符号整数 -3。

进阶：

如果多次调用这个函数，你将如何优化你的算法？

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	循环位计算	O(1)	O(1)
02(最优)	位计算 $n \& (n-1)$ ，会把该整数的最右边的1变成0	O(1)	O(1)

```
// 循环位计算
func hammingWeight(num uint32) int {
    count := 0
    for num != 0 {
        if num&1 == 1 {
            count++
        }
        num = num >> 1
    }
    return count
}

// 
func hammingWeight(num uint32) int {
    count := 0
    for num != 0 {
        num = num & (num - 1)
        count++
    }
    return count
}
```

198. 打家劫舍(4)

- 题目

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下能够盗取的最大金额。

示例 1: 输入: [1,2,3,1] 输出: 4

解释: 偷窃 1 号房屋 (金额 = 1)，然后偷窃 3 号房屋 (金额 = 3)。偷窃两个房屋 = 1 + 3 = 4。

示例 2: 输入: [2,7,9,3,1] 输出: 12

解释: 偷窃 1 号房屋 (金额 = 2)，偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。偷窃到的最高金额 = 2 + 9 + 1 = 12。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	动态规划	$O(n)$	$O(1)$
02	动态规划+一维数组	$O(n)$	$O(n)$
03	动态规划+二维数组	$O(n)$	$O(n)$
04	奇偶法	$O(n)$	$O(1)$

```

func rob(nums []int) int {
    if len(nums) == 0 {
        return 0
    }
    if len(nums) == 1 {
        return nums[0]
    }
    a := nums[0]
    b := max(a, nums[1])

    for i := 2; i < len(nums); i++ {
        a, b = b, max(a+nums[i], b)
    }
    return b
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

#
func rob(nums []int) int {
    n := len(nums)
    if n == 0 {
        return 0
    }
    if n == 1 {
        return nums[0]
    }
    dp := make([]int, n)
    dp[0] = nums[0]
    if nums[0] > nums[1] {
        dp[1] = nums[0]
    } else {
        dp[1] = nums[1]
    }
    for i := 2; i < n; i++ {
        dp[i] = max(dp[i-1], dp[i-2]+nums[i])
    }
    return dp[n-1]
}

func max(a, b int) int {
    if a > b {
        return a
    }
}

```

```

        }
        return b
    }

    #

func rob(nums []int) int {
    if len(nums) == 0 {
        return 0
    }
    if len(nums) == 1 {
        return nums[0]
    }
    n := len(nums)
    dp := make([][]int, n)
    for n := range dp {
        dp[n] = make([]int, 2)
    }
    dp[0][0], dp[0][1] = 0, nums[0]
    for i := 1; i < n; i++ {
        dp[i][0] = max(dp[i-1][0], dp[i-1][1])
        dp[i][1] = dp[i-1][0] + nums[i]
    }
    return max(dp[n-1][0], dp[n-1][1])
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

#

func rob(nums []int) int {
    var a, b int
    for i, v := range nums {
        if i%2 == 0 {
            a = max(a+v, b)
        } else {
            b = max(a, b+v)
        }
    }
    return max(a, b)
}

func max(a, b int) int {
    if a > b {
        return a
    }
}

```

```
    }  
    return b  
}
```

0201-0300-Easy

- 0201-0300-Easy
 - 202.快乐数(2)
 - 203.移除链表元素(2)
 - 204.计数质数(2)
 - 205.同构字符串(3)
 - 206.反转链表(2)
 - 217.存在重复元素(2)
 - 219.存在重复元素 II(2)
 - 225.用队列实现栈(4)
 - 226.翻转二叉树(2)
 - 231.2的幂(3)
 - 232.用栈实现队列(3)
 - 234.回文链表(4)
 - 235.二叉搜索树的最近公共祖先(2)
 - 237.删除链表中的节点(1)
 - 242.有效的字母异位词(2)
 - 257.二叉树的所有路径(2)
 - 258.各位相加(4)
 - 263.丑数(2)
 - 268.缺失数字(5)
 - 278.第一个错误的版本(2)
 - 283.移动零(3)
 - 290.单词规律(2)
 - 292.Nim 游戏(1)
 - 299.猜数字游戏(2)

202.快乐数(2)

- 题目

编写一个算法来判断一个数 n 是不是快乐数。

「快乐数」定义为：对于一个正整数，每一次将该数替换为它每个位置上的数字之和，然后重复这个过程直到这个数变为 1，也可能是 无限循环 但始终变不到 1。

如果 可以变为 1，那么这个数就是快乐数。

如果 n 是快乐数就返回 True ；不是，则返回 False 。

示例： 输入：19 输出：true

解释：

12 + 92 = 82

82 + 22 = 68

62 + 82 = 100

12 + 02 + 02 = 1

- 解题思路

No. | 思路 | 时间复杂度 | 空间复杂度 -|-|-| 01 | 哈希+遍历 | $O(\log(n))$

| $O(\log(n))$

02 | 遍历+快慢判断环 | $O(\log(n))$ | $O(1)$

```

func isHappy(n int) bool {
    now, next := n, nextValue(n)
    m := make(map[int]int)
    m[now] = 1
    for {
        if next == 1 {
            break
        }
        _, ok := m[next]; ok {
            break
        } else {
            m[next] = 1
        }
        next = nextValue(next)
    }

    if next == 1 {
        return true
    }

    return false
}

func nextValue(n int) int {
    ret := 0
    for n != 0 {
        ret = ret + (n%10)*(n%10)
        n = n / 10
    }
    return ret
}

#
func isHappy(n int) bool {
    now, next := n, nextValue(n)
    for now != next {
        now = nextValue(now)
        next = nextValue(nextValue(next))
    }
    if now == 1 {
        return true
    }
    return false
}

func nextValue(n int) int {
    ret := 0
    for n != 0 {

```

```

        ret = ret + (n%10)*(n%10)
        n = n / 10
    }
    return ret
}

```

203. 移除链表元素(2)

- 题目

删除链表中等于给定值 val 的所有节点。

示例：

输入： $1 \rightarrow 2 \rightarrow 6 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$, $\text{val} = 6$

输出： $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	哨兵结点+链表遍历	$O(n)$	$O(1)$
02	递归	$O(n)$	$O(n)$

```

func removeElements(head *ListNode, val int) *ListNode {
    headPre := &ListNode{Next: head}
    temp := headPre

    for temp.Next != nil {
        if temp.Next.Val == val {
            //delete
            temp.Next = temp.Next.Next
        } else {
            temp = temp.Next
        }
    }

    return headPre.Next
}

# 递归
func removeElements(head *ListNode, val int) *ListNode {
    if head == nil {
        return nil
    }
    head.Next = removeElements(head.Next, val)
    if head.Val == val {
        return head.Next
    }
    return head
}

```

204. 计数质数(2)

- 题目

统计所有小于非负整数 n 的质数的数量。

示例：

输入：10

输出：4

解释：小于 10 的质数一共有 4 个，它们是 2, 3, 5, 7。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	筛选质数(厄拉多塞筛法)	$O(n^2)$	$O(n)$
02	筛选质数优化	$O(n^{(1/4)})$	$O(n)$

```

func countPrimes(n int) int {
    if n < 3 {
        return 0
    }
    notPrimes := make([]bool, n)
    count := 0
    for i := 2; i < n; i++ {
        if notPrimes[i] {
            continue
        }
        for j := i*2 ; j < n; j += i {
            notPrimes[j] = true
        }
        count++
    }
    return count
}

#
func countPrimes(n int) int {
    if n < 3 {
        return 0
    }
    isPrimes := make([]bool, n)
    for i := range isPrimes {
        isPrimes[i] = true
    }
    for i := 2; i*i < n; i++ {
        if !isPrimes[i] {
            continue
        }
        for j := i * i; j < n; j += i {
            isPrimes[j] = false
        }
    }
    count := 0
    for i := 2; i < n; i++ {
        if isPrimes[i] {
            count++
        }
    }
    return count
}

```

205.同构字符串(3)

- 题目

给定两个字符串 s 和 t ，判断它们是否是同构的。

如果 s 中的字符可以被替换得到 t ，那么这两个字符串是同构的。

所有出现的字符都必须用另一个字符替换，同时保留字符的顺序。

两个字符不能映射到同一个字符上，但字符可以映射自己本身。

示例 1: 输入: $s = "egg"$, $t = "add"$ 输出: true

示例 2: 输入: $s = "foo"$, $t = "bar"$ 输出: false

示例 3: 输入: $s = "paper"$, $t = "title"$ 输出: true

说明:你可以假设 s 和 t 具有相同的长度。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	数组映射	$O(n)$	$O(n)$
02	哈希映射	$O(n)$	$O(n)$
03	字符串查找匹配	$O(n)$	$O(1)$

```

func isIsomorphic(s string, t string) bool {
    if len(s) != len(t) {
        return false
    }

    m1 := make([]int, 256)
    m2 := make([]int, 256)

    for i := 0; i < len(s); i++ {
        a := int(s[i])
        b := int(t[i])
        if m1[a] != m2[b] {
            return false
        }
        m1[a] = i + 1
        m2[b] = i + 1
    }
    return true
}

#
func isIsomorphic(s string, t string) bool {
    if len(s) != len(t) {
        return false
    }

    m := make(map[int]int)
    n := make(map[int]int)

    for i := 0; i < len(s); i++ {
        a := int(s[i])
        b := int(t[i])
        if m[a] == 0 && n[b] == 0 {
            m[a] = b
            n[b] = a
        } else if m[a] != b || n[b] != a {
            return false
        }
    }
    return true
}

#
func isIsomorphic(s string, t string) bool {
    for i := 0; i < len(s); i++ {
        if strings.IndexByte(s[i+1:], s[i]) != strings.IndexByte(t[i+1:], t[i])
            return false
    }
}

```

```
        }
    }
    return true
}
◀ ▶
```

206. 反转链表(2)

- 题目

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？

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(n)$
02	迭代	$O(n)$	$O(1)$

```

func reverseList(head *ListNode) *ListNode {
    if head == nil || head.Next == nil {
        return head
    }

    result := reverseList(head.Next)
    head.Next.Next = head
    head.Next = nil
    return result
}

// 
func reverseList(head *ListNode) *ListNode {
    var result *ListNode
    var temp *ListNode
    for head != nil {
        temp = head.Next
        head.Next = result
        result = head
        head = temp
    }
    return result
}

```

217. 存在重复元素(2)

- 题目

给定一个整数数组，判断是否存在重复元素。

如果任意一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

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

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	哈希辅助遍历	O(n)	O(n)
02	排序后遍历	O(nlog(n))	O(1)

```

func containsDuplicate(nums []int) bool {
    m := make(map[int]int)
    for i := 0; i < len(nums); i++ {
        if _, ok := m[nums[i]]; ok {
            return true
        } else {
            m[nums[i]] = 1
        }
    }
    return false
}

#
func containsDuplicate(nums []int) bool {
    sort.Ints(nums)
    for i := 0; i < len(nums)-1; i++{
        if nums[i] == nums[i+1]{
            return true
        }
    }
    return false
}

```

219. 存在重复元素 II(2)

- 题目

给定一个整数数组和一个整数 k ，判断数组中是否存在两个不同的索引 i 和 j 使得 $\text{nums}[i] = \text{nums}[j]$ ，并且 i 和 j 的差的 绝对值 至多为 k 。

示例 1: 输入: $\text{nums} = [1,2,3,1]$, $k = 3$ 输出: true

示例 2: 输入: $\text{nums} = [1,0,1,1]$, $k = 1$ 输出: true

示例 3: 输入: $\text{nums} = [1,2,3,1,2,3]$, $k = 2$ 输出: false

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	哈希辅助遍历	$O(n)$	$O(n)$
02	哈希表+滑动窗口	$O(n)$	$O(n)$

```

#
func containsNearbyDuplicate(nums []int, k int) bool {
    m := make(map[int]int)

    for i, n := range nums {
        if m[n] != 0 && (i+1)-m[n] <= k {
            return true
        }
        m[n] = i + 1
    }
    return false
}

#
func containsNearbyDuplicate(nums []int, k int) bool {
    m := make(map[int]int)

    for i, n := range nums {
        if m[n] != 0 {
            return true
        }
        m[n] = i + 1
        if len(m) > k {
            delete(m, nums[i-k])
        }
    }
    return false
}

```

225.用队列实现栈(4)

- 题目

使用队列实现栈的下列操作：

```

push(x) -- 元素 x 入栈
pop() -- 移除栈顶元素
top() -- 获取栈顶元素
empty() -- 返回栈是否为空

```

注意：

你只能使用队列的基本操作-- 也就是 push to back, peek/pop front 和 is empty 这些操作是合法的。

你所使用的语言也许不支持队列。你可以使用 list 或者 deque (双端队列) 只要是标准的队列操作即可。

你可以假设所有操作都是有效的 (例如, 对一个空的栈不会调用 pop 或者

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	使用切片实现	$O(1)$	$O(n)$
02	使用1个list实现	$O(1)$	$O(n)$
03	使用2个list实现	$O(n)$	$O(n)$
04	使用2个双端队列deque实现	$O(n)$	$O(n)$

```

type MyStack struct {
    arr []int
}

func Constructor() MyStack {
    return MyStack{}
}

func (m *MyStack) Push(x int) {
    m.arr = append(m.arr, x)
}

func (m *MyStack) Pop() int {
    if len(m.arr) == 0 {
        return 0
    }
    last := m.arr[len(m.arr)-1]
    m.arr = m.arr[0 : len(m.arr)-1]
    return last
}

func (m *MyStack) Top() int {
    if len(m.arr) == 0 {
        return 0
    }
    return m.arr[len(m.arr)-1]
}

func (m *MyStack) Empty() bool {
    if len(m.arr) == 0 {
        return true
    }
    return false
}

# 使用1个list实现
type MyStack struct {
    *list.List
}
func Constructor() MyStack {
    return MyStack{
        list.New(),
    }
}

func (m *MyStack) Push(x int) {
    m.PushBack(x)
}

```

```

func (m *MyStack) Pop() int {
    if m.Len() == 0 {
        return -1
    }
    return m.Remove(m.Back()).(int)
}

func (m *MyStack) Top() int {
    if m.Len() == 0 {
        return -1
    }
    return m.Back().Value.(int)
}

func (m *MyStack) Empty() bool {
    return m.Len() == 0
}

# 使用2个list实现
/*
入栈过程:
1、q1 为空，放入 q2，否则放入 q1
出栈过程:
1、q1为空：依次取出q2中的元素（除了最后一个），并且放入q1中 取出q2中
否则 依次取出q1中的元素（除了最后一个），并且放入q2中 取出q1中的最后-
*/
type MyStack struct {
    l1 *list.List
    l2 *list.List
}

func Constructor() MyStack {
    return MyStack{
        l1: list.New(),
        l2: list.New(),
    }
}

func (m *MyStack) Push(x int) {
    if m.l1.Len() == 0 {
        m.l2.PushBack(x)
    } else {
        m.l1.PushBack(x)
    }
}

func (m *MyStack) Pop() int {
}

```

```

var top int
if m.l1.Len() > 0 {
    for m.l1.Len() > 1 {
        m.l2.PushBack(m.l1.Remove(m.l1.Front()))
    }
    top = m.l1.Remove(m.l1.Front()).(int)
} else {
    for m.l2.Len() > 1 {
        m.l1.PushBack(m.l2.Remove(m.l2.Front()))
    }
    top = m.l2.Remove(m.l2.Front()).(int)
}
return top
}

func (m *MyStack) Top() int {
    var top int
    if m.l1.Len() > 0 {
        for m.l1.Len() > 1 {
            m.l2.PushBack(m.l1.Remove(m.l1.Front()))
        }
        top = m.l1.Back().Value.(int)
        m.l2.PushBack(m.l1.Remove(m.l1.Front()))
    } else {
        for m.l2.Len() > 1 {
            m.l1.PushBack(m.l2.Remove(m.l2.Front()))
        }
        top = m.l2.Back().Value.(int)
        m.l1.PushBack(m.l2.Remove(m.l2.Front()))
    }
    return top
}

func (m *MyStack) Empty() bool {
    return m.l1.Len() == 0 && m.l2.Len() == 0
}

# 使用2个双端队列deque实现
type MyStack struct {
    l1 *Queue
    l2 *Queue
}

func Constructor() MyStack {
    return MyStack{
        l1: NewQueue(),
        l2: NewQueue(),
    }
}

```

```

        }

    }

    func (m *MyStack) Push(x int) {
        m.l1.Push(x)
    }

    func (m *MyStack) Pop() int {
        if m.l2.Len() == 0 {
            m.l1, m.l2 = m.l2, m.l1
        }

        for m.l2.Len() > 1 {
            m.l1.Push(m.l2.Pop())
        }
        return m.l2.Pop()
    }

    func (m *MyStack) Top() int {
        res := m.Pop()
        m.l1.Push(res)
        return res
    }

    func (m *MyStack) Empty() bool {
        return (m.l1.Len() + m.l2.Len()) == 0
    }

    type Queue struct {
        nums []int
    }

    func NewQueue() *Queue {
        return &Queue{
            nums: []int{},
        }
    }

    func (q *Queue) Push(n int) {
        q.nums = append(q.nums, n)
    }

    func (q *Queue) Pop() int {
        if len(q.nums) == 0 {
            return 0
        }
        res := q.nums[0]
        q.nums = q.nums[1:]
    }
}

```

```

        return res
    }

func (q *Queue) Len() int {
    return len(q.nums)
}

func (q *Queue) IsEmpty() bool {
    return q.Len() == 0
}

```

226. 翻转二叉树(2)

- 题目

翻转一棵二叉树。

示例：

输入：

```

        4
      /   \
     2     7
    / \   / \
   1   3  6   9

```

输出：

```

        4
      /   \
     7     2
    / \   / \
   9   6  3   1

```

备注：

这个问题是受到 Max Howell 的 原问题 启发的：

谷歌：我们90%的工程师使用您编写的软件(Homebrew)，

但是您却无法在面试时在白板上写出翻转二叉树这道题，这太糟糕了。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	O(n)	O(n)
02	迭代	O(n)	O(n)

```

func invertTree(root *TreeNode) *TreeNode {
    if root == nil || (root.Left == nil && root.Right == nil)
        return root
    }
    root.Left, root.Right = invertTree(root.Right), invertTree(root.Left)
    return root
}

#
func invertTree(root *TreeNode) *TreeNode {
    if root == nil {
        return root
    }

    var queue []*TreeNode
    queue = append(queue, root)
    for len(queue) > 0 {
        node := queue[0]
        queue = queue[1:]
        node.Left, node.Right = node.Right, node.Left
        if node.Left != nil {
            queue = append(queue, node.Left)
        }
        if node.Right != nil {
            queue = append(queue, node.Right)
        }
    }
    return root
}

```

231.2的幂(3)

- 题目

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

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	迭代	O(log(n))	O(1)
02	位运算	O(1)	O(1)
03	递归	O(log(n))	O(log(n))

```

func isPowerOfTwo(n int) bool {
    if n < 1 {
        return false
    }
    for n > 1 {
        if n%2 == 1 {
            return false
        }
        n = n / 2
    }
    return true
}

#
func isPowerOfTwo(n int) bool {
    if n < 1 {
        return false
    }
    return n & (n-1) == 0
}

#
func isPowerOfTwo(n int) bool {
    if n < 1 {
        return false
    }
    if n == 1{
        return true
    }
    if n % 2 != 0{
        return false
    }
    return isPowerOfTwo(n/2)
}

```

232.用栈实现队列(3)

- 题目

使用栈实现队列的下列操作：

```
push(x) -- 将一个元素放入队列的尾部。
pop() -- 从队列头部移除元素。
peek() -- 返回队列头部的元素。
empty() -- 返回队列是否为空。
```

示例：

```
MyQueue queue = new MyQueue();
queue.push(1);
queue.push(2);
queue.peek(); // 返回 1
queue.pop(); // 返回 1
queue.empty(); // 返回 false
```

说明：

你只能使用标准的栈操作 --- 也就是只有 `push to top`, `peek/pop from` 和 `is empty` 操作是合法的。

你所使用的语言也许不支持栈。你可以使用 `list` 或者 `deque` (双端队列) 来假设所有操作都是有效的 (例如, 一个空的队列不会调用 `pop` 或者 `peek` 操

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	使用切片实现	O(1)	O(n)
02	使用2个栈实现	O(n)	O(n)
03	使用2个切片实现	O(n)	O(n)

```

type MyQueue struct {
    a []int
}

func Constructor() MyQueue {
    return MyQueue{}
}

func (m *MyQueue) Push(x int) {
    m.a = append(m.a, x)
}

func (m *MyQueue) Pop() int {
    if len(m.a) == 0 {
        return 0
    }
    first := m.a[0]
    m.a = m.a[1:]
    return first
}

func (m *MyQueue) Peek() int {
    if len(m.a) == 0 {
        return 0
    }
    return m.a[0]
}

func (m *MyQueue) Empty() bool {
    if len(m.a) == 0 {
        return true
    }
    return false
}

# 使用2个栈实现
/*
入队：直接入栈a
出队：栈b为空，则把栈a中全部数据出栈进入栈b，然后出栈b，不为空直接出栈
*/
type MyQueue struct {
    a, b *Stack
}

func Constructor() MyQueue {
    return MyQueue{
        a: NewStack(),
        b: NewStack(),
}

```

```

        }

    }

    func (m *MyQueue) Push(x int) {
        m.a.Push(x)
    }

    func (m *MyQueue) Pop() int {
        if m.b.Len() == 0 {
            for m.a.Len() > 0 {
                m.b.Push(m.a.Pop())
            }
        }
        return m.b.Pop()
    }

    func (m *MyQueue) Peek() int {
        res := m.Pop()
        m.b.Push(res)
        return res
    }

    func (m *MyQueue) Empty() bool {
        return m.a.Len() == 0 && m.b.Len() == 0
    }

    type Stack struct {
        nums []int
    }

    func NewStack() *Stack {
        return &Stack{
            nums: []int{},
        }
    }

    func (s *Stack) Push(n int) {
        s.nums = append(s.nums, n)
    }

    func (s *Stack) Pop() int {
        res := s.nums[len(s.nums)-1]
        s.nums = s.nums[:len(s.nums)-1]
        return res
    }

    func (s *Stack) Len() int {
        return len(s.nums)
    }
}

```

```

}

func (s *Stack) IsEmpty() bool {
    return s.Len() == 0
}

# 使用2个切片实现
type MyQueue struct {
    a []int
    b []int
}

func Constructor() MyQueue {
    return MyQueue{}
}

func (m *MyQueue) Push(x int) {
    m.a = append(m.a, x)
}

func (m *MyQueue) Pop() int {
    m.Peek()
    temp := m.b[len(m.b)-1]
    m.b = m.b[:len(m.b)-1]
    return temp
}

func (m *MyQueue) Peek() int {
    if len(m.b) == 0 {
        for len(m.a) > 0 {
            m.b = append(m.b, m.a[len(m.a)-1])
            m.a = m.a[:len(m.a)-1]
        }
    }
    if len(m.b) == 0 {
        return -1
    }
    return m.b[len(m.b)-1]
}

func (m *MyQueue) Empty() bool {
    return len(m.a) == 0 && len(m.b) == 0
}

```

234.回文链表(4)

- 题目

请判断一个链表是否为回文链表。

示例 1: 输入: 1->2 输出: false

示例 2: 输入: 1->2->2->1 输出: true

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	哈希辅助遍历	$O(n)$	$O(n)$
02	快慢指针反转链表	$O(n)$	$O(1)$
03	栈辅助	$O(n)$	$O(n)$
04	递归	$O(n)$	$O(n)$

```

func isPalindrome(head *ListNode) bool {
    m := make([]int, 0)
    for head != nil {
        m = append(m, head.Val)
        head = head.Next
    }
    i, j := 0, len(m)-1
    for i < j {
        if m[i] != m[j] {
            return false
        }
        i++
        j--
    }
    return true
}

#
func isPalindrome(head *ListNode) bool {
    fast, slow := head, head
    for fast != nil && fast.Next != nil {
        fast = fast.Next.Next
        slow = slow.Next
    }
    var pre *ListNode
    cur := slow
    for cur != nil{
        next := cur.Next
        cur.Next = pre
        pre = cur
        cur = next
    }
    for pre != nil{
        if head.Val != pre.Val{
            return false
        }
        pre = pre.Next
        head = head.Next
    }
    return true
}

#
func isPalindrome(head *ListNode) bool {
    m := make([]int, 0)
    temp := head
    for temp != nil {
        m = append(m, temp.Val)

```

```

        temp = temp.Next
    }
    for head != nil {
        val := m[len(m)-1]
        m = m[:len(m)-1]
        if head.Val != val {
            return false
        }
        head = head.Next
    }
    return true
}

#
var p *ListNode
func isPalindrome(head *ListNode) bool {
    if head == nil{
        return true
    }
    if p == nil{
        p = head
    }
    if isPalindrome(head.Next) && (p.Val == head.Val){
        p = p.Next
        return true
    }
    p = nil
    return false
}

```

235.二叉搜索树的最近公共祖先(2)

- 题目

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 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： $\text{root} = [6, 2, 8, 0, 4, 7, 9, \text{null}, \text{null}, 3, 5]$

示例 1: 输入: $\text{root} = [6, 2, 8, 0, 4, 7, 9, \text{null}, \text{null}, 3, 5]$, $p = 2$, $q = 8$
解释: 节点 2 和节点 8 的最近公共祖先是 6。

示例 2: 输入: $\text{root} = [6, 2, 8, 0, 4, 7, 9, \text{null}, \text{null}, 3, 5]$, $p = 2$, $q = 4$
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值大于等于左子树所有节点值，小于等于右子树所有节点值。

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。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(\log(n))$	$O(\log(n))$
02	迭代	$O(\log(n))$	$O(1)$

```

func lowestCommonAncestor(root, p, q *TreeNode) *TreeNode {
    if p.Val > root.Val && q.Val > root.Val{
        return lowestCommonAncestor(root.Right, p, q)
    }else if p.Val < root.Val && q.Val < root.Val{
        return lowestCommonAncestor(root.Left, p, q)
    }else {
        return root
    }
}

#
func lowestCommonAncestor(root, p, q *TreeNode) *TreeNode {
    for root != nil{
        if p.Val > root.Val && q.Val > root.Val{
            root = root.Right
        }else if p.Val < root.Val && q.Val < root.Val{
            root = root.Left
        }else {
            return root
        }
    }
    return nil
}

```

237.删除链表中的节点(1)

- 题目

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，该链表变为：

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，该链表变为：

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。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	把当前节点替换成下一个节点	O(1)	O(1)

```
func deleteNode(node *ListNode) {
    node.Val = node.Next.Val
    node.Next = node.Next.Next
}
```

242.有效的字母异位词(2)

- 题目

给定两个字符串 s 和 t，编写一个函数来判断 t 是否是 s 的字母异位词。

示例 1: 输入: s = "anagram", t = "nagaram" 输出: true

示例 2: 输入: s = "rat", t = "car" 输出: false

说明:你可以假设字符串只包含小写字母。

进阶:如果输入字符串包含 unicode 字符怎么办? 你能否调整你的解法来应对?

