剑指 offer

链表(8 道)：

三：从尾到头打印链表
十四：链表中倒数第 k 个结点
十五：反转链表
十六：合并两个排序的链表
二十五：复杂链表的复制
三十六：两个链表的第一个公共结点
五十五：链表中环的入口结点
五十六：删除链表中重复的结点

二叉树(12 道)：

四：重建二叉树
十七：树的子结构
十八：二叉树的镜像
二十二：从上往下打印二叉树
二十四：二叉树中和为某一值的路径
三十八：二叉树的深度
三十九：平衡二叉树
五十七：二叉树的下一个结点
五十八：对称的二叉树
五十九：按之字顺序打印二叉树
六十：把二叉树打印成多行
六十一：序列化二叉树

二叉搜索树(3 道)：

二十三：二叉搜索树的后序遍历序列
二十六：二叉搜索树与双向链表
六十二：二叉搜索树的第 k 个结点

数组(11 道)：

一：二维数组中的查找
六：旋转数组的最小数字
十三：调整数组顺序使奇数位于偶数前面
二十八：数组中出现次数超过一半的数字
三十：连续子数组的最大和
三十二：把数组排成最小的数
三十五：数组中的逆序对
三十七：数字在排序数组中出现的次数
四十：数组中只出现一次的数字
五十：数组中重复的数字
五十一：构建乘积数组

字符串(8 道)：

二：替换空格
二十七：字符串的排列
三十四：第一个只出现一次的字符
四十三：左旋转字符串
四十四：翻转单词顺序序列
四十九：把字符串转换成整数
五十二：正则表达式匹配
五十三：表示数值的字符串

栈(3 道)：

五：用两个栈实现队列
二十：包含 min 函数的栈
二十一：栈的压入、弹出序列

递归(4 道)：

七：裴波那契数列
八：跳台阶
九：变态跳台阶
十：矩形覆盖

回溯法(2 道)：

六十五：矩阵中的路径
六十六：机器人的运动范围

其他(15 道)：

十一：二进制中 1 的个数
十二：数值的整数次方
十九：顺时针打印矩阵
二十九：最小的 K 个数
三十一：整数中 1 出现的次数(从 1-n)
三十三：丑数
四十一：和为 S 的连续正数序列
四十二：和为 S 的两个数字
四十五：扑克牌顺子
四十六：孩子们的游戏(圆圈中最后剩下的数)
四十七：求 1+2+3+…+n
四十八：不用加减乘除的加法
五十四：字符流中第一个不重复的字符
六十三：数据流中的中位数
六十四：滑动窗口的最大值

动态规划(5 道)：

Leetcode343：整数拆分
Offer：剪绳子
Leetcode121：买卖股票的最佳时机
Leetcode122：买卖股票的最佳时 II
Leetcode123：买卖股票的最佳时III

排序(5 道)：

冒泡排序
选择排序
快速排序
插入排序
归并排序

链表(8 道)：

三：从尾到头打印链表

```
type NodeList struct {
    Val  int
    Next *NodeList
}
func printList(node *NodeList) {
    // 逆序打印链表
    if node.Next != nil {
        printList(node.Next)
    }
    fmt.Println(node.Val)
}
```

十四：链表中倒数第 k 个结点

```
func kthNode(head *NodeList, k int) *NodeList {
    if head == nil {return head}
    tail := head
    for k > 1 && tail != nil {
        tail = tail.Next
        k--
    }
    if tail == nil {return nil}
    for tail.Next != nil {
        tail = tail.Next
        head = head.Next
    }
    return head
}
```

十五：反转链表

```
func reverse(head *NodeList) *NodeList {
    if head == nil || head.Next == nil {
        return head
    }
    vide := &NodeList{-1, nil}
    for head != nil {
        next := head.Next
        head.Next = vide.Next
        vide.Next = head
        head = next
    }
    return vide.Next
}
func print(head *NodeList) {
    for head != nil {
        fmt.Printf("%d -> ", head.Val)
        head = head.Next
    }
}
```

十六：合并两个排序的链表

```
func mergeTwoLists(l1 *NodeList, l2 *NodeList) *NodeList{
    res := &NodeList{}
    cur := res
    for l1 != nil || l2 != nil {
        if l1 == nil {
            cur.Next = l2
            break
        } else if l2 == nil {
            cur.Next = l1
            break
        }
        if l1.Val > l2.Val {
            cur.Next = l2
            cur = cur.Next
            l2 = l2.Next
        } else {
            cur.Next = l1
            cur = cur.Next
            l1 = l1.Next
        }
    }
    return res.Next
}
```

二十五：复杂链表的复制

```
type RandNodeList struct {
    Val  int
    Next *RandNodeList
    Rand *RandNodeList
}
// 用 next 自己写 A->A`->B->B`写
func RandNodeList(head *RandNodeList) *RandNodeList {
    if head == nil {return nil}
    m := make(map[*RandNodeList]*RandNodeList)
    cur := head
    for cur != nil {
        if _, ok := m[cur]; !ok {
            m[cur] = &RandNodeList{cur.Val, nil, nil}
        }
        if cur.Next != nil {
            if _, ok := m[cur.Next]; !ok {
            m[cur.Next] = &RandNodeList{cur.Next.Val, nil, nil}
            }
            m[cur].Next = m[cur.Next]
        }
        if cur.Rand != nil {
            if _, ok := m[cur.Rand]; !ok {
```

```
                m[cur.Rand] = &RandNodeList{cur.Rand.Val, nil, nil}
                }
                m[cur].Rand = m[cur.Rand]
            }
            cur = cur.Next
        }
        return m[head]
}
```

三十六：两个链表的第一个公共结点

```
type LinkNode struct {
    Val  int
    Next *LinkNode
}
func firstCommon(h1, h2 *LinkNode) *LinkNode {
    // 长链表先走，实现右对齐
    start, l1 := h1, 0
    for start != nil {
        start = start.Next
        l1++
    }
    start, l2 := h2, 0
    for start != nil {
        start = start.Next
        l2++
    }
    s1, s2 := h1, h2
    if l1 > l2 {
        diff := l1 - l2
        for s1 != nil && diff > 0 {
            s1 = s1.Next
            diff--
        }
    } else if l1 < l2 {
        diff := l2 - l1
        for s2 != nil && diff > 0 {
            s2 = s2.Next
            diff--
        }
    }
    for s1 != nil && s2 != nil && s1 != s2 {
        s1 = s1.Next
        s2 = s2.Next
    }
    return s1
}
// Hashmap
func firstCommonMap(h1, h2 *LinkNode) *LinkNode {
    m := make(map[*LinkNode]bool)
    for h1 != nil {
        m[h1] = true
        h1 = h1.Next
    }
    for h2 != nil {
        if _, ok := m[h2]; ok {
            return h2
        }
        h2 = h2.Next
    }
    return h2
}
```

