Table of Contents

go-Leetcode

gitbook

0.参考

- https://leetcode-cn.com/

1.完成的题目

Title	Tag	难度	完成情况
第一期			
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
第二期			
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
第三期			
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
第四期			
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
第五期			
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
第六期			
193. 有效电话号码	Bash	Easy	完成
195. 第十行	Bash	Easy	完成

0001-0100

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
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

示例 3:输入: 10      输出: false
解释: 从右向左读, 为 01 。因此它不是一个回文数。

进阶:
你能不将整数转为字符串来解决这个问题吗?
```

- 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))

```go
// 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 不
    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)

```go
// 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(nlog(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)

```go
// 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
}
```

```go
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
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)

```go
// 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

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)

```go
// 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
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,
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
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
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
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
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)

```go
// 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
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)

```go
// 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 ≤ n ≤ 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)

```go
// 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

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

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(nlog(n))	O(log(n))

```go
// 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)

```go
// 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.Trim(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)

```go
// 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)

```go
// 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...,
     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)

```go
// 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)

```go
// 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)

```go
// 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(nlog(n))	O(1)
02(最优)	双指针法	O(n)	O(1)
03	拷贝后插入	O(n)	O(n)

```go
// 合并后排序
func merge(n ums1 []int, m int, nums2 []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))

```go
// 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

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)

```go
// 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 && left
            leftQ = append(leftQ, leftCur.Left, leftCur.Rig
            rightQ = append(rightQ, rightCur.Right, rightCu
        } 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

```go
// 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],
    3
   / \
  9  20
    /  \
   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)

```go
// 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.Va
        } 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

      0
     / \
   -3   9
   /   /
 -10  5
```

- 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)

```go
// 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))

```go
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],

```
    3
   / \
  9  20
    /  \
   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)

```go
// 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.R
    }
}

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，
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，
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \      \
        7    2      1
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.Val
            }
        }
    }
    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)

```go
// 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 ≤ 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

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）
      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{前一天最大利润, 今天的价格 - 之前最低价格}	O(n)	O(1)
03	动态规划(从后到前)	O(n)	O(1)

```go
// 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（
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
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)

```go
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
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)

```go
// 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
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)

```go
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], skip
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 ski
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
```

```
        } 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 in

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)

```go
// 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))

```go
// 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
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(nlog(n))	O(log(n))

```go
// 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)

```go
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)

```go
// 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：00000010100101000001111010011100
输出：00111001011110000010100101000000
解释：输入的二进制串 00000010100101000001111010011100 表示无符号
      因此返回 964176192，其二进制表示形式为 00111001011110000010

示例 2：
输入：11111111111111111111111111111101
输出：10111111111111111111111111111111
解释：输入的二进制串 11111111111111111111111111111101 表示无符号
      因此返回 3221225471 其二进制表示形式为 10101111110010110010

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
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)

```go
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
}

//
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 & 0xaaaaaaaa) >> 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：00000000000000000000000010000000
输出：1
解释：输入的二进制串 00000000000000000000000010000000 中，共有一

示例 3：
输入：11111111111111111111111111111101
输出：31
解释：输入的二进制串 11111111111111111111111111111101 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
}
```

1

0201-0300

name	价格	数量	
香蕉	$1	5	
苹果	$1	6	
草莓	$1	7	

mysql

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) valu

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 Addre
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
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(25
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(
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

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.Custome
where b.CustomerId is null;

#
select name Customers
from customers c
where not exists (
    select 1 from orders o
    where o.customerid=c.id
)
```

Bash

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
grep -E "^([0-9]{3}-|\([0-9]{3}\) )[0-9]{3}-[0-9]{4}$" file

#
awk "/^([0-9]{3}-|\([0-9]{3}\) )[0-9]{3}-[0-9]{4}$/" file.t

#
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	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