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	哈希辅助	O(n)	O(1)
02	排序比较	O(nlog(n))	O(n)

```

func isAnagram(s string, t string) bool {
    if len(s) != len(t) {
        return false
    }

    sr := []rune(s)
    tr := []rune(t)

    rec := make(map[rune]int, len(sr))
    for i := range sr {
        rec[sr[i]]++
        rec[tr[i]]--
    }

    for _, n := range rec {
        if n != 0 {
            return false
        }
    }
    return true
}

#
func isAnagram(s string, t string) bool {
    if len(s) != len(t) {
        return false
    }
    sArr := make([]int, len(s))
    tArr := make([]int, len(t))
    for i := 0; i < len(s); i++ {
        sArr[i] = int(s[i] - 'a')
        tArr[i] = int(t[i] - 'a')
    }
    sort.Ints(sArr)
    sort.Ints(tArr)
    for i := 0; i < len(s); i++ {
        if sArr[i] != tArr[i] {
            return false
        }
    }
    return true
}

```

257.二叉树的所有路径(2)

- 题目

给定一个二叉树，返回所有从根节点到叶子节点的路径。

说明：叶子节点是指没有子节点的节点。

示例：

输入：



输出：["1->2->5", "1->3"]

解释：所有根节点到叶子节点的路径为：1->2->5, 1->3

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(\log(n))$
02	迭代	$O(n)$	$O(n)$

```

#
func binaryTreePaths(root *TreeNode) []string {
    if root == nil {
        return nil
    }

    res := make([]string, 0)
    var dfs func(string, *TreeNode)
    dfs = func(pre string, root *TreeNode) {
        if pre == "" {
            pre = strconv.Itoa(root.Val)
        } else {
            pre += "->" + strconv.Itoa(root.Val)
        }

        if root.Left != nil {
            dfs(pre, root.Left)
        }

        if root.Right != nil {
            dfs(pre, root.Right)
        }

        if root.Left == nil && root.Right == nil {
            res = append(res, pre)
        }
    }

    dfs("", root)
    return res
}

#
func binaryTreePaths(root *TreeNode) []string {
    res := make([]string, 0)
    if root == nil {
        return res
    }

    var queue []*TreeNode
    var stringQueue []string
    queue = append(queue, root)
    stringQueue = append(stringQueue, strconv.Itoa(root.Val))
    for len(queue) > 0 {
        node := queue[0]
        path := stringQueue[0]
        queue = queue[1:]
        stringQueue = stringQueue[1:]
        if node.Left == nil && node.Right == nil {

```

```

        res = append(res, path)
    }
    if node.Left != nil {
        queue = append(queue, node.Left)
        stringQueue = append(stringQueue, path+"->" + str)
    }
    if node.Right != nil {
        queue = append(queue, node.Right)
        stringQueue = append(stringQueue, path+"->" + str)
    }
}
return res
}

```

258. 各位相加(4)

- 题目

给定一个非负整数 num，反复将各个位上的数字相加，直到结果为一位数。

示例：输入：38 输出：2

解释：各位相加的过程为： $3 + 8 = 11$, $1 + 1 = 2$ 。由于 2 是一位数，进阶：

你可以不使用循环或者递归，且在 $O(1)$ 时间复杂度内解决这个问题吗？

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	找规律1	$O(1)$	$O(1)$
02	找规律2	$O(1)$	$O(1)$
03	模拟计算-字符串	$O(\log(n))$	$O(1)$
04	模拟计算-递归	$O(\log(n))$	$O(\log(n))$

```

# 找规律1
func addDigits(num int) int {
    if num < 10 {
        return num
    }
    if num%9 == 0 {
        return 9
    }
    return num % 9
}

# 找规律2
func addDigits(num int) int {
    return (num-1)%9 + 1
}

# 模拟计算-字符串
func addDigits(num int) int {
    for num >= 10 {
        num = sumDigits(num)
    }
    return num
}

func sumDigits(num int) int {
    sumVal := 0
    str := strconv.Itoa(num)
    for i := range str {
        sumVal = sumVal + int(str[i]-'0')
    }
    return sumVal
}

# 模拟计算-递归
func addDigits(num int) int {
    sum := 0
    for num != 0 {
        sum = sum + num%10
        num = num / 10
    }
    if sum/10 == 0 {
        return sum
    }
    return addDigits(sum)
}

```

263.丑数(2)

- 题目

丑数就是只包含质因数 $2, 3, 5$ 的正整数。

示例 1: 输入: 6 输出: true 解释: $6 = 2 \times 3$

示例 2: 输入: 8 输出: true 解释: $8 = 2 \times 2 \times 2$

示例 3: 输入: 14 输出: false 解释: 14 不是丑数, 因为它包含了另外-

说明:

1 是丑数。

输入不会超过 32 位有符号整数的范围: $[-2^{31}, 2^{31} - 1]$ 。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(\log(n))$	$O(\log(n))$
02	迭代	$O(\log(n))$	$O(1)$

```

func isUgly(num int) bool {
    if num <= 0 {
        return false
    }
    if num <= 6 {
        return true
    }
    if num%2 == 0 {
        return isUgly(num / 2)
    }
    if num%3 == 0 {
        return isUgly(num / 3)
    }
    if num%5 == 0 {
        return isUgly(num / 5)
    }
    return false
}

# 迭代
func isUgly(num int) bool {
    if num <= 0 {
        return false
    }
    for num != 1 {
        if num%2 == 0 {
            num = num / 2
        } else if num%3 == 0 {
            num = num / 3
        } else if num%5 == 0 {
            num = num / 5
        } else {
            return false
        }
    }
    return true
}

```

268. 缺失数字(5)

- 题目

给定一个包含 $0, 1, 2, \dots, n$ 中 n 个数的序列，找出 $0 \dots n$ 中没有^t

示例 1: 输入: [3, 0, 1] 输出: 2

示例 2: 输入: [9, 6, 4, 2, 3, 5, 7, 0, 1] 输出: 8

说明: 你的算法应具有线性时间复杂度。你能否仅使用额外常数空间来实现?

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	数学计算	$O(n)$	$O(1)$
02	排序遍历)	$O(1)$
03	异或-位运算	$O(n)$	$O(1)$
04	交换排序(就地排序)	$O(n)$	$O(1)$
05	哈希辅助	$O(n)$	$O(n)$

```

func missingNumber(nums []int) int {
    n := len(nums)
    sum := n * (n + 1) / 2
    for i := 0; i < n; i++ {
        sum = sum - nums[i]
    }
    return sum
}

#
func missingNumber(nums []int) int {
    sort.Ints(nums)
    for i := 0; i < len(nums); i++ {
        if nums[i] != i {
            return i
        }
    }
    return len(nums)
}

#
func missingNumber(nums []int) int {
    res := 0
    for i := 0; i < len(nums); i++ {
        res = res ^ (i+1) ^ nums[i]
    }
    return res
}

#
func missingNumber(nums []int) int {
    n := len(nums)
    index := n
    for i := 0; i < n; {
        if nums[i] == n{
            index = i
            i++
            continue
        }
        if i == nums[i]{
            i++
            continue
        }
        nums[i], nums[nums[i]] = nums[nums[i]], nums[i]
    }
    return index
}

```

```

#
func missingNumber(nums []int) int {
    n := len(nums)
    // 假设index=n
    index := n
    for i := 0; i < n; {
        // nums[i]到指定位置i后往后走
        if i == nums[i] {
            i++
            continue
        }
        if nums[i] == n {
            index = i
            i++
            continue
        }
        nums[i], nums[nums[i]] = nums[nums[i]], nums[i]
    }
    return index
}

#
func missingNumber(nums []int) int {
    m := make(map[int]bool)
    for i := range nums{
        m[nums[i]] = true
    }
    for i := 0; i <= len(nums); i++{
        if m[i] == false{
            return i
        }
    }
    return 0
}

```

278.第一个错误的版本(2)

- 题目

你是产品经理，目前正在带领一个团队开发新的产品。不幸的是，你的产品的最近一个版本出错了，由于每个版本都是基于之前的版本开发的，所以错误的版本之后的所有版本都是错误的。假设你有 n 个版本 $[1, 2, \dots, n]$ ，你想找出导致之后所有版本出错的第一个错误的版本。你可以通过调用 `bool isBadVersion(version)` 接口来判断版本号 `version` 是否为错误的。实现一个函数来查找第一个错误的版本。你应该尽量减少对调用 API 的次数。

示例：

给定 $n = 5$ ，并且 `version = 4` 是第一个错误的版本。

调用 `isBadVersion(3) -> false`

调用 `isBadVersion(5) -> true`

调用 `isBadVersion(4) -> true`

所以，`4` 是第一个错误的版本。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	二分查找	$O(\log(n))$	$O(1)$
02	二分查找	$O(\log(n))$	$O(1)$

```

func firstBadVersion(n int) int {
    low := 1
    high := n
    for low <= high {
        mid := low + (high-low)/2
        if isBadVersion(mid) == false {
            low = mid + 1
        } else if isBadVersion(mid) == true && isBadVersion(mid+1) == false {
            high = mid - 1
        } else if isBadVersion(mid) == true && isBadVersion(mid+1) == true {
            return mid
        }
    }
    return -1
}

#
func firstBadVersion(n int) int {
    low := 1
    high := n
    for low < high {
        mid := low + (high-low)/2
        if isBadVersion(mid) {
            high = mid
        } else {
            low = mid + 1
        }
    }
    return low
}

```

283. 移动零(3)

- 题目

给定一个数组 `nums`, 编写一个函数将所有 `0` 移动到数组的末尾, 同时保持非零元素相对顺序不变。

示例:

输入: `[0,1,0,3,12]`

输出: `[1,3,12,0,0]`

说明:

必须在原数组上操作, 不能拷贝额外的数组。

尽量减少操作次数。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	前移补零	O(n)	O(1)
02	遇零交换	O(n)	O(1)
03	数组辅助	O(n)	O(n)

```

func moveZeroes(nums []int) {
    length := 0
    for i := 0; i < len(nums); i++ {
        if nums[i] != 0 {
            nums[length] = nums[i]
            length++
        }
    }

    for i := length; i < len(nums); i++ {
        nums[i] = 0
    }
}

#
func moveZeroes(nums []int) {
    length := 0
    for i:= 0; i < len(nums); i++ {
        nums[i], nums[length] = nums[length], nums[i]
        if nums[length] != 0 {
            length++
        }
    }
}

#
func moveZeroes(nums []int) {
    arr := make([]int, len(nums))
    count := 0
    for i := range nums{
        if nums[i] != 0{
            arr[count] = nums[i]
            count++
        }
    }

    copy(nums, arr)
}

```

290. 单词规律(2)

- 题目

给定一种规律 pattern 和一个字符串 str , 判断 str 是否遵循相同的规律

这里的 遵循 指完全匹配,

例如, pattern 里的每个字母和字符串 str 中的每个非空单词之间存在着双

示例1:输入: pattern = "abba", str = "dog cat cat dog"输出: tr

示例 2:输入:pattern = "abba", str = "dog cat cat fish"输出: f

示例 3:输入: pattern = "aaaa", str = "dog cat cat dog"输出: f

示例 4:输入: pattern = "abba", str = "dog dog dog dog" 输出:

说明:

你可以假设 pattern 只包含小写字母, str 包含了由单个空格分隔的小写字

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	双哈希相互映射	O(n)	O(n)
02	单哈希相互映射	O(n)	O(n)

```

func wordPattern(pattern string, str string) bool {
    pa := strings.Split(pattern, "")
    sa := strings.Split(str, " ")
    if len(pa) != len(sa) {
        return false
    }
    length := len(pa)
    pMap := make(map[string]string, length)
    sMap := make(map[string]string, length)

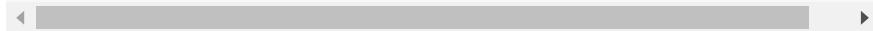
    for i := 0; i < length; i++ {
        pStr, ok := pMap[pa[i]]
        sStr, ok1 := sMap[sa[i]]

        if (ok && pStr != sa[i]) || (ok1 && sStr != pa[i])
            return false
        } else {
            pMap[pa[i]] = sa[i]
            sMap[sa[i]] = pa[i]
        }
    }
    return true
}

func wordPattern(pattern string, str string) bool {
    pa := strings.Split(pattern, "")
    sa := strings.Split(str, " ")
    if len(pa) != len(sa) {
        return false
    }
    return isMatch(pa, sa) && isMatch(sa, pa)
}

func isMatch(pa, sa []string) bool {
    length := len(pa)
    m := make(map[string]string, length)
    for i := 0; i < length; i++ {
        if w, ok := m[pa[i]]; ok && w != sa[i] {
            return false
        } else {
            m[pa[i]] = sa[i]
        }
    }
    return true
}

```



292.Nim 游戏(1)

- 题目

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：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

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

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	推理	O(1)	O(1)

```
func canWinNim(n int) bool {
    // return n&3 != 0
    return n%4 != 0
}
```

299.猜数字游戏(2)

- 题目

你正在和你的朋友玩 猜数字 (Bulls and Cows) 游戏：你写下一个数字让你的朋友猜测。每次他猜测后，你给他一个提示，告诉他有多少位数字和确切位置都猜对了（称为“Bulls”，公牛），有多少位数字猜对了但是位置不对（称为“Cows”，奶牛）。

你的朋友将会根据提示继续猜，直到猜出秘密数字。

请写出一个根据秘密数字和朋友的猜测数返回提示的函数，用 A 表示公牛，用 B 表示奶牛。请注意秘密数字和朋友的猜测数都可能含有重复数字。

示例 1: 输入: secret = "1807", guess = "7810" 输出: "1A3B"

解释: 1 公牛和 3 奶牛。公牛是 8，奶牛是 0, 1 和 7。

示例 2: 输入: secret = "1123", guess = "0111" 输出: "1A1B"

解释: 朋友猜测数中的第一个 1 是公牛，第二个或第三个 1 可被视为奶牛。

说明: 你可以假设秘密数字和朋友的猜测数都只包含数字，并且它们的长度永远相等。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	双哈希辅助	$O(n)$	$O(1)$
02	单哈希辅助	$O(n)$	$O(1)$

```

func getHint(secret string, guess string) string {
    length := len(secret)
    right := 0
    wrongLoc := 0
    m := make(map[byte]int)
    n := make(map[byte]int)
    for i := 0; i < length; i++ {
        if secret[i] == guess[i] {
            right++
        } else {
            m[secret[i]]++
            n[guess[i]]++
        }
    }
    for i := range m {
        if m[i] < n[i] {
            wrongLoc = wrongLoc + m[i]
        } else {
            wrongLoc = wrongLoc + n[i]
        }
    }
}

return fmt.Sprintf("%d%A%dB", right, wrongLoc)
}

#
func getHint(secret string, guess string) string {
    length := len(secret)
    right := 0
    wrongNum := 0
    m := make(map[int]int)
    for i := 0; i < length; i++ {
        if secret[i] == guess[i] {
            right++
        }
        m[int(secret[i]-'0')]++
        m[int(guess[i]-'0')]--
    }
    for i := range m {
        if m[i] > 0{
            wrongNum = wrongNum + m[i]
        }
    }
    // wrongLoc = 总数 - 猜对的数 - 猜错的数
    wrongLoc := length - right - wrongNum
    return fmt.Sprintf("%d%A%dB", right, wrongLoc)
}

```

0001-0100

0301-0400-Easy

- 0301-0400-Easy
 - 303.区域和检索-数组不可变(2)
 - 326.3的幂(3)
 - 342.4的幂(4)
 - 344.反转字符串(3)
 - 345.反转字符串中的元音字母(2)
 - 349.两个数组的交集(3)
 - 350.两个数组的交集 II(3)
 - 367.有效的完全平方数(4)
 - 371.两整数之和(2)
 - 374.猜数字大小(2)
 - 383.赎金信(3)
 - 387.字符串中的第一个唯一字符(2)
 - 389.找不同(5)
 - 392.判断子序列(4)

303.区域和检索-数组不可变(2)

- 题目

给定一个整数数组 `nums`, 求出数组从索引 `i` 到 `j` ($i \leq j$) 范围内元素的示例:

给定 `nums = [-2, 0, 3, -5, 2, -1]`, 求和函数为 `sumRange()`
`sumRange(0, 2) -> 1`
`sumRange(2, 5) -> -1`
`sumRange(0, 5) -> -3`

说明:

你可以假设数组不可变。
 会多次调用 `sumRange` 方法。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	一维前缀和	O(1)	O(n)
02	遍历计算	O(n)	O(1)

```

type NumArray struct {
    arr []int
}

func Constructor(nums []int) NumArray {
    size := len(nums)
    arr := make([]int, size+1)
    for i := 1; i <= size; i++ {
        arr[i] = arr[i-1] + nums[i-1]
    }
    return NumArray{arr: arr}
}

func (n *NumArray) SumRange(i int, j int) int {
    return n.arr[j+1] - n.arr[i]
}

#
type NumArray struct {
    arr []int
}

func Constructor(nums []int) NumArray {
    return NumArray{nums}
}

func (n *NumArray) SumRange(i int, j int) int {
    sum := 0
    for ; i <= j; i++ {
        sum = sum + n.arr[i]
    }
    return sum
}

```

326.3的幂(3)

- 题目

给定一个整数，写一个函数来判断它是否是 3 的幂次方。

示例 1: 输入: 27 输出: true

示例 2: 输入: 0 输出: false

示例 3: 输入: 9 输出: true

示例 4: 输入: 45 输出: false

进阶: 你能不使用循环或者递归来完成本题吗?

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	迭代	$O(1)$	$O(1)$
02	转3进制判断	$O(\log(n))$	$O(1)$
03	递归	$O(\log(n))$	$O(\log(n))$

```

func isPowerOfThree(n int) bool {
    if n <= 0 {
        return false
    }
    for n > 1 {
        if n % 3 != 0{
            return false
        }
        n = n / 3
    }
    return n == 1
}

#
func isPowerOfThree(n int) bool {
    if n <= 0 {
        return false
    }
    str := strconv.FormatInt(int64(n), 3)
    return str[0:1] == "1" && strings.Count(str, "0") == len(str)-1
}

#
func isPowerOfThree(n int) bool {
    if n <= 0 {
        return false
    }
    if n == 1 {
        return true
    }
    if n%3 != 0 {
        return false
    }
    return isPowerOfThree(n / 3)
}

```

342.4的幂(4)

- 题目

给定一个整数（32 位有符号整数），请编写一个函数来判断它是否是 4 的幂次

示例 1：输入：16 输出：true

示例 2：输入：5 输出：false

进阶：你能不使用循环或者递归来完成本题吗？



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	迭代	$O(\log(n))$	$O(1)$
02	递归	$O(\log(n))$	$O(\log(n))$
03	位运算	$O(1)$	$O(1)$
04	转4进制	$O(\log(n))$	$O(1)$

```

func isPowerOfFour(num int) bool {
    if num <= 0 {
        return false
    }

    for num > 1 {
        if num%4 != 0 {
            return false
        }
        num = num / 4
    }
    return num == 1
}

#
func isPowerOfFour(num int) bool {
    if num <= 0 {
        return false
    }
    if num == 1{
        return true
    }
    if num % 4 != 0{
        return false
    }

    return isPowerOfFour(num/4)
}

#
func isPowerOfFour(num int) bool {
    if num <= 0 {
        return false
    }
    // return (num & (num-1) == 0) && (num-1)%3 == 0
    return (num&(num-1) == 0) && (num&0xffffffff == 0)
}

#
func isPowerOfFour(num int) bool {
    if num <= 0 {
        return false
    }
    str := strconv.FormatInt(int64(num), 4)
    return str[0:1] == "1" && strings.Count(str, "0") == len(str)-1
}

```

344. 反转字符串(3)

- 题目

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` 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。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"]

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	双指针	$O(n)$	$O(1)$
02	递归	$O(n)$	$O(n)$
03	单指针	$O(n)$	$O(1)$

```

func reverseString(s []byte) {
    i, j := 0, len(s)-1
    for i < j {
        s[i], s[j] = s[j], s[i]
        i++
        j--
    }
}

#
func reverseString(s []byte) {
    var reverse func(int, int)
    reverse = func(left, right int) {
        if left < right {
            s[left], s[right] = s[right], s[left]
            reverse(left+1, right-1)
        }
    }
    reverse(0, len(s)-1)
}

#
func reverseString(s []byte) {
    for i := 0; i < len(s)/2; i++ {
        s[i], s[len(s)-1-i] = s[len(s)-1-i], s[i]
    }
}

```

345. 反转字符串中的元音字母(2)

- 题目

编写一个函数，以字符串作为输入，反转该字符串中的元音字母。

示例 1: 输入: "hello" 输出: "holle"

示例 2: 输入: "leetcode" 输出: "leotcede"

说明: 元音字母不包含字母"y"。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	双指针	O(n)	O(1)
02	数组辅助替换	O(n)	O(n)

```

func reverseVowels(s string) string {
    bytes := []byte(s)
    length := len(s)
    i, j := 0, length-1
    for i < j {
        if !isvowels(bytes[i]) {
            i++
            continue
        }
        if !isvowels(bytes[j]) {
            j--
            continue
        }
        bytes[i], bytes[j] = bytes[j], bytes[i]
        i++
        j--
    }
    return string(bytes)
}

func isvowels(b byte) bool {
    return b == 'a' || b == 'e' || b == 'i' || b == 'o' ||
           b == 'A' || b == 'E' || b == 'I' || b == 'O' || b ==
}

#
func reverseVowels(s string) string {
    bytes := []byte(s)
    length := len(s)
    temp := make([]byte, 0)
    for i := 0; i < length; i++ {
        if isvowels(bytes[i]) {
            temp = append(temp, bytes[i])
        }
    }
    count := 0
    for i := 0; i < length; i++ {
        if isvowels(bytes[i]) {
            bytes[i] = temp[len(temp)-1-count]
            count++
        }
    }
    return string(bytes)
}

func isvowels(b byte) bool {
    return b == 'a' || b == 'e' || b == 'i' || b == 'o' ||

```

```
b == 'A' || b == 'E' || b == 'I' || b == 'O' || b =
}
```



349. 两个数组的交集(3)

- 题目

给定两个数组，编写一个函数来计算它们的交集。

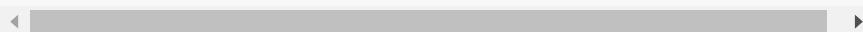
示例 1: 输入: `nums1 = [1,2,2,1], nums2 = [2,2]` 输出: `[2]`

示例 2: 输入: `nums1 = [4,9,5], nums2 = [9,4,9,8,4]` 输出: `[9,4]`

说明:

输出结果中的每个元素一定是唯一的。

我们可以不考虑输出结果的顺序。



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	单哈希辅助	$O(n)$	$O(n)$
02	双哈希辅助	$O(n)$	$O(n)$
03	排序双指针	$O(n \log(n))$	$O(n)$

```

func intersection(nums1 []int, nums2 []int) []int {
    res := make([]int, 0)
    m := make(map[int]int)
    for _, v := range nums1 {
        m[v] = 1
    }
    for _, v := range nums2 {
        if m[v] == 1 {
            res = append(res, v)
            m[v] += 1
        }
    }
    return res
}

#
func intersection(nums1 []int, nums2 []int) []int {
    m1 := make(map[int]bool)
    m2 := make(map[int]bool)
    res := make([]int, 0)
    for _, v := range nums1 {
        m1[v] = true
    }

    for _, v := range nums2 {
        if m1[v] != false {
            m2[v] = true
        }
    }

    for k := range m2 {
        res = append(res, k)
    }
    return res
}

#
func intersection(nums1 []int, nums2 []int) []int {
    sort.Ints(nums1)
    sort.Ints(nums2)
    res := make([]int, 0)
    i := 0
    j := 0
    for i < len(nums1) && j < len(nums2) {
        if nums1[i] < nums2[j] {
            i++
        } else if nums1[i] > nums2[j] {
            j++
        } else {
            res = append(res, nums1[i])
            i++
            j++
        }
    }
    return res
}

```

```

    } else {
        if len(res) == 0 || res[len(res)-1] != nums1[i]
            res = append(res, nums1[i])
        }
        i++
        j++
    }
}
return res
}

```

350.两个数组的交集 II(3)

- 题目

给定两个数组，编写一个函数来计算它们的交集。

示例 1: 输入: nums1 = [1,2,2,1], nums2 = [2,2] 输出: [2,2]

示例 2: 输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4] 输出: [4,9]

说明: 输出结果中每个元素出现的次数，应与元素在两个数组中出现的次数一致。

我们可以不考虑输出结果的顺序。

进阶:

