1. 3. 无重复字符的最长子串
2. 206. 反转链表
3. 146. LRU 缓存机制
4. 215. 数组中的第K个最大元素
5. 25. K 个一组翻转链表
6. 15. 三数之和
7. 53. 最大子序和
8. 补充题4. 手撕快速排序 (912. 排序数组)
9. 21. 合并两个有序链表
10. 1. 两数之和
11. 5. 最长回文子串
12. 102. 二叉树的层序遍历
13. 33. 搜索旋转排序数组
14. 200. 岛屿数量
15. 20. 有效的括号
16. 121. 买卖股票的最佳时机
17. 141. 环形链表
18. 236. 二叉树的最近公共祖先
19. 46. 全排列
20. 47. 全排列 II 补充
21. 88. 合并两个有序数组
22. 103. 二叉树的锯齿形层序遍历
23. 92. 反转链表 II
24. 54. 螺旋矩阵
25. 23. 合并K个升序链表
26. 300. 最长递增子序列
27. 160. 相交链表
28. 415. 字符串相加
29. 143. 重排链表
30. 42. 接雨水
31. 142. 环形链表 II
32. 56. 合并区间
33. 124. 二叉树中的最大路径和
34. 72. 编辑距离
35. 19. 删除链表的倒数第N个节点
36. 93. 复原 IP 地址
37. 1143. 最长公共子序列
38. 94. 二叉树的中序遍历
39. 82. 删除排序链表中的重复元素 II
40. 704. 二分查找
41. 199. 二叉树的右视图
42. 31. 下一个排列
43. 4. 寻找两个正序数组的中位数
44. 232. 用栈实现队列
45. 148. 排序链表
46. 69. x 的平方根

47. 8. 字符串转换整数 (atoi)
48. 22. 括号生成
49. 70. 爬楼梯
50. 2. 两数相加
51. 165. 比较版本号
52. 239. 滑动窗口最大值
53. 41. 缺失的第一个正数
54. 剑指 Offer 22. 链表中倒数第k个节点
55. 322. 零钱兑换
56. 518. 零钱兑换 II
57. 76. 最小覆盖子串
58. 78. 子集
59. 105. 从前序与中序遍历序列构造二叉树
60. 43. 字符串相乘
61. 32. 最长有效括号
62. 155. 最小栈
63. 151. 翻转字符串里的单词
64. 129. 求根节点到叶节点数字之和
65. 104. 二叉树的最大深度
66. 101. 对称二叉树
67. 144. 二叉树的前序遍历
68. 110. 平衡二叉树
69. 39. 组合总和
70. 543. 二叉树的直径
71. 470. 用 Rand7() 实现 Rand10()
72. 48. 旋转图像
73. 98. 验证二叉搜索树
74. 394. 字符串解码
75. 34. 在排序数组中查找元素的第一个和最后一个位置
76. 113. 路径总和 II
77. 240. 搜索二维矩阵 II
78. 64. 最小路径和
79. 221. 最大正方形
80. 162. 寻找峰值
81. 14. 最长公共前缀
82. 128. 最长连续序列
83. 234. 回文链表
84. 112. 路径总和
85. 662. 二叉树最大宽度
86. 169. 多数元素
87. 62. 不同路径
88. 179. 最大数
89. 718. 最长重复子数组
90. 227. 基本计算器 II
91. 122. 买卖股票的最佳时机 II
92. 198. 打家劫舍

93. 152. 乘积最大子数组
94. 83. 删除排序链表中的重复元素
95. 695. 岛屿的最大面积
96. 226. 翻转二叉树
97. 139. 单词拆分
98. 560. 和为 K 的子数组
99. 209. 长度最小的子数组
100. 补充题6. 手撕堆排序 912. 排序数组
101. 24. 两两交换链表中的节点
102. 224. 基本计算器

3. 无重复字符的最长子串

```
func lengthOfLongestSubstring(s string) int {
    longest, n := 0, len(s)
    freq := make(map[byte]int, n)
    for i, j := 0, 0; j < n; j++ {
        freq[s[j]]++
        for freq[s[j]] > 1 {
            freq[s[i]]--
            i++
        }
        longest = max(longest, j-i+1)
    }
    return longest
}
```

```
func lengthOfLongestSubstring(s string) int {
    longest, n := 0, len(s)
    freq := make(map[byte]int, n)
    for l, r := 0, 0; r < n; r++ {
        freq[s[r]]++
        for freq[s[r]] > 1 {
            freq[s[l]]--
            l++
        }
        longest = max(longest, r-l+1)
    }
    return longest
}
```

206. 反转链表

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
```

```
 *     Val int
 *     Next *ListNode
 * }
*/
// 方法一：迭代
func reverseList(head *ListNode) *ListNode {
    var prev *ListNode
    curr := head
    for curr != nil {
        temp := curr.Next
        curr.Next = prev
        prev = curr
        curr = temp
    }
    return prev
}

// 方法二：递归
func reverseList_2(head *ListNode) *ListNode {
    if head == nil || head.Next == nil { // 递归出口：没有节点或只有一个节点
        return head
    }
    newHead := reverseList(head.Next) // 子问题
    head.Next.Next = head             // 翻转
    head.Next = nil                   // 断开旧链
    return newHead
}

// 方法三：穿针引线
func reverseList_3(head *ListNode) *ListNode {
    dummy, curr := &ListNode{Next: head}, head
    for curr != nil && curr.Next != nil { // 至少有2个节点
        temp := curr.Next
        curr.Next = temp.Next
        temp.Next = dummy.Next // 如果等于 curr，将导致断开链表
        dummy.Next = temp
    }
    return dummy.Next
}
```

146. LRU 缓存机制

