0001-0100

1. 两数之和

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的元素。

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

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)

```go
// 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
```

```go
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

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）读都是一样的整数。

示例 1: 输入: 121 输出: true

示例 2:输入: -121 输出: false
解释: 从左向右读, 为 -121 。 从右向左读, 为 121- 。因此它不是一个回文数。

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

进阶:
你能不将整数转为字符串来解决这个问题吗?
```

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

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 + II 。 27 写做  XXVII, 即为 XX + V + II 。
通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况:
    I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
    X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
    C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。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 = 3.

示例 5:
输入: "MCMXCIV" 输出: 1994
解释: M = 1000, CM = 900, XC = 90, IV = 4.
```

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

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最长前缀，继续比较，直到最后
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++ {
```

```go
    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

题目

```
给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串，判断字符串是否有效。
有效字符串需满足：
    左括号必须用相同类型的右括号闭合。
    左括号必须以正确的顺序闭合。
注意空字符串可被认为是有效字符串。

示例 1:
输入: "()"
输出: true

示例 2:
输入: "()[]{}"
输出: true

示例 3:
输入: "(]"
输出: false

示例 4:
输入: "([)]"
输出: false

示例 5:
输入: "{[]}"
输出: true
```

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

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

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，返回移除后数组的新长度。
不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

示例 1:
给定数组 nums = [1,1,2],
函数应该返回新的长度 2, 并且原数组 nums 的前两个元素被修改为 1, 2。
你不需要考虑数组中超出新长度后面的元素。

```

```
示例 2:
给定 nums = [0,0,1,1,1,2,2,3,3,4],
函数应该返回新的长度 5, 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4。
你不需要考虑数组中超出新长度后面的元素。

说明:
为什么返回数值是整数，但输出的答案是数组呢?
请注意，输入数组是以“引用”方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。
你可以想象内部操作如下:
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

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

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)

```go
// 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

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，返回移除后数组的新长度。
不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。
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。
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)

```
// 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()

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)

```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

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)

```go
// 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

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。其前五项如下：
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

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），返回其最大和。

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

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
```

```go
}

// 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]
    }
```

```
65
66      if dp > result {
67        result = dp
68      }
69    }
70    return result
71  }
72
73  // 分治法
74  func maxSubArray(nums []int) int {
75    result := maxSubArr(nums, 0, len(nums)-1)
76    return result
77  }
78
79  func maxSubArr(nums []int, left, right int) int {
80    if left == right {
81      return nums[left]
82    }
83
84    mid := (left + right) / 2
85    leftSum := maxSubArr(nums, left, mid)         // 最大子序在左边
86    rightSum := maxSubArr(nums, mid+1, right)     // 最大子序在右边
87    midSum := findMaxArr(nums, left, mid, right) // 跨中心
88    result := max(leftSum, rightSum)
89    result = max(result, midSum)
90    return result
91  }
92
93  func findMaxArr(nums []int, left, mid, right int) int {
94    leftSum := math.MinInt32
95    sum := 0
96    // 从右到左
97    for i := mid; i >= left; i-- {
98      sum += nums[i]
99      leftSum = max(leftSum, sum)
100   }
101
102   rightSum := math.MinInt32
103   sum = 0
104   // 从左到右
105   for i := mid + 1; i <= right; i++ {
106     sum += nums[i]
107     rightSum = max(rightSum, sum)
108   }
109   return leftSum + rightSum
110 }
111
112 func max(a, b int) int {
113   if a > b {
```

```
    return a
  }
  return b
}
```

58.最后一个单词的长度

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

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 4321。
```

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

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
```

```go
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的平方跟

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

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

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
```

```go
}

// 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

题目

```
给定两个有序整数数组 nums1 和 nums2，将 nums2 合并到 nums1 中，使得 num1 成为一个有序数组。
说明：
    初始化 nums1 和 nums2 的元素数量分别为 m 和 n。
    你可以假设 nums1 有足够的空间（空间大小大于或等于 m + n）来保存 nums2 中的元素。

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

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--
    }
  }
}

```

```go
// 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

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
}
```