如果给定的数组已经排好序呢？你将如何优化你的算法？

如果 nums1 的大小比 nums2 小很多，哪种方法更优？

如果 nums2 的元素存储在磁盘上，磁盘内存是有限的，并且你不能一次加

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	单哈希辅助	O(n)	O(n)
02	双哈希辅助	O(n)	O(n)
03	排序双指针	O(nlog(n))	O(n)

```

func intersect(nums1 []int, nums2 []int) []int {
    m1 := make(map[int]int)
    res := make([]int, 0)
    for _, v := range nums1 {
        m1[v] += 1
    }

    for _, v := range nums2 {
        if m1[v] > 0 {
            res = append(res, v)
            m1[v]--
        }
    }
    return res
}

#
func intersect(nums1 []int, nums2 []int) []int {
    m1 := make(map[int]int)
    m2 := make(map[int]int)
    res := make([]int, 0)
    for _, v := range nums1 {
        m1[v]++
    }

    for _, v := range nums2 {
        if m1[v] != 0 && m1[v] > m2[v] {
            m2[v]++
        }
    }

    for k := range m2 {
        for i := 0; i < m2[k]; i++ {
            res = append(res, k)
        }
    }
    return res
}

#
func intersect(nums1 []int, nums2 []int) []int {
    sort.Ints(nums1)
    sort.Ints(nums2)
    res := make([]int, 0)
    i := 0
    j := 0
    for i < len(nums1) && j < len(nums2) {
        if nums1[i] < nums2[j] {
            i++
        } else if nums1[i] > nums2[j] {
            j++
        } else {
            res = append(res, nums1[i])
            i++
            j++
        }
    }
    return res
}

```

```

        i++
    } else if nums1[i] > nums2[j] {
        j++
    } else {
        res = append(res, nums1[i])
        i++
        j++
    }
}
return res
}

```

367.有效的完全平方数(4)

- 题目

给定一个正整数 num，编写一个函数，如果 num 是一个完全平方数，则返回 true。

说明：不要使用任何内置的库函数，如 sqrt。

示例 1：输入：16 输出：True

示例 2：输入：14 输出：False

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	二分查找	O(log(n))	O(1)
02	牛顿迭代法	O(log(n))	O(1)
03	数学法	O(n ^{1/2})	O(1)
04	暴力法	O(n ^{1/2})	O(1)

```

func isPerfectSquare(num int) bool {
    if num < 2 {
        return true
    }
    left := 2
    right := num / 2
    for left <= right {
        mid := left + (right-left)/2
        if mid*mid == num {
            return true
        } else if mid*mid > num {
            right = mid - 1
        } else {
            left = mid + 1
        }
    }
    return false
}

#
func isPerfectSquare(num int) bool {
    if num < 2 {
        return true
    }
    x := num / 2
    for x*x > num {
        x = (x + num/x) / 2
    }
    return x*x == num
}

#
func isPerfectSquare(num int) bool {
    i := 1
    for num > 0 {
        num = num - i
        i = i + 2
    }
    return num == 0
}

#
func isPerfectSquare(num int) bool {
    i := 1
    for i * i < num{
        i++
    }
}

```

```
        return i * i == num
    }
```

371.两整数之和(2)

- 题目

不使用运算符 + 和 - , 计算两整数a, b之和。

示例 1: 输入: a = 1, b = 2 输出: 3

示例 2: 输入: a = -2, b = 3 输出: 1

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	迭代	O(1)	O(1)
02	递归	O(1)	O(1)

```
func getSum(a int, b int) int {
    for b != 0 {
        a, b = a^b, (a&b)<<1
    }
    return a
}

# 
func getSum(a int, b int) int {
    if b == 0 {
        return a
    }
    return getSum(a^b, (a&b)<<1)
}
```

374.猜数字大小(2)

- 题目

我们正在玩一个猜数字游戏。 游戏规则如下：
我从 1 到 n 选择一个数字。 你需要猜我选择了哪个数字。
每次你猜错了，我会告诉你这个数字是大了还是小了。
你调用一个预先定义好的接口 `guess(int num)`，它会返回 3 个可能的结果
-1 : 我的数字比较小
1 : 我的数字比较大
0 : 恭喜！你猜对了！

示例：输入：n = 10, pick = 6 输出：6

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	二分查找	$O(\log(n))$	$O(1)$
02	递归	$O(\log(n))$	$O(\log(n))$

```

func guessNumber(n int) int {
    low := 1
    high := n
    for low < high{
        mid := low + (high-low)/2
        if guess(mid) == 0{
            return mid
        }else if guess(mid) == 1{
            low = mid + 1
        }else {
            high = mid - 1
        }
    }
    return low
}

#
func guessNumber(n int) int {
    return binary(1, n)
}

func binary(left, right int) int {
    mid := left + (right-left)/2
    if guess(mid) == 1 {
        return binary(mid+1, right)
    } else if guess(mid) == -1 {
        return binary(left, mid-1)
    } else {
        return mid
    }
}

```

383. 贼金信(3)

- 题目

给定一个赎金信 (`ransom`) 字符串和一个杂志(`magazine`)字符串，
判断第一个字符串 `ransom` 能不能由第二个字符串 `magazines` 里面的字符构成。
如果可以构成，返回 `true` ；否则返回 `false`。

(题目说明：为了不暴露赎金信字迹，要从杂志上搜索各个需要的字母，组成单词。
杂志字符串中的每个字符只能在赎金信字符串中使用一次。)

注意：你可以假设两个字符串均只含有小写字母。

```
canConstruct("a", "b") -> false  
canConstruct("aa", "ab") -> false  
canConstruct("aa", "aab") -> true
```

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	数组辅助	$O(n)$	$O(1)$
02	哈希辅助	$O(n)$	$O(1)$
03	排序双指针	$O(n \log(n))$	$O(n)$

```

func canConstruct(ransomNote string, magazine string) bool
    index := [26]int{}
    for i := 0; i < len(magazine); i++ {
        index[magazine[i]-'a']++
    }

    for i := 0; i < len(ransomNote); i++ {
        index[ransomNote[i]-'a']--
        if index[ransomNote[i]-'a'] < 0 {
            return false
        }
    }
    return true
}

#
func canConstruct(ransomNote string, magazine string) bool
    index := make(map[uint8]int)
    for i := 0; i < len(magazine); i++ {
        index[magazine[i]-'a']++
    }

    for i := 0; i < len(ransomNote); i++ {
        index[ransomNote[i]-'a']--
        if index[ransomNote[i]-'a'] < 0 {
            return false
        }
    }
    return true
}

#
func canConstruct(ransomNote string, magazine string) bool
    ransomNoteArr := strings.Split(ransomNote, "")
    magazineArr := strings.Split(magazine, "")
    sort.Strings(ransomNoteArr)
    sort.Strings(magazineArr)

    i := 0
    j := 0
    for i < len(ransomNoteArr) && j < len(magazineArr) {
        if ransomNoteArr[i] > magazineArr[j] {
            j++
        } else if ransomNoteArr[i] < magazineArr[j] {
            return false
        } else {
            i++
            j++
        }
    }
}

```

```

        }
    }
    return i == len(ransomNote)
}

```

387. 字符串中的第一个唯一字符(2)

- 题目

给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在案例：

s = "leetcode" 返回 0.
s = "loveleetcode", 返回 2.

注意事项：您可以假定该字符串只包含小写字母。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	数组辅助	O(n)	O(1)
02	哈希辅助	O(n)	O(1)

```

func firstUniqChar(s string) int {
    m := [26]int{}
    for i := 0; i < len(s); i++ {
        m[s[i]-'a']++
    }
    for i := 0; i < len(s); i++ {
        if m[s[i]-'a'] == 1 {
            return i
        }
    }
    return -1
}

#
func firstUniqChar(s string) int {
    m := make(map[uint8]int)
    for i := 0; i < len(s); i++ {
        m[s[i]-'a']++
    }
    for i := 0; i < len(s); i++ {
        if m[s[i]-'a'] == 1 {
            return i
        }
    }
    return -1
}

```

389. 找不同(5)

- 题目

给定两个字符串 s 和 t，它们只包含小写字母。
 字符串 t 由字符串 s 随机重排，然后在随机位置添加一个字母。
 请找出在 t 中被添加的字母。
 示例：输入：s = "abcd"t = "abcde"输出：e
 解释：'e' 是那个被添加的字母。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	数组辅助	$O(n)$	$O(1)$
02	哈希辅助	$O(n)$	$O(1)$
03	位计算	$O(n)$	$O(1)$
04	数学计算	$O(n)$	$O(1)$
05	排序遍历	$O(n\log(n))$	$O(1)$

```

func findTheDifference(s string, t string) byte {
    m := [26]int{}
    bytest := []byte(t)
    bytess := []byte(s)
    for _, v := range bytest {
        m[v-'a']++
    }
    for _, v := range bytess {
        m[v-'a']--
    }
    for k, _ := range m {
        if m[k] == 1 {
            return byte(k + 'a')
        }
    }
    return 0
}

#
func findTheDifference(s string, t string) byte {
    m := make(map[byte]int)
    bytest := []byte(t)
    bytess := []byte(s)
    for _, v := range bytest {
        m[v]++
    }
    for _, v := range bytess {
        m[v]--
    }
    for k, _ := range m {
        if m[k] == 1 {
            return k
        }
    }
    return 0
}

#
func findTheDifference(s string, t string) byte {
    ch := byte(0)
    for _, value := range s {
        ch ^= byte(value)
    }
    for _, value := range t {
        ch ^= byte(value)
    }
    return ch
}

```

```

#
func findTheDifference(s string, t string) byte {
    ch := byte(0)
    for _, value := range t {
        ch += byte(value)
    }
    for _, value := range s {
        ch -= byte(value)
    }
    return ch
}

#
func findTheDifference(s string, t string) byte {
    sArr := strings.Split(s, "")
    tArr := strings.Split(t, "")
    sort.Strings(sArr)
    sort.Strings(tArr)
    for i := 0; i < len(sArr); i++{
        if sArr[i] != tArr[i]){
            return []byte(tArr[i])[0]
        }
    }
    return []byte(tArr[len(tArr)-1])[0]
}

```

392. 判断子序列(4)

- 题目

给定字符串 s 和 t ，判断 s 是否为 t 的子序列。

你可以认为 s 和 t 中仅包含英文小写字母。

字符串 t 可能会很长（长度 $\sim 500,000$ ），而 s 是个短字符串（长度 ≤ 100 ）。

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余！
(例如，“ace”是“abcde”的一个子序列，而“aec”不是）。

示例 1: $s = "abc"$, $t = "ahbgdc"$ 返回 true.

示例 2: $s = "axc"$, $t = "ahbgdc"$ 返回 false.

后续挑战：

如果有大量输入的 S ，称作 S_1, S_2, \dots, S_k 其中 $k \geq 10$ 亿，

你需要依次检查它们是否为 T 的子序列。在这种情况下，你会怎样改变代码？



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	双指针	$O(n)$	$O(1)$
02	单指针遍历	$O(n^2)$	$O(1)$
03	二分查找	$O(n \log(n))$	$O(1)$
04	动态规划	$O(n^2)$	$O(n^2)$

```

func isSubsequence(s string, t string) bool {
    if len(s) > len(t){
        return false
    }
    i := 0
    j := 0
    for i < len(s) && j < len(t){
        if s[i] == t[j]){
            i++
        }
        j++
    }
    return i == len(s)
}

#
func isSubsequence(s string, t string) bool {
    for _, v := range s{
        idx := strings.IndexRune(t, v)
        if idx == -1{
            return false
        }
        t = t[idx+1:]
    }
    return true
}

#
func isSubsequence(s string, t string) bool {
    m := make(map[uint8][]int)
    for i := 0; i < len(t); i++ {
        value := t[i] - 'a'
        if m[value] == nil {
            m[value] = make([]int, 0)
        }
        m[value] = append(m[value], i)
    }
    prev := -1
    for i := 0; i < len(s); i++ {
        value := s[i] - 'a'
        left := 0
        right := len(m[value]) - 1
        if len(m[value]) == 0 {
            return false
        }
        for left < right {
            mid := left + (right-left)/2
            if m[value][mid] > prev {

```

```

        right = mid
    } else {
        left = mid + 1
    }
}
if left > right || m[value][left] <= prev {
    return false
}
prev = m[value][left]
}
return true
}

#
/*
状态定义: dp[i][j] 表示长度为i的字符串s是否为长度为j的字符串t的子序列
状态转移方程: 如果s[i] == t[j], 则dp[i][j] = dp[i-1][j-1]
如果s[i] != t[j], 则dp[i][j] = dp[i][j-1]
初始: dp[0][j] = true 表示空串s 是任意长度串t的子串
dp[i][0] = false 表示任意长度非空串s 不是空串t的子串
dp[i][0] = false 表示任意长度非空串s 不是空串t的子串
*/
func isSubsequence(s string, t string) bool {
    if len(s) == 0 {
        return true
    } else if len(s) > len(t) {
        return false
    }

    dp := make([][]bool, len(s)+1)
    for i := 0; i < len(s)+1; i++ {
        dp[i] = make([]bool, len(t)+1)
        dp[i][0] = false
    }
    for i := 0; i <= len(t); i++ {
        dp[0][i] = true
    }

    for i := 1; i <= len(s); i++ {
        for j := 1; j <= len(t); j++ {
            if s[i-1] == t[j-1] {
                dp[i][j] = dp[i-1][j-1]
            } else {
                dp[i][j] = dp[i][j-1]
            }
        }
    }
}

```

```
    return dp[len(s)][len(t)]  
}
```



0401-0500-Easy

- 0401-0500-Easy
 - 401.二进制手表(3)
 - 404.左叶子之和(2)
 - 405.数字转换为十六进制数(2)
 - 409.最长回文串(2)
 - 412.Fizz Buzz(1)
 - 414.第三大的数(2)
 - 415.字符串相加(2)
 - 434.字符串中的单词数(2)
 - 437.路径总和III(4)
 - 441.排列硬币(3)
 - 443.压缩字符串(1)
 - 447.回旋镖的数量(1)
 - 448.找到所有数组中消失的数字(3)
 - 453.最小移动次数使数组元素相等(2)
 - 455.分发饼干(1)
 - 459.重复的子字符串(2)
 - 461.汉明距离(3)
 - 463.岛屿的周长(3)
 - 475.供暖器(2)
 - 476.数字的补数(3)
 - 482.密钥格式化(2)
 - 485.最大连续1的个数(2)
 - 492.构造矩形(1)
 - 496.下一个更大元素 I(3)
 - 500.键盘行(4)

401.二进制手表(3)

- 题目

二进制手表顶部有 4 个 LED 代表小时（0-11），底部的 6 个 LED 代表分钟。每个 LED 代表一个 0 或 1，最低位在右侧。

例如，上面的二进制手表读取 “3:25”。

给定一个非负整数 n 代表当前 LED 亮着的数量，返回所有可能的时间。

案例：输入：n = 1

返回：["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:04"]

注意事项：

输出的顺序没有要求。

小时不会以零开头，比如 “01:00” 是不允许的，应为 “1:00”。

分钟必须由两位数组成，可能会以零开头，比如 “10:2” 是无效的，应为 “10:02”。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	暴力法	O(1)	O(1)
02	暴力法	O(1)	O(1)
03	回溯法	O(2^n)	O(n)

```

func binCount(num int) int {
    count := make([]int, 0)
    for num != 0 {
        temp := num % 2
        count = append(count, temp)
        num = num / 2
    }
    countNum := 0
    for i := 0; i < len(count); i++ {
        if count[i] == 1 {
            countNum++
        }
    }
    return countNum
}

func readBinaryWatch(num int) []string {
    res := make([]string, 0)
    for i := 0; i < 12; i++ {
        for j := 0; j < 60; j++ {
            if binCount(i)+binCount(j) == num {
                res = append(res, fmt.Sprintf("%d:%02d", i,
                j))
            }
        }
    }
    return res
}

#
func readBinaryWatch(num int) []string {
    res := make([]string, 0)
    for i := 0; i < 12; i++ {
        for j := 0; j < 60; j++ {
            hour := fmt.Sprintf("%b", i)
            minute := fmt.Sprintf("%b", j)
            if strings.Count(hour, "1")+strings.Count(minute, "1") == num {
                res = append(res, fmt.Sprintf("%d:%02d", i,
                j))
            }
        }
    }
    return res
}

#
func readBinaryWatch(num int) []string {
    res := make([]string, 0)
    ledS := make([]bool, 10)

```

```

var dfs func(int, int)
dfs = func(idx, num int) {
    if num == 0 {
        // 满足条件
        m, h := get(leds[:6]), get(leds[6:])
        if h < 12 && m < 60 {
            res = append(res, fmt.Sprintf("%d:%02d", h,
        }
        return
    }
    for i := idx; i < 11-num; i++ {
        ledS[i] = true
        dfs(i+1, num-1)
        ledS[i] = false
    }
}
dfs(0, num)
return res
}

func get(leds []bool) int {
    bs := []int{1, 2, 4, 8, 16, 32}
    var sum int
    size := len(leds)
    for i := 0; i < size; i++ {
        if ledS[i] {
            sum += bs[i]
        }
    }
    return sum
}

```

404. 左叶子之和(2)

- 题目

计算给定二叉树的所有左叶子之和。

示例：

```

      3
     / \
    9   20
     /   \
    15   7

```

在这个二叉树中，有两个左叶子，分别是 9 和 15，所以返回 24

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	O(n)	O(log(n))
02	迭代	O(n)	O(n)

```

func sumOfLeftLeaves(root *TreeNode) int {
    if root == nil {
        return 0
    }
    if root.Left == nil {
        return sumOfLeftLeaves(root.Right)
    }
    if root.Left.Left == nil && root.Left.Right == nil {
        return root.Left.Val + sumOfLeftLeaves(root.Right)
    }
    return sumOfLeftLeaves(root.Left) + sumOfLeftLeaves(ro
}

# 
func sumOfLeftLeaves(root *TreeNode) int {
    sum := 0
    if root == nil{
        return 0
    }
    queue := make([]*TreeNode,0)
    queue = append(queue, root)
    for len(queue) > 0{
        node := queue[0]
        queue = queue[1:]
        if node.Left != nil && node.Left.Left == nil && no
            sum = sum + node.Left.Val
        }
        if node.Left != nil{
            queue = append(queue, node.Left)
        }
        if node.Right != nil{
            queue = append(queue, node.Right)
        }
    }

    return sum
}

```

405. 数字转换为十六进制数(2)

- 题目

给定一个整数，编写一个算法将这个数转换为十六进制数。对于负整数，我们通常注意：

十六进制中所有字母(a-f)都必须是小写。

十六进制字符串中不能包含多余的前导零。

如果要转化的数为0，那么以单个字符'0'来表示；对于其他情况，十六进制字符串中的数确保在32位有符号整数范围内。

不能使用任何由库提供的将数字直接转换或格式化为十六进制的方法。

示例 1： 输入：26 输出："1a"

示例 2： 输入：-1 输出："ffffffff"

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	位运算	O(1)	O(1)
02	遍历	O(1)	O(1)

```

var h = []string{
    "0", "1", "2", "3", "4", "5", "6", "7",
    "8", "9", "a", "b", "c", "d", "e", "f",
}

func toHex(num int) string {
    hex := ""
    if num == 0 {
        return "0"
    }

    for i := 0; i < 8 && num != 0; i++ {
        hex = h[num&15] + hex
        num = num >> 4
    }
    return hex
}

#
var h = []string{
    "0", "1", "2", "3", "4", "5", "6", "7",
    "8", "9", "a", "b", "c", "d", "e", "f",
}

func toHex(num int) string {
    res := ""
    if num == 0{
        return "0"
    }
    if num < 0 {
        num = num + 4294967296
    }

    for num != 0{
        temp := num % 16
        res = h[temp] + res
        num = num / 16
    }
    return res
}

```

409.最长回文串(2)

- 题目

给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造成的最长的回文子串。
在构造过程中，请注意区分大小写。比如 "Aa" 不能当做一个回文字符串。

注意：假设字符串的长度不会超过 1010。

示例 1：输入："abccccdd" 输出：7

解释：我们可以构造的最长的回文串是"dccaccd"，它的长度是 7。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	数组辅助	$O(n)$	$O(1)$
02	哈希辅助	$O(n)$	$O(1)$

```

func longestPalindrome(s string) int {
    ret := 0
    a := [123]int{}
    for i := range s {
        a[s[i]]++
    }
    hasOdd := 0
    for i := range a {
        if a[i] == 0 {
            continue
        }
        if a[i] % 2 == 0 {
            ret += a[i]
        } else {
            ret += a[i] - 1
            hasOdd = 1
        }
    }
    return ret + hasOdd
}

#
func longestPalindrome(s string) int {
    ret := 0
    a := make(map[byte]int)
    for i := range s {
        a[s[i]]++
    }
    hasOdd := 0
    for i := range a {
        if a[i] == 0 {
            continue
        }
        if a[i]%2 == 0 {
            ret += a[i]
        } else {
            ret += a[i] - 1
            hasOdd = 1
        }
    }
    return ret + hasOdd
}

```

412.Fizz Buzz(1)

- 题目

写一个程序，输出从 1 到 n 数字的字符串表示。

1. 如果 n 是3的倍数，输出“Fizz”；
2. 如果 n 是5的倍数，输出“Buzz”；
3. 如果 n 同时是3和5的倍数，输出 “FizzBuzz”。

示例：n = 15，

返回：

```
[
    "1",
    "2",
    "Fizz",
    "4",
    "Buzz",
    "Fizz",
    "7",
    "8",
    "Fizz",
    "Buzz",
    "11",
    "Fizz",
    "13",
    "14",
    "FizzBuzz"
]
```

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历	$O(n)$	$O(1)$

```

func fizzBuzz(n int) []string {
    ret := make([]string, n)
    for i := range ret {
        x := i + 1
        switch {
        case x%15 == 0:
            ret[i] = "FizzBuzz"
        case x%5 == 0:
            ret[i] = "Buzz"
        case x%3 == 0:
            ret[i] = "Fizz"
        default:
            ret[i] = strconv.Itoa(x)
        }
    }
    return ret
}

```

414. 第三大的数(2)

- 题目

给定一个非空数组，返回此数组中第三大的数。如果不存在，则返回数组中最大的数。

示例 1: 输入: [3, 2, 1] 输出: 1

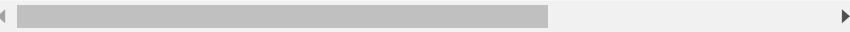
解释: 第三大的数是 1.

示例 2: 输入: [1, 2] 输出: 2

解释: 第三大的数不存在，所以返回最大的数 2 .