```
type LRUCache struct {
    cache          map[int]*DLinkedNode
    head, tail     *DLinkedNode
    size, capacity int
}

type DLinkedNode struct {
    key, value int
    prev, next *DLinkedNode
```

```
}

func initDLinkedNode(key, value int) *DLinkedNode {
    return &DLinkedNode{
        key:   key,
        value: value,
    }
}

func Constructor(capacity int) LRUCache {
    l := LRUCache{
        cache:    map[int]*DLinkedNode{},
        head:     initDLinkedNode(0, 0),
        tail:     initDLinkedNode(0, 0),
        capacity: capacity,
    }
    l.head.next = l.tail
    l.tail.prev = l.head
    return l
}

func (this *LRUCache) Get(key int) int {
    if _, ok := this.cache[key]; !ok {
        return -1
    }
    node := this.cache[key] // 如果 key 存在，先通过哈希表定位，再移到头部
    this.moveToHead(node)
    return node.value
}

func (this *LRUCache) Put(key int, value int) {
    if _, ok := this.cache[key]; !ok { // 如果 key 不存在，创建一个新的节点
        node := initDLinkedNode(key, value)
        this.cache[key] = node // 添加进哈希表
        this.addToHead(node)   // 添加至双向链表的头部
        this.size++
        if this.size > this.capacity {
            removed := this.removeTail()    // 如果超出容量，删除双向链表的尾部
节点
            delete(this.cache, removed.key) // 删除哈希表中对应的项
            this.size--
        }
    } else { // 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部
        node := this.cache[key]
        node.value = value
        this.moveToHead(node)
    }
}

func (this *LRUCache) addToHead(node *DLinkedNode) {
    node.prev = this.head
    node.next = this.head.next
    this.head.next.prev = node
    this.head.next = node
```

```
}

func (this *LRUCache) removeNode(node *DLinkedNode) {
    node.prev.next = node.next
    node.next.prev = node.prev
}

func (this *LRUCache) moveToHead(node *DLinkedNode) {
    this.removeNode(node)
    this.addToHead(node)
}

func (this *LRUCache) removeTail() *DLinkedNode {
    node := this.tail.prev
    this.removeNode(node)
    return node
}

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

215. 数组中的第K个最大元素

方法一：基于快速排序的选择方法

快速选择算法思路：

只要某次划分的 q 为倒数第 k 个下标的时候，我们就已经找到了答案。 如果划分得到的 q 正好就是我们需要的下标，就直接返回 a[q]； 否则，如果 q 比目标下标小，就递归右子区间，否则递归左子区间。

```
func findKthLargest(A []int, k int) int {
    n := len(A)
    return quickSelect(A, 0, n-1, n-k)
}

func quickSelect(A []int, l, r, k int) int { // kSmallest
    if l == r {
        return A[k]
    }
    x := A[l+((r-l)>>1)]
    i, j := l-1, r+1
    for i < j {
        for i++; A[i] < x; i++ {
        }
        for j--; A[j] > x; j-- {
        }
        if i < j {
```

```
            A[i], A[j] = A[j], A[i]
        }
    }
    if k <= j {
        return quickSelect(A, l, j, k)
    } else {
        return quickSelect(A, j+1, r, k)
    }
}
```

复杂度分析

- 时间复杂度：O(n)，如上文所述，证明过程可以参考「《算法导论》9.2：期望为线性的选择算法」。
- 空间复杂度：O(logn)，递归使用栈空间的空间代价的期望为 O(logn)。

- 考点1：能否实现解法的优化
- 考点2：是否了解快速选择算法
- 考点3：能否说明堆算法和快速选择算法的适用场景

方法二：基于堆排序的选择方法

思路和算法

建立一个大根堆，做 k - 1 次删除操作后堆顶元素就是我们要找的答案。

```
// 在大根堆中、最大元素总在根上，堆排序使用堆的这个属性进行排序
func findKthLargest(A []int, k int) int {
    heapSize, n := len(A), len(A)
    buildMaxHeap(A, heapSize) // A[0]为堆顶
    for i := heapSize - 1; i >= n-k+1; i-- {
        A[0], A[i] = A[i], A[0]    // 交换堆顶元素 A[0] 与堆底元素 A[i]，最大值
A[0] 放置在数组末尾
        heapSize--                 // 删除堆顶元素 A[0]
        maxHeapify(A, 0, heapSize) // 向下调整堆顶元素 A[0]
    }
    return A[0]
}

// 建堆 O(n)
func buildMaxHeap(A []int, heapSize int) {
    for i := heapSize >> 1; i >= 0; i-- { // heapSize / 2 后面都是叶子节点，不
需要向下调整
        maxHeapify(A, i, heapSize)
    }
}

// 迭代: 调整大根堆 O(n)
func maxHeapify(A []int, i, heapSize int) {
    for i<<1+1 < heapSize {
        l, r, largest := i<<1+1, i<<1+2, i
        for l < heapSize && A[l] > A[largest] { // 左儿子存在并大于根
            largest = l
```

```
        }
        for r < heapSize && A[r] > A[largest] { // 右儿子存在并大于根
            largest = r
        }
        if i != largest { // 找到左右儿子的最大值
            A[i], A[largest] = A[largest], A[i]
            i = largest // 堆顶调整为最大值
        } else {
            break
        }
    }
}

// 递归: 调整大根堆 O(nlogn)
func MaxHeapify(A []int, i, heapSize int) {
    l, r, largest := i*2+1, i*2+2, i
    for l < heapSize && A[l] > A[largest] {
        largest = l
    }
    for r < heapSize && A[r] > A[largest] {
        largest = r
    }
    if largest != i {
        A[i], A[largest] = A[largest], A[i]
        MaxHeapify(A, largest, heapSize) // 递归调整子树
    }
}
```

复杂度分析

- 时间复杂度：O(nlogn)，建堆的时间代价是 O(n)，删除的总代价是 O(klogn)，因为 k < n，故渐进时间复杂为 O(n+klogn)=O(nlogn)。
- 空间复杂度：O(logn)，即递归使用栈空间的空间代价。

25. K 个一组翻转链表

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func reverseKGroup(head *ListNode, k int) *ListNode {
    dummy := &ListNode{Next: head}
    prev := dummy
    for head != nil {
        tail := prev
        for i := 0; i < k; i++ {
            tail = tail.Next
            if tail == nil {
```

```
                return dummy.Next
            }
        }
        temp := tail.Next
        tail.Next = nil
        prev.Next = reverse(head)
        prev = head
        prev.Next = temp
        head = temp
    }
    return dummy.Next
}

func reverse(head *ListNode) *ListNode {
    var prev *ListNode
    curr := head
    for curr != nil {
        temp := curr.Next
        curr.Next = prev
        prev = curr
        curr = temp
    }
    return prev
}
```

15. 三数之和