五十五：链表中环的入口结点

```
func detectCycle(head *NodeList) *NodeList {
    if head == nil || head.Next == nil {
        return nil
    }
    slow := head.Next
    fast := head.Next.Next
    for fast != nil && fast.Next != nil {
        if slow == fast {
            break
        }
        slow, fast = slow.Next, fast.Next.Next
    }
    tmp := head
    // n = (q-1)m + (m-k)
    // slow 和 head 同时开始走，当 slow==head 是 entrance
    for tmp != nil && slow != nil {
        if slow == tmp {
            return slow
        }
        slow, tmp = slow.Next, tmp.Next
    }
    return nil
}
```

五十六：删除链表中重复的结点

```
func deleteDuplicates(head *NodeList) *NodeList {
    // 长度 <=1 的 list ，可以直接返回
    if head == nil || head.Next == nil {
        return head
    }
    // 要么 head 重复了，那就删除 head
    if head.Val == head.Next.Val {
        for head.Next != nil && head.Val == head.Next.Val {
            head = head.Next
        }
```

```
        // 有重复所以直接不带 head
        return deleteDuplicates(head.Next)
    }
    // 要么 head 不重复，递归处理 head 后面的节点
    head.Next = deleteDuplicates(head.Next)
    return head
}
```

二叉树(12 道)：

```
type TreeNode struct {
    Val   int
    Left  *TreeNode
    Right *TreeNode
}
```

四：重建二叉树

```
func printPreOrder(root *TreeNode) {
    if root != nil {
        fmt.Printf("%d ", root.Val)
        printPreOrder(root.Left)
        printPreOrder(root.Right)
    }
}
func printInOrder(root *TreeNode) {
    if root != nil {
        printInOrder(root.Left)
        fmt.Printf("%d ", root.Val)
        printInOrder(root.Right)
    }
}
func reConstructBinaryTree(pre []int, in []int) *TreeNode {
    if len(pre) != len(in) || len(pre) == 0 {
        return nil
    }
    // find root and root Index in inOrder
    rootVal := pre[0]
    rootIndex := 0
    for i := 0; i < len(in); i++ {
        if in[i] == rootVal {
            rootIndex = i
        }
    }
    // pre and in for left and right
    inL, inR := in[:rootIndex], in[rootIndex+1:]
    preL, preR := pre[1:rootIndex+1], pre[rootIndex+1:]
    // revursive
    left := reConstructBinaryTree(preL, inL)
    right := reConstructBinaryTree(preR, inR)
    return &TreeNode{Val: rootVal, Left: left, Right: right}
}
```

十七：树的子结构

```
func hasSubRootOrTree(p *TreeNode, c *TreeNode) bool {
    if c == nil {return true}
    if p == nil {return false}
    if p.Val == c.Val {
        if hasSub(p, c) {return true}
    }
    return hasSubRootOrTree(p.Left,c) || hasSubRootOrTree(p.Right, c)
}
func hasSub(p *TreeNode, c *TreeNode) bool {
    if c == nil {return true}
    if p == nil {return false}
    if p.Val != c.Val {return true}
    return hasSub(p.Left, c.Left) && hasSub(p.Right, c.Right)
}
```

十八：二叉树的镜像

```
func MirrorTree(p *TreeNode) {
    if p == nil {
        return
    }
    p.Left, p.Right = p.Right, p.Left
    MirrorTree(p.Left)
    MirrorTree(p.Right)
}
func Print(root *TreeNode) {
    if root == nil {
        return
    }
    fmt.Printf("%d ", root.Val)
    Print(root.Left)
    Print(root.Right)
}
```

二十二：从上往下打印二叉树

```
func getTreeByLevel(root *TreeNode) []int {
    var queue []*TreeNode
    var res []int
    queue = append(queue, root)
    for len(queue) != 0 {
        node := queue[0]
        queue = queue[1:]
        res = append(res, node.Val)
        if node.Left != nil {
            queue = append(queue, node.Left)
        }
        if node.Right != nil {
            queue = append(queue, node.Right)
        }   }
    return res
}
```

二十四：二叉树中和为某一值的路径

```
func pathSum(root *TreeNode, sum int) [][]int {
    //写个递归函数，从左到右把每条路线都试一下，有符合的就把路径加入 paths
    var res [][]int
    solution := []int{}
    var dfs func(root *TreeNode, sum int)
    dfs = func(root *TreeNode, sum int) {
        if root == nil {return}
        rest := sum - root.Val
        if rest == 0 && root.Left == nil&&root.Right ==nil{
            solution = append(solution, root.Val)
            tmp := make([]int, len(solution))
            copy(tmp, solution) //注意这里需要 copy
            res = append(res, tmp)
            solution = solution[:len(solution)-1]
            return
        }
        solution = append(solution, root.Val)
        dfs(root.Left, rest)
        dfs(root.Right, rest)
        solution = solution[:len(solution)-1]
    }
    dfs(root, sum)
    return res
}
```

三十八：二叉树的深度

```
func maxDepth(root *TreeNode) int {
    max := 0
    var dfs func(root *TreeNode, level int)
    dfs = func(root *TreeNode, level int) {
        if root == nil {return}
        if level > max {max = level}
        dfs(root.Left, level+1)
        dfs(root.Right, level+1)
    }
    dfs(root, 1)
    return max
}
```

三十九：平衡二叉树

```
func isBalanced(root *TreeNode) bool {
    // 只要遇到以下情况就返回 false
    // 1.左右子树只要有一个不平衡
    // 2.左右子树深度相差大于一
    // 注意返回当前深度时为 max(ldepth, rdepth)+1
    if root == nil {
        return true
    }
    var recur func(root *TreeNode) (int, bool)
    recur = func(root *TreeNode) (int, bool) {
        if root == nil {
            return 0, true
        }
        rightD, rightB := recur(root.Right)
        leftD, leftB := recur(root.Left)
        return max(rightD, leftD) + 1, abs(rightD-leftD) <= 1 && rightB && leftB
    }
    _, res := recur(root)
    return res
}
func abs(a int) int {
    if a < 0 {return -a}
    return a
}
func max(a, b int) int {
    if a > b {return a}
    return b
}
```

五十七：二叉树的下一个结点

```
type TreeNodeNext struct {
    Val    int
    Left   *TreeNodeNext
    Right  *TreeNodeNext
    Parent *TreeNodeNext
}

func getNext(node *TreeNodeNext) *TreeNodeNext {
    if node == nil {return node}
//如果节点有右子树，那么它的下一个节点就是它的右子树中最左边的节点
    if node.Right != nil {
        node = node.Right
        for node.Left != nil {
            node = node.Left
        }
        return node
    }
    // 先取目标的父节点
    p := node.Parent
    n := node
    for p != nil {
        // 如果 p 节点是 p 的父节点的右节点 =》 继续向上找
        if n == p.Right {
```

```
                n = p
                p = p.Parent
                continue
            }
            // p 是 p 父节点的左节点 =》 返回父节点
            return p
        }
        // 目标节点没有下一个节点
        return nil
}
func inOrder(root *TreeNodeNext) {
        if root == nil {
            return
        }
        inOrder(root.Left)
        fmt.Printf("%d -> ", root.Val)
        inOrder(root.Right)
}
```