示例 3: 输入: [2, 2, 3, 1] 输出: 1

解释: 注意，要求返回第三大的数，是指第三大且唯一出现的数。存在两个值为



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历	O(n)	O(1)
02	排序遍历	O(nlog(n))	O(1)

```

func thirdMax(nums []int) int {
    max1, max2, max3 := math.MinInt64, math.MinInt64, math.
    for _, n := range nums {
        if n == max1 || n == max2 {
            continue
        }
        switch {
        case max1 < n:
            max1, max2, max3 = n, max1, max2
        case max2 < n:
            max2, max3 = n, max2
        case max3 < n:
            max3 = n
        }
    }

    if max3 == math.MinInt64 {
        return max1
    }
    return max3
}

#
func thirdMax(nums []int) int {

    sort.Ints(nums)
    if len(nums) < 3 {
        return nums[len(nums)-1]
    }

    k := 2
    maxValue := nums[len(nums)-1]
    for i := len(nums) - 2; i >= 0; i-- {
        if nums[i] != nums[i+1] {
            k--
        }
        if k == 0 {
            return nums[i]
        }
    }
    return maxValue
}

```

415. 字符串相加(2)

- 题目

给定两个字符串形式的非负整数 num1 和num2 ，计算它们的和。

注意：

num1 和num2 的长度都小于 5100.

num1 和num2 都只包含数字 0-9.

num1 和num2 都不包含任何前导零。

你不能使用任何內建 BigInteger 库，也不能直接将输入的字符串转换为整数。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	模拟遍历	$O(n)$	$O(1)$
02	逆置进位模拟	$O(n)$	$O(1)$

```

func addStrings(num1 string, num2 string) string {
    if len(num1) > len(num2) {
        num1, num2 = num2, num1
    }
    n1, n2 := len(num1), len(num2)
    a1, a2 := []byte(num1), []byte(num2)
    carry := byte(0)
    buf := make([]byte, n2+1)
    buf[0] = '1'

    for i := 1; i <= n2; i++ {
        if i <= n1 {
            buf[n2+1-i] = a1[n1-i] - '0'
        }
        buf[n2+1-i] = buf[n2+1-i] + a2[n2-i] + carry

        if buf[n2+1-i] > '9' {
            buf[n2+1-i] = buf[n2+1-i] - 10
            carry = byte(1)
        } else {
            carry = byte(0)
        }
    }
    if carry == 1 {
        return string(buf)
    }
    return string(buf[1:])
}

#
func addStrings(num1 string, num2 string) string {
    if len(num1) > len(num2) {
        num1, num2 = num2, num1
    }
    n1, n2 := len(num1), len(num2)
    a1, a2 := []byte(num1), []byte(num2)
    a1 = reverse(a1)
    a2 = reverse(a2)

    carry := 0
    buf := make([]byte, 0)
    for i := 0; i < n2; i++ {
        temp := 0
        if i < n1 {
            temp = int(a1[i] - '0')
        }
        temp = int(a2[i]-'0') + temp + carry
        if temp > 9 {

```

```

        buf = append(buf, byte(temp-'0'))
        carry = 1
    } else {
        buf = append(buf, byte(temp+'0'))
        carry = 0
    }
}
if carry == 1 {
    buf = append(buf, byte('1'))
}
return string(reverse(buf))
}

func reverse(arr []byte) []byte {
    for i := 0; i < len(arr)/2; i++ {
        arr[i], arr[len(arr)-1-i] = arr[len(arr)-1-i], arr[
    }
    return arr
}

```

434. 字符串中的单词数(2)

- 题目

统计字符串中的单词个数，这里的单词指的是连续的不是空格的字符。

请注意，你可以假定字符串里不包括任何不可打印的字符。

示例：输入："Hello, my name is John"输出：5

解释：这里的单词是指连续的不是空格的字符，所以 "Hello," 算作 1 个单

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	内置函数	O(n)	O(n)
02	遍历	O(n)	O(1)

```

func countSegments(s string) int {
    if len(s) == 0 {
        return 0
    }
    return len(strings.Fields(s))
}

#
func countSegments(s string) int {
    count := 0
    for i := 0; i < len(s); i++{
        if (i == 0 || s[i-1] == ' ') && s[i] != ' '{
            count++
        }
    }
    return count
}

```

437. 路径总和III(4)

- 题目

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下！

二叉树不超过1000个节点，且节点数值范围是 $[-1000000, 1000000]$ 的整数。
示例：

`root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8`

```

      10
     / \
    5   -3
   / \   \
  3   2   11
 / \   \
3   -2   1

```

返回 3。和等于 8 的路径有：

1. 5 -> 3
2. 5 -> 2 -> 1
3. -3 -> 11

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n^2)$	$O(n)$
02	2次递归	$O(n^2)$	$O(n)$
03	迭代+递归	$O(n^2)$	$O(n)$
04	保存路径	$O(n^2)$	$O(n)$

```

func pathSum(root *TreeNode, sum int) int {
    if root == nil {
        return 0
    }
    res := 0
    var helper func(*TreeNode, int)
    helper = func(node *TreeNode, sum int) {
        if node == nil {
            return
        }
        sum = sum - node.Val
        // 路径不需要从根节点开始，也不需要在叶子节点结束
        if sum == 0 {
            res++
        }
        helper(node.Left, sum)
        helper(node.Right, sum)
    }
    helper(root, sum)
    return res + pathSum(root.Left, sum) + pathSum(root.Right, sum)
}

# 迭代+递归
func helper(node *TreeNode, sum int, curSum int) int {
    res := 0
    curSum = curSum + node.Val
    if curSum == sum {
        res++
    }
    if node.Left != nil {
        res += helper(node.Left, sum, curSum)
    }
    if node.Right != nil {
        res += helper(node.Right, sum, curSum)
    }
    return res
}

```

```

        }
        if node.Left != nil {
            res += helper(node.Left, sum, curSum)
        }
        if node.Right != nil {
            res += helper(node.Right, sum, curSum)
        }
        return res
    }

func pathSum(root *TreeNode, sum int) int {
    if root == nil {
        return 0
    }
    queue := make([]*TreeNode, 0)
    queue = append(queue, root)
    res := 0
    for len(queue) > 0 {
        node := queue[0]
        queue = queue[1:]
        tempSum := 0
        res += helper(node, sum, tempSum)
        if node.Left != nil {
            queue = append(queue, node.Left)
        }
        if node.Right != nil {
            queue = append(queue, node.Right)
        }
    }
    return res
}

#
func helper(node *TreeNode, sum int, path []int, level int)
    if node == nil {
        return 0
    }
    res := 0
    if sum == node.Val {
        res = 1
    }
    temp := node.Val
    for i := level - 1; i >= 0; i-- {
        temp = temp + path[i]
        if temp == sum {
            res++
        }
    }
}

```

```

    path[level] = node.Val
    return res + helper(node.Left, sum, path, level+1) +
           helper(node.Right, sum, path, level+1)
}

func pathSum(root *TreeNode, sum int) int {
    return helper(root, sum, make([]int, 1001), 0)
}

```

441. 排列硬币(3)

- 题目

你总共有 n 枚硬币，你需要将它们摆成一个阶梯形状，第 k 行就必须正好有给定一个数字 n ，找出可形成完整阶梯行的总行数。

n 是一个非负整数，并且在32位有符号整型的范围内。

示例 1: $n = 5$

硬币可排列成以下几行：

□
□ □
□ □

因为第三行不完整，所以返回2。

示例 2: $n = 8$

硬币可排列成以下几行：

□
□ □
□ □ □
□ □

因为第四行不完整，所以返回3。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	数学法	$O(1)$	$O(1)$
02	迭代	$O(n^{1/2})$	$O(1)$
03	二分查找	$O(\log(n))$	$O(1)$

```

func arrangeCoins(n int) int {
    return int(math.Sqrt(float64(2*n)+0.25) - 0.5)
}

#
func arrangeCoins(n int) int {
    i := 1
    for i <= n{
        n = n - i
        i++
    }
    return i-1
}

#
func arrangeCoins(n int) int {
    if n == 0{
        return 0
    }
    left, right := 1, n
    for left < right{
        mid := left + (right-left)/2
        if mid * (mid+1)/2 < n{
            left = mid + 1
        }else {
            right = mid
        }
    }
    if left*(left+1)/2 == n{
        return left
    }
    return left-1
}

```

443. 压缩字符串(1)

- 题目

给定一组字符，使用原地算法将其压缩。
压缩后的长度必须始终小于或等于原数组长度。
数组的每个元素应该是长度为1 的字符（不是 int 整数类型）。
在完成原地修改输入数组后，返回数组的新长度。
进阶：你能否仅使用O(1) 空间解决问题？

示例 1: 输入: ["a", "a", "b", "b", "c", "c", "c"]
输出: 返回6, 输入数组的前6个字符应该是: ["a", "2", "b", "2", "c", "3"]
说明: "aa"被"2"替代。"bb"被"2"替代。"ccc"被"3"替代。

示例 2：输入：["a"]
输出：返回1，输入数组的前1个字符应该是：["a"]
说明：没有任何字符串被替代。

名刀腰田吃

No.	思路	时间复杂度	空间复杂度
01	双指针	$O(n)$	$O(1)$

```

func compress(chars []byte) int {
    j := 0
    count := 1
    for i := 0; i < len(chars); i++ {
        char := chars[i]
        if i+1 < len(chars) && char == chars[i+1] {
            count++
        } else {
            chars[j] = char
            j++
            if count > 1 {
                for _, num := range strconv.Itoa(count) {
                    chars[j] = byte(num)
                    j++
                }
            }
            count = 1
        }
    }
    return j
}

```

447.回旋镖的数量(1)

- 题目

给定平面上 n 对不同的点，“回旋镖” 是由点表示的元组 (i, j, k) ，其中 i 和 j 之间的距离和 i 和 k 之间的距离相等（需要考虑元组的顺序）

找到所有回旋镖的数量。你可以假设 n 最大为 500，所有点的坐标在闭区间 [示例：

输入：[[0, 0], [1, 0], [2, 0]]

输出：2

解释：两个回旋镖为 [[1, 0], [0, 0], [2, 0]] 和 [[1, 0], [2, 0], [0, 0]]

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	哈希辅助+遍历	$O(n^2)$	$O(n)$

```

func numberOfBoomerangs(points [][]int) int {
    res := 0
    size := len(points)
    if size < 3 {
        return 0
    }
    for i := 0; i < size; i++ {
        dMap := make(map[int]int, size)
        for j := 0; j < size; j++ {
            if i == j {
                continue
            }
            d := dSquare(points[i], points[j])
            if _, ok := dMap[d]; ok {
                dMap[d]++
            } else {
                dMap[d] = 1
            }
        }
        // 相同距离的v个点，总共有 v*(v-1)种排列
        for _, v := range dMap {
            res = res + v*(v-1)
        }
    }
    return res
}

func dSquare(p1, p2 []int) int {
    x := p2[0] - p1[0]
    y := p2[1] - p1[1]
    return x*x + y*y
}

```

448. 找到所有数组中消失的数字(3)

- 题目

给定一个范围在 $1 \leq a[i] \leq n$ ($n = \text{数组大小}$) 的 整型数组，数组中找到所有在 $[1, n]$ 范围之间没有出现在数组中的数字。

您能在不使用额外空间且时间复杂度为 $O(n)$ 的情况下完成这个任务吗？你可以假

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

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历交换	$O(n)$	$O(1)$
02	遍历置反	$O(n)$	$O(1)$
03	哈希辅助	$O(n)$	$O(n)$

```

func findDisappearedNumbers(nums []int) []int {
    for i := 0; i < len(nums); i++ {
        for nums[i] != nums[nums[i]-1] {
            nums[i], nums[nums[i]-1] = nums[nums[i]-1], num
        }
    }
    res := make([]int, 0)
    for i, n := range nums {
        if n != i+1 {
            res = append(res, i+1)
        }
    }
    return res
}

#
func findDisappearedNumbers(nums []int) []int {
    for i := 0; i < len(nums); i++ {
        value := nums[i]
        if value < 0{
            value = -value
        }
        if nums[value-1] > 0{
            nums[value-1] = -nums[value-1]
        }
    }
    res := make([]int, 0)
    for key, value := range nums {
        if value > 0{
            res = append(res, key+1)
        }
    }
    return res
}

#
func findDisappearedNumbers(nums []int) []int {
    m := make(map[int]int)
    for i := 0; i < len(nums); i++ {
        m[nums[i]] = 1
    }
    res := make([]int, 0)
    for i := 0; i < len(nums); i++ {
        if _, ok := m[i+1]; !ok {
            res = append(res, i+1)
        }
    }
}

```

```

    return res
}

```

453. 最小移动次数使数组元素相等(2)

- 题目

给定一个长度为 n 的非空整数数组，找到让数组所有元素相等的最小移动次数。

示例：输入：[1, 2, 3] 输出：3

解释：只需要3次移动（注意每次移动会增加两个元素的值）：

[1, 2, 3] \Rightarrow [2, 3, 3] \Rightarrow [3, 4, 3] \Rightarrow [4, 4, 4]

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	数学公式	$O(n)$	$O(1)$
02	排序遍历	$O(n \log(n))$	$O(1)$

```

func minMoves(nums []int) int {
    sum := 0
    min := nums[0]
    for _, n := range nums {
        sum += n
        if min > n {
            min = n
        }
    }
    return sum - min*len(nums)
}

#
func minMoves(nums []int) int {
    sum := 0
    sort.Ints(nums)
    for i := 1; i < len(nums); i++{
        sum = sum + nums[i] - nums[0]
    }
    return sum
}

```

455. 分发饼干(1)

- 题目

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多对每个孩子 i ，都有一个胃口值 g_i ，这是能让孩子们满足胃口的饼干的最小并且每块饼干 j ，都有一个尺寸 s_j 。

如果 $s_j \geq g_i$ ，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

注意：你可以假设胃口值为正。一个小朋友最多只能拥有一块饼干。

示例 1: 输入: [1,2,3], [1,1] 输出: 1

解释: 你有三个孩子和两块小饼干，3个孩子的胃口值分别是: 1, 2, 3。

虽然你有两块小饼干，由于他们的尺寸都是1，你只能让胃口值是1的孩子满足。

示例 2: 输入: [1,2], [1,2,3] 输出: 2

解释: 你有两个孩子和三块小饼干，2个孩子的胃口值分别是1, 2。

你拥有的饼干数量和尺寸都足以让所有孩子满足。所以你应该输出2.

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	排序双指针	$O(n \log n)$	$O(1)$

```
func findContentChildren(g []int, s []int) int {
    sort.Ints(g)
    sort.Ints(s)
    var i, j int
    for i < len(g) && j < len(s) {
        if g[i] <= s[j] {
            i++
        }
        j++
    }
    return i
}
```

459. 重复的子字符串(2)

- 题目

给定一个非空的字符串，判断它是否可以由它的一个子串重复多次构成。

给定的字符串只含有小写英文字母，并且长度不超过10000。

示例 1: 输入: "abab"输出: True

解释: 可由子字符串 "ab" 重复两次构成。

示例 2: 输入: "aba"输出: False

示例 3: 输入: "abcabcabcabc"输出: True

解释: 可由子字符串 "abc" 重复四次构成。 (或者子字符串 "abcabc" 重复两次)

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	2倍去除首尾匹配	O(n)	O(1)
02	暴力匹配	O(n^2)	l

```

func repeatedSubstringPattern(s string) bool {
    if len(s) == 0 {
        return false
    }

    size := len(s)
    ss := (s + s)[1 : size*2-1]
    return strings.Contains(ss, s)
}

#
func repeatedSubstringPattern(s string) bool {
    if len(s) == 0 {
        return false
    }
    size := len(s)
    for i := 1; i < size; i++ {
        if size%i == 0 {
            count := size / i
            if strings.Repeat(s[0:i], count) == s {
                return true
            }
        }
    }
    return false
}

```

461.汉明距离(3)

- 题目

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 x 和 y , 计算它们之间的汉明距离。

注意:

$0 \leq x, y < 231.$

示例:

输入: $x = 1, y = 4$ 输出: 2

解释:

1 (0 0 0 1)

4 (0 1 0 0)

↑ ↑

上面的箭头指出了对应二进制位不同的位置。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	位运算+遍历统计	O(1)	O(1)
02	位运算	O(1)	O(1)
03	内置函数	O(1)	O(1)

```
func hammingDistance(x int, y int) int {
    x = x ^ y
    res := 0
    for x > 0 {
        if x&1 == 1{
            res++
        }
        x = x >> 1
    }
    return res
}

#
func hammingDistance(x int, y int) int {
    x = x ^ y
    res := 0
    for x > 0 {
        res++
        x = x & (x-1)
    }
    return res
}

#
func hammingDistance(x int, y int) int {
    x = x ^ y
    return bits.OnesCount(uint(x))
}
```

463. 岛屿的周长(3)

- 题目

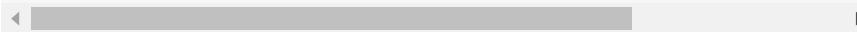
给定一个包含 0 和 1 的二维网格地图，其中 1 表示陆地 0 表示水域。
 网格中的格子水平和垂直方向相连（对角线方向不相连）。
 整个网格被水完全包围，但其中恰好有一个岛屿（或者说，一个或多个表示陆地的格子是岛屿，岛屿中没有“湖”（“湖”指水域在岛屿内部且不和岛屿周围的水相连）。格子是长方形，且宽度和高度均不超过 100。计算这个岛屿的周长。

示例：

输入：

```
[[0,1,0,0],
 [1,1,1,0],
 [0,1,0,0],
 [1,1,0,0]]
```

输出：16



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	暴力法	$O(n^2)$	$O(1)$
02	暴力法	$O(n^2)$	$O(1)$
03	深度优先搜索	$O(n^2)$	$O(n^2)$

```

func islandPerimeter(grid [][]int) int {
    var dx = []int{-1, 1, 0, 0}
    var dy = []int{0, 0, -1, 1}
    m, n := len(grid), len(grid[0])
    res := 0
    for i := 0; i < m; i++ {
        for j := 0; j < n; j++ {
            if grid[i][j] == 0 {
                continue
            }
            res += 4
            for k := 0; k < 4; k++ {
                x := i + dx[k]
                y := j + dy[k]
                if (0 <= x && x < m && 0 <= y && y < n) &&
                    res--
            }
        }
    }
    return res
}

#
func islandPerimeter(grid [][]int) int {
    m, n := len(grid), len(grid[0])
    res := 0
    for i := 0; i < m; i++ {
        for j := 0; j < n; j++ {
            if grid[i][j] == 0 {
                continue
            }
            res += 4
            if i > 0 && grid[i-1][j] == 1 {
                res -= 2
            }
            if j > 0 && grid[i][j-1] == 1 {
                res -= 2
            }
        }
    }
    return res
}

#
func islandPerimeter(grid [][]int) int {
    m, n := len(grid), len(grid[0])
    for i := 0; i < m; i++ {

```

```
        for j := 0; j < n; j++ {
            if grid[i][j] == 1 {
                return dfs(grid, i, j)
            }
        }
    }
    return 0
}

func dfs(grid [][]int, i, j int) int {
    // 边界+1
    if !(0 <= i && i < len(grid) && 0 <= j && j < len(grid[0])) {
        return 1
    }
    // 水域+1
    if grid[i][j] == 0 {
        return 1
    }
    if grid[i][j] != 1 {
        return 0
    }
    grid[i][j] = 2
    return dfs(grid, i-1, j) +
        dfs(grid, i+1, j) +
        dfs(grid, i, j-1) +
        dfs(grid, i, j+1)
}
```

475. 供暖器(2)

- 题目

冬季已经来临。 你的任务是设计一个有固定加热半径的供暖器向所有房屋供暖。现在，给出位于一条水平线上的房屋和供暖器的位置，找到可以覆盖所有房屋的最小加热半径。

说明：

给出的房屋和供暖器的数目是非负数且不会超过 25000。

给出的房屋和供暖器的位置均是非负数且不会超过 10^9 。

只要房屋位于供暖器的半径内(包括在边缘上)，它就可以得到供暖。

所有供暖器都遵循你的半径标准，加热的半径也一样。

示例 1: 输入： [1, 2, 3], [2] 输出： 1

解释： 仅在位置2上有一个供暖器。如果我们将加热半径设为1，那么所有房屋就

示例 2: 输入： [1, 2, 3, 4], [1, 4] 输出： 1

解释： 在位置1, 4上有两个供暖器。我们需要将加热半径设为1，这样所有房屋

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	排序双指针	$O(n \log(n))$	$O(1)$
02	排序二分查找	$O(n \log(n))$	$O(1)$

```

func findRadius(houses []int, heaters []int) int {
    if len(heaters) == 0 {
        return 0
    }
    sort.Ints(houses)
    sort.Ints(heaters)
    res := 0
    j := 0
    for i := 0; i < len(houses); i++ {
        // 找到最近的一个供暖器, >=确保出现重复的供暖器会往后走
        for j < len(heaters)-1 &&
            Abs(houses[i], heaters[j]) >= Abs(houses[i], heaters[j+1]) {
            j++
        }
        res = Max(Abs(houses[i], heaters[j]), res)
    }
    return res
}

func Abs(a, b int) int {
    if a > b {
        return a - b
    }
    return b - a
}

func Max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

#
func findRadius(houses []int, heaters []int) int {
    if len(heaters) == 0 {
        return 0
    }
    sort.Ints(houses)
    sort.Ints(heaters)
    res := 0
    length := len(heaters)
    for i := 0; i < len(houses); i++ {
        left := 0
        right := length - 1
        for left < right {
            mid := left + (right-left)/2
            if heaters[mid] < houses[i] {
                left = mid + 1
            } else {
                right = mid
            }
        }
        res = Max(res, Abs(houses[i], heaters[left]))
    }
    return res
}

```

```

        left = mid + 1
    } else {
        right = mid
    }
}
dis := 0
if heaters[left] < houses[i] {
    dis = houses[i] - heaters[left]
} else if heaters[left] > houses[i] {
    if left == 0 {
        dis = heaters[0] - houses[i]
    } else {
        dis = Min(heaters[left]-houses[i], houses[i])
    }
}
res = Max(res, dis)
}
return res
}

func Min(a, b int) int {
    if a > b {
        return b
    }
    return a
}

func Max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

```

476.数字的补数(3)

- 题目

给定一个正整数，输出它的补数。补数是对该数的二进制表示取反。

示例 1: 输入: 5 输出: 2

解释: 5 的二进制表示为 101 (没有前导零位)，其补数为 010。所以你需要

示例 2: 输入: 1 输出: 0

解释: 1 的二进制表示为 1 (没有前导零位)，其补数为 0。所以你需要输出注意:

给定的整数保证在 32 位带符号整数的范围内。

你可以假定二进制数不包含前导零位。

本题与 1009 <https://leetcode-cn.com/problems/complement-of-base-10-integer/>

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	位运算	$O(\log(n))$	$O(1)$
02	位运算	$O(\log(n))$	$O(1)$
03	遍历	$O(\log(n))$	$O(1)$

```

func findComplement(num int) int {
    temp := 1
    for num >= temp {
        temp = temp << 1
    }
    return temp - 1 - num
}

#
func findComplement(num int) int {
    temp := num
    res := 0
    for temp > 0 {
        temp = temp >> 1
        res = res << 1
        res++
    }
    return res ^ num
}

#
func findComplement(num int) int {
    res := 0
    if num == 0 {
        return 1
    }
    if num == 1 {
        return 0
    }

    exp := 1
    for num > 0 {
        temp := num % 2
        if temp == 0 {
            res = res + exp
            exp = exp * 2
        } else {
            exp = exp * 2
        }
        num = num / 2
    }
    return res
}

```

482.密钥格式化(2)

- 题目

有一个密钥字符串 S ，只包含字母，数字以及 '-'（破折号）。其中， N 个 '-' 将字符串分成了 $N+1$ 组。

给你一个数字 K ，请你重新格式化字符串，除了第一个分组以外，每个分组要包而第一个分组中，至少要包含 1 个字符。

两个分组之间需要用 '-'（破折号）隔开，并且将所有的小写字母转换为大写字给定非空字符串 S 和数字 K ，按照上面描述的规则进行格式化。

示例 1：输入： $S = "5F3Z-2e-9-w"$ ， $K = 4$ 输出："5F3Z-2E9W"

解释：字符串 S 被分成了两个部分，每部分 4 个字符；注意，两个额外的破折号被忽略。

示例 2：输入： $S = "2-5g-3-J"$ ， $K = 2$ 输出："2-5G-3J"

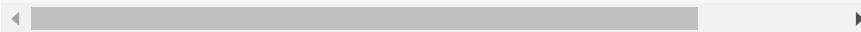
解释：字符串 S 被分成了 3 个部分，按照前面的规则描述，第一部分的字符可以少于给定的数量，其余部分皆为 2 个字符。

提示：

S 的长度可能很长，请按需分配大小。 K 为正整数。

S 只包含字母数字 (a-z, A-Z, 0-9) 以及破折号 '-'

S 非空



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	内置函数	$O(n)$	$O(1)$
02	遍历	$O(n)$	$O(1)$

```

func licenseKeyFormatting(S string, K int) string {
    arr := strings.Join(strings.Split(strings.ToUpper(S), ''),
    count := len(arr) / K
    first := len(arr) % K
    if first > 0 {
        count++
    }
    str := arr[:first]
    if first != 0 {
        count = count - 1
    }
    for i := 0; i < count; i++ {
        str = str + "-" + arr[first+i*K:first+(i+1)*K]
    }
    return strings.Trim(str, "-")
}

#
func licenseKeyFormatting(S string, K int) string {
    res := make([]rune, 0)
    temp := []rune(S)
    count := 0
    for i := len(temp) - 1; i >= 0; i-- {
        value := temp[i]
        if value >= 'a' {
            value = value - 'a' + 'A'
        }
        if value == '-' {
            continue
        }
        count++
        res = append([]rune{value}, res...)
        if count == K {
            res = append([]rune{'-'}, res...)
            count = 0
        }
    }
    if len(res) == 0 {
        return ""
    }
    if res[0] == '-' {
        res = res[1:]
    }
    return string(res)
}

```

485. 最大连续1的个数(2)

- 题目

给定一个二进制数组， 计算其中最大连续1的个数。

示例 1: 输入: [1, 1, 0, 1, 1, 1] 输出: 3

解释: 开头的两位和最后的三位都是连续1， 所以最大连续1的个数是 3 .

注意:

输入的数组只包含 0 和1。

输入数组的长度是正整数， 且不超过 10,000。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	双指针	$O(n)$	$O(1)$
02	单指针	$O(n)$	$O(1)$

```

func findMaxConsecutiveOnes(nums []int) int {
    max := 0
    for i, j := 0, -1; i < len(nums); i++ {
        if nums[i] == 0 {
            j = i
        } else {
            if max < i-j {
                max = i - j
            }
        }
    }
    return max
}

#
func findMaxConsecutiveOnes(nums []int) int {
    max := 0
    count := 0
    for _, v := range nums {
        if v == 1 {
            count++
        } else {
            if count > max {
                max = count
            }
            count = 0
        }
    }
    if count > max {
        max = count
    }
    return max
}

```

492.构造矩形(1)

- 题目

作为一位web开发者，懂得怎样去规划一个页面的尺寸是很重要的。
现给定一个具体的矩形页面面积，你的任务是设计一个长度为 L 和宽度为 W 的矩形。

1. 你设计的矩形页面必须等于给定的目标面积。
2. 宽度 W 不应大于长度 L ，换言之，要求 $L \geq W$ 。
3. 长度 L 和宽度 W 之间的差距应当尽可能小。

你需要按顺序输出你设计的页面的长度 L 和宽度 W 。

示例：

输入：4 输出：[2, 2]

解释：目标面积是 4，所有可能的构造方案有 [1,4], [2,2], [4,1]。

但是根据要求2, [1,4] 不符合要求；根据要求3, [2,2] 比 [4,1] 更能符合要求，所以输出长度 L 为 2，宽度 W 为 2。

说明：

给定的面积不大于 10,000,000 且为正整数。

你设计的页面的长度和宽度必须都是正整数。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	开方向下遍历	$O(n)$	$O(1)$

```
func constructRectangle(area int) []int {
    for i := int(math.Sqrt(float64(area))); i > 1; i-- {
        if area%i == 0 {
            return []int{area / i, i}
        }
    }
    return []int{area, 1}
}
```

496. 下一个更大元素 I(3)

- 题目

给定两个没有重复元素的数组 nums1 和 nums2 ，其中 nums1 是 nums2 的子集。找到 nums1 中每个元素在 nums2 中的下一个比其大的值。

nums1 中数字 x 的下一个更大元素是指 x 在 nums2 中对应位置的右边的第一个比 x 大的元素。如果不存在，对应位置输出 -1。

示例 1：

输入: $\text{nums1} = [4, 1, 2]$, $\text{nums2} = [1, 3, 4, 2]$.

输出: $[-1, 3, -1]$

解释:

对于 num1 中的数字 4，你无法在第二个数组中找到下一个更大的数字，因此输出 -1。
对于 num1 中的数字 1，第二个数组中数字 1 右边的下一个较大数字是 3。

对于 num1 中的数字 2，第二个数组中没有下一个更大的数字，因此输出 -1。

示例 2：

输入: $\text{nums1} = [2, 4]$, $\text{nums2} = [1, 2, 3, 4]$.