```
func threeSum(nums []int) [][]int {
    sort.Ints(nums)
    res := [][]int{}
    for i := 0; i < len(nums)-2; i++ {
        n1 := nums[i]
        if n1 > 0 { //如果最小的数大于0，break
            break
        }
        if i > 0 && n1 == nums[i-1] { //如果和前一个相同，跳过
            continue
        }
        start, end := i+1, len(nums)-1 //转换为两数之和，双指针解法
        for start < end {
            n2, n3 := nums[start], nums[end]
            if n1+n2+n3 == 0 {
                res = append(res, []int{n1, n2, n3})
                for start < end && nums[start] == n2 { //去重移位
                    start++
                }
                for start < end && nums[end] == n3 {
                    end--
                }
            } else if n1+n2+n3 < 0 {
                start++
```

```
            } else {
                end--
            }
        }
    }
    return res
}
```

53. 最大子序和

```
func maxSubArray(nums []int) int {
    prev, maxSum := 0, nums[0]
    for _, curr := range nums {
        // 若当前指针所指元素之前的和小于0，则丢弃当前元素之前的数列
        prev = max(prev+curr, curr)
        // 将当前值与最大值比较，取最大
        maxSum = max(maxSum, prev)
    }
    return maxSum
}
```

```
func maxSubArray(nums []int) int {
    maxSum := nums[0]
    for i := 1; i < len(nums); i++ {
        if nums[i-1]+nums[i] > nums[i] { // 若前一个元素大于0，将其加到当前元素上
            nums[i] += nums[i-1] // nums[i-1] > 0
        }
        maxSum = max(maxSum, nums[i])
    }
    return maxSum
}
```

补充题4. 手撕快速排序 (912. 排序数组)

「快速排序 quick sort」是一种基于分治策略的排序算法，运行高效，应用广泛。

1. 选取数组 "3数中值" 为基准数；
2. 将所有小于基准数的元素移动到其左边，大于基准数的元素移动到其右边；
3. 递归处理左右两边。

简写

```
func sortArray(nums []int) []int {
    quickSort(nums, 0, len(nums)-1)
    return nums
}
```

```
func quickSort(nums []int, start, end int) {
    if start >= end { // 子数组长度为 1 时终止递归
        return
    }
    pivot := nums[start+(end-start)>>1] // 选取中值 pivot 划分
    i, j := start-1, end+1
    for i < j {
        for i++; nums[i] < pivot; i++ { // 从左向右扫描，找到大于 pivot 的数，停
止
        }
        for j--; nums[j] > pivot; j-- { // 从右向左扫描，找到小于 pivot 的数，停
止
        }
        if i < j {
            nums[i], nums[j] = nums[j], nums[i] // 交换，使得左边小于 pivot，
右边大于 pivot
        }
    }
    quickSort(nums, start, j) // 递归处理左边
    quickSort(nums, j+1, end) // 递归处理左边
}
```

标准版

```
func sortArray(nums []int) []int {
    quickSort(nums, 0, len(nums)-1)
    return nums
}

func quickSort(nums []int, start, end int) {
    if start >= end { // 子数组长度为 1 时终止递归
        return
    }
    piv_pos := partition(nums, start, end) // 获取分区索引
    quickSort(nums, start, piv_pos)        // 递归处理左边
    quickSort(nums, piv_pos+1, end)        // 递归处理右边
}

func partition(nums []int, start, end int) int {
    pivot := nums[start+(end-start)>>1] // 以中值作为基准数
    i, j := start-1, end+1
    for {
        for i++; nums[i] < pivot; i++ { // 从左向右找首个大于基准数的元素
        }
        for j--; nums[j] > pivot; j-- { // 从左向右找首个大于基准数的元素
        }
        if i < j {
            nums[i], nums[j] = nums[j], nums[i] // 交换元素到正确的区间
        } else {
            break
        }
```

```
    }
    return j // 返回基准数的索引
}
```

3数中值分割优化

```
func sortArray(nums []int) []int {
    quickSort(nums, 0, len(nums)-1)
    return nums
}

func quickSort(nums []int, start, end int) {
    if start >= end { // 子数组长度为 1 时终止递归
        return
    }
    piv_pos := partition(nums, start, end) // 获取分区索引
    quickSort(nums, start, piv_pos)        // 递归处理左边
    quickSort(nums, piv_pos+1, end)        // 递归处理右边
}

func partition(nums []int, start, end int) int {
    pivot := median3(nums, start, end) // 三数中值分割
    i, j := start-1, end+1
    for {
        for i++; nums[i] < pivot; i++ { // 从左向右找首个大于基准数的元素
        }
        for j--; nums[j] > pivot; j-- { // 从左向右找首个大于基准数的元素
        }
        if i < j {
            nums[i], nums[j] = nums[j], nums[i] // 交换元素到正确的区间
        } else {
            break
        }
    }
    return j // 返回基准数的索引
}

// 三数中值分割，减少 5% 运行时间
func median3(nums []int, start, end int) int {
    mid := start + (end-start)>>1
    if nums[start] > nums[mid] {
        nums[start], nums[mid] = nums[mid], nums[start]
    }
    if nums[start] > nums[end] {
        nums[start], nums[end] = nums[end], nums[start]
    }
    if nums[mid] > nums[end] {
        nums[end], nums[mid] = nums[mid], nums[end]
    }
    return nums[mid] // A[start] <= A[mid] <= A[end]
}
```

算法特性

- **时间复杂度 O(nlog(n))、自适应排序**：在平均情况下，哨兵划分的递归层数为 log n，每层中的总循环数为 n，总体使用 O(nlog(n)) 时间。在最差情况下，每轮哨兵划分操作都将长度为 n 的数组划分为长度为 0 和 n-1 的两个子数组，此时递归层数达到 n 层，每层中的循环数为n，总体使用 O(n^{2})时间。
- **空间复杂度 O(n)、原地排序**：在输入数组完全倒序的情况下，达到最差递归深度 n，使用 O(n) 栈帧空间。排序操作是在原数组上进行的，未借助额外数组。
- **非稳定排序**：在哨兵划分的最后一步，基准数可能会被交换至相等元素的右侧。

快排为什么快?

从名称上就能看出，快速排序在效率方面应该具有一定的优势。尽管快速排序的平均时间复杂度与“归并排序”和“堆排序”相同，但通常快速排序的效率更高，主要有以下原因。

- **出现最差情况的概率很低**：虽然快速排序的最差时间复杂度为 O(n^{2})，没有归并排序稳定，但在绝大多数情况下，快速排序能在 O(nlog(n)) 的时间复杂度下运行。
- **缓存使用效率高**：在执行哨兵划分操作时，系统可将整个子数组加载到缓存，因此访问元素的效率较高。而像“堆排序”这类算法需要跳跃式访问元素，从而缺乏这一特性。
- **复杂度的常数系数低**：在上述三种算法中，快速排序的比较、赋值、交换等操作的总数量最少。这与“插入排序”比“冒泡排序”更快的原因类似。

最优解