五十八：对称的二叉树

```
func isSymmetric(root *TreeNode) bool {
        if root == nil {
            return true
        }
        var isMirror func(*TreeNode, *TreeNode) bool
        isMirror = func(t1, t2 *TreeNode) bool {
            // 判断两个树是否都存在
            if t1 == nil && t2 == nil {
                return true
            }
            if t1 == nil || t2 == nil {
                return false
            }
//t1=t2，同时 t1.Left/t2.Right，t1.Right/t2.Left 都需要对称
            return t1.Val == t2.Val && isMirror(t1.Left, t2.Right) && isMirror(t1.Right, t2.Left)
        }
        return isMirror(root.Left, root.Right)
}
```

五十九：按之字顺序打印二叉树

```
func Zprint(root *TreeNode) {
        var q1 []*TreeNode
        var q2 []*TreeNode
        q1 = append(q1, root)
        for len(q1) != 0 || len(q2) != 0 {
            if len(q2) == 0 {
                for len(q1) != 0 {
                    node := q1[len(q1)-1]
                    q1 = q1[:len(q1)-1]
                    fmt.Printf("%d ", node.Val)
                    if node.Left != nil {
                        q2 = append(q2, node.Left)
                    }
                    if node.Right != nil {
                        q2 = append(q2, node.Right)
                    }
                }
            } else {
                for len(q2) != 0 {
                    node := q2[len(q2)-1]
                    q2 = q2[:len(q2)-1]
                    fmt.Printf("%d ", node.Val)
                    if node.Right != nil {
                        q1 = append(q1, node.Right)
                    }
                    if node.Left != nil {
                        q1 = append(q1, node.Left)
                    }
                }
            }
        }
}
```

六十：把二叉树打印成一行

```
func print(root *TreeNode) {
        var q1 []*TreeNode
        q1 = append(q1, root)
        for len(q1) != 0 {
            node := q1[0]
            q1 = q1[1:]
            fmt.Printf("%d ", node.Val)
            if node.Left != nil {
                q1 = append(q1, node.Left)
            }
            if node.Right != nil {
                q1 = append(q1, node.Right)
            }
        }
}
# 多行
func levelOrder(root *TreeNode) [][]int {
        var (
            result [][]int
            queue    []*TreeNode
        )
        if root == nil {return result}
        queue = append(queue, root)
```

```
    for len(queue) > 0 {
        var currentLevel []int
        currentLenth := len(queue)
        for i := 0; i < currentLenth; i++ {
            node := queue[0]
            queue = queue[1:]
            currentLevel = append(currentLevel, node.Val)
            if node.Left != nil {
                queue = append(queue, node.Left)
            }
            if node.Right != nil {
                queue = append(queue, node.Right)
            }
        }
        result = append(result, currentLevel)
    }
    return result
}
```

六十一：序列化二叉树

```
func Serialize(root *TreeNode) string {
    str := ""
    if root == nil {
        str += "#,"
        return str
    }
    var cur func(root *TreeNode) string
    cur = func(root *TreeNode) string {
        if root == nil {
            str += "#,"
            return str
        }
        str += strconv.Itoa(root.Val) + ","
        str += Serialize(root.Left)
        str += Serialize(root.Right)
        return str
    }
    return cur(root)
}

func Deserialize(str string) *TreeNode {
    if str == "" {
        return nil
    }
    index := 0
    var recur func(str string) *TreeNode
    recur = func(str string) *TreeNode {
        if str[index] == '#' {
            index += 2
            return nil
        }
        num := 0
        for str[index] != ',' && index < len(str) {
            num = num*10 + int(str[index]-'0')
            index++
        }
        index++
        root := &TreeNode{num, nil, nil}
        root.Left = recur(str)
        root.Right = recur(str)
        return root
    }
    root := recur(str)
    return root
}
func print(root *TreeNode) {
    if root == nil {return}
    fmt.Printf("%d -> ", root.Val)
    print(root.Left)
    print(root.Right)
}
// 层次遍历
func SerializeLevel(root *TreeNode) string {
    var (
        str       string
        q1        []*TreeNode
        falseNode = &TreeNode{-1, nil, nil}//构建假节点
    )
    q1 = append(q1, root)
    for len(q1) != 0 {
        node := q1[0]
        q1 = q1[1:]
        if node.Val == -1 {str += "#" + ","
        } else {
            str += strconv.Itoa(node.Val) + ","
            if node.Left != nil {
                q1 = append(q1, node.Left)
            } else {
                q1 = append(q1, falseNode)
            }
            if node.Right != nil {
                q1 = append(q1, node.Right)
            } else {
                q1 = append(q1, falseNode)
            }
        }
    }
    return str
```

```
}

func DeserializeLevel(str string) *TreeNode {
    val := strings.Split(str, ",")
    val = val[:len(val)-1] //最后一位不要
    if len(val) == 0 {return nil}
    var (
        i       = 1
        queue []*TreeNode
    )
    build := func(str string) *TreeNode { //匿名函数
        nodeValue, _ := strconv.Atoi(str)
        buildNode := &TreeNode{nodeValue, nil, nil}
        queue = append(queue, buildNode)
        return buildNode
    }
    root := build(val[0])
    for len(queue) > 0 {
        node := queue[0]
        queue = queue[1:]
        if val[i] != "#" {
            node.Left = build(val[i])
        }
        i += 1
        if val[i] != "#" {
            node.Right = build(val[i])
        }
        i += 1
    }
    return root
}
```

二叉搜索树(3 道)：

二十三：二叉搜索树的后序遍历序列

```
func isPostOrder(post []int) bool {
    if len(post) <= 2 {return true}
    root := post[len(post)-1]
    left := -1
    for i := len(post) - 2; i >= 0; i-- {
        if post[i] < root {
            left = i
            break
        }
    }
    for _, v := range post[:left+1] {
        if v > root {
            return false
        }
    }
    return isPostOrder(post[0:left+1]) &&
isPostOrder(post[left+1:len(post)-1])
}
func isPostOrder1(post []int) bool {
    if len(post) <= 2 {return true}
    root := post[len(post)-1]
    split := 0
    // 找出分界点
    for i := 0; i <= len(post)-2; i++ {
        if post[i] > root {split = i
            break
        }
    }
    // 分界点之后的都要大于 root
    for _, v := range post[split:] {
        if v < root {return false}
    }
    return isPostOrder(post[0:split]) &&
isPostOrder(post[split:len(post)-1])
}
```