输出: $[3, -1]$

解释:

对于 num1 中的数字 2，第二个数组中的下一个较大数字是 3。

对于 num1 中的数字 4，第二个数组中没有下一个更大的数字，因此输出 -1。

提示:

nums1 和 nums2 中所有元素是唯一的。

nums1 和 nums2 的数组大小都不超过 1000。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	哈希辅助	$O(n^2)$	$O(n)$
02	哈希辅助	$O(n^2)$	$O(n)$
02	栈+哈希辅助	$O(n)$	$O(n)$

```

func nextGreaterElement(nums1 []int, nums2 []int) []int {
    m := make(map[int]int)
    for i, n := range nums2 {
        m[n] = i
    }
    res := make([]int, len(nums1))
    for i, n := range nums1 {
        res[i] = -1
        for j := m[n] + 1; j < len(nums2); j++ {
            if n < nums2[j] {
                res[i] = nums2[j]
                break
            }
        }
    }
    return res
}

#
func nextGreaterElement(nums1 []int, nums2 []int) []int {
    m := make(map[int]int)
    res := make([]int, len(nums1))
    for i := 0; i < len(nums2); i++ {
        for j := i + 1; j < len(nums2); j++ {
            if nums2[j] > nums2[i] {
                m[nums2[i]] = nums2[j]
                break
            }
        }
    }
    for key, value := range nums1 {
        if _, ok := m[value]; ok {
            res[key] = m[value]
        } else {
            res[key] = -1
        }
    }
    return res
}

#
func nextGreaterElement(nums1 []int, nums2 []int) []int {
    m := make(map[int]int)
    res := make([]int, len(nums1))
    stack := make([]int, 0)
    for i := 0; i < len(nums2); i++ {
        if len(stack) > 0 {
            for len(stack) > 0 && nums2[i] > stack[len(stack)-1] {
                res[stack[len(stack)-1]] = nums2[i]
                stack = stack[:len(stack)-1]
            }
        }
        m[nums2[i]] = i
        stack = append(stack, i)
    }
    return res
}

```

```

        top := stack[len(stack)-1]
        m[top] = nums2[i]
        stack = stack[:len(stack)-1]
    }
}
stack = append(stack, nums2[i])
}
for key, value := range nums1 {
    if _, ok := m[value]; ok {
        res[key] = m[value]
    } else {
        res[key] = -1
    }
}
return res
}

```

500. 键盘行(4)

- 题目

给定一个单词列表，只返回可以使用在键盘同一行的字母打印出来的单词。键盘示例：

输入： ["Hello", "Alaska", "Dad", "Peace"]

输出： ["Alaska", "Dad"]

注意：

你可以重复使用键盘上同一字符。

你可以假设输入的字符串将只包含字母。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	哈希辅助	O(n^2)	O(1)
02	哈希辅助	O(n^2)	O(1)
03	遍历	O(n^2)	O(1)
04	内置函数	O(n^2)	O(1)

```

func findWords(words []string) []string {
    m := make(map[byte]int)
    m['q'] = 1
    m['w'] = 1
    m['e'] = 1
    m['r'] = 1
    m['t'] = 1
    m['y'] = 1
    m['u'] = 1
    m['i'] = 1
    m['o'] = 1
    m['p'] = 1
    m['a'] = 2
    m['s'] = 2
    m['d'] = 2
    m['f'] = 2
    m['g'] = 2
    m['h'] = 2
    m['j'] = 2
    m['k'] = 2
    m['l'] = 2
    m['z'] = 3
    m['x'] = 3
    m['c'] = 3
    m['v'] = 3
    m['b'] = 3
    m['n'] = 3
    m['m'] = 3

    res := make([]string, 0)
    for i := 0; i < len(words); i++ {
        b := []byte(strings.ToLower(words[i]))
        level := m[b[0]]
        flag := true
        for j := 1; j < len(b); j++ {
            if m[b[j]] != level {
                flag = false
                break
            }
        }
        if flag {
            res = append(res, words[i])
        }
    }
    return res
}

#

```

```

var qRow = map[byte]bool{
    'q': true,
    'w': true,
    'e': true,
    'r': true,
    't': true,
    'y': true,
    'u': true,
    'i': true,
    'o': true,
    'p': true,
}

var aRow = map[byte]bool{
    'a': true,
    's': true,
    'd': true,
    'f': true,
    'g': true,
    'h': true,
    'j': true,
    'k': true,
    'l': true,
}

var zRow = map[byte]bool{
    'z': true,
    'x': true,
    'c': true,
    'v': true,
    'b': true,
    'n': true,
    'm': true,
}

func findWords(words []string) []string {
    res := make([]string, 0, len(words))
    for _, word := range words {
        w := strings.ToLower(word)
        if isAllIn(w, qRow) || isAllIn(w, aRow) || isAllIn(
            res = append(res, word)
        )
    }
    return res
}

func isAllIn(s string, Row map[byte]bool) bool {
    for i := range s {

```

```

        if !Row[s[i]] {
            return false
        }
    }
    return true
}

#



func findWords(words []string) []string {
    res := make([]string, 0, len(words))
    for _, word := range words {
        w := strings.ToLower(word)
        flag := 0
        for _, m := range w {
            switch m {
            case 'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'c':
                if flag != 0 && flag != 1 {
                    flag = 4
                    break
                }
                flag = 1
            case 'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l':
                if flag != 0 && flag != 2 {
                    flag = 4
                    break
                }
                flag = 2
            case 'z', 'x', 'c', 'v', 'b', 'n', 'm':
                if flag != 0 && flag != 3 {
                    flag = 4
                    break
                }
                flag = 3
            default:
                flag = 4
            }
        }
        if flag != 0 && flag != 4 {
            res = append(res, word)
        }
    }
    return res
}

#



func findWords(words []string) []string {
    res := make([]string, 0, len(words))
    q := "qwertyuiopQWERTYUIOP"

```

```
a := "asdfghjklASDFGHJKL"
z := "zxcvbnmZXCVBNM"
for _, word := range words {
    qLen, aLen, zLen := 0, 0, 0
    for i := 0; i < len(word); i++ {
        if strings.Contains(q, string(word[i])) {
            qLen++
        }
        if strings.Contains(a, string(word[i])) {
            aLen++
        }
        if strings.Contains(z, string(word[i])) {
            zLen++
        }
    }
    if qLen == len(word) || aLen == len(word) || zLen =
        res = append(res, word)
    }
}
return res
}
```



0501-0600-Easy

- 0501-0600-Easy
 - 501.二叉搜索树中的众数(2)
 - 504.七进制数(3)
 - 506.相对名次(1)
 - 507.完美数(1)
 - 509.斐波那契数(4)
 - 520.检测大写字母(2)
 - 521.最长特殊序列 I (1)
 - 530.二叉搜索树的最小绝对差(2)
 - 532.数组中的K-diff数对(3)
 - 538.把二叉搜索树转换为累加树(2)
 - 541.反转字符串II(2)
 - 543.二叉树的直径(2)
 - 551.学生出勤记录 I(2)
 - 557.反转字符串中的单词 III(2)
 - 559.N叉树的最大深度(2)
 - 561.数组拆分 I(2)
 - 563.二叉树的坡度(2)
 - 566.重塑矩阵(2)
 - 572.另一个树的子树(3)
 - 575.分糖果(2)
 - 581.最短无序连续子数组(3)
 - 589.N叉树的前序遍历(2)
 - 590.N叉树的后序遍历(2)
 - 594.最长和谐子序列(2)
 - 598.范围求和 II(1)
 - 599.两个列表的最小索引总和(2)

501.二叉搜索树中的众数(2)

- 题目

给定一个有相同值的二叉搜索树（BST），找出 BST 中的所有众数（出现频率最高的元素）。

假定 BST 有如下定义：

结点左子树中所含结点的值小于等于当前结点的值

结点右子树中所含结点的值大于等于当前结点的值

左子树和右子树都是二叉搜索树

例如：

给定 BST [1, null, 2, 2],

1

\

2

/

2

返回[2].

提示：如果众数超过1个，不需考虑输出顺序

进阶：你可以不使用额外的空间吗？（假设由递归产生的隐式调用栈的开销不被限制）

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归+哈希辅助	O(n)	O(n)
02	递归+中序遍历	O(n)	O(log(n))

```

func findMode(root *TreeNode) []int {
    m := map[int]int{}
    dfs(root, m)
    max := -1
    res := make([]int, 0)
    for i, v := range m {
        if max <= v {
            if max < v {
                max = v
                res = res[0:0]
            }
            res = append(res, i)
        }
    }
    return res
}

func dfs(root *TreeNode, rec map[int]int) {
    if root == nil {
        return
    }
    rec[root.Val]++
    dfs(root.Left, rec)
    dfs(root.Right, rec)
}

#
var max int
var res []int
var cur int
var count int

func findMode(root *TreeNode) []int {
    res = make([]int, 0)
    max, cur, count = 0, 0, 0
    dfs(root)
    return res
}

// 中序遍历保证利用二叉搜索树的性质，得到的结果是升序的
func dfs(root *TreeNode) {
    if root == nil {
        return
    }
    dfs(root.Left)
    if root.Val != cur {
        count = 0
    }
}

```

```

count++
if max < count {
    max = count
    res = []int{root.Val}
} else if max == count {
    res = append(res, root.Val)
}
cur = root.Val
dfs(root.Right)
}

```

504. 七进制数(3)

- 题目

给定一个整数，将其转化为7进制，并以字符串形式输出。

示例 1: 输入: 100 输出: "202"

示例 2: 输入: -7 输出: "-10"

注意：输入范围是 $[-1e7, 1e7]$ 。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历	$O(\log(n))$	$O(1)$
02	内置函数	$O(\log(n))$	$O(1)$
03	递归	$O(\log(n))$	$O(\log(n))$

```

func convertToBase7(num int) string {
    if num == 0 {
        return "0"
    }

    minus := ""
    if num < 0 {
        minus = "-"
        num = -1 * num
    }

    s := ""
    for num > 0 {
        s = fmt.Sprintf("%d", num%7) + s
        num = num / 7
    }
    return minus + s
}

#
func convertToBase7(num int) string {
    return strconv.FormatInt(int64(num), 7)
}

#
func convertToBase7(num int) string {
    if num < 0 {
        return "-" + convertToBase7(-1*num)
    }
    if num < 7 {
        return strconv.Itoa(num)
    }
    return convertToBase7(num/7) + strconv.Itoa(num%7)
}

```

506.相对名次(1)

- 题目

给出 N 名运动员的成绩，找出他们的相对名次并授予前三名对应的奖牌。
 前三名运动员将会被分别授予“金牌”，“银牌”和“铜牌”
 ("Gold Medal", "Silver Medal", "Bronze Medal")。
 (注：分数越高的选手，排名越靠前。)

示例 1：

输入：[5, 4, 3, 2, 1]

输出：["Gold Medal", "Silver Medal", "Bronze Medal", "4", "!"

解释：前三名运动员的成绩为前三高的，

因此将会分别被授予“金牌”，“银牌”和“铜牌” ("Gold Medal", "Silver
 余下的两名运动员，我们只需要通过他们的成绩计算将其相对名次即可。

提示：

N 是一个正整数并且不会超过 10000。

所有运动员的成绩都不相同。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	排序+遍历	O(nlog(n))	O(n)

```
func findRelativeRanks(nums []int) []string {
    temp := make([]int, len(nums))
    copy(temp, nums)
    sort.Ints(temp)
    m := make(map[int]string)
    for i := 0; i < len(temp); i++ {
        if i == len(temp)-1 {
            m[temp[i]] = "Gold Medal"
        } else if i == len(temp)-2 {
            m[temp[i]] = "Silver Medal"
        } else if i == len(temp)-3 {
            m[temp[i]] = "Bronze Medal"
        } else {
            m[temp[i]] = strconv.Itoa(len(temp) - i)
        }
    }
    res := make([]string, 0)
    for i := 0; i < len(nums); i++ {
        res = append(res, m[nums[i]])
    }
    return res
}
```

507. 完美数(1)

- 题目

对于一个 正整数，如果它和除了它自身以外的所有正因子之和相等，我们称它为
给定一个 整数 n， 如果他是完美数，返回 True，否则返回 False

示例：输入：28 输出：True 解释： $28 = 1 + 2 + 4 + 7 + 14$

提示：输入的数字 n 不会超过 $100,000,000$. ($1e8$)

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历	$O(n^{1/2})$	$O(1)$

```
func checkPerfectNumber(num int) bool {
    if num == 1 {
        return false
    }
    sum := 1
    for i := 2; i <= num/i; i++ {
        if num%i == 0 {
            sum = sum + i + (num / i)
        }
    }
    return sum == num
}
```

509.斐波那契数(4)

- 题目

斐波那契数，通常用 $F(n)$ 表示，形成的序列称为斐波那契数列。

该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：

$F(0) = 0, F(1) = 1$

$F(N) = F(N - 1) + F(N - 2)$, 其中 $N > 1$.

给定 N，计算 $F(N)$ 。

示例 1：输入：2输出：1

解释： $F(2) = F(1) + F(0) = 1 + 0 = 1$.

示例 2：输入：3输出：2

解释： $F(3) = F(2) + F(1) = 1 + 1 = 2$.

示例 3：输入：4输出：3

解释： $F(4) = F(3) + F(2) = 2 + 1 = 3$.

提示：

$0 \leq N \leq 30$

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历	$O(n)$	$O(1)$
02	遍历+数组	$O(n)$	$O(n)$
03	递归	$O(2^n)$	$O(n)$
04	公式法	$O(1)$	$O(1)$

```

func fib(N int) int {
    if N == 0 {
        return 0
    }
    if N == 1 {
        return 1
    }
    n1, n2 := 0, 1
    for i := 2; i <= N; i++ {
        n1, n2 = n2, n1+n2
    }
    return n2
}

#
func fib(N int) int {
    if N == 0 {
        return 0
    }
    if N == 1 {
        return 1
    }
    res := make([]int, N+1)
    res[0] = 0
    res[1] = 1
    for i := 2; i <= N; i++ {
        res[i] = res[i-1] + res[i-2]
    }
    return res[N]
}

#
func fib(N int) int {
    if N == 0 {
        return 0
    }
    if N == 1 {
        return 1
    }
    return fib(N-1) + fib(N-2)
}

#
func fib(N int) int {
    temp1 := (1 + math.Sqrt(5)) / 2
    temp2 := (1 - math.Sqrt(5)) / 2
    fn := math.Round((math.Pow(temp1, float64(N))- math.Pow

```

```
    return int(fn)
}
```

520. 检测大写字母(2)

- 题目

给定一个单词，你需要判断单词的大写使用是否正确。

我们定义，在以下情况时，单词的大写用法是正确的：

全部字母都是大写，比如"USA"。

单词中所有字母都不是大写，比如"leetcode"。

如果单词不只含有一个字母，只有首字母大写，比如 "Google"。

否则，我们定义这个单词没有正确使用大写字母。

示例 1: 输入: "USA" 输出: True

示例 2: 输入: "FlaG" 输出: False

注意：输入是由大写和小写拉丁字母组成的非空单词。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历	O(n)	O(1)
02	正则	O(n)	O(1)

```

func detectCapitalUse(word string) bool {
    if word == "" {
        return false
    }
    count := 0
    for i := 0; i < len(word); i++ {
        if word[i] >= 'A' && word[i] <= 'Z' {
            count++
        }
    }

    if count == 0 || count == len(word) ||
        (count == 1 && word[0] >= 'A' && word[0] <= 'Z') {
        return true
    }
    return false
}

#
func detectCapitalUse(word string) bool {
    pattern := "([a-z]+)$|([A-Z]+)$|([A-Z]{1}[a-z]*)$"
    isMatch, _ := regexp.MatchString(pattern, word)
    return isMatch
}

```

521.最长特殊序列 I (1)

- 题目

给你两个字符串，请你从这两个字符串中找出最长的特殊序列。

「最长特殊序列」定义如下：该序列为某字符串独有的最长子序列（即不能是其子序列）可以通过删去字符串中的某些字符实现，但不能改变剩余字符的相对顺序。

输入为两个字符串，输出最长特殊序列的长度。如果不存在，则返回 -1。

示例 1：输入： "aba", "cdc" 输出： 3

解释：最长特殊序列可为 "aba"（或 "cdc"），两者均为自身的子序列且不是

示例 2：输入： a = "aaa", b = "bbb" 输出： 3

示例 3：输入： a = "aaa", b = "aaa" 输出： -1

提示：

两个字符串长度均处于区间 [1 - 100]。

字符串中的字符仅含有 'a'~'z'。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	比较	O(1)	O(1)

```
func findLUSlength(a string, b string) int {
    if a == b {
        return -1
    }
    return max(len(a), len(b))
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}
```

530.二叉搜索树的最小绝对差(2)

- 题目

给你一棵所有节点为非负值的二叉搜索树，请你计算树中任意两节点的差的绝对值。

示例：

输入：

```
1
 \
 3
 /
2
```

输出：1

解释：

最小绝对差为 1，其中 2 和 1 的差的绝对值为 1（或者 2 和 3）。

提示：

树中至少有 2 个节点。

本题与 783 <https://leetcode-cn.com/problems/minimum-distance-between-bst-nodes/>

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归+中序遍历	$O(n)$	$O(\log(n))$
02	递归+遍历	$O(n)$	$O(n)$

```

var minDiff, previous int
func getMinimumDifference(root *TreeNode) int {
    minDiff, previous = math.MaxInt32, math.MaxInt32
    dfs(root)
    return minDiff
}

func dfs(root *TreeNode) {
    if root == nil {
        return
    }
    dfs(root.Left)

    newDiff := diff(previous, root.Val)
    if minDiff > newDiff {
        minDiff = newDiff
    }
    previous = root.Val
    dfs(root.Right)
}

func diff(a, b int) int {
    if a > b {
        return a - b
    }
    return b - a
}

#
func getMinimumDifference(root *TreeNode) int {
    arr := make([]int, 0)
    dfs(root, &arr)
    minDiff := arr[1] - arr[0]
    for i := 2; i < len(arr); i++ {
        if minDiff > arr[i]-arr[i-1] {
            minDiff = arr[i] - arr[i-1]
        }
    }
    return minDiff
}

func dfs(root *TreeNode, arr *[]int) {
    if root == nil {
        return
    }
    dfs(root.Left, arr)
    *arr = append(*arr, root.Val)
}

```

```
    dfs(root.Right, arr)
}
```

532. 数组中的K-diff数对(3)

- 题目

给定一个整数数组和一个整数 k , 你需要在数组里找到不同的 k -diff 数对。
这里将 k -diff 数对定义为一个整数对 (i, j) , 其中 i 和 j 都是数组中不同的元素。

示例 1: 输入: [3, 1, 4, 1, 5], $k = 2$ 输出: 2

解释: 数组中有两个 2-diff 数对, (1, 3) 和 (3, 5)。

尽管数组中有两个1, 但我们只应返回不同的数对的数量。

示例 2: 输入: [1, 2, 3, 4, 5], $k = 1$ 输出: 4

解释: 数组中有四个 1-diff 数对, (1, 2), (2, 3), (3, 4) 和 (4, 5)。

示例 3: 输入: [1, 3, 1, 5, 4], $k = 0$ 输出: 1

解释: 数组中只有一个 0-diff 数对, (1, 1)。

注意:

数对 (i, j) 和数对 (j, i) 被算作同一数对。

数组的长度不超过10,000。

所有输入的整数的范围在 $[-1e7, 1e7]$ 。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	单哈希辅助	$O(n)$	$O(n)$
02	双哈希辅助	$O(n)$	$O(n)$
03	排序遍历	$O(n \log(n))$	$O(1)$

```

func findPairs(nums []int, k int) int {
    if k < 0 {
        return 0
    }
    record := make(map[int]int)
    for _, num := range nums {
        record[num]++
    }
    res := 0
    if k == 0 {
        for _, count := range record {
            if count > 1 {
                res++
            }
        }
        return res
    } else {
        for n := range record {
            if record[n-k] > 0 {
                res++
            }
        }
        return res
    }
}

#
func findPairs(nums []int, k int) int {
    if k < 0 {
        return 0
    }
    m := make(map[int]bool)
    res := make(map[int]bool)
    for _, value := range nums {
        if m[value-k] {
            res[value-k] = true
        }
        if m[value+k] {
            res[value] = true
        }
        m[value] = true
    }
    return len(res)
}

```

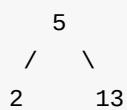
538.把二叉搜索树转换为累加树(2)

- 题目

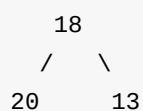
给定一个二叉搜索树（Binary Search Tree），把它转换成为累加树（Greater Sum Tree），使得每个节点的值是原来的节点值加上所有大于它的节点值之和。

例如：

输入：原始二叉搜索树：



输出：转换为累加树：



注意：

本题和 1038: <https://leetcode-cn.com/problems/binary-search-tree-inorder-traversal/>

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(\log(n))$
02	栈辅助	$O(n)$	$O(n)$

```

func convertBST(root *TreeNode) *TreeNode {
    sum := 0
    dfs(root, &sum)
    return root
}

func dfs(root *TreeNode, sum *int) {
    if root == nil {
        return
    }
    dfs(root.Right, sum)
    *sum = *sum + root.Val
    root.Val = *sum
    dfs(root.Left, sum)
}

#
func convertBST(root *TreeNode) *TreeNode {
    if root == nil {
        return root
    }
    stack := make([]*TreeNode, 0)
    temp := root
    sum := 0
    for {
        if temp != nil {
            stack = append(stack, temp)
            temp = temp.Right
        } else if len(stack) != 0 {
            temp = stack[len(stack)-1]
            stack = stack[:len(stack)-1]
            temp.Val = temp.Val + sum
            sum = temp.Val
            temp = temp.Left
        } else {
            break
        }
    }
    return root
}

```

541. 反转字符串II(2)

- 题目

给定一个字符串和一个整数 k ，你需要对从字符串开头算起的每个 $2k$ 个字符的子串进行反转。如果剩余少于 k 个字符，则将剩余的所有全部反转。

如果有小于 $2k$ 但大于或等于 k 个字符，则反转前 k 个字符，并将剩余的字符按原样保留。

示例：

输入： $s = "abcdefg"$, $k = 2$

输出： "bacdfeg"

要求：

该字符串只包含小写的英文字母。

给定字符串的长度和 k 在 $[1, 10000]$ 范围内。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历	$O(n)$	$O(1)$
02	遍历	$O(n)$	$O(1)$

```

func reverseStr(s string, k int) string {
    arr := []byte(s)
    for i := 0; i < len(s); i = i + 2*k {
        j := min(i+k, len(s))
        reverse(arr[i:j])
    }
    return string(arr)
}

func reverse(arr []byte) {
    i, j := 0, len(arr)-1
    for i < j {
        arr[i], arr[j] = arr[j], arr[i]
        i++
        j--
    }
}

func min(a, b int) int {
    if a < b {
        return a
    }
    return b
}

func reverseStr(s string, k int) string {
    arr := []byte(s)
    for i := 0; i < len(s); i = i + k {
        if i%(2*k) == 0 {
            j := i + k
            if len(arr) < j {
                j = len(arr)
            }
            reverse(arr[i:j])
        }
    }
    return string(arr)
}

func reverse(arr []byte) {
    i, j := 0, len(arr)-1
    for i < j {
        arr[i], arr[j] = arr[j], arr[i]
        i++
        j--
    }
}

```

543.二叉树的直径(2)

- 题目

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点间最长的路径长度。这条路径可能穿过也可能不穿过根结点。

示例：

给定二叉树



返回 3，它的长度是路径 [4, 2, 1, 3] 或者 [5, 2, 1, 3]。

注意：两结点之间的路径长度是以它们之间边的数目表示。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(\log(n))$
02	栈辅助	$O(n)$	$O(n)$

```

var res int
func diameterOfBinaryTree(root *TreeNode) int {
    res = 0
    dfs(root)
    return res
}

func dfs(root *TreeNode) int {
    if root == nil {
        return 0
    }
    left := dfs(root.Left)
    right := dfs(root.Right)
    path := max(left, right)
    res = max(left+right, res) // 当前节点最大直径与当前保存最大
    return path + 1 // 以该节点为根的最大深度
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

#
func diameterOfBinaryTree(root *TreeNode) int {
    if root == nil {
        return 0
    }
    max := 0
    stack := make([]*TreeNode, 0)
    m := make(map[*TreeNode]int)

    cur := root
    var prev *TreeNode
    for cur != nil || len(stack) != 0 {
        for cur != nil {
            stack = append(stack, cur)
            cur = cur.Left
        }
        cur = stack[len(stack)-1]
        if cur.Right == nil || cur.Right == prev {
            cur = stack[len(stack)-1]
            stack = stack[:len(stack)-1]

            leftLen := 0
            rightLen := 0

```

```

        if v, ok := m[cur.Left]; ok {
            leftLen = v
        }
        if v, ok := m[cur.Right]; ok {
            rightLen = v
        }
        if leftLen > rightLen {
            m[cur] = leftLen + 1
        } else {
            m[cur] = rightLen + 1
        }
        if max < leftLen+rightLen {
            max = leftLen + rightLen
        }
        prev = cur
        cur = nil
    } else {
        cur = cur.Right
    }
}
return max
}

```

551. 学生出勤记录 I(2)

- 题目

给定一个字符串来代表一个学生的出勤记录，这个记录仅包含以下三个字符：

'A' : Absent, 缺勤
'L' : Late, 迟到
'P' : Present, 到场

如果一个学生的出勤记录中不超过一个'A'(缺勤)并且不超过两个连续的'L'(迟到)，你需要根据这个学生的出勤记录判断他是否会被奖赏。

示例 1: 输入: "PPALLP" 输出: True

示例 2: 输入: "PPALLL" 输出: False

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	内置函数	O(n)	O(1)
02	遍历	O(n)	O(1)

```

func checkRecord(s string) bool {
    if strings.Count(s, "A") <= 1 && strings.Count(s, "LLL")
        return true
    }
    return false
}

#
func checkRecord(s string) bool {
    aNum := 0
    lNum := 0
    for i := 0; i < len(s); i++ {
        if s[i] == 'A' {
            aNum++
        }
        if s[i] == 'L' {
            lNum++
        } else {
            lNum = 0
        }
        if aNum == 2 {
            return false
        }
        if lNum == 3 {
            return false
        }
    }
    return true
}

```

557. 反转字符串中的单词 III(2)

- 题目

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，同时仍保留空格和点。

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！

- 解题思路分析

No.	思路	时间复杂度	空间复杂度
01	内置函数	$O(n)$	$O(n)$
02	遍历	$O(n)$	$O(n)$

```

func reverseWords(s string) string {
    strS := strings.Split(s, " ")
    for i, s := range strS {
        strS[i] = reverse(s)
    }
    return strings.Join(strS, " ")
}

func reverse(s string) string {
    arr := []byte(s)
    i, j := 0, len(arr)-1
    for i < j {
        arr[i], arr[j] = arr[j], arr[i]
        i++
        j--
    }
    return string(arr)
}

#
func reverseWords(s string) string {
    arr := []byte(s)
    j := 0
    for i := 0; i < len(arr); i++ {
        if arr[i] == ' ' {
            reverse(arr, j, i-1)
            j = i + 1
        }
    }
    reverse(arr, j, len(arr)-1)
    return string(arr)
}

func reverse(arr []byte, i, j int) []byte {
    for i < j {
        arr[i], arr[j] = arr[j], arr[i]
        i++
        j--
    }
    return arr
}

```

559.N叉树的最大深度(2)

- 题目

给定一个 N 叉树，找到其最大深度。

最大深度是指从根节点到最远叶子节点的最长路径上的节点总数。

例如，给定一个 3叉树：

我们应返回其最大深度，3。

说明：

树的深度不会超过 1000。

树的节点总不会超过 5000。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(\log(n))$
02	迭代	$O(n)$	$O(n)$

```

func maxDepth(root *Node) int {
    if root == nil {
        return 0
    }
    depth := 0
    for _, node := range root.Children {
        depth = max(depth, maxDepth(node))
    }
    return depth + 1
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

#
func maxDepth(root *Node) int {
    if root == nil {
        return 0
    }
    queue := make([]*Node, 0)
    depth := 0
    queue = append(queue, root)
    for len(queue) > 0 {
        length := len(queue)
        for i := 0; i < length; i++ {
            temp := queue[0]
            for _, node := range temp.Children {
                queue = append(queue, node)
            }
            queue = queue[1:]
        }
        depth++
    }
    return depth
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

```

561. 数组拆分 I(2)

- 题目

给定长度为 $2n$ 的数组，你的任务是将这些数分成 n 对，例如 $(a_1, b_1), \dots, (a_n, b_n)$ ，使得从 1 到 n 的 $\min(a_i, b_i)$ 总和最大。

示例 1: 输入: [1, 4, 3, 2] 输出: 4

解释: n 等于 2, 最大总和为 $4 = \min(1, 2) + \min(3, 4)$.