```go
func sortArray(nums []int) []int {
    quickSort(nums, 0, len(nums)-1)
    return nums
}

func quickSort(A []int, l, r int) {
    Cutoff := 3
    if l+Cutoff <= r {
        piv_pos := partition(A, l, r)
        quickSort(A, l, piv_pos-1)
        quickSort(A, piv_pos+1, r)
    } else { // Do an insortion sort on the subarry
        InsertionSort(A, l, r)
    }
}

func partition(A []int, l, r int) int {
    pivot := median3(A, l, r)
    i, j := l, r-1
    for {
        for i++; A[i] < pivot; i++ {
        }
        for j--; A[j] > pivot; j-- {
        }
        if i < j {
```

```
            A[i], A[j] = A[j], A[i]
        } else {
            break
        }
    }
    A[i], A[r-1] = A[r-1], A[i] // Restore pivot
    return i
}

// 三数中值分割，减少 5% 运行时间
func median3(A []int, l, r int) int {
    mid := l + (r - l) >> 1
    if A[l] > A[mid] {
        A[l], A[mid] = A[mid], A[l]
    }
    if A[l] > A[r] {
        A[l], A[r] = A[r], A[l]
    }
    if A[mid] > A[r] {
        A[r], A[mid] = A[mid], A[r]
    }
    // A[l] <= A[mid] <= A[r]
    A[mid], A[r-1] = A[r-1], A[mid] // Hide pivot
    return A[r-1]                   // return pivot
}

// 很小数组（n <= 20),快排不如插入排序，减少 15% 运行时间,一种好的截止范围 cutoff =
10
func InsertionSort(A []int, l, r int) {
    for i := l; i <= r; i++ {
        temp, j := A[i], i           // temp 插入元素
        for j > 0 && temp < A[j-1] { // 如果新元素小于有序元素
            A[j] = A[j-1] // 右移
            j--           // 向左扫描
        }
        A[j] = temp // 插入新元素
    }
}
```

21. 合并两个有序链表

方法一：递归

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
```

```go
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
    }
}
```

方法二：迭代

当 l1 和 l2 都不是空链表时，判断 l1 和 l2 哪一个链表的头节点的值更小，将较小值的节点添加到结果里， 当一个节点被添加到结果里之后，将对应链表中的节点向后移一位。

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func mergeTwoLists(l1 *ListNode, l2 *ListNode) *ListNode {
    dummy := new(ListNode)
    prev := dummy
    for l1 != nil && l2 != nil {
        if l1.Val < l2.Val {
            prev.Next = l1
            l1 = l1.Next
        } else {
            prev.Next = l2
            l2 = l2.Next
        }
        prev = prev.Next
    }
    if l1 == nil {
        prev.Next = l2
    } else {
        prev.Next = l1
    }
    return dummy.Next
}
```

1. 两数之和

```go
func twoSum(nums []int, target int) []int {
    hash := map[int]int{}
    for i, v := range nums {
        if j, ok := hash[target-v]; ok {
            return []int{i, j}
        }
        hash[v] = i
    }
    return nil
}
```

5. 最长回文子串

```go
func longestPalindrome(s string) string {
    res, n := "", len(s)
    var extend func(int, int)

    extend = func(i, j int) {  // 中心扩展算法
        for i >= 0 && j < n && s[i] == s[j] {
            if len(res) < j-i+1 {
                res = s[i : j+1]
            }
            i-- // 扩展
            j++
        }
    }
    for i := 0; i < n; i++ {
        extend(i, i)   // 以自身为中心点
        extend(i, i+1) // 以自身和自身的下一个元素为中心点
    }
    return res
}
```

102. 二叉树的层序遍历

方法一：DFS递归

```go
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */

func levelOrder(root *TreeNode) (res [][]int) {
```

```
    var dfs func(*TreeNode, int)

    dfs = func(node *TreeNode, level int) {
        if node == nil {
            return
        }
        if len(res) == level { // 首次进入，加入空列表
            res = append(res, []int{})
        }
        res[level] = append(res[level], node.Val) // 将当前节点的值加入当前层
        dfs(node.Left, level+1)                   // 递归扫描下一层节点
        dfs(node.Right, level+1)
    }

    dfs(root, 0)
    return
}
```

方法二：BFS(queue)迭代

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func levelOrder(root *TreeNode) (res [][]int) {
    if root == nil {
        return
    }
    queue := []*TreeNode{root}                // 存储当前层节点到队列
    for level := 0; len(queue) > 0; level++ { // 队列不为空,遍历队列，检查下一层
        res = append(res, []int{})
        next := []*TreeNode{}
        for j := 0; j < len(queue); j++ { // 遍历当前层
            node := queue[j]
            res[level] = append(res[level], node.Val) // 存储当前层节点的值
            if node.Left != nil {                     // 遍历左子树，加入下一
层队列
                next = append(next, node.Left)
            }
            if node.Right != nil { // 遍历右子树，加入下一层队列
                next = append(next, node.Right)
            }
        }
        queue = next // 扫描下一层
    }
    return
}
```

33. 搜索旋转排序数组

```
func search(nums []int, target int) int {
    l, r := 0, len(nums)-1
    for l <= r {
        mid := l + (r-l)>>1
        if nums[mid] == target {
            return mid
        }
        if nums[l] <= nums[mid] { // 左边有序
            if nums[l] <= target && target < nums[mid] { // 答案在左边
                r = mid - 1
            } else {
                l = mid + 1
            }
        } else { // 右边有序
            if nums[mid] < target && target <= nums[r] { // 答案在右边
                l = mid + 1
            } else {
                r = mid - 1
            }
        }
    }
    return -1
}
```