二十六：二叉搜索树与双向链表

```
func TreeToList(root *TreeNode) (*TreeNode, *TreeNode) {
    head, tail := root, root
    if root == nil {
        return head, tail
    }
    if root.Left == nil && root.Right == nil {
        head.Left, tail.Right = root, root
        return head, tail
    }
    if root.Left != nil {
        leftHead, leftTail := TreeToList(root.Left)
        head = leftHead
        root.Left = leftTail
        leftTail.Right = root
    }
    if root.Right != nil {
        rightHead, rightTail := TreeToList(root.Right)
        tail = rightTail
        root.Right = rightHead
        rightHead.Left = root
    }
    head.Left, tail.Right = head, tail
    return head, tail
}
func printRight(root *TreeNode) {
    for root.Right != root {
        fmt.Printf("%d ", root.Val)
```

```
        root = root.Right
    }
    fmt.Printf("%d ", root.Val)
}
func printLeft(root *TreeNode) {
    for root.Left != root {
        fmt.Printf("%d ", root.Val)
        root = root.Left
    }
    fmt.Printf("%d ", root.Val)
}
```

六十二：二叉搜索树的第 k 个结点

```
func getKth(root *TreeNode, k int) *TreeNode {
    if root == nil || k < 1 {
        return nil
    }
    var res *TreeNode
    found := false
    var inOrder func(*TreeNode)
    inOrder = func(node *TreeNode) {
        if found == true || node == nil {
            return
        }
        inOrder(node.Left)
        if k == 1 {
            res = node
            found = true
        }
        k--
        inOrder(node.Right)
    }
    inOrder(root)
    return res
}
```

数组(11 道)：

一：二维数组中的查找

```
func Find(board2 [][]int, target int) bool {
    rlen := len(board2)       // 行数
    clen := len(board2[0]) // 列数
    for c, r := 0, rlen-1; c < clen && r >= 0; {
        if board2[r][c] == target {
            return true
        }
        if board2[r][c] > target { r--
        } else {c++}
    }
    return false
}
```

六：旋转数组的最小数字

```
func minNumberInRotateArray(nums []int) int {
    mid := 0
    for low, high :=0, len(nums)-1; nums[low]>= nums[high];{
        if high-low == 1 {
            mid = high
            break
        }
        mid = (low + high) / 2
        if nums[mid] == nums[low]&&nums[mid] == nums[high]{
            return FindMin(nums[low :high+1])//包 high
        }
        if nums[mid] >= nums[low] {
            low = mid
        } else {
            high = mid
        }
    }
    return nums[mid]
}
func FindMin(nums []int) int {
    min := nums[0]
    for _, value := range nums {
        if value <= min {
            min = value
        }
    }
    return min
}
```

十三：调整数组顺序使奇数位于偶数前面

```
func oddFirst(s []int) {
    left, right := 0, len(s)-1
    for left < right {
        for s[right]%2 == 0 && left < right {
            right--
        }
        for s[left]%2 == 1 && left < right {
            left++
        }
        if left == right {
            break
        }
        if left < right {
            s[left], s[right] = s[right], s[left]
        }
    }
}
```

二十八：数组中出现次数超过一半的数字

```
func getMostFreq(nums []int) (int, error) {
    if len(nums) == 0 {
        return -1, errors.New("Array is empty")
    }
    count := 1
    value := nums[0]
    for i := 1; i < len(nums); i++ {
        if nums[i] == value {
            count++
        } else {
            count--
            if count == 0 {
                value = nums[i]
                count = 1
            }
        }
    }
    return value, nil
}
```

三十：连续子数组的最大和

```
func MaxSubset(nums []int) int {
    // dp[i] 代表以 i 结尾时最大连续子数组的最大和（同时）
    // 复制 nums 到 dp
    // 如果 dp[i-1]>0 那就把 dp[i]=dp[i-1]+dp[i]
    // 再利用全局变量，保存出出现过的最大值
    if len(nums) == 0 {
        return 0
    }
    dp := make([]int, len(nums))
    copy(dp, nums)
    max := dp[0]
    for i := 1; i < len(dp); i++ {
        if dp[i-1] > 0 {
            dp[i] = dp[i-1] + dp[i]
        }
        if dp[i] > max {
            max = dp[i]
        }
    }
    return max
}
```

三十二：把数组排成最小的数

```
func getMinStr(nums []int) string {
    quickSort(nums, 0, len(nums)-1)
    noZero := 0
    for nums[noZero] == 0 {
        noZero++
    }
    res := ""
    fmt.Println(nums)
    for i := noZero; i <= len(nums)-1; i++ {
        res += strconv.Itoa(nums[i])
    }
    return res
}// 注意 sup 的条件，确定一个数应该在前面还是后面
func quickSort(nums []int, left int, right int) {
    if left < right {
        tmp := nums[left]
        l, r := left, right
        for {
            // 先从右向左!!!
            for l < r && sup(nums[r], tmp) {
                r--
            }
            for l < r && inf(nums[l], tmp) {
                l++
            }
            if l >= r {
                break
            }
            nums[l], nums[r] = nums[r], nums[l]
        }
        nums[left], nums[l] = nums[l], nums[left]
        quickSort(nums, left, l-1)
        quickSort(nums, l+1, right)
    }
}
func sup(a, b int) bool {
    aStr := strconv.Itoa(a)
    bStr := strconv.Itoa(b)
    if aStr+bStr >= bStr+aStr {
        return true
    }
    return false
}
func inf(a, b int) bool {
    aStr := strconv.Itoa(a)
    bStr := strconv.Itoa(b)
    if aStr+bStr <= bStr+aStr {
        return true
    }
    return false
}
type bytes [][]byte
func (b bytes) Less(i, j int) bool {
```

```
    size := len(b[i]) + len(b[j])
    bij := make([]byte, 0, size)
    bij = append(bij, b[i]...)
    bij = append(bij, b[j]...)
    bji := make([]byte, 0, size)
    bji = append(bji, b[j]...)
    bji = append(bji, b[i]...)
    for k := 0; k < size; k++ {
        if bij[k] > bji[k] {
            return true
        } else if bij[k] < bji[k] {
            return false
        }
    }
    return false
}
```

三十五：数组中的逆序对(之前没看)

```
func InversePairs(nums []int) int {
    // 用来存储排序好的数组
    tmp := make([]int, len(nums))
    // 合并两个排序好的 array
    var merge func(start int, mid int, end int) int
    merge = func(start int, mid int, end int) int {
        if start >= end {
            return 0
        }
        p1, p2 := mid, end
        k, count := 0, 0
        for p1 >= start && p2 >= mid+1 {
            if nums[p1] <= nums[p2] {
                tmp[k] = nums[p2]
                p2--
                k++
            } else {
                // nums[p1] > nums[p2]
    //因为两个数组是排好序的，所以说明对于 num[p1] 来说至少有 p2-mid 个逆序对
                tmp[k] = nums[p1]
                count += p2 - mid
                p1--
                k++
            }
        }
        for p1 >= start {
            tmp[k] = nums[p1]
            k++
            p1--
        }
        for p2 >= mid+1 {
            tmp[k] = nums[p2]
            k++
            p2--
        }
        for i := 0; i <= k-1; i++ {
            nums[end-i] = tmp[i]
        }
        return count
    }
    var sort func(start, end int) int
    sort = func(start, end int) int {
        count := 0
        if start < end {
            mid := (start + end) / 2
            count += sort(start, mid)
            count += sort(mid+1, end)
            count += merge(start, mid, end)
            return count
        }
        return 0
    }
    return sort(0, len(nums)-1)
}
```