提示:

n 是正整数, 范围在 [1, 10000].

数组中的元素范围在 [-10000, 10000].

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	排序遍历	$O(n\log(n))$	P
02	数组辅助	$O(n)$	$O(1)$

```

func arrayPairSum(nums []int) int {
    sort.Ints(nums)
    sum := 0
    for k, v := range nums {
        if k%2 == 0 {
            sum = sum + v
        }
    }
    return sum
}

#
func arrayPairSum(nums []int) int {
    var arr [20010]int
    for _, num := range nums {
        arr[num+10000]++
    }
    sum := 0
    needAdd := true
    for num, count := range arr {
        for count > 0 {
            if needAdd {
                sum = sum + num - 10000
            }
            needAdd = !needAdd
            count--
        }
    }
    return sum
}

```

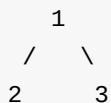
563.二叉树的坡度(2)

- 题目

给定一个二叉树，计算整个树的坡度。
一个树的节点的坡度定义即为，该节点左子树的结点之和和右子树结点之和的差|
整个树的坡度就是其所有节点的坡度之和。

示例：

输入：



输出：1

解释：

结点的坡度 2 : 0

结点的坡度 3 : 0

结点的坡度 1 : $|2-3| = 1$

树的坡度 : $0 + 0 + 1 = 1$

注意：

任何子树的结点的和不会超过32位整数的范围。

坡度的值不会超过32位整数的范围。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(\log(n))$
02	迭代	$O(n)$	$O(n)$

```

var total int

func findTilt(root *TreeNode) int {
    total = 0
    dfs(root)
    return total
}

func dfs(root *TreeNode) int {
    if root == nil {
        return 0
    }
    left := dfs(root.Left)
    right := dfs(root.Right)
    total = total + abs(left, right)
    return left + right + root.Val // 返回节点之和
}

func abs(a, b int) int {
    if a > b {
        return a - b
    }
    return b - a
}

#
func findTilt(root *TreeNode) int {
    if root == nil {
        return 0
    }
    stack := make([]*TreeNode, 0)
    stack = append(stack, root)
    list := make([]*TreeNode, 0)
    total := 0
    for len(stack) > 0 {
        node := stack[len(stack)-1]
        stack = stack[0 : len(stack)-1]
        list = append([]*TreeNode{node}, list...)
        if node.Left != nil {
            stack = append(stack, node.Left)
        }
        if node.Right != nil {
            stack = append(stack, node.Right)
        }
    }
    for i := range list {
        node := list[i]
        left := 0

```

```
    right := 0
    if node.Left != nil {
        left = node.Left.Val
    }
    if node.Right != nil {
        right = node.Right.Val
    }
    total = total + abs(left, right)
    node.Val = left + right + node.Val
}
return total
}

func abs(a, b int) int {
    if a > b {
        return a - b
    }
    return b - a
}
```

566. 重塑矩阵(2)

- 题目

在MATLAB中，有一个非常有用的函数 `reshape`，它可以将一个矩阵重塑为另一个由二维数组表示的矩阵，以及两个正整数r和c，分别表示想要的重构的矩阵。重构后的矩阵需要将原始矩阵的所有元素以相同的行遍历顺序填充。

如果具有给定参数的`reshape`操作是可行且合理的，则输出新的重塑矩阵；否则

示例 1: 输入:

```
nums =
[[1,2],
 [3,4]]
r = 1, c = 4
```

输出:

```
[[1,2,3,4]]
```

解释: 行遍历`nums`的结果是 `[1, 2, 3, 4]`。新的矩阵是 `1 * 4` 矩阵，用之前的

示例 2: 输入:

```
nums =
[[1,2],
 [3,4]]
r = 2, c = 4
```

输出:

```
[[1,2],
 [3,4]]
```

解释: 没有办法将 `2 * 2` 矩阵转化为 `2 * 4` 矩阵。所以输出原矩阵。

注意:

给定矩阵的宽和高范围在 `[1, 100]`。

给定的 `r` 和 `c` 都是正数。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历	$O(n^2)$	$O(n^2)$
02	遍历	$O(n^2)$	$O(n^2)$

```

func matrixReshape(nums [][]int, r int, c int) [][]int {
    row, col := len(nums), len(nums[0])
    if (row*col != r*c) || (row == r && col == c) {
        return nums
    }
    res := make([][]int, r)
    for i := 0; i < len(res); i++ {
        res[i] = make([]int, c)
    }
    for i := 0; i < r*c; i++ {
        res[i/c][i%c] = nums[i/col][i%col]
    }
    return res
}

#
func matrixReshape(nums [][]int, r int, c int) [][]int {
    row, col := len(nums), len(nums[0])
    if (row*col != r*c) || (row == r && col == c) {
        return nums
    }
    res := make([][]int, 0)
    arr := make([]int, 0)
    count := 0
    for _, num := range nums {
        for _, value := range num {
            arr = append(arr, value)
            count++
            if count == c {
                res = append(res, arr)
                arr = []int{}
                count = 0
            }
        }
    }
    return res
}

```

572. 另一个树的子树(3)

- 题目

给定两个非空二叉树 s 和 t , 检验 s 中是否包含和 t 具有相同结构和节点值的一个子树包括 s 的一个节点和这个节点的所有子孙。 s 也可以看做它自身

示例 1:

给定的树 s :

```
    3
   / \
  4   5
 / \
1   2
```

给定的树 t :

```
    4
   / \
  1   2
```

返回 `true`, 因为 t 与 s 的一个子树拥有相同的结构和节点值。

示例 2:

给定的树 s :

```
    3
   / \
  4   5
 / \
1   2
 /
  0
```

给定的树 t :

```
    4
   / \
  1   2
```

返回 `false`。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n^2)$	$O(\log(n))$
02	递归+字符串辅助	$O(n)$	$O(\log(n))$
03	栈辅助	$O(n)$	$O(n)$

```

func isSubtree(s *TreeNode, t *TreeNode) bool {
    if s == nil {
        return false
    }
    return isSame(s, t) || isSubtree(s.Left, t) || isSubtree(s.Right, t)
}

func isSame(s *TreeNode, t *TreeNode) bool {
    if s == nil || t == nil{
        return t == s
    }
    return isSame(s.Left, t.Left) && isSame(s.Right, t.Right)
}

#dfs
func isSubtree(s *TreeNode, t *TreeNode) bool {
    sStr := dfs(s, "")
    tStr := dfs(t, "")
    return strings.Contains(sStr, tStr)
}

func dfs(s *TreeNode, pre string) string {
    if s == nil {
        return pre
    }
    return fmt.Sprintf("#%d%s%", s.Val, dfs(s.Left, "1"),
}
}

#preOrder
func preOrder(root *TreeNode) string {
    if root == nil {
        return ""
    }
    res := "!"
    stack := make([]*TreeNode, 0)
    temp := root
    for {
        for temp != nil{
            res += strconv.Itoa(temp.Val)
            res += "!"
            stack = append(stack, temp)
            temp = temp.Left
        }
        res += "#!"
        if len(stack) > 0{
            node := stack[len(stack)-1]
            stack = stack[:len(stack)-1]
            temp = node.Right
        }
    }
}

```

```

        }else {
            break
        }
    }
    return res
}

```

575. 分糖果(2)

- 题目

给定一个偶数长度的数组，其中不同的数字代表着不同种类的糖果，每一个数字你需要把这些糖果平均分给一个弟弟和一个妹妹。返回妹妹可以获得的最大糖果数。

示例 1: 输入: candies = [1,1,2,2,3,3] 输出: 3

解析: 一共有三种种类的糖果，每一种都有两个。

最优分配方案: 妹妹获得[1, 2, 3], 弟弟也获得[1, 2, 3]。这样使妹妹获得的糖果种类数最多。

示例 2 : 输入: candies = [1,1,2,3] 输出: 2

解析: 妹妹获得糖果[2, 3], 弟弟获得糖果[1, 1]，妹妹有两种不同的糖果，弟弟有一种。这样使得妹妹可以获得的糖果种类数最多。

注意:

数组的长度为[2, 10,000]，并且确定为偶数。

数组中数字的大小在范围[-100,000, 100,000]内。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	哈希辅助	O(n)	O(n)
02	排序遍历	O(nlog(n))	O(1)

```

func distributeCandies(candies []int) int {
    n := len(candies)
    r := make(map[int]bool, n)
    for _, c := range candies {
        r[c] = true
    }
    return min(len(r), n/2)
}

func min(a, b int) int {
    if a < b {
        return a
    }
    return b
}

#
func distributeCandies(candies []int) int {
    length := len(candies)
    half := length / 2
    count := 1
    sort.Ints(candies)
    for i := 1; i < length; i++ {
        if candies[i] != candies[i-1] {
            count++
        }
    }
    if count >= half {
        return half
    }
    return count
}

```

581.最短无序连续子数组(3)

- 题目

给定一个整数数组，你需要寻找一个连续的子数组，如果对这个子数组进行升序：你找到的子数组应是最短的，请输出它的长度。

示例 1: 输入: [2, 6, 4, 8, 10, 9, 15] 输出: 5

解释: 你只需要对 [6, 4, 8, 10, 9] 进行升序排序，那么整个表都会变为：

说明：

输入的数组长度范围在 [1, 10,000]。

输入的数组可能包含重复元素，所以升序的意思是 \leq 。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	双指针	$O(n)$	$O(1)$
02	2次遍历	$O(n)$	$O(1)$
03	排序遍历	$O(n \log(n))$	$O(n)$

```

func findUnsortedSubarray(nums []int) int {
    length := len(nums)
    left, right := 0, -1
    min, max := nums[length-1], nums[0]
    for i := 1; i < length; i++ {
        if max <= nums[i] {
            max = nums[i]
        } else {
            right = i
        }
        j := length - i - 1
        if min >= nums[j] {
            min = nums[j]
        } else {
            left = j
        }
    }
    return right - left + 1
}

#
func findUnsortedSubarray(nums []int) int {
    length := len(nums)
    right := -1
    max := nums[0]
    for i := 1; i < length; i++ {
        if max <= nums[i] {
            max = nums[i]
        } else {
            right = i
        }
    }
    if right == 0 {
        // 针对升序，特殊处理
        // 如去掉判断
        // 需要保证left,right初始值满足right-left+1=0
        return 0
    }
    left := 0
    min := nums[length-1]
    for i := length - 2; i >= 0; i-- {
        if min >= nums[i] {
            min = nums[i]
        } else {
            left = i
        }
    }
    return right - left + 1
}

```

```

}

#
func findUnsortedSubarray(nums []int) int {
    temp := make([]int, len(nums))
    copy(temp, nums)
    sort.Ints(temp)
    i, j := 0, len(nums)-1
    for i < len(nums) && nums[i] == temp[i]{
        i++
    }
    for i+1 < j && nums[j] == temp[j]{
        j--
    }
    return j-i+1
}

```

589.N叉树的前序遍历(2)

- 题目

给定一个 N 叉树，返回其节点值的前序遍历。

例如，给定一个 3叉树：

返回其前序遍历：[1, 3, 5, 6, 2, 4]。

说明：递归法很简单，你可以使用迭代法完成此题吗？

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	O(n)	O(log(n))
02	迭代	O(n)	O(n)

```

var res []int

func preorder(root *Node) []int {
    res = make([]int, 0)
    dfs(root)
    return res
}

func dfs(root *Node) {
    if root == nil {
        return
    }
    res = append(res, root.Val)
    for _, value := range root.Children {
        dfs(value)
    }
}

#
func preorder(root *Node) []int {
    res := make([]int, 0)
    if root == nil {
        return res
    }
    stack := make([]*Node, 0)
    stack = append(stack, root)
    for len(stack) > 0 {
        temp := stack[len(stack)-1]
        stack = stack[:len(stack)-1]
        res = append(res, temp.Val)
        for i := len(temp.Children) - 1; i >= 0; i-- {
            stack = append(stack, temp.Children[i])
        }
    }
    return res
}

```

590.N叉树的后序遍历(2)

- 题目

给定一个 N 叉树，返回其节点值的后序遍历。

例如，给定一个 3叉树：

返回其后序遍历：[5, 6, 3, 2, 4, 1].

说明：递归法很简单，你可以使用迭代法完成此题吗？

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(\log(n))$
02	迭代	$O(n)$	$O(n)$

```

var res []int
func postorder(root *Node) []int {
    res = make([]int, 0)
    dfs(root)
    return res
}

func dfs(root *Node) {
    if root == nil {
        return
    }
    for _, value := range root.Children {
        dfs(value)
    }
    res = append(res, root.Val)
}

#
// 后序: (左右)根
// 前序: 根(左右)=>根(右左)=>左右根
func postorder(root *Node) []int {
    res := make([]int, 0)
    if root == nil {
        return res
    }
    stack := make([]*Node, 0)
    stack = append(stack, root)
    for len(stack) > 0 {
        temp := stack[len(stack)-1]
        stack = stack[:len(stack)-1]
        res = append(res, temp.Val)
        for i := 0; i < len(temp.Children); i++ {
            stack = append(stack, temp.Children[i])
        }
    }
    for i := 0; i < len(res)/2; i++ {
        res[i], res[len(res)-1-i] = res[len(res)-1-i], res[i]
    }
    return res
}

```

594.最长和谐子序列(2)

- 题目

和谐数组是指一个数组里元素的最大值和最小值之间的差别正好是1。
现在，给定一个整数数组，你需要在所有可能的子序列中找到最长的和谐子序列！
示例 1: 输入: [1, 3, 2, 2, 5, 2, 3, 7] 输出: 5
原因: 最长的和谐数组是: [3, 2, 2, 2, 3].
说明: 输入的数组长度最大不超过20,000.

• 解题思路

No.	思路	时间复杂度	空间复杂度
01	哈希辅助	$O(n)$	$O(n)$
02	排序遍历	$O(n\log(n))$	$O(1)$

```

func findLHS(nums []int) int {
    m := make(map[int]int, len(nums))
    for _, n := range nums {
        m[n]++
    }
    res := 0
    for key, value := range m {
        value2, ok := m[key+1]
        if ok {
            t := value + value2
            if res < t {
                res = t
            }
        }
    }
    return res
}

#
func findLHS(nums []int) int {
    sort.Ints(nums)
    res := 0
    left := 0
    for i := 0; i < len(nums); i++ {
        for nums[i]-nums[left] > 1 {
            left++
        }
        if nums[i]-nums[left] == 1 {
            if res < i-left+1 {
                res = i - left + 1
            }
        }
    }
    return res
}

```

598. 范围求和 II(1)

- 题目

给定一个初始元素全部为 0，大小为 $m \times n$ 的矩阵 M 以及在 M 上的一系列更新操作用二维数组表示，其中的每个操作用一个含有两个正整数 a 和 b 的数组表示。含义是将所有符合 $0 \leq i < a$ 以及 $0 \leq j < b$ 的元素 $M[i][j]$ 的值都加 1。在执行给定的一系列操作后，你需要返回矩阵中含有最大整数的元素个数。

示例 1: 输入: $m = 3$, $n = 3$ operations = [[2,2],[3,3]] 输出: 4

解释: 初始状态, $M =$

```
[[0, 0, 0],
 [0, 0, 0],
 [0, 0, 0]]
```

执行完操作 [2,2] 后, $M =$

```
[[1, 1, 0],
 [1, 1, 0],
 [0, 0, 0]]
```

执行完操作 [3,3] 后, $M =$

```
[[2, 2, 1],
 [2, 2, 1],
 [1, 1, 1]]
```

M 中最大的整数是 2，而且 M 中有 4 个值为 2 的元素。因此返回 4。

注意：

m 和 n 的范围是 [1, 40000]。

a 的范围是 [1, m]， b 的范围是 [1, n]。

操作数目不超过 10000。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	数学	$O(n)$	$O(1)$

```
func maxCount(m int, n int, ops [][]int) int {
    for _, o := range ops {
        m = min(m, o[0])
        n = min(n, o[1])
    }
    return m * n
}

func min(a, b int) int {
    if a < b {
        return a
    }
    return b
}
```

599.两个列表的最小索引总和(2)

- 题目

假设Andy和Doris想在晚餐时选择一家餐厅，并且他们都有一个表示最喜爱餐厅的列表。你需要帮助他们用最少的索引和找出他们共同喜爱的餐厅。如果答案不止一个，你可以假设总是存在一个答案。

示例 1: 输入:

```
["Shogun", "Tapioca Express", "Burger King", "KFC"]
["Piatti", "The Grill at Torrey Pines", "Hungry Hunter Steakhouse", "Shogun"]
输出: ["Shogun"]
```

解释: 他们唯一共同喜爱的餐厅是“Shogun”。

示例 2:

输入:

```
["Shogun", "Tapioca Express", "Burger King", "KFC"]
["KFC", "Shogun", "Burger King"]
```

输出: ["Shogun"]

解释: 他们共同喜爱且具有最小索引和的餐厅是“Shogun”，它有最小的索引和。

提示:

两个列表的长度范围都在 [1, 1000] 内。

两个列表中的字符串的长度将在 [1, 30] 的范围内。

下标从 0 开始，到列表的长度减 1。

两个列表都没有重复的元素。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	哈希辅助	O(n)	O(n)
02	暴力法	O(n^2)	

```

func findRestaurant(list1 []string, list2 []string) []string {
    if len(list1) > len(list2) {
        list1, list2 = list2, list1
    }
    m2 := make(map[string]int, len(list2))
    for i := range list2 {
        m2[list2[i]] = i
    }
    min := 2000
    res := make([]string, 0, 1000)
    for key, value := range list1 {
        if key2, ok := m2[value]; ok {
            if min == key+key2 {
                res = append(res, value)
            }
            if min > key+key2 {
                min = key + key2
                res = []string{value}
            }
        }
    }
    return res
}

#
func findRestaurant(list1 []string, list2 []string) []string {
    min := 2000
    res := make([]string, 0, 1000)
    for key1, value1 := range list1 {
        for key2, value2 := range list2{
            if value1 == value2{
                if min == key1+key2 {
                    res = append(res, value1)
                }
                if min > key1+key2 {
                    min = key1 + key2
                    res = []string{value1}
                }
            }
        }
    }
    return res
}

```

0601-0700-Easy

- 0601-0700-Easy
 - 605.种花问题(3)
 - 606.根据二叉树创建字符串(2)
 - 617.合并二叉树(2)
 - 628.三个数的最大乘积(2)
 - 633.平方数之和(2)
 - 637.二叉树的层平均值(2)
 - 643.子数组最大平均数 I(3)
 - 645.错误的集合(5)
 - 653.两数之和IV输入BST(4)
 - 657.机器人能否返回原点(2)
 - 661.图片平滑器(2)
 - 665.非递减数列(3)
 - 669.修剪二叉搜索树(2)
 - 671.二叉树中第二小的节点(3)
 - 674.最长连续递增序列(3)
 - 680.验证回文字符串 II (2)
 - 682.棒球比赛(1)
 - 686.重复叠加字符串匹配

605.种花问题(3)

- 题目

假设你有一个很长的花坛，一部分地块种植了花，另一部分却没有。

可是，花卉不能种植在相邻的地块上，它们会争夺水源，两者都会死去。

给定一个花坛（表示为一个数组包含0和1，其中0表示没种植花，1表示种植了花）能否在不打破种植规则的情况下种入 n 朵花？能则返回True，不能则返回False。

示例 1: 输入: flowerbed = [1, 0, 0, 0, 1], n = 1 输出: True

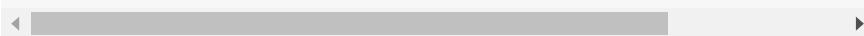
示例 2: 输入: flowerbed = [1, 0, 0, 0, 1], n = 2 输出: False

注意：

数组内已种好的花不会违反种植规则。

输入的数组长度范围为 [1, 20000]。

n 是非负整数，且不会超过输入数组的大小。



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历	$O(n)$	$O(1)$
02	遍历统计	$O(n)$	$O(1)$
03	补数+遍历统计	$O(n)$	$O(1)$

```

func canPlaceFlowers(flowerbed []int, n int) bool {
    length := len(flowerbed)
    // 判断条件
    // 1:当前元素是0
    // 2.前一个元素是0, 或者当前是第一个元素
    // 3.后一个元素是0, 或者当前是最后一个元素
    for i := 0; i < length; i++ {
        if flowerbed[i] == 0 &&
            (i == 0 || flowerbed[i-1] == 0) &&
            (i == length-1 || flowerbed[i+1] == 0) {
            flowerbed[i] = 1
            n--
            if n <= 0 {
                return true
            }
        }
    }
    return n <= 0
}

#
func canPlaceFlowers(flowerbed []int, n int) bool {
    length := len(flowerbed)
    count := 0
    temp := 1
    // 以0开头, 计算情况同以0结束, 为向中间情况靠齐, 可以特殊处理把temp设为1
    // 中间计算可以种花, value = (temp-1)/2
    // 最后结束如果为偶数, value=temp/2
    for i := 0; i < length; i++ {
        if flowerbed[i] == 1 {
            count = count + (temp-1)/2
            temp = 0
        } else {
            temp++
        }
    }
    count = count + temp/2
    return n <= count
}

#
func canPlaceFlowers(flowerbed []int, n int) bool {
    flowerbed = append([]int{0}, flowerbed...)
    flowerbed = append(flowerbed, []int{0, 1}...)
    count := 0
    temp := 0
    // 首补0, 尾补0, 1, 统一一种情况
    for i := 0; i < len(flowerbed); i++ {

```

```

        if flowerbed[i] == 1 {
            count = count + (temp-1)/2
            temp = 0
        } else {
            temp++
        }
    }
    return n <= count
}

```

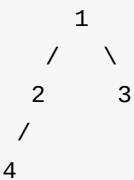
606. 根据二叉树创建字符串(2)

- 题目

你需要采用前序遍历的方式，将一个二叉树转换成一个由括号和整数组成的字符串。空节点则用一对空括号 "()" 表示。而且你需要省略所有不影响字符串与原始二叉树的一致性的空括号对。

示例 1：

输入：二叉树： [1, 2, 3, 4]



输出： "1(2(4))(3)"

解释：原本将是"1(2(4)())(3())"，

在你省略所有不必要的空括号对之后，

它将是"1(2(4))(3)"。

示例 2：

输入：二叉树： [1, 2, 3, null, 4]



输出： "1(2()(4))(3)"

解释：和第一个示例相似，

除了我们不能省略第一个对括号来中断输入和输出之间的一对一映射关系。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(\log(n))$
02	迭代	$O(n)$	$O(n)$

```

func tree2str(t *TreeNode) string {
    if t == nil {
        return ""
    }
    res := strconv.Itoa(t.Val)
    if t.Left == nil && t.Right == nil {
        return res
    }
    res += "(" + tree2str(t.Left) + ")"
    if t.Right != nil{
        res += "(" + tree2str(t.Right) + ")"
    }
    return res
}

#
func tree2str(t *TreeNode) string {
    if t == nil {
        return ""
    }
    stack := make([]*TreeNode, 0)
    m := make(map[*TreeNode]bool)
    stack = append(stack, t)
    res := ""
    for len(stack) > 0 {
        node := stack[len(stack)-1]
        if _, ok := m[node]; ok {
            stack = stack[:len(stack)-1]
            res = res + ")"
        } else {
            m[node] = true
            res = res + "(" + strconv.Itoa(node.Val)
            if node.Left == nil && node.Right != nil {
                res = res + "("
            }
            if node.Right != nil {
                stack = append(stack, node.Right)
            }
            if node.Left != nil {
                stack = append(stack, node.Left)
            }
        }
    }
    return res[1 : len(res)-1]
}