200. 岛屿数量

```
func numIslands(grid [][]byte) int {
    count := 0
    for i := 0; i < len(grid); i++ { // 行
        for j := 0; j < len(grid[0]); j++ { // 列
            if grid[i][j] == '1' { // 如果找到岛屿
                count++          // 岛屿数量加1
                dfs(grid, i, j) // dfs标记此岛屿所有节点已遍历
            }
        }
    }
    return count
}

func dfs(grid [][]byte, i, j int) {
    if 0 <= i && i < len(grid) && 0 <= j && j < len(grid[0]) && grid[i][j]
== '1' {
        grid[i][j] = '0'  // 标记此节点已遍历
        dfs(grid, i+1, j) //右（顺序无关）
        dfs(grid, i-1, j) //左
        dfs(grid, i, j+1) //上
```

```
            dfs(grid, i, j-1) //下
        }
    }
```

20. 有效的括号

```
func isValid(s string) bool {
    if len(s) == 0 {
        return true
    }
    stack := make([]rune, 0)
    for _, v := range s {
        if v == '(' || v == '[' || v == '{' { // 遇到左括号入栈，等待右括号
            stack = append(stack, v)
        } else if len(stack) > 0 && stack[len(stack)-1] == '(' && v == ')'
||
            len(stack) > 0 && stack[len(stack)-1] == '[' && v == ']' ||
            len(stack) > 0 && stack[len(stack)-1] == '{' && v == '}' {
            stack = stack[:len(stack)-1] // 遇到右括号与前面左括号组成有效的括
号，出栈
        } else {
            return false // 无法组成有效的括号
        }
    }
    return len(stack) == 0
}
```

121. 买卖股票的最佳时机

```
func maxProfit(prices []int) int {
    min_price, max_profit := math.MaxInt64, 0
    for _, price := range prices {
        min_price = min(min_price, price)
        max_profit = max(max_profit, price-min_price)
    }
    return max_profit
}

func maxProfit1(prices []int) int {
    min_price, max_profit := 1<<63-1, 0
    for _, price := range prices {
        if price < min_price {
            min_price = price
        }
        if max_profit < price-min_price {
            max_profit = price - min_price
        }
    }
```

```
    return max_profit
}
```

141. 环形链表

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func hasCycle(head *ListNode) bool {
    slow, fast := head, head
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next // 需提前判断不为 nil
        if slow == fast {
            return true
        }
    }
    return false
}
func hasCycle1(head *ListNode) bool {
    if head == nil || head.Next == nil {
        return false
    }
    slow, fast := head, head.Next
    for slow != fast {
        if fast == nil || fast.Next == nil {
            return false
        }
        slow = slow.Next
        fast = fast.Next.Next
    }
    return true
}
```

236. 二叉树的最近公共祖先

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func lowestCommonAncestor(root, p, q *TreeNode) *TreeNode {
```

```
    if root == nil || root == p || root == q { // 越过叶节点，返回 nil
        return root // root 等于 p、q，返回root
    }
    left := lowestCommonAncestor(root.Left, p, q)
    right := lowestCommonAncestor(root.Right, p, q)
    if left == nil { // 左子树为空，p,q 都不在 root 的左子树中，返回 right
        return right
    }
    if right == nil {
        return left
    }
    return root // 左右子树都不为空，p、q 在 root 异侧，root 为最近公共祖先
}
```

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func lowestCommonAncestor(root, p, q *TreeNode) *TreeNode {
    if root == nil { // 越过叶节点，返回 nil
        return nil
    }
    if root == p || root == q { // root 恰好是 p 节点或 q 节点
        return root
    }
    left := lowestCommonAncestor(root.Left, p, q)
    right := lowestCommonAncestor(root.Right, p, q)
    if left != nil && right != nil { // 如果左子树和右子树均包含 p 节点或 q 节点
        return root
    }
    if left != nil { // 如果左子树包含 p 节点，那么右子树只能包含 q 节点
        return left
    }
    return right
}
```

46. 全排列

```
func permute(nums []int) (res [][]int) {
    n := len(nums)
    var dfs func(int)

    dfs = func(pos int) {
        if pos == n { // 所有数都填完了
            res = append(res, append([]int{}, nums...))
```

```
            return
        }
        for i := pos; i < n; i++ {
            nums[i], nums[pos] = nums[pos], nums[i] // 枚举 pos 位置的所有选择
            dfs(pos + 1)                            // 枚举下一个位置
            nums[i], nums[pos] = nums[pos], nums[i] // 撤销选择\回溯
        }
    }

    dfs(0)
    return
}
```

```
func permute(nums []int) [][]int {
    used, path, res, n := make(map[int]bool, len(nums)), []int{}, []
[]int{}, len(nums)
    var dfs func(int)

    dfs = func(pos int) { // 枚举位置
        if len(path) == n {
            res = append(res, append([]int{}, path...))
            return
        }
        for i := 0; i < n; i++ { // 枚举所有的选择
            if !used[i] { // 第i个位置未使用
                path = append(path, nums[i]) // 做出选择，记录路径
                used[i] = true               // 第i个位置已使用
                dfs(pos + 1)                 // 枚举下一个位置
                used[i] = false              // 撤销选择
                path = path[:len(path)-1]    // 取消记录
            }
        }
    }
    dfs(0)
    return res
}
```

47. 全排列 II 补充

```
func permuteUnique(nums []int) (res [][]int) {
    n := len(nums)
    var dfs func(int)

    dfs = func(pos int) {
        if pos == n-1 {
            res = append(res, append([]int{}, nums...))
            return
        }
        exist := make(map[int]bool, n)
```

```
        for i := pos; i < n; i++ {
            if _, ok := exist[nums[i]]; ok {
                continue
            }
            exist[nums[i]] = true
            nums[i], nums[pos] = nums[pos], nums[i]
            dfs(pos + 1)
            exist[nums[i]] = false
            nums[i], nums[pos] = nums[pos], nums[i]
        }
    }
    dfs(0)
    return
}
```