三十七：数字在排序数组中出现的次数

```
func count(nums []int, target int) int {
    left, right := 0, len(nums)-1
    var mid int
    for left < right {
        mid = (left + right) / 2
        if nums[mid] == target {
            for nums[mid] != nums[right] {
                right--
            }
            for nums[mid] != nums[left] {
                left++
            }
            break
        }
        if nums[mid] > target {left = mid + 1
        } else {right = mid - 1}
    }
    if left < right {return right - left + 1}
    return -1
}
```

四十：数组中只出现一次的数字

```
func singleNumber(nums []int) int {
```

```
    res := 0
    for _, n := range nums {
        // n^n == 0
        // a^b^a^b^a == a
        res ^= n
    }
    return res
}
// 两个都出现了一次，找出来
func singleNumber3(nums []int) []int {
    xor := 0
    for _, v := range nums {
        xor ^= v
    }
    // 取 xor 最低位为 1 的数
    lowest := xor & -xor
    a, b := 0, 0
    for _, v := range nums {
        if lowest&v == 0 {
            a ^= v
        } else {
            b ^= v
        }
    }
    return []int{a, b}
}
```

五十：数组中重复的数字

```
func again(nums []int) int {
    appear := make(map[int]bool)
    for _, v := range nums {
        if _, ok := appear[v]; ok {
            return v
        }
        appear[v] = true
    }
    // Not Found Error
    return len(nums) + 1
}
```

五十一：构建乘积数组

```
func multiArray(nums []int) []int {
    res := make([]int, len(nums))
    tmp := 1
    for i := range res {
        res[i] = tmp
        tmp *= nums[i]
    }
    tmp = 1
    for i := range res {
        res[len(res)-1-i] *= tmp
        tmp *= nums[len(res)-1-i]
    }
    return res
}
```

字符串(8 道)：

二：替换空格

```
func replaceSpace1(str []byte, length int) {
    // 首先计算空格长度
    count := 0
    for i := 0; i < length; i++ {
        if str[i] == ' ' {count++}
    }
    newLen := length + count*2
    for l, nl := length-1, newLen-1; l >= 0 && nl >= 0; {
        if str[l] == ' ' {
            str[nl] = '0'
            nl--
            str[nl] = '2'
            nl--
            str[nl] = '%'
            nl--
        } else {
            str[nl] = str[l]
            nl--
        }
        l--
    }
}
```

二十七：字符串的排列

```
func stringPermutation(str string) []string {
    var res []string
    s := []byte(str)
    length := len(s)
    var dfs func(idx int)
    dfs = func(idx int) {
        if idx == length {
            str := string(s)
            res = append(res, str)
            return
        }
        m := make(map[byte]bool)
        // 交换
        for i := idx; i < length; i++ {
            if _, ok := m[s[i]]; ok {
                continue
            }
            m[s[i]] = true
            s[idx], s[i] = s[i], s[idx]
```

```
            dfs(idx + 1)
            s[i], s[idx] = s[idx], s[i]
        }
    }
    dfs(0)
    return res
}
```

三十四：第一个只出现一次的字符

```
func count(str string) int {
    bytes := []byte(str)
    m := make(map[byte]int)
    for _, v := range bytes {m[v]++}
    for i, v := range bytes {if m[v] == 1 {return i}}
    return -1
}
```

四十三：左旋转字符串

```
func rotateString(str string, shift int) string {
    shift = shift % len(str)
    list := []byte(str)
    reverseString(list[:shift])
    reverseString(list[shift:])
    reverseString(list)
    return string(list)
}
func reverseString(s []byte) {
    start, end := 0, len(s)-1
    for start <= len(s)/2-1 {
        s[start], s[end] = s[end], s[start]
        start++
        end--
    }
}
```

四十四：翻转单词顺序序列

```
func reverseWord(str string) string {
    list := []byte(str)
    reverseString(list)
    left, right := 0, 0
    for right <= len(list)-1 {
        if list[right] != ' ' {
            right++
            continue
        }
        reverseString(list[left:right])
        left, right = right+1, right+1
    }
    reverseString(list[left:right])
    return string(list)
}
```

四十九：把字符串转换成整数

```
func myAtoi(str string) int {
    res := 0
    flag := 1
    start := 0
    // space
    for start <= len(str)-1 && str[start] == ' ' {
        start++
    }
    // +/-
    if start <= len(str)-1 {
        if str[start] == '-' {
            flag = -1
            start++
        } else if str[start] == '+' {
            start++
        }
    }
    // is digital ?
    for start <= len(str)-1 && isDigital(str[start]) {
        if res == 0 {
            res += int(str[start] - '0')
            start++
        } else {
            res = res*10 + int(str[start]-'0')
            start++
        }
        // overflow int32
        if res*flag > math.MaxInt32 {
            return math.MaxInt32
        } else if res*flag < math.MinInt32 {
            return math.MinInt32
        }
    }
    res *= flag
    return res
}
func isDigital(b byte) bool {
    if b <= '9' && b >= '0' {
        return true
    }
    return false
}
```

五十二：正则表达式匹配

五十三：表示数值的字符串（pass）

栈(3 道)：

五：用两个栈实现队列

```
// 定义 queue
type Queue struct {
    in  utils.Stack
    out utils.Stack
}
func (q *Queue) IsEmpty() bool {
    return q.out.IsEmpty() && q.out.IsEmpty()
}
func (q *Queue) Push(value interface{}) {
    q.in.Push(value)
}
func (q *Queue) Pop() (interface{}, error) {
    if q.IsEmpty() {
        return nil, errors.New("nil")
    }
    if len(q.out) != 0 {
        value, _ := q.out.Pop()
        return value, nil
    } else {
        // 将 in 数据导入到 out 中
        for {
            if len(q.in) == 1 {
                value, _ := q.in.Pop()
                return value, nil
            } else {
                value, _ := q.in.Pop()
                q.out.Push(value)
            }
        }
    }
}
```

二十：包含 min 函数的栈

```
type MinStack struct {
    stack []item
}
type item struct {
    min, x int
}
// Constructor 构造 MinStack
func Constructor() MinStack {
    return MinStack{}
}
// Push 存入数据
func (s *MinStack) Push(x int) {
    min := x
    if len(s.stack) > 0 && s.GetMin() < x {
        min = s.GetMin()
    }
    s.stack = append(s.stack, item{min: min, x: x})
}
// Pop 抛弃最后一个入栈的值
func (s *MinStack) Pop() {
    s.stack = s.stack[:len(s.stack)-1]
}
// Top 返回最大值
func (s *MinStack) Top() int {
    return s.stack[len(s.stack)-1].x
}
// GetMin 返回最小值
func (s *MinStack) GetMin() int {
    return s.stack[len(s.stack)-1].min
}
```