```

617. 合并二叉树(2)

- 题目

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一！你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将！否则不为 NULL 的节点将直接作为新二叉树的节点。

示例 1：

输入：

```
Tree 1
      1
     / \
    3   2
   /
  5
```

```
Tree 2
      2
     / \
    1   3
   \   \
    4   7
```

输出：

合并后的树：

```
      3
     / \
    4   5
   / \   \
  5   4   7
```

注意：合并必须从两个树的根节点开始。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(\log(n))$
02	迭代	$O(n)$	$O(n)$

```

func mergeTrees(t1 *TreeNode, t2 *TreeNode) *TreeNode {
    if t1 == nil {
        return t2
    }
    if t2 == nil {
        return t1
    }
    t1.Val = t1.Val + t2.Val
    t1.Left = mergeTrees(t1.Left, t2.Left)
    t1.Right = mergeTrees(t1.Right, t2.Right)
    return t1
}

#
func mergeTrees(t1 *TreeNode, t2 *TreeNode) *TreeNode {
    if t1 == nil {
        return t2
    }
    if t2 == nil {
        return t1
    }
    list := make([]*TreeNode, 0)
    list = append(list, t1)
    list = append(list, t2)
    for len(list) > 0 {
        node1 := list[0]
        node2 := list[1]
        node1.Val = node1.Val + node2.Val
        if node1.Left != nil && node2.Left != nil {
            list = append(list, node1.Left)
            list = append(list, node2.Left)
        } else if node1.Left == nil && node2.Left != nil {
            node1.Left = node2.Left
        }
        if node1.Right != nil && node2.Right != nil {
            list = append(list, node1.Right)
            list = append(list, node2.Right)
        } else if node1.Right == nil && node2.Right != nil {
            node1.Right = node2.Right
        }
        list = list[2:]
    }
    return t1
}

```

628.三个数的最大乘积(2)

- 题目

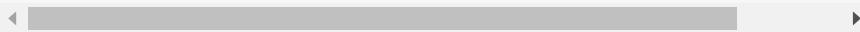
给定一个整型数组，在数组中找出由三个数组成的最大乘积，并输出这个乘积。

示例 1: 输入: [1, 2, 3] 输出: 6

示例 2: 输入: [1, 2, 3, 4] 输出: 24

注意:

给定的整型数组长度范围是[3, 104]，数组中所有的元素范围是[-1000, 1000]。
输入的数组中任意三个数的乘积不会超出32位有符号整数的范围。



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	排序	$O(n \log(n))$	$O(1)$
02	遍历	$O(n)$	$O(1)$

```

func maximumProduct(nums []int) int {
    sort.Ints(nums)
    return max(nums[0]*nums[1]*nums[len(nums)-1],
               nums[len(nums)-3]*nums[len(nums)-2]*nums[len(nums)-
})
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

#  

func maximumProduct(nums []int) int {
    max1, max2, max3 := math.MinInt32, math.MinInt32, math.
    min1, min2 := math.MaxInt32, math.MaxInt32
    for i := 0; i < len(nums); i++ {
        if nums[i] <= min1 {
            min2 = min1
            min1 = nums[i]
        } else if nums[i] <= min2 {
            min2 = nums[i]
        }
        if nums[i] >= max1 {
            max3 = max2
            max2 = max1
            max1 = nums[i]
        } else if nums[i] >= max2 {
            max3 = max2
            max2 = nums[i]
        } else if nums[i] >= max3 {
            max3 = nums[i]
        }
    }
    return max(min1*min2*max1, max1*max2*max3)
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

```

633. 平方数之和(2)

- 题目

给定一个非负整数 c ，你要判断是否存在两个整数 a 和 b ，使得 $a^2 + b^2$

示例1: 输入: 5 输出: True 解释: $1 * 1 + 2 * 2 = 5$

示例2: 输入: 3 输出: False

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	双指针	$O(\log(n))$	$O(1)$
02	遍历	$O(\log(n))$	$O(1)$

```

func judgeSquareSum(c int) bool {
    if c < 0 {
        return false
    }
    i, j := 0, int(math.Sqrt(float64(c)))
    for i <= j {
        current := i*i + j*j
        if current < c {
            i++
        } else if current > c {
            j--
        } else {
            return true
        }
    }
    return false
}

#
func judgeSquareSum(c int) bool {
    for i := 0; i <= int(math.Sqrt(float64(c))); i++ {
        b := c - i*i
        s := int(math.Sqrt(float64(b)))
        if s*s == b {
            return true
        }
    }
    return false
}

```

637.二叉树的层平均值(2)

- 题目

给定一个非空二叉树，返回一个由每层节点平均值组成的数组。

示例 1：

输入：

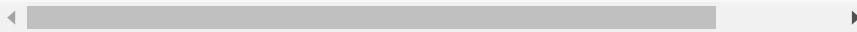
```
3
 / \
9  20
 /  \
15  7
```

输出：[3, 14.5, 11]

解释：第0层的平均值是 3， 第1层是 14.5， 第2层是 11。因此返回 [3, 11]。

注意：

节点值的范围在32位有符号整数范围内。



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(\log(n))$
02	迭代	$O(n)$	$O(n)$

```

func averageOfLevels(root *TreeNode) []float64 {
    var sum, node []int
    res := make([]float64, 0)
    sum = append(sum, root.Val)
    node = append(node, 1)
    sum, node = dfs(root, sum, node, 1)
    for i := 0; i < len(sum); i++ {
        res = append(res, float64(sum[i])/float64(node[i]))
    }
    return res
}

func dfs(root *TreeNode, sum, node []int, level int) ([]int {
    if root == nil || (root.Left == nil && root.Right == nil) {
        return sum, node
    }
    if level >= len(sum) {
        sum = append(sum, 0)
        node = append(node, 0)
    }
    if root.Left != nil {
        sum[level] += root.Left.Val
        node[level]++
    }
    if root.Right != nil {
        sum[level] += root.Right.Val
        node[level]++
    }
    sum, node = dfs(root.Left, sum, node, level+1)
    sum, node = dfs(root.Right, sum, node, level+1)
    return sum, node
}

#
func averageOfLevels(root *TreeNode) []float64 {
    res := make([]float64, 0)
    list := make([]*TreeNode, 0)
    list = append(list, root)
    for len(list) > 0 {
        length := len(list)
        sum := 0
        for i := 0; i < length; i++ {
            sum = sum + list[i].Val
            if list[i].Left != nil {
                list = append(list, list[i].Left)
            }
            if list[i].Right != nil {
                list = append(list, list[i].Right)
            }
        }
        res = append(res, float64(sum)/float64(length))
    }
    return res
}

```

```

        }
    }
    res = append(res, float64(sum)/float64(length))
    list = list[length:]
}
return res
}

```

643. 子数组最大平均数 I(3)

- 题目

给定 n 个整数，找出平均数最大且长度为 k 的连续子数组，并输出该最大平均数。

示例 1: 输入: [1, 12, -5, -6, 50, 3], $k = 4$ 输出: 12.75

解释: 最大平均数 $(12-5-6+50)/4 = 51/4 = 12.75$

注意:

$1 \leq k \leq n \leq 30,000$ 。

所给数据范围 $[-10,000, 10,000]$ 。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历+滑动窗口	$O(n)$	$O(1)$
02	遍历+暴力法	$O(n^2)$	$O(1)$
03	遍历+累计求和	$O(n)$	$O(n)$

```

func findMaxAverage(nums []int, k int) float64 {
    temp := 0
    for i := 0; i < k; i++ {
        temp = temp + nums[i]
    }
    max := temp
    for i := k; i < len(nums); i++ {
        temp = temp + nums[i] - nums[i-k]
        if max < temp {
            max = temp
        }
    }
    return float64(max) / float64(k)
}

#
func findMaxAverage(nums []int, k int) float64 {
    max := math.MinInt32
    for i := 0; i < len(nums); i++ {
        if i + k > len(nums){
            break
        }
        sum := 0
        for j := i; j < i+k; j++{
            sum = sum+nums[j]
        }
        if sum > max{
            max = sum
        }
    }
    return float64(max) / float64(k)
}

#
func findMaxAverage(nums []int, k int) float64 {
    sum := make([]int, len(nums))
    sum[0] = nums[0]
    for i := 1; i < len(nums); i++ {
        sum[i] = sum[i-1] + nums[i]
    }
    max := sum[k-1]
    for i := k; i < len(nums); i++ {
        if sum[i]-sum[i-k] > max {
            max = sum[i] - sum[i-k]
        }
    }
    return float64(max) / float64(k)
}

```

645. 错误的集合(5)

- 题目

集合 S 包含从1到 n 的整数。不幸的是，因为数据错误，导致集合里面某一个元素复制了成了集合里面的另外一个元素的值，导致集合丢失了一个元素。给定一个数组 nums 代表了集合 S 发生错误后的结果。

你的任务是首先寻找到重复出现的整数，再找到丢失的整数，将它们以数组的形式返回。

示例 1: 输入: $\text{nums} = [1, 2, 2, 4]$ 输出: $[2, 3]$

注意:

给定数组的长度范围是 $[2, 10000]$ 。

给定的数组是无序的。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	数组辅助	$O(n)$	$O(n)$
02	置反	$O(n)$	$O(1)$
03	位运算	$O(n)$	$O(1)$
04	哈希辅助	$O(n)$	$O(n)$
05	排序	$O(n \log(n))$	$O(1)$

```

func findErrorNums(nums []int) []int {
    newNums := make([]int, len(nums))
    var repeatNum int
    for _, v := range nums {
        if newNums[v-1] != 0 {
            repeatNum = v
        }
        newNums[v-1] = v
    }
    for i, v := range newNums {
        if v == 0 {
            return []int{repeatNum, i + 1}
        }
    }
    return []int{0, 0}
}

#
func findErrorNums(nums []int) []int {
    repeatNum := 0
    for i := 0; i < len(nums); i++ {
        n := abs(nums[i])
        if nums[n-1] < 0 {
            repeatNum = n
        } else {
            nums[n-1] = -nums[n-1]
        }
    }
    misNum := 0
    for i, v := range nums {
        if v > 0 {
            misNum = i + 1
            break
        }
    }
    return []int{repeatNum, misNum}
}

func abs(a int) int {
    if a > 0 {
        return a
    }
    return -a
}

#
func findErrorNums(nums []int) []int {
    res := 0

```

```

// 异或得到repeatedNum^misNum
for i := 0; i < len(nums); i++ {
    res = res ^ (i + 1) ^ (nums[i])
}
// 找到第一位不是0的
h := 1
for res&h == 0 {
    h = h << 1
}
a := 0
b := 0
for i := range nums {
    if h&nums[i] == 0 {
        a ^= nums[i]
    } else {
        b ^= nums[i]
    }
    if h&(i+1) == 0 {
        a ^= i + 1
    } else {
        b ^= i + 1
    }
}
for i := range nums {
    if nums[i] == b {
        return []int{b, a}
    }
}
return []int{a, b}
}

#
func findErrorNums(nums []int) []int {
    m := make(map[int]int)
    n := len(nums)
    sum := 0
    repeatNum := 0
    for i := 0; i < len(nums); i++ {
        sum = sum + nums[i]
        if _, ok := m[nums[i]]; ok {
            repeatNum = nums[i]
        }
        m[nums[i]] = 1
    }
    return []int{repeatNum, n*(n+1)/2 - sum + repeatNum}
}
#

```

```

func findErrorNums(nums []int) []int {
    sort.Ints(nums)
    n := len(nums)
    sum := 0
    repeatNum := nums[0]
    for i := 0; i < len(nums); i++ {
        sum = sum + nums[i]
        if i < len(nums)-1 && nums[i] == nums[i+1] {
            repeatNum = nums[i]
        }
    }
    return []int{repeatNum, n*(n+1)/2 - sum + repeatNum}
}

```

653. 两数之和IV 输入 BST(4)

- 题目

给定一个二叉搜索树和一个目标结果，如果 BST 中存在两个元素且它们的和等于目标结果。

案例 1：

输入：

```

      5
     / \
    3   6
   / \   \
  2   4   7
Target = 9

```

输出：True

案例 2：

输入：

```

      5
     / \
    3   6
   / \   \
  2   4   7

```

Target = 28 输出：False

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归+哈希辅助	$O(n)$	$O(n)$
02	递归	$O(n \log(n))$	$O(\log(n))$
03	迭代	$O(n)$	$O(n)$
04	递归+二分查找	$O(n)$	$O(n)$

```

func findTarget(root *TreeNode, k int) bool {
    if root == nil {
        return false
    }
    m := map[int]int{}
    return dfs(root, k, m)
}

func dfs(node *TreeNode, k int, m map[int]int) bool {
    if node == nil {
        return false
    }
    if _, ok := m[k-node.Val]; ok {
        return true
    }
    m[node.Val] = node.Val
    return dfs(node.Left, k, m) || dfs(node.Right, k, m)
}

#
func dfs(root, searchRoot *TreeNode, k int) bool {
    if root == nil {
        return false
    }
    found := findNode(searchRoot, k-root.Val)
    if found != nil && found != root {
        return true
    }
    return dfs(root.Left, searchRoot, k) ||
        dfs(root.Right, searchRoot, k)
}

func findNode(root *TreeNode, target int) *TreeNode {
    if root == nil {
        return nil
    }
    if root.Val == target {
        return root
    }
    if root.Val < target {
        return findNode(root.Right, target)
    }
    return findNode(root.Left, target)
}

#
func findTarget(root *TreeNode, k int) bool {
    if root == nil {

```

```

        return false
    }
    m := make(map[int]int)
    queue := make([]*TreeNode, 0)
    queue = append(queue, root)
    for len(queue) > 0 {
        node := queue[len(queue)-1]
        queue = queue[:len(queue)-1]
        if _, ok := m[k-node.Val]; ok {
            return true
        }
        if node.Left != nil {
            queue = append(queue, node.Left)
        }
        if node.Right != nil {
            queue = append(queue, node.Right)
        }
        m[node.Val] = 1
    }
    return false
}

#
var arr []int

func findTarget(root *TreeNode, k int) bool {
    if root == nil {
        return false
    }
    arr = make([]int, 0)
    dfs(root)
    i := 0
    j := len(arr) - 1
    for i < j {
        if arr[i]+arr[j] == k {
            return true
        } else if arr[i]+arr[j] > k {
            j--
        } else {
            i++
        }
    }
    return false
}

func dfs(node *TreeNode) {
    if node == nil {
        return
    }
}

```

```

    }
    dfs(node.Left)
    arr = append(arr, node.Val)
    dfs(node.Right)
}

```

657. 机器人能否返回原点(2)

- 题目

在二维平面上，有一个机器人从原点 $(0, 0)$ 开始。

给出它的移动顺序，判断这个机器人在完成移动后是否在 $(0, 0)$ 处结束。

移动顺序由字符串表示。字符 $move[i]$ 表示其第 i 次移动。

机器人的有效动作有 R (右) , L (左) , U (上) 和 D (下)。

如果机器人在完成所有动作后返回原点，则返回 `true`。否则，返回 `false`。

注意：机器人“面朝”的方向无关紧要。“R”将始终使机器人向右移动一次，“L”此外，假设每次移动机器人的移动幅度相同。

示例 1: 输入: "UD" 出: true

解释：机器人向上移动一次，然后向下移动一次。

所有动作都具有相同的幅度，因此它最终回到它开始的原点。因此，我们返回 `true`。

示例 2: 输入: "LL"输出: false

解释：机器人向左移动两次。它最终位于原点的左侧，距原点有两次“移动”的幅度。因此，我们返回 `false`，因为它在移动结束时没有返回原点。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	内置函数-字符统计	$O(n)$	$O(1)$
02	模拟	$O(n)$	$O(1)$

```
func judgeCircle(moves string) bool {
    return strings.Count(moves, "U") == strings.Count(moves, "D") && strings.Count(moves, "L") == strings.Count(moves, "R")
}

#
func judgeCircle(moves string) bool {
    x, y := 0, 0
    for i := range moves {
        switch i {
        case 'U':
            y = y + 1
        case 'D':
            y = y - 1
        case 'L':
            x = x - 1
        case 'R':
            x = x + 1
        }
    }
    return x == 0 && y == 0
}
```

661. 图片平滑器(2)

- 题目

包含整数的二维矩阵 M 表示一个图片的灰度。你需要设计一个平滑器来让每一个平均灰度的计算是周围的8个单元和它本身的值求平均，如果周围的单元格不足/

示例 1:

输入:

```
[[1,1,1],
 [1,0,1],
 [1,1,1]]
```

输出:

```
[[0, 0, 0],
 [0, 0, 0],
 [0, 0, 0]]
```

解释:

对于点 $(0,0), (0,2), (2,0), (2,2)$: 平均($3/4$) = 平均(0.75) = 0

对于点 $(0,1), (1,0), (1,2), (2,1)$: 平均($5/6$) = 平均(0.83333333

对于点 $(1,1)$: 平均($8/9$) = 平均(0.88888889) = 0

注意:

给定矩阵中的整数范围为 $[0, 255]$ 。

矩阵的长和宽的范围均为 $[1, 150]$ 。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	遍历	$O(n^2)$	$O(n^2)$
02	遍历	$O(n^2)$	$O(n^2)$

```

func imageSmoother(M [][]int) [][]int {
    res := make([][]int, len(M))
    for i := range res {
        res[i] = make([]int, len(M[0]))
        for j := range res[i] {
            res[i][j] = getValue(M, i, j)
        }
    }
    return res
}

func getValue(M [][]int, r, c int) int {
    value, count := 0, 0
    for i := r - 1; i < r+2; i++ {
        for j := c - 1; j < c+2; j++ {
            if 0 <= i && i < len(M) && 0 <= j && j < len(M) {
                value = value + M[i][j]
                count++
            }
        }
    }
    return value / count
}

func imageSmoother(M [][]int) [][]int {
    res := make([][]int, len(M))
    for i := range res {
        res[i] = make([]int, len(M[0]))
        for j := range res[i] {
            value, count := 0, 0
            for r := i - 1; r <= i+1; r++ {
                for c := j - 1; c <= j+1; c++ {
                    if 0 <= r && r < len(M) && 0 <= c && c < len(M) {
                        value = value + M[r][c]
                        count++
                    }
                }
            }
            res[i][j] = value / count
        }
    }
    return res
}

```

665. 非递减数列(3)

- 题目

给你一个长度为 n 的整数数组，请你判断在 最多 改变 1 个元素的情况下，是否能将数组变成非递减数列。我们是这样定义一个非递减数列的：对于数组中所有的 i ($1 \leq i < n$)，总满足 $nums[i] \leq nums[i + 1]$ 。

示例 1: 输入: $nums = [4, 2, 3]$ 输出: `true`

解释: 你可以通过把第一个4变成1来使得它成为一个非递减数列。

示例 2: 输入: $nums = [4, 2, 1]$ 输出: `false`

解释: 你不能在只改变一个元素的情况下将其变为非递减数列。

说明:

```
1 <= n <= 10 ^ 4
- 10 ^ 5 <= nums[i] <= 10 ^ 5
```

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	暴力法	$O(n^2)$	$O(n)$
02	遍历修改前后	$O(n)$	$O(n)$
03	遍历	$O(n)$)

```

func checkPossibility(nums []int) bool {
    for i := 0; i < len(nums); i++ {
        res := make([]int, 0)
        res = append(res, nums[0:i]...)
        res = append(res, nums[i+1:]...)
        if isSort(res) {
            return true
        }
    }
    return false
}

func isSort(nums []int) bool {
    for i := 0; i < len(nums)-1; i++ {
        if nums[i] > nums[i+1] {
            return false
        }
    }
    return true
}

func checkPossibility(nums []int) bool {
    for i := 1; i < len(nums); i++{
        if nums[i-1] > nums[i]{
            pre := deepcopy(nums)
            pre[i-1] = pre[i]
            next := deepcopy(nums)
            next[i] = next[i-1]
            return sort.IsSorted(sort.IntSlice(pre)) || sort.
                IsSorted(sort.IntSlice(next))
        }
    }
    return true
}

func deepcopy(nums []int) []int {
    res := make([]int, len(nums))
    copy(res,nums)
    return res
}

func checkPossibility(nums []int) bool {
    count := 0
    for i := 0; i < len(nums)-1; i++ {
        if nums[i] > nums[i+1] {
            if count == 1 {
                return false
            }
            count++
        }
    }
}

```

```
    } else if i == 0 {
        // 4 2 3 => 2 2 3
        nums[i] = nums[i+1]
        count++
    } else if nums[i-1] > nums[i+1] {
        // 3 4 2 => 3 4 4
        nums[i+1] = nums[i]
        count++
    } else {
        // 1 4 2 => 1 2 2
        nums[i] = nums[i+1]
        count++
    }
}
return true
}
```

669.修剪二叉搜索树(2)

- 题目

给定一个二叉搜索树，同时给定最小边界 L 和最大边界 R 。
 通过修剪二叉搜索树，使得所有节点的值在 $[L, R]$ 中 ($R \geq L$)。
 你可能需要改变树的根节点，所以结果应当返回修剪好的二叉搜索树的新的根节点。

示例 1：

输入：

```

1
/
0   2

```

```

L = 1
R = 2

```

输出：

```

1
 \
 2

```

示例 2：

输入：

```

3
/
0   4
 \
  \
  2
 /
1

```

```

L = 1
R = 3

```

输出：

```

3
/
2
/
1

```



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	$O(n)$	$O(\log(n))$
02	迭代	$O(n)$	$O(n)$

```

func trimBST(root *TreeNode, L int, R int) *TreeNode {
    if root == nil {
        return nil
    }
    if root.Val < L {
        return trimBST(root.Right, L, R)
    }
    if R < root.Val {
        return trimBST(root.Left, L, R)
    }
    root.Left = trimBST(root.Left, L, R)
    root.Right = trimBST(root.Right, L, R)
    return root
}

#
func trimBST(root *TreeNode, L int, R int) *TreeNode {
    if root == nil {
        return nil
    }
    // 找到根节点
    for root.Val < L || root.Val > R {
        if root.Val < L {
            root = root.Right
        } else {
            root = root.Left
        }
    }

    stack := make([]*TreeNode, 0)
    stack = append(stack, root)
    cur := root
    temp := root
    for len(stack) > 0 {
        cur = stack[len(stack)-1]
        stack = stack[:len(stack)-1]
        if cur.Left != nil {
            if cur.Left.Val >= L {
                // 左节点>=L, 继续向左
                stack = append(stack, cur.Left)
            } else {
                // 在当前左节点, 向它的右节点找到满足<L的值,
                // 并把当前左指针指向找到的值
                // 如示例2里面的, 3的Left指向找到的2
                // 然后入栈继续在2找
                temp = cur.Left
                for temp != nil && temp.Val < L {
                    temp = temp.Right
                }
            }
        }
    }
}

```

```
        }
        cur.Left = temp
        if temp != nil {
            stack = append(stack, temp)
        }
    }
    if cur.Right != nil {
        if cur.Right.Val <= R {
            stack = append(stack, cur.Right)
        } else {
            temp = cur.Right
            for temp != nil && temp.Val > R {
                temp = temp.Left
            }
            cur.Right = temp
            if temp != nil {
                stack = append(stack, temp)
            }
        }
    }
    return root
}
```

671.二叉树中第二小的节点(3)

- 题目

给定一个非空特殊的二叉树，每个节点都是正数，并且每个节点的子节点数量只有一或两个。如果一个节点有两个子节点的话，那么这个节点的值不大于它的子节点的值。

给出这样的一个二叉树，你需要输出所有节点中的第二小的值。如果第二小的值不存在，输出 -1。

示例 1：

输入：

```

    2
   / \
  2   5
     / \
    5   7
  
```

输出：5

说明：最小的值是 2，第二小的值是 5。

示例 2：

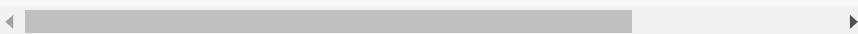
输入：

```

    2
   / \
  2   2
  
```

输出：-1

说明：最小的值是 2，但是不存在第二小的值。



- 解题思路

No.	思路	时间复杂度	空间复杂度
01	递归+数组辅助	$O(n)$	$O(n)$
02	递归	$O(n)$	$O(\log(n))$
03	迭代	$O(n)$	$O(n)$

```

var arr []int

func findSecondMinimumValue(root *TreeNode) int {
    arr = make([]int, 0)
    dfs(root)
    min, second := math.MaxInt32, math.MaxInt32
    flag := 0
    for i := 0; i < len(arr); i++ {
        if arr[i] < min {
            second = min
            min = arr[i]
        } else if min < arr[i] && arr[i] <= second {
            flag = 1
            second = arr[i]
        }
    }
    if second == math.MaxInt32 && flag == 0 {
        return -1
    }
    return second
}

func dfs(root *TreeNode) {
    if root == nil {
        return
    }
    arr = append(arr, root.Val)
    dfs(root.Left)
    dfs(root.Right)
}

#
func dfs(root *TreeNode, val int) int {
    if root == nil {
        return -1
    }
    if root.Val > val {
        return root.Val
    }
    left := dfs(root.Left, val)
    right := dfs(root.Right, val)
    if left == -1 {
        return right
    }
    if right == -1 {
        return left
    }
    return min(left, right)
}

```

```

}

func min(a, b int) int {
    if a > b {
        return b
    }
    return a
}

# 
func findSecondMinimumValue(root *TreeNode) int {
    min, second := root.Val, math.MaxInt32
    queue := make([]*TreeNode, 0)
    queue = append(queue, root)
    flag := 0
    for len(queue) > 0 {
        node := queue[len(queue)-1]
        queue = queue[:len(queue)-1]
        if node.Val < min {
            second = min
            min = node.Val
        } else if min < node.Val && node.Val <= second {
            flag = 1
            second = node.Val
        }
        if node.Left != nil {
            // 有0个或2节点
            queue = append(queue, node.Left)
            queue = append(queue, node.Right)
        }
    }
    if second == math.MaxInt32 && flag == 0 {
        return -1
    }
    return second
}

```

674.最长连续递增序列(3)