```
func permuteUnique(nums []int) [][]int {
    sort.Ints(nums)
    used, res, path := make([]bool, len(nums)), [][]int{}, []int{}
    var dfs func(int)

    dfs = func(pos int) {
        if len(path) == len(nums) {
            res = append(res, append([]int{}, path...))
            return
        }
        for i := 0; i < len(nums); i++ {
            if used[i] || i > 0 && !used[i-1] && nums[i-1] == nums[i] { //
已使用 或 重复
                continue // 去重，跳过
            }
            used[i] = true
            path = append(path, nums[i])
            dfs(pos + 1)
            used[i] = false
            path = path[:len(path)-1]
        }
    }

    dfs(0)
    return res
}
```

88. 合并两个有序数组

```
func merge(nums1 []int, m int, nums2 []int, n int) {
    for tail := m + n; m > 0 && n > 0; tail-- {
        if nums1[m-1] < nums2[n-1] {
            nums1[tail-1] = nums2[n-1]
            n--
```

```
        } else {
            nums1[tail-1] = nums1[m-1]
            m--
        }
    }
    for ; n > 0; n-- {
        nums1[n-1] = nums2[n-1]
    }
}

func merge2(nums1 []int, m int, nums2 []int, n int) {
    i, j := m-1, n-1
    for tail := m + n - 1; tail >= 0; tail-- {
        if i < 0 || (j >= 0 && nums1[i] <= nums2[j]) {
            nums1[tail] = nums2[j]
            j--
        } else {
            nums1[tail] = nums1[i]
            i--
        }
    }
}
```

103. 二叉树的锯齿形层序遍历

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func zigzagLevelOrder(root *TreeNode) (res [][]int) {
    var dfs func(*TreeNode, int)

    dfs = func(root *TreeNode, level int) {
        if root != nil {
            if len(res) == level {
                res = append(res, []int{})
            }
            if level%2 == 0 { // 偶数层
                res[level] = append(res[level], root.Val) // 先从左往右
            } else {
                res[level] = append([]int{root.Val}, res[level]...) // 再从
右往左进行下一层遍历
            }
            dfs(root.Left, level+1)
            dfs(root.Right, level+1)
        }
```

```
    }

    dfs(root, 0)
    return
}
```

92. 反转链表 II

- curr：指向待反转区域的第一个节点 left；
- next：永远指向 curr 的下一个节点，循环过程中，curr 变化以后 next 会变化；
- pre：永远指向待反转区域的第一个节点 left 的前一个节点，在循环过程中不变。

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func reverseBetween(head *ListNode, left int, right int) *ListNode {
    dummy := &ListNode{Next: head}
    prev := dummy
    for i := 0; i < left-1; i++ {
        prev = prev.Next
    }
    curr := prev.Next
    for i := 0; i < right-left; i++ {
        next := curr.Next
        curr.Next = next.Next
        next.Next = prev.Next
        prev.Next = next
    }
    return dummy.Next
}
```

54. 螺旋矩阵

```
func spiralOrder(matrix [][]int) []int {
    if len(matrix) == 0 {
        return []int{}
    }
    res := []int{}
    top, right, bottom, left := 0, len(matrix[0])-1, len(matrix)-1, 0
    size := len(matrix) * len(matrix[0])
    for len(res) != size {            // 仍未遍历结束
        for i := left; i <= right; i ++ { res = append(res, matrix[top]
[i])}    // 上层  top 行 i 列
```

```
        top ++
        for i := top; i <= bottom; i ++ { res = append(res, matrix[i]
[right])}  // 右层  i 行 right 列
        right --
        if len(res) == size { break } // 遍历结束
        for i := right; i >= left; i -- { res = append(res, matrix[bottom]
[i])} // 下层  bottom 行 i 列
        bottom --
        for i := bottom; i >= top; i -- { res = append(res, matrix[i]
[left])}   // 左层  i 行 left 列
        left ++                         // 四个边界同时收缩，进入内层
    }
    return res
}
```

```
func spiralOrder(matrix [][]int) []int {
    if len(matrix) == 0 {
        return []int{}
    }
    res := []int{}
    top, right, bottom, left := 0, len(matrix[0])-1, len(matrix)-1, 0
    for top <= bottom && left <= right {    // 一条边从头遍历到底（包括最后一个
元素）
        for i := left; i <= right; i ++ { res = append(res, matrix[top]
[i])}    // 上层  top 行 i 列
        top ++
        for i := top; i <= bottom; i ++ { res = append(res, matrix[i]
[right])}  // 右层  i 行 right 列
        right --
        if top > bottom || left > right { break }
        for i := right; i >= left; i -- { res = append(res, matrix[bottom]
[i])} // 下层  bottom 行 i 列
        bottom --
        for i := bottom; i >= top; i -- { res = append(res, matrix[i]
[left])}   // 左层  i 行 left 列
        left ++     // 四个边界同时收缩，进入内层
    }
    return res
}
```

```
func spiralOrder(matrix [][]int) []int {
    if len(matrix) == 0 {
        return []int{}
    }
    res := []int{}
    top, right, bottom, left := 0, len(matrix[0])-1, len(matrix)-1, 0
    for top < bottom && left < right {  // 一条边不从头遍历到底（不包括最后一个元
素）
        for i := left; i < right; i ++ { res = append(res, matrix[top]
```

```
[i])}    // 上层 (top 行 i 列)
        for i := top; i < bottom; i ++ { res = append(res, matrix[i]
[right])}  // 右层 (i 行 right 列)
        for i := right; i > left; i -- { res = append(res, matrix[bottom]
[i])} // 下层 (bottom 行 i 列)
        for i := bottom; i > top; i -- { res = append(res, matrix[i]
[left])}   // 左层 (i 行 left 列)
        top ++      // 四个边界同时收缩，进入内层
        right --
        bottom --
        left ++
    }
    if top == bottom {
        for i := left; i <= right; i ++ { res = append(res, matrix[top]
[i])}  // 只剩一行，从左到右依次添加
    } else if left == right {
        for i := top; i <= bottom; i ++ { res = append(res, matrix[i]
[left])} // 只剩一列，从上到下依次添加
    }
    return res
}
```

23. 合并K个升序链表

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func mergeKLists(lists []*ListNode) *ListNode {
    n := len(lists)
    if n == 0 {
        return nil
    }
    if n == 1 {
        return lists[0]
    }
    mid := n >> 1
    left, right := mergeKLists(lists[:mid]), mergeKLists(lists[mid:])
    return mergeTwoList(left, right)
}

func mergeTwoList(l1, l2 *ListNode) *ListNode {
    if l1 == nil {
        return l2
    }
    if l2 == nil {
        return l1
    }
```