二十一：栈的压入、弹出序列

```
func stackOrder(in []int, out []int) bool {
    var s utils.Stack
    if len(out) != len(in) {
        return false
    }
    pi, po := 0, 0
    for pi < len(in) {
        if out[po] != in[pi] {
            s.Push(in[pi])
            pi++
            continue
        }
        po++
        pi++
        value, _ := s.Top()
        for po < len(out) && out[po] == value {
            s.Pop()
            po++
            value, _ = s.Top()
        }
    }
    return po == len(out)
}
```

动态规划：(5 道)：

七：裴波那契数列

```
func Fibonacci(n int) int {
    if n <= 1 {
        return n
    }
    f1, f2 := 0, 1
```

```
    res := 0
    for i := 2; i <= n; i++ {
        res = f1 + f2
        f1, f2 = f2, res
    }
    return res
}
```

八：跳台阶

```
func jumpFloor(N int) int {
    if N <= 0 {
        return 0
    }
    if N == 1 || N == 2 {
        return N
    }
    a, b := 1, 2
    for i := 3; i <= N; i++ {
        a, b = b, a+b
    }
    return b
}
```

九：变态跳台阶

```
func jumpFloor2(N int) int {
    if N <= 0 {
        return 0
    }
    if N == 1 || N == 2 {
        return N
    }
    b := 2
    for i := 3; i <= N; i++ {
        b = 2 * b
    }
    return b
}
```

十：矩形覆盖

```
func rectCover(n int) int {
    if n < 1 {
        return 0
    }
    if n == 1 || n == 2 {
        return n
    }
    return rectCover(n-1) + rectCover(n-2)
}
```

三十三：丑数

```
func getUgly(N int) []int {
    res := make([]int, N)
    if N < 1 {
        return res
    }
    res[0] = 1
    index2, index3, index5 := 0, 0, 0
    index := 1
    for index <= N-1 {
    minValue:=min(min(res[index2]*2,res[index3]*3),res[index5]*5)
        if minValue == res[index2]*2 {index2++}
        // 不是 elseif 因为可能重复
        if minValue == res[index3]*3 {index3++}
        if minValue == res[index5]*5 {index5++}
        res[index] = minValue
        index++
    }
    return res
}
func min(a, b int) int {
    if a < b {return a}
    return b
}
```

回溯法(2 道)：

六十五：矩阵中的路径

```
func hasPath(matrix []rune, rows, cols int, str []rune) bool {
    lengthM := len(matrix)
    lengthS := len(str)
    if lengthM == 0 || lengthS == 0 || rows <= 0 || cols <= 0
|| lengthM != rows*cols {
        return false
    }
    //标记是否走过
//此处用数组无法指定长度，用切片操作下标越界，所以
用 map
    flag := make(map[int]int)
    //循环匹配字符
    for i := 0; i < rows; i++ {
        for j := 0; j < cols; j++ {
            //找到 str 路径，返回 true
            if hasPathHandler(matrix, rows, cols, i, j, str,
0, flag){
                return true
            }
        }
    }
    return false
```

```
}
func hasPathHandler(matrix []rune, rows, cols, i, j int, str []rune, k int, flag map[int]int) bool {
    index := i*cols + j
//i 越界,j 越界,矩阵元素不等于 str 字符,矩阵元素已经走过了
    if i < 0 || i >= rows || j < 0 || j >= cols || matrix[index] != str[k] || flag[index] == 1 {
        return false
    }
    //匹配结束
    if k == len(str)-1 {
        return true
    }
    //元素匹配上，falg 变为 1
    flag[index] = 1
    //下一位匹配上下左右，返回 bool，所以是或
   if hasPathHandler(matrix, rows, cols, i-1, j, str, k+1, flag) ||
hasPathHandler(matrix, rows, cols, i+1,j,str k+1,flag)||
hasPathHandler(matrix, rows, cols, i,j-1,str,k+1,flag)||
hasPathHandler(matrix, rows, cols, i,j+1,str,k+1,flag){
        return true
    }
    //没匹配上，当前位 flag 变为 0，因为从下一个元素开始时，当前位有可能会成为下一位
    flag[index] = 0
    return false
}
```

六十六：机器人的运动范围

```
func movingCount(threshold, rows, cols int) int {
//标记是否走过
//此处用数组无法指定长度，用切片操作下标越界，所以用 map
    flag := make(map[int]int)
    return movingCountHandler(threshold, rows,cols,0, 0, flag)
}
func movingCountHandler(threshold,rows,cols,i,j int,flag map[int]int)int{
    index := i*cols + j
    count := 0
    //不用循环，因为从头开始,i 合法,j 合法,此位置没走过
    //坐标和小于 threshold
    if i >= 0 && i < rows && j >= 0 && j < cols && flag[index] == 0 && movingCountCheck(threshold, i, j) {
        flag[index] = 1
        //返回 int，所以是加
        count = 1 +
movingCountHandler(threshold, rows, cols, i-1, j, flag)+
movingCountHandler(threshold, rows, cols, i+1, j, flag)+
movingCountHandler(threshold, rows, cols, i, j-1, flag)+
movingCountHandler(threshold, rows, cols, i, j+1, flag)
    }
    return count
}
func movingCountCheck(threshold, i, j int) bool {
    sum := 0
    for i > 0 {
        sum += i % 10 //取个位数
        i = i / 10      //移除个位
    }
    for j > 0 {
        sum += j % 10 //取个位数
        j = j / 10      //移除个位
    }
    if sum > threshold {
        return false
    }
    return true
}
```

其他(15 道)：

十一：二进制中 1 的个数

```
func Ones(n int) int {
    count := 0
    for n != 0 {
        count++
        n = n & (n - 1)
    }
    return count
}
```

十二：数值的整数次方

```
func Pow(base float64, exp int) (float64, error) {
    if exp == 0 {
        return 1, nil
    }
    if exp < 0 && base == 0 {
        return -1, errors.New("base == 0 and exp < 0")
    }
    if exp > 0 {
        return PowNormal(base, exp), nil
    } else {
        res := PowNormal(base, -exp)
        res = 1 / res
        return res, nil
    }
}
```

```
func PowNormal(base float64, exp int) float64 {
    res, temp := 1.0, base
    for exp != 0 {
        if exp&1 == 1 {res *= temp}
        temp *= temp
        exp >>= 1
    }
    return res
}
```

十九：顺时针打印矩阵

```
func spiralOrder(matrix [][]int) []int {
    var res []int
    n := len(matrix)
    if n == 0 {return res}
    m := len(matrix[0])
    // 计算元素个数
    nb := n * m
    // 初始为 0 层皮
    layer := 0
    // 初始边界
    startN, endN, startM, endM := 0, 0, 0, 0
//每放一个元素，nb 就减一，若 nb==0，说明扒皮结束
    for nb > 0 {
        // 按照 layer 数计算边界
        startN, endN = layer, n-layer-1
        startM, endM = layer, m-layer-1
        // 4 个 for 来扒皮
        for i := startM; i <= endM && nb > 0; i++ {
            res = append(res, matrix[startN][i])
            nb--
        }
        for i := startN + 1; i <= endN && nb > 0; i++ {
            res = append(res, matrix[i][endM])
            nb--
        }
        for i := endM - 1; i >= startM && nb > 0; i-- {
            res = append(res, matrix[endN][i])
            nb--
        }
        for i := endN - 1; i >= startN+1 && nb > 0; i-- {
            res = append(res, matrix[i][startM])
            nb--
        }
        // 层数递增
        layer++
    }
    return res
}
```