- 题目

给定一个未经排序的整数数组，找到最长且连续的递增序列。

示例 1: 输入: [1, 3, 5, 4, 7] 输出: 3

解释: 最长连续递增序列是 [1, 3, 5]，长度为3。

尽管 [1, 3, 5, 7] 也是升序的子序列，但它不是连续的，因为5和7在原数组里:

示例 2: 输入: [2, 2, 2, 2, 2] 输出: 1

解释: 最长连续递增序列是 [2]，长度为1。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	双指针	$O(n)$	$O(1)$
02	动态规划	$O(n)$	$O(n)$
03	遍历	$O(n)$	$O(1)$

```

func findLengthOfLCIS(nums []int) int {
    if len(nums) == 0 {
        return 0
    }
    res := 1
    i, j := 0, 1
    for j < len(nums) {
        for j < len(nums) && nums[j-1] < nums[j] {
            j++
        }
        if res < j-i {
            res = j - i
        }
        i = j
        j++
    }
    return res
}

#
// 状态转移方程
// 若nums[i-1]<nums[i], 则dp[i]=dp[i-1]+1; 否则dp[i]=1
func findLengthOfLCIS(nums []int) int {
    if len(nums) == 0 {
        return 0
    }
    res := 1
    dp := make([]int, len(nums))
    for i := 0; i < len(nums); i++{
        dp[i] = 1
    }
    for i := 1; i < len(nums); i++{
        if nums[i-1] < nums[i]{
            dp[i] = dp[i-1]+1
        }
        if dp[i] > res{
            res = dp[i]
        }
    }
    return res
}

```

680. 验证回文字符串 II (2)

- 题目

给定一个非空字符串 s，最多删除一个字符。判断是否能成为回文字符串。

示例 1: 输入: "aba" 输出: True

示例 2: 输入: "abca"输出: True 解释: 你可以删除c字符。

注意:

字符串只包含从 a-z 的小写字母。字符串的最大长度是50000。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	双指针	$O(n)$	$O(1)$
02	递归	$O(n)$	$O(n)$

```

func validPalindrome(s string) bool {
    i := 0
    j := len(s) - 1
    for i < j {
        if s[i] != s[j] {
            return isPalindrome(s, i, j-1) || isPalindrome(
                i,
                j)
        }
        i++
        j--
    }
    return true
}

func isPalindrome(s string, i, j int) bool {
    for i < j {
        if s[i] != s[j] {
            return false
        }
        i++
        j--
    }
    return true
}

func validPalindrome(s string) bool {
    length := len(s)
    if length < 2 {
        return true
    }
    if s[0] == s[length-1] {
        return validPalindrome(s[1 : length-1])
    }
    return isPalindrome(s[0:length-1]) || isPalindrome(s[1:
}
}

func isPalindrome(s string) bool {
    i := 0
    j := len(s) - 1
    for i < j {
        if s[i] != s[j] {
            return false
        }
        i++
        j--
    }
    return true
}

```

682. 棒球比赛(1)

- 题目

你现在是棒球比赛记录员。

给定一个字符串列表，每个字符串可以是以下四种类型之一：

1. 整数（一轮的得分）：直接表示您在本轮中获得的积分数。
2. "+"（一轮的得分）：表示本轮获得的得分是前两轮有效 回合得分的总和。
3. "D"（一轮的得分）：表示本轮获得的得分是前一轮有效 回合得分的两倍。
4. "C"（一个操作，这不是一个回合的分数）：表示您获得的最后一个有效 回

每一轮的操作都是永久性的，可能会对前一轮和后一轮产生影响。

你需要返回你在所有回合中得分的总和。

示例 1: 输入: ["5", "2", "C", "D", "+"] 输出: 30

解释:

第1轮：你可以得到5分。总和是：5。

第2轮：你可以得到2分。总和是：7。

操作1：第2轮的数据无效。总和是：5。

第3轮：你可以得到10分（第2轮的数据已被删除）。总数是：15。

第4轮：你可以得到5 + 10 = 15分。总数是：30。

示例 2: 输入: ["5", "-2", "4", "C", "D", "9", "+", "+"] 输出: 27

解释:

第1轮：你可以得到5分。总和是：5。

第2轮：你可以得到-2分。总数是：3。

第3轮：你可以得到4分。总和是：7。

操作1：第3轮的数据无效。总数是：3。

第4轮：你可以得到-4分（第三轮的数据已被删除）。总和是：-1。

第5轮：你可以得到9分。总数是：8。

第6轮：你可以得到-4 + 9 = 5分。总数是13。

第7轮：你可以得到9 + 5 = 14分。总数是27。

注意:

输入列表的大小将介于1和1000之间。

列表中的每个整数都将介于 -30000 和 30000 之间。

- 解题思路

No.	思路	时间复杂度	空间复杂度
01	模拟-栈辅助	O(n)	O(n)

```

func calPoints(ops []string) int {
    stacks := make([]int, 0)
    for i := range ops {
        switch ops[i] {
        case "+":
            r1 := stacks[len(stacks)-1]
            r2 := stacks[len(stacks)-2]
            stacks = append(stacks, r1+r2)
        case "D":
            r1 := stacks[len(stacks)-1]
            stacks = append(stacks, 2*r1)
        case "C":
            stacks = stacks[:len(stacks)-1]
        default:
            tempInt, _ := strconv.Atoi(ops[i])
            stacks = append(stacks, tempInt)
        }
    }
    res := 0
    for _, value := range stacks {
        res = res + value
    }
    return res
}

```

686. 重复叠加字符串匹配

一 题目

给定两个字符串 A 和 B，寻找重复叠加字符串A的最小次数，使得字符串B成为叠加后的字符串A的子串，如果不存在则返回 -1。

举个例子，A = "abcd"，B = "cdabcdab"。

答案为 3，因为 A 重复叠加三遍后为 "abcdabcdabcd"，此时 B 是其子串；A 重复叠加两遍后为"abcdabcd"，B 并不是其子串。

注意：

A 与 B 字符串的长度在1和10000区间范围内。

一 解题思路

No.	思路	时间复杂度	空间复杂度
01	模拟-栈辅助	O(n)	O(n)

0001-0100



0001-0100

0701-0800-Easy

mysql

- mysql
 - 175.组合两个表(2)
 - 176.第二高的薪水(3)
 - 181.超过经理收入的员工(3)
 - 182.查找重复的电子邮箱(2)
 - 183.从不订购的客户(3)
 - 193.删除重复的电子邮箱(2)
 - 197.上升的温度(4)
 - 595.大的国(2)
 - 596.超过5名学生的课(2)
 - 620.有趣的电影(2)
 - 627.交换工资(3)

175.组合两个表(2)

- 题目

SQL架构

```
Create table Person (PersonId int, FirstName varchar(255),
Create table Address (AddressId int, PersonId int, City var
Truncate table Person
insert into Person (PersonId, LastName, FirstName) values (
Truncate table Address
insert into Address (AddressId, PersonId, City, State) val
```

表1: Person

列名	类型
PersonId	int
FirstName	varchar
LastName	varchar

PersonId 是上表主键

表2: Address

列名	类型
AddressId	int
PersonId	int
City	varchar
State	varchar

AddressId 是上表主键

编写一个 SQL 查询，满足条件：无论 person 是否有地址信息，都需要基于 FirstName, LastName, City, State

- 解题思路

No.	思路
01	考察join的基本使用
02	考察join的基本使用

```

select FirstName, LastName, City, State
from Person left join Address on Person.PersonId = Address.

#
select A.FirstName, A.LastName, B.City, B.State
from Person A
left join (select distinct PersonId, City, State from Address)
on A.PersonId=B.PersonId;

```



176. 第二高的薪水(3)

- 题目

SQL架构

```

Create table If Not Exists Employee (Id int, Salary int)
Truncate table Employee
insert into Employee (Id, Salary) values ('1', '100')
insert into Employee (Id, Salary) values ('2', '200')
insert into Employee (Id, Salary) values ('3', '300')

```

编写一个 SQL 查询，获取 Employee 表中第二高的薪水 (Salary)。

Id	Salary
1	100
2	200
3	300

例如上述 Employee 表，SQL查询应该返回 200 作为第二高的薪水。

如果不存在第二高的薪水，那么查询应返回 null。

SecondHighestSalary
200

- 解题思路

No.	思路
01	把select语句包起来，使空的时候为null
02	使用ifnull
03	先查出最大的，然后查出比最大小的

```

select(
    select distinct Salary
    from Employee
    order by Salary desc
    limit 1 offset 1
) as SecondHighestSalary;

#
select ifnull(
    (select distinct Salary
    from Employee
    order by Salary desc
    limit 1 offset 1), null
) as SecondHighestSalary;

#
select max(Salary) as SecondHighestSalary
from Employee
where Salary < (select max(Salary) from Employee);

```

181.超过经理收入的员工(3)

- 题目

SQL架构

```
Create table If Not Exists Employee (Id int, Name varchar(2
Truncate table Employee
insert into Employee (Id, Name, Salary, ManagerId) values (
insert into Employee (Id, Name, Salary, ManagerId) values (
insert into Employee (Id, Name, Salary, ManagerId) values (
insert into Employee (Id, Name, Salary, ManagerId) values (
```

Employee 表包含所有员工，他们的经理也属于员工。

每个员工都有一个 Id，此外还有一列对应员工的经理的 Id。

Id	Name	Salary	ManagerId
1	Joe	70000	3
2	Henry	80000	4
3	Sam	60000	NULL
4	Max	90000	NULL

给定 Employee 表，编写一个 SQL 查询，该查询可以获取收入超过他们经理的员工。在上面的表格中，Joe 是唯一一个收入超过他的经理的员工。

Employee
Joe

- 解题思路

No.	思路
01	使用笛卡尔乘积，和方法2一样
02	使用内链接
03	子查询

```

SELECT a.Name AS 'Employee'
FROM Employee AS a, Employee AS b
WHERE a.ManagerId = b.Id AND a.Salary > b.Salary;

#
SELECT a.Name AS 'Employee'
FROM Employee AS a join Employee AS b
on a.ManagerId = b.Id AND a.Salary > b.Salary;

#
select name as Employee
from employee a
where salary > (select salary from employee where a.manager

```

182. 查找重复的电子邮箱(2)

- 题目

SQL架构

```

Create table If Not Exists Person (Id int, Email varchar(255))
Truncate table Person
insert into Person (Id, Email) values ('1', 'a@b.com')
insert into Person (Id, Email) values ('2', 'c@d.com')
insert into Person (Id, Email) values ('3', 'a@b.com')

```

编写一个 SQL 查询，查找 Person 表中所有重复的电子邮箱。

示例：

Id	Email
1	a@b.com
2	c@d.com
3	a@b.com

根据以上输入，你的查询应返回以下结果：

Email
a@b.com

说明：所有电子邮箱都是小写字母。

- 解题思路

No.	思路
01	使用临时表
02	使用having子句

```
select Email from
(
    select Email, count(Email) as num
    from Person
    Group by Email
) as temp_table
where num > 1;

// 
select Email
from Person
group by Email
having count(Email) > 1;
```

183.从不订购的客户(3)

- 题目

SQL架构

```
Create table If Not Exists Customers (Id int, Name varchar(40))
Create table If Not Exists Orders (Id int, CustomerId int)
Truncate table Customers
insert into Customers (Id, Name) values ('1', 'Joe')
insert into Customers (Id, Name) values ('2', 'Henry')
insert into Customers (Id, Name) values ('3', 'Sam')
insert into Customers (Id, Name) values ('4', 'Max')
Truncate table Orders
insert into Orders (Id, CustomerId) values ('1', '3')
insert into Orders (Id, CustomerId) values ('2', '1')
```

某网站包含两个表，Customers 表和 Orders 表。编写一个 SQL 查询，找出所有购买了商品的客户。

Customers 表：

Id	Name
1	Joe
2	Henry
3	Sam
4	Max

Orders 表：

Id	CustomerId
1	3
2	1

例如给定上述表格，你的查询应返回：

Customers
Henry
Max

- 解题思路

No.	思路
01	使用not in
02	左连接
03	使用not exists

```

select Customers.Name as Customers
from Customers
where Customers.Id not in (
    select CustomerId from Orders
);

#
select a.Name as Customers
from Customers as a left join Orders as b on a.Id=b.CustomerId
where b.CustomerId is null;

#
select name Customers
from customers c
where not exists (
    select 1 from orders o
    where o.customerid=c.id
)

```

193.删除重复的电子邮箱(2)

- 题目

编写一个 SQL 查询，来删除 Person 表中所有重复的电子邮箱，重复的邮箱是

Id	Email
1	john@example.com
2	bob@example.com
3	john@example.com

Id 是这个表的主键。

例如，在运行你的查询语句之后，上面的 Person 表应返回以下几行：

Id	Email
1	john@example.com
2	bob@example.com

提示：

执行 SQL 之后，输出是整个 Person 表。

使用 delete 语句。

- 解题思路

No.	思路
01	使用delete+自连接
02	使用delete，根据group by和min()查询出最小的id

```

#
delete p1 from Person p1, Person P2
where p1.Email = p2.Email and p1.Id > p2.Id

#
delete from Person
where id not in (
    select id from (
        select min(id) as id
        from Person
        group by Email
    ) as temp_table
)

```

197.上升的温度(4)

- 题目

SQL架构

```
Create table If Not Exists Weather (Id int, RecordDate date)
Truncate table Weather
insert into Weather (Id, RecordDate, Temperature) values ('1', '2015-01-01', 10)
insert into Weather (Id, RecordDate, Temperature) values ('2', '2015-01-02', 25)
insert into Weather (Id, RecordDate, Temperature) values ('3', '2015-01-03', 20)
insert into Weather (Id, RecordDate, Temperature) values ('4', '2015-01-04', 30)
```

给定一个 Weather 表，编写一个 SQL 查询，来查找与之前（昨天的）日期相

Id(INT)	RecordDate(DATE)	Temperature(INT)
1	2015-01-01	10
2	2015-01-02	25
3	2015-01-03	20
4	2015-01-04	30

例如，根据上述给定的 Weather 表格，返回如下 Id:

Id
2
4

- 解题思路

No.	思路
01	自连接和datediff()的使用
02	自连接和adddate()的使用
03	自连接和unix_timestamp()的使用
04	自连接和subdate()的使用

```
#  
select A.Id as "Id"  
from Weather A join Weather B  
on datediff(A.RecordDate, B.RecordDate) = 1 and A.Temperature < B.Temperature  
  
#  
select A.Id as "Id"  
from Weather A join Weather B  
on A.Temperature > B.Temperature and A.RecordDate = adddate(B.RecordDate, interval 1 day)  
  
#  
select A.Id as "Id"  
from Weather A join Weather B  
on unix_timestamp(A.RecordDate) = unix_timestamp(B.RecordDate) + 1 and A.Temperature > B.Temperature  
  
#  
select A.Id as "Id"  
from Weather A join Weather B  
on A.Temperature > B.Temperature and B.RecordDate = subdate(A.RecordDate, interval 1 day)
```

595.大的国(2)

- 题目

```
Create table If Not Exists World (name varchar(255), continent varchar(50), area float, population int, gdp float)
Truncate table World
insert into World (name, continent, area, population, gdp)
values ('Russia', 'Europe', 171000000, 1449600000, 147000000000.0)
insert into World (name, continent, area, population, gdp)
values ('China', 'Asia', 9597000, 1340000000, 100000000000.0)
insert into World (name, continent, area, population, gdp)
values ('USA', 'North America', 9629000, 317000000, 137000000000.0)
insert into World (name, continent, area, population, gdp)
values ('India', 'Asia', 2970000, 1210000000, 100000000000.0)
insert into World (name, continent, area, population, gdp)
```

这里有张 World 表

name	continent	area	population
Afghanistan	Asia	652230	25500100
Albania	Europe	28748	2831741
Algeria	Africa	2381741	37100000
Andorra	Europe	468	78115
Angola	Africa	1246700	20609294

如果一个国家的面积超过300万平方公里，或者人口超过2500万，那么这个国家编写一个SQL查询，输出表中所有大国家的名称、人口和面积。

例如，根据上表，我们应该输出：

name	population	area
Afghanistan	25500100	652230
Algeria	37100000	2381741

- 解题思路

No.	思路
01	or的使用
02	union的使用

```
select name, population, area from world
where area > 3000000 or population > 25000000

#
# or具有全表扫描机制
# union具有索引列查询速度快
# Union: 对两个结果集进行并集操作，不包括重复行，同时进行默认规则的排序
# Union All: 对两个结果集进行并集操作，包括重复行，不进行排序;
select name, population, area from world where area > 3000000
union
select name, population, area from world where population >
```

596.超过5名学生的课(2)

- 题目

```

Create table If Not Exists courses (student varchar(255), c
Truncate table courses
insert into courses (student, class) values ('A', 'Math')
insert into courses (student, class) values ('B', 'English')
insert into courses (student, class) values ('C', 'Math')
insert into courses (student, class) values ('D', 'Biology')
insert into courses (student, class) values ('E', 'Math')
insert into courses (student, class) values ('F', 'Computer')
insert into courses (student, class) values ('G', 'Math')
insert into courses (student, class) values ('H', 'Math')
insert into courses (student, class) values ('I', 'Math')

```

有一个courses 表，有：student (学生) 和 class (课程)。

请列出所有超过或等于5名学生的课。

例如,表:

student	class
A	Math
B	English
C	Math
D	Biology
E	Math
F	Computer
G	Math
H	Math
I	Math

应该输出：

class
Math

Note:

学生在每个课中不应被重复计算。

- 解题思路

No.	思路
01	group by + having
02	group by + 临时表

```

select class from courses
group by class
having count(distinct student) >= 5

#
select class from
(select class, count(distinct student) as num from courses
where num >= 5

```

620.有趣的电影(2)

- 题目

```

Create table If Not Exists cinema (id int, movie varchar(255)
Truncate table cinema
insert into cinema (id, movie, description, rating) values

```

某城市开了一家新的电影院，吸引了很多人过来看电影。

该电影院特别注意用户体验，专门有个 LED显示板做电影推荐，上面公布着影评。作为该电影院的信息部主管，您需要编写一个 SQL查询，

找出所有影片描述为非 boring (不无聊) 的并且 id 为奇数 的影片，结果请例如，下表 cinema:

id	movie	description	rating
1	War	great 3D	8.9
2	Science	fiction	8.5
3	irish	boring	6.2
4	Ice song	Fantacy	8.6
5	House card	Interesting	9.1

对于上面的例子，则正确的输出是为：

id	movie	description	rating
5	House card	Interesting	9.1
1	War	great 3D	8.9

- 解题思路

No.	思路
01	mod的使用
02	常用操作

```

select * from cinema
where mod(id,2)=1 and description != 'boring'
order by rating desc

#
select * from cinema
where id%2=1 and description != 'boring'
order by rating desc

```

627. 交换工资(3)

- 题目

```

create table if not exists salary(id int, name varchar(100))
Truncate table salary
insert into salary (id, name, sex, salary) values ('1', 'A')
insert into salary (id, name, sex, salary) values ('2', 'B')
insert into salary (id, name, sex, salary) values ('3', 'C')
insert into salary (id, name, sex, salary) values ('4', 'D')
给定一个 salary 表，如下所示，有 m = 男性 和 f = 女性 的值。

```

交换所有的 f 和 m 值（例如，将所有 f 值更改为 m，反之亦然）。

要求只使用一个更新（Update）语句，并且没有中间的临时表。

注意，您必只能写一个 Update 语句，请不要编写任何 Select 语句。

例如：

id name sex salary
----- ----- ----- -----
1 A m 2500
2 B f 1500
3 C m 5500
4 D f 500

运行你所编写的更新语句之后，将会得到以下表：

id name sex salary
----- ----- ----- -----
1 A f 2500
2 B m 1500
3 C f 5500
4 D m 500

- 解题思路

No.	思路
01	update+case
02	update+if
03	update+ascii互转

```
update salary
set sex=
CASE sex
when 'm' then 'f'
else 'm'
END

#
update salary
set sex=if(sex='f', 'm', 'f')

#
update salary
set sex=char(ascii('m')+ascii('f')-ascii(sex))
```

Bash

- Bash
 - 193.有效电话号码(4)
 - 195. 第十行(4)

193.有效电话号码(4)

- 题目

给定一个包含电话号码列表（一行一个电话号码）的文本文件 `file.txt`，写一个 `bash` 脚本输出所有有效的电话号码。

你可以假设一个有效的电话号码必须满足以下两种格式：

`(xxx) xxx-xxxx` 或 `xxx-xxx-xxxx`。 (x 表示一个数字)

你也可以假设每行前后没有多余的空格字符。

示例：

假设 `file.txt` 内容如下：

987-123-4567

123 456 7890

(123) 456-7890

你的脚本应当输出下列有效的电话号码：

987-123-4567

(123) 456-7890

- 解题思路分析

No.	思路
01	正则表达式_cat_grep
02	正则表达式_grep
03	正则表达式_awk
04	正则表达式_grep

```

# 正则表达式_cat_grep
# (xxx) xxx-xxxx 或 xxx-xxx-xxxx
# ^([0-9]{3}\) [0-9]{3}-[0-9]{4}$ (xxx) xxx-xxxx
# ^[0-9]{3}-[0-9]{3}-[0-9]{4}$ xxx-xxx-xxxx
# grep -P 匹配正则
cat file.txt | grep -P "^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$|^[
# (xxx) xxx-xxxx 或 xxx-xxx-xxxx
grep -P "^\([0-9]{3}-|\([0-9]{3}\)\) [0-9]{3}-[0-9]{4}$" file.txt
grep -E "^\([0-9]{3}-|\([0-9]{3}\)\) [0-9]{3}-[0-9]{4}$" file.txt

#
awk "/^\([0-9]{3}-|\([0-9]{3}\)\) [0-9]{3}-[0-9]{4}$/" file.txt

#
grep -P "\(\d{3}-|\(\d{3}\)\) \d{3}-\d{4}$" file.txt

```

195. 第十行(4)

- 题目

给定一个文本文件 `file.txt`, 请只打印这个文件中的第十行。

示例:

假设 `file.txt` 有如下内容:

```

Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10

```

你的脚本应当显示第十行:

`Line 10`

说明:

1. 如果文件少于十行, 你应当输出什么?
2. 至少有三种不同的解法, 请尝试尽可能多的方法来解题。

- 解题思路分析

No.	思路
01	awk NR
02	tail head
03	sed

```
#  
awk 'NR==10' file.txt  
  
#  
tail -n +10 file.txt | head -1  
  
#  
sed -n 10p file.txt
```

初级算法(49)

- 49题
- <https://leetcode-cn.com/explore/interview/card/top-interview-questions-easy/>

1.数组(11)

Title	Tag	难度	完成情况
26.删除排序数组中的重复项	数组、双指针	Easy	完成
买卖股票的最佳时机 II			
189.旋转数组	数组	Easy	完成
217.存在重复元素	数组、哈希表	Easy	完成
136.只出现一次的数字	位运算、哈希表	Easy	完成
350.两个数组的交集 II	排序、哈希表、双指针、二分查找	Easy	完成
66.加一	数组	Easy	完成
283.移动零	数组、双指针	Easy	完成
1.两数之和	数组、哈希表	Easy	完成
有效的数独			
旋转图像			

2.字符串(9)

Title	Tag	难度	完成情况
344.反转字符串	双指针、字符串	Easy	完成
7.整数反转	数学	Easy	完成
387.字符串中的第一个唯一字符	哈希表、字符串	Easy	完成
有效的字母异位词			
验证回文字符串 125.验证回文串	双指针、字符串	Easy	完成
字符串转换整数 (atoi)			
实现 strStr()			
外观数列			
最长公共前缀			

3.链表(6)

Title	Tag	难度	完成情况
删除链表中的节点			
删除链表的倒数第N个节点			
反转链表			
合并两个有序链表			
回文链表			
环形链表			

4.树(5)

Title	Tag	难度	完成情况
104.二叉树的最大深度	树、深度优先搜索	Easy	完成
验证二叉搜索树			
对称二叉树			
二叉树的层序遍历			
将有序数组转换为二叉搜索树			

5.排序和搜索(2)

Title	Tag	难度	完成情况
合并两个有序数组			
第一个错误的版本			

6.动态规划(4)

Title	Tag	难度	完成情况
70.爬楼梯	动态规划	Easy	完成
买卖股票的最佳时机			
53.最大子序和	数组、分治算法、动态规划	Easy	完成
198.打家劫舍	动态规划	Easy	完成

7.设计问题(2)

Title	Tag	难度	完成情况
Shuffle an Array			
最小栈			

8.数学(4)

Title	Tag	难度	完成情况
412.Fizz Buzz		Easy	完成
计数质数			
326.3的幂	数学	Easy	完成
罗马数字转整数			

9.其他(6)

Title	Tag	难度	完成情况
191.位1的个数	位运算	Easy	完成
461.汉明距离	位运算	Easy	完成
190.颠倒二进制位	位运算	Easy	完成
帕斯卡三角形			
有效的括号			
268.缺失数字	位运算、数组、数学	Easy	完成

剑指offer

参考资料

- leetcode专栏 <https://leetcode-cn.com/problemset/lcof/>
-

面试题

No.	Title	Tag	难度	完成情况
第二章：面试需要的基本知识				
面试题1	赋值运算符函数			
面试题2	实现Singleton模式			
面试题3	数组中重复的数字			
面试题4	二维数组中的查找			
面试题5	替换空格			
面试题6	从尾到头打印链表			
面试题7	重建二叉树			
面试题8	二叉树的下一个节点			
面试题9	用两个栈实现队列			
面试题10	斐波那契数列			
面试题11	旋转数组的最小数字			
面试题12	矩阵中的路径			
面试题13	机器人的运动范围			
面试题14	剪绳子			
面试题15	二进制中1的个数			
第三章：高质量的代码				
面试题16	数值的整数次方			
面试题17				

No.	Title	Tag	难度	完成情况
面试题18				
面试题19				
面试题20				
面试题21				
面试题22				
面试题23				
面试题24				
面试题25				
面试题26				
第四章：解决面试题的思路				