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

300. 最长递增子序列

方法一：动态规划

思路与算法

定义 dp[i] 为考虑前 i 个元素，以第 i 个数字结尾的最长上升子序列的长度，注意 nums[i] 必须被选取。

```
func lengthOfLIS(nums []int) int {
    if len(nums) == 0 {
        return 0
    }
    dp, res := make([]int, len(nums)), 0
    for i := 0; i < len(nums); i++ {
        dp[i] = 1
        for j := 0; j < i; j++ {
            if nums[j] < nums[i] {          // 贪心上升：以nums[i]结尾的上升子
序列
                dp[i] = max(dp[i], dp[j]+1) // 前 i 个数字结尾的最长上升子序列的
长度
            }
        }
        res = max(res, dp[i]) // 计算dp数组的最大值
    }
    return res
}

func max(x, y int) int {
    if x > y {
        return x
    }
    return y
}
```

方法二： nlogn 动态规划

```
func lengthOfLIS(nums []int) int {
    dp := []int{} // 维护单调递增数组 sorted
    for _, num := range nums {
        // 在递增顺序的数组dp中搜索num，返回num的索引。如果查找不到，返回值是num应该
```

```
插入dp的位置
        i := sort.SearchInts(dp, num)
        if i == len(dp) { // dp 中不存在 num
            dp = append(dp, num)
        } else {          // dp 中存在 num
            dp[i] = num   //更新
        }
    }
    return len(dp)
}
```

方法三：贪心 + 二分查找

考虑一个简单的贪心，如果我们要使上升子序列尽可能的长，则我们需要让序列上升得尽可能慢，因此我们希望每次在上升子序列最后加上的那个数尽可能的小。

最后整个算法流程为：

设当前已求出的最长上升子序列的长度为 len（初始时为 1），从前往后遍历数组 nums，在遍历到 nums[i] 时：

- 如果 nums[i]>d[len] ，则直接加入到 d 数组末尾，并更新 len=len+1；
- 否则，在 d 数组中二分查找，找到第一个比 nums[i] 小的数 d[k] ，并更新 d[k+1]=nums[i]。

以输入序列 [0,8,4,12,2] 为例：

第一步插入 0，d=[0]；

第二步插入 8，d=[0,8]；

第三步插入 4，d=[0,4]；

第四步插入 12，d=[0,4,12]；

第五步插入 2，d=[0,2,12]。

最终得到最大递增子序列长度为 3。

```
func lengthOfLIS(nums []int) int {
    dp := []int{} // 维护单调递增数组 sorted
    for _, num := range nums {
        if len(dp) == 0 || dp[len(dp)-1] < num { // dp 中不存在 num
            dp = append(dp, num)
        } else { // dp 中存在 num
            l, r, pos := 0, len(dp)-1, 0
            for l <= r { // 二分查找
                mid := l + (r-l)>>1
                if dp[mid] >= num { // 第一个大于 num 的数的下标
                    pos = mid
                    r = mid - 1
                } else {
```

```
                    l = mid + 1
                }
            }
            dp[pos] = num // 更新
        }
    }
    return len(dp)
}
```

160. 相交链表

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func getIntersectionNode(headA, headB *ListNode) *ListNode {
    A, B := headA, headB
    for A != B {
        if A != nil {
            A = A.Next
        } else { // 首次遍历到尾，以另一个链表的头为起点从头开始遍历
            A = headB
        }
        if B != nil {
            B = B.Next
        } else {
            B = headA
        }
    }
    return A // 直到链表相交 A == B，退出循环返回
}
```

415. 字符串相加

```
func addStrings(num1 string, num2 string) string {
    res, carry := "", 0
    for i, j := len(num1)-1, len(num2)-1; i >= 0 || j >= 0 || carry != 0;
i, j = i-1, j-1 {
        var x, y int
        if i >= 0 {
            x = int(num1[i] - '0')
        }
        if j >= 0 {
            y = int(num2[j] - '0')
        }
        temp := x + y + carry // 必须加进位
```

```
        res = strconv.Itoa(temp%10) + res
        carry = temp / 10
    }
    return res
}
```

143. 重排链表

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func reorderList(head *ListNode) {
    mid := middleNode(head)
    l1, l2 := head, mid.Next
    mid.Next = nil
    l2 = reverseList(l2)
    mergeList(l1, l2)
}

func middleNode(head *ListNode) *ListNode {
    slow, fast := head, head
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
    }
    return slow
}

func reverseList(head *ListNode) *ListNode {
    var prev *ListNode
    curr := head
    for curr != nil {
        next := curr.Next
        curr.Next = prev
        prev = curr
        curr = next
    }
    return prev
}

func mergeList(l1, l2 *ListNode) {
    var l1Tmp, l2Tmp *ListNode
    for l1 != nil && l2 != nil {
        l1Tmp, l2Tmp = l1.Next, l2.Next
        l1.Next = l2
        l1 = l1Tmp
        l2.Next = l1
```

```
            l2 = l2Tmp
        }
    }
```

42. 接雨水

```
func trap(height []int) int {
    left, right, res := 0, len(height)-1, 0
    leftMax, rightMax := 0, 0
    for left < right {
        leftMax = max(leftMax, height[left])
        rightMax = max(rightMax, height[right])
        if height[left] < height[right] { // 有低洼，能接到雨水
            res += leftMax - height[left] // 计算每个单位接到的雨水量，并累加
            left++
        } else {
            res += rightMax - height[right]
            right--
        }
    }
    return res
}
```

142. 环形链表 II

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func detectCycle(head *ListNode) *ListNode {
    slow, fast := head, head
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
        if slow == fast {
            fast = head
            for slow != fast {
                slow = slow.Next
                fast = fast.Next
            }
            return fast
        }
    }
    return nil
}
```

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func detectCycle(head *ListNode) *ListNode {
    slow, fast := head, head
    for fast != nil {
        slow = slow.Next
        if fast.Next == nil {
            return nil
        }
        fast = fast.Next.Next
        if slow == fast {
            p := head
            for p != slow {
                p = p.Next
                slow = slow.Next
            }
            return p
        }
    }
    return nil
}
```