二十九：最小的 K 个数(排序+循环)

```
func minK(nums []int, k int) ([]int, error) {
    res := []int{}
    if k > len(nums) {
        return res, errors.New("k > length of nums")
    }
    maxHeap := NewMaxHeap()
    for _, v := range nums {
        if maxHeap.Length() < k {
            maxHeap.Insert(v)
        } else {
            max, _ := maxHeap.Max()
            if max > v {
                maxHeap.DeleteMax()
                maxHeap.Insert(v)
            }
        }
    }
    for maxHeap.Length() > 0 {
        v, _ := maxHeap.DeleteMax()
        res = append(res, v)
    }
    return res, nil
}
```

三十一：整数中 1 出现的次数(从 1-n)

```
func AddOnes(n int) int {
    if n == 0 {return 0}
    if n > 1 && n < 10 {return 1}
    count := 0
    highest := n
    bit := 0
    for highest >= 10 {
        highest /= 10
        bit++
    }
    weight := pow(10, bit)
    if highest == 1 {
    count = AddOnes(weight-1) + AddOnes(n-weight) + (n -weight+ 1)
    } else {
    count=AddOnes(weight-1) + AddOnes(n-highest*weight)+weight
    }
    return count
}
func pow(a, b int) int {
    res := 1
    for i := b; i > 0; i-- {res *= a}
    return res
```

```
}
```

四十一：和为 S 的连续正数序列

```
func arraySum(target int) []int {
    if target < 3 {return []int{}}
    start, end := 1, 2
    sum := 3
    for end <= target/2+1 {
        if sum == target {break
        } else if sum < target {
            end++
            sum += end
        } else {
            sum -= start
            start++
        }
    }
    res := []int{}
    if sum == target {
        for i := start; i <= end; i++ {
            res = append(res, i)
        }
    }
    return res
}
```

四十二：和为 S 的两个数字

```
func twoSum(nums []int, target int) []int {
    m := make(map[int]int)
    for i, v := range nums {
        if val, ok := m[v]; ok {
            return []int{val, i}
        }
        m[target-v] = i
    }
    return []int{}
}
```

四十五：扑克牌顺子

```
func poker(nums []int) string {
    sort.Ints(nums)
    start := 0
    for nums[start] == 0 {start++}
    for i := start + 1; i < len(nums); i++ {
        if nums[i]-nums[i-1]-1 >start ||nums[i] == nums[i-1]{
            return "Oh my god"
        }
        start -= nums[i] - nums[i-1] - 1
        fmt.Println(i, start)
    }
    if start >= 0 {return "So lucky"}
    return "Oh my god"
}
```

四十六：孩子们的游戏(圆圈中最后剩下的数)

```
func cycle1(n, m int) int {
    var (
        children []int
        num      int // 计算去除第几个
    )
    // 编号 0,1,2,...,n-1
    for i := 0; i < n; i++ {children = append(children, i)}
    for len(children) > 1 {
        switch {
        case m <= len(children): // 去除 m 个数
            children = append(children[m:], children[:m-1]...)
        default:
            // 小于那个数量
            num = m % len(children)
    children=append(children[num:],children[:num-1]...)
        }}
    return children[0]
}
array := []int{1,2,3,4,5,6,7,8,9}
index := 3
num := 3
1、去除第一个数
fmt.Println("array[1:]", "=",array[1:])//(包括)
2、去除最后一个数   //(不包括)
fmt.Println("array[:len(array)-1]", "=",array[:len(array)-1])
3、去除第 m 个数据
array = append(array[:num-1], array[num:]...)
fmt.Println("append(array[:num-1], array[num:]...)", "=",array )
// 去除第三个元素的元素,下标为 2
4、去除下标为第 m 的数据
array = append(array[:index], array[index+1:]...)
fmt.Println("append(array[:3], array[3+1:]...)", "=", array)
// 去除下标为 3 的元素
5、打印下标为 3 的数
fmt.Println("array[index]", "=", array[index])
```

四十七：求 1+2+3+…+n(递归)

```
func Nadd(n int) int {
    if n == 0 {return 0}
    return Nadd(n-1) + n
}
```

四十八：不用加减乘除的加法

```
func Add(n, m int) int {
    tmp := 0
    for m != 0 {
        tmp = n ^ m
```

```
        m = (n & m) << 1
        n = tmp
    }
    return n
}
```

五十四：字符流中第一个不重复的字符

```
type Solution struct {
    str    string
    count []int
}
func (s *Solution) Insert(char byte) {
    s.str += string(char)
    s.count[char]++
}
func (s *Solution) GetOnceAppear() byte {
    for i, v := range s.count {
        if v == 1 {return byte(i)}
    }
    return '#'
}
```

六十三：数据流中的中位数

```
type Solution struct {
    max *utils.MaxHeap
    min *utils.MinHeap
}
func (s *Solution) Insert(num int) {
    if ((s.max.Length() + s.min.Length()) & 1) == 0 {
        // 偶数数据的情况下
        // 直接将新的数据插入到数组的后半段
        // 即在最小堆中插入元素
        // 此时最小堆中多出一个元素,
        // 即最小元素,将其弹出后,压入前半段(即最大堆中)
        if s.max.Length() > 0 && num < s.max.GetMax() {
            s.max.Insert(num)
            val, _ := s.max.DeleteMax()
            s.min.Insert(val)
        } else {
            s.min.Insert(num)
        }
    } else {
        // 奇数情况
        if s.max.Length() > 0 && num < s.max.GetMax() {
            s.max.Insert(num)
        } else {
            s.min.Insert(num)
            val, _ := s.min.DeleteMin()
            s.max.Insert(val)
        }
    }
}
func (s *Solution) GetMedian() float64 {
    size := s.max.Length() + s.min.Length()
    if size == 0 {return -1    }
    var median float64
    if size&1 != 0 {
        median = float64(s.min.GetMin())
    } else {
    median=float64(s.min.GetMin()+s.max.GetMax())/2
    }
    return median
}
```

六十四：滑动窗口的最大值

```
// deque  双向队列
func maxSlidingWindow(nums []int, k int) []int {
    var res []int
    var window []int
    for index := 0; index < len(nums); index++ {
        // 最大值已经超出窗口范围
        if index >= k && window[0] <= index-k {
            window = window[1:]
        }
        // 当前值比队列末尾大则替换末尾
        for len(window)>0 && nums[window[len(window)-1]] < nums[index]{
            window = window[:len(window)-1]
        }
        // 添加元素进队列
        window = append(window, index)
        // 记录当前最大值
        if index >= k-1 {
            res = append(res, nums[window[0]])
        }
    }
    return res
}
```

动态规划

整数拆分-将一个整数拆分成乘积最大的

```
func integerBreak(n int) int {
    dp := make([]int, n+1)
    if n < 2 {return 1}
    else if n <= 3 {return n - 1}
    dp[1], dp[2], dp[3] = 1, 2, 3
    for i := 4; i <= n; i++ {
        for j := 1; j <= i/2; j++ { // 尝试所有的剪法
```

```
                dp[i] = max(dp[i], dp[j]*dp[i-j])
            }
        }
        return dp[n]
}
func max(n1, n2 int) int {
        if n1 > n2 {return n1} else {return n2}
}
func min(n1, n2 int) int {
        if n1 > n2 {return n2} else {return n1}
}
```

剪绳子

```
func cuttingRope(n int) int {
        dp := make([]int, n+1)
        if n < 2 {return 1}
        else if n <= 3 {return n - 1}
        dp[2] = 1
        for i := 3; i <= n; i++ {
            for j := 1; j <= i; j++ {// 尝试所有的剪法
                dp[i] = max(dp[i], j*max(i-j, dp[i-j]))
            }
        }
        return dp[n]
}
```

买卖股票的最佳时机-一次

```
func maxProfit1(prices []int) int {
// 前 i 天的最大收益 = max{前 i-1 天的最大收益，第 i 天的价格-前 i-1 天中的最小价格}
        if len(prices) <= 1 {return 0}
        minNum, maxNum := prices[0], 0
        for i := 1; i < len(prices); i++ {
            maxNum = max(maxNum, prices[i]-minNum)
            minNum = min(minNum, prices[i])
        }
        return maxNum
}
```

买卖股票的最佳时II-多次买卖

```
func maxProfit2(prices []int) int {
        //只要今天比昨天大，就卖
        lastNum, maxNum := prices[0], 0
        for i := 1; i < len(prices); i++ {
            if prices[i]-lastNum > 0 {
                maxNum += prices[i] - lastNum
            }
            lastNum = prices[i]
        }
        return maxNum
}
```

买卖股票的最佳时III-最多可以卖两次

```
func maxProfit3(prices []int) int {
        n := len(prices)
        buy1, profit1 := prices[0], 0
        buy2, profit2 := prices[0], 0
        for i := 1; i < n; i++ {
            buy1 = min(buy1, prices[i])
            profit1 = max(profit1, prices[i]-buy1)
            buy2 = min(buy2, prices[i]-profit1) //第二次买入要减去第一次的利润,相当于降低第二次买入的成本了
            profit2 = max(profit2, prices[i]-buy2)
        }
        return profit2
}
```

五种排序

冒泡-越到后面越大

```
func bubbleSort(list []int) []int {
        for i := 1; i < len(list); i++ {
            for j := 0; j < len(list)-i; j++ {
                if list[j] > list[j+1] {
                    list[j], list[j+1] = list[j+1], list[j]
                }
            }
        }
        return list
}
```

冒泡 2-固定最后一个元素

```
func bubbleSort2(list []int) []int {
        for i := len(list) - 1; i > 0; i-- {
            for j := 0; j < i; j++ {
                if list[j] > list[i] {
                    list[j], list[i] = list[i], list[j]
                }
            }
        }
        return list
}
```

选择-找最小的放在前面

```
func selectSort(sli []int) []int {
        len := len(sli)
        for i := 0; i < len-1; i++ {
            k := i // 设最小值的位置为当前
            for j := i + 1; j < len; j++ {
                if sli[k] > sli[j] {k = j}// 记录最小值的位置
```

```
            }
            if k != i {sli[k], sli[i] = sli[i], sli[k]}
        }
        return sli
}
```

快排-分治

```
func quickSort(sli []int) []int {
    len := len(sli)   //先判断是否需要继续进行
    if len <= 1 {return sli}
    base_num := sli[0]     //选择第一个元素作为基准
    left_sli := []int{}       //小于基准的
    right_sli := []int{}      //大于基准的
    for i := 1; i < len; i++ {
        if base_num > sli[i] {
            left_sli = append(left_sli, sli[i]) //放入左边
        } else {
            right_sli = append(right_sli, sli[i]) //放入右边
        }
    }
    left_sli = quickSort(left_sli)
    right_sli = quickSort(right_sli)
    left_sli = append(left_sli, base_num) //  合并
    return append(left_sli, right_sli...)
}
```

插入排序

```
func insertSort(sli []int) []int {
    len := len(sli)
    for i := 1; i < len; i++ {
        //  每次保证前面的数字有序
        tmp := sli[i] //  每次往前要插的数
        for j := i - 1; j >= 0; j-- {
            if tmp < sli[j] {//当前数字比前面小则交换
                sli[j+1], sli[j] = sli[j], tmp
            } else {   //  不比前面的小则退出
                break
            }
        }
    }
    return sli
}
```

归并排序

```
func mergeSort(arr []int) []int {
    if len(arr) == 1 {return arr}
    var left = mergeSort(arr[0 : len(arr)/2]) //不断地分左右
    var right = mergeSort(arr[len(arr)/2:])
    return merge(left, right)   //  合并
}
func merge(left []int, right []int) []int {
    var result = make([]int, len(left)+len(right))
    var leftIndex = 0     //  左数组索引
    var rightIndex = 0    //  右数组索引
    var resultIndex = 0 //  结果索引
    //  按照从小到大的顺序往 result 写数据
    for leftIndex < len(left) && rightIndex < len(right) {
        if left[leftIndex] < right[rightIndex] {
            result[resultIndex] = left[leftIndex]
            leftIndex++
        } else if right[rightIndex] < left[leftIndex] {
            result[resultIndex] = right[rightIndex]
            rightIndex++
        }
        resultIndex++
    }
    for leftIndex < len(left) {//左边还有数字，加在后面
        result[resultIndex] = left[leftIndex]
        leftIndex++
        resultIndex++
    }
    for rightIndex < len(right) {//右边还有数字，加在后面
        result[resultIndex] = right[rightIndex]
        rightIndex++
        resultIndex++
    }
    return result
}
//  堆排序-算了

func main() {
    fmt.Println(integerBreak(6)) //  数字拆分
    fmt.Println(cuttingRope(6)) //  剪绳子
    //  排序
    list := []int{2, 6, 9, 7, 5, 3, 4, 8, 1}
    var aa = make([]int, len(list))
    copy(aa, list)
    bubbleSort2(aa)
    fmt.Println(list) // list 不变
    fmt.Println(aa) //  冒泡排序
    fmt.Println(selectSort([]int{2, 6, 9, 7, 5, 3, 4, 8, 1}))
    fmt.Println(quickSort([]int{2, 6, 9, 7, 5, 3, 4, 8, 1}))
    fmt.Println(insertSort([]int{2, 6, 9, 7, 5, 3, 4, 8, 1}))
    fmt.Println(mergeSort([]int{1, 39, 2, 9, 7, 54, 11, 8}))
}
```