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func detectCycle(head *ListNode) *ListNode {
    slow, fast := head, head
    for {
        if fast == nil || fast.Next == nil {
            return nil
        }
        fast = fast.Next.Next
        slow = slow.Next
        if slow == fast {
            break
        }
    }
    fast = head
    for slow != fast {
```

```
        slow = slow.Next
        fast = fast.Next
    }
    return fast
}
```

876. 链表的中间结点

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func middleNode(head *ListNode) *ListNode {
    slow, fast := head, head
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
    }
    return slow
}
```

56. 合并区间

```
func merge(intervals [][]int) (res [][]int) {
    n, k := len(intervals), 0
    startArr, endArr := make([]int, n), make([]int, n)
    for _, interval := range intervals {
        startArr[k], endArr[k] = interval[0], interval[1]
        k++
    }
    sort.Ints(startArr)
    sort.Ints(endArr)
    start := 0 // 全局起点
    for i := 0; i < n; i++ {
        if i == n-1 || startArr[i+1] > endArr[i] { // 无重叠：最后一个结点  ||
下一个结点大于当前终点
            res = append(res, []int{startArr[start], endArr[i]}) // 合并区
间：[全局起点，当前终点]
            start = i + 1                                        // 更新
global start
        }
    }
    return
}
```

124. 二叉树中的最大路径和

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func maxPathSum(root *TreeNode) int {
    maxSum := math.MinInt64
    var maxGain func(*TreeNode) int // 二叉树中的一个节点的最大贡献值

    maxGain = func(node *TreeNode) int {
        if node == nil {
            return 0
        }
        // 只有在最大贡献值大于 0 时，才会选取对应子节点
        left := max(maxGain(node.Left), 0)
        right := max(maxGain(node.Right), 0)
        // 节点的最大路径和取决于该节点的值与该节点的左右子节点的最大贡献值
        currPathSum := left + node.Val + right
        maxSum = max(maxSum, currPathSum)  // 更新答案
        return node.Val + max(left, right) // 返回节点的最大贡献值
    }

    maxGain(root)
    return maxSum
}
```

72. 编辑距离

```

```

19. 删除链表的倒数第N个节点

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func removeNthFromEnd(head *ListNode, n int) *ListNode {
    dummy := &ListNode{0, head}
    slow, fast := dummy, head
```

```
    for i := 0; fast != nil; i++ {
        if i >= n {
            slow = slow.Next
        }
        fast = fast.Next
    }
    slow.Next = slow.Next.Next
    return dummy.Next
}

func removeNthFromEnd1(head *ListNode, n int) *ListNode {
    dummy := &ListNode{0, head}
    slow, fast := dummy, head
    for fast != nil {
        if n > 0 {
            fast = fast.Next
            n--
        } else {
            slow = slow.Next
            fast = fast.Next
        }
    }
    slow.Next = slow.Next.Next
    return dummy.Next
}
```

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func removeNthFromEnd(head *ListNode, n int) *ListNode {
    dummy := &ListNode{0, head}
    slow, fast := dummy, head
    for ; n > 0; n-- {
        fast = fast.Next // 1. 快指针先走n步
    }
    for ; fast != nil; fast = fast.Next {
        slow = slow.Next // 2. 快慢指针同时走到头，找到要删除节点的前一个节点
    }
    slow.Next = slow.Next.Next // 3. 删除
    return dummy.Next
}
```

93. 复原 IP 地址

```

```

```go
func restoreIpAddresses(s string) []string {
    res := []string{}
    var dfs func([]string, int)

    dfs = func(sub []string, start int) {
        if len(sub) == 4 && start == len(s) { // 片段满4段，且耗尽所有字符
            res = append(res, strings.Join(sub, ".")) // 拼成字符串，加入解集
            return
        }
        if len(sub) == 4 && start < len(s) { // 满4段，字符未耗尽，不用往下选了
            return
        }
        for length := 1; length <= 3; length++ { // 枚举出选择，三种切割长度
            if start+length-1 >= len(s) { // 加上要切的长度就越界，不能切这个长度
                return
            }
            if length != 1 && s[start] == '0' { // 不能切出'0x'、'0xx'
                return
            }
            str := s[start : start+length]          // 当前选择切出的片段
            if n, _ := strconv.Atoi(str); n > 255 { // 不能超过255
                return
            }
            sub = append(sub, str) // 作出选择，将片段加入sub
            dfs(sub, start+length) // 基于当前选择，继续选择，注意更新指针
            sub = sub[:len(sub)-1] // 上面一句的递归分支结束，撤销最后的选择，进入下一轮迭代，考察下一个切割长度
        }
    }
    dfs([]string{}, 0)
    return res
}
```

1143. 最长公共子序列

设字符串 text1 和 text2 的长度分别为 m 和 n，创建 m+1 行 n+1 列的二维数组 dp， 其中 dp[i][j] 表示 text1 [0:i] 和 text2 [0:j] 的最长公共子序列的长度。

```go
func longestCommonSubsequence(text1 string, text2 string) int {
    m, n := len(text1), len(text2)
    dp := make([][]int, m+1)
    for i := range dp { // 创建 m+1 行 n+1 列的二维数组 dp
        dp[i] = make([]int, n+1)
    }
    for i, c1 := range text1 {
        for j, c2 := range text2 {
            if c1 == c2 { // 如果存在公共子序列
                dp[i+1][j+1] = dp[i][j] + 1
            } else {
                dp[i+1][j+1] = max(dp[i+1][j], dp[i][j+1])
```

```
            }
        }
    }
    return dp[m][n]
}
```

94. 二叉树的中序遍历

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func inorderTraversal(root *TreeNode) (res []int) {
    var inorder func(*TreeNode)
    inorder = func(node *TreeNode) {
        if node == nil {
            return
        }
        inorder(node.Left)
        res = append(res, node.Val)
        inorder(node.Right)
    }
    inorder(root)
    return
}
```

82. 删除排序链表中的重复元素 II

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func deleteDuplicates(head *ListNode) *ListNode {
    dummy := &ListNode{0, head}
    prev := dummy
    for prev.Next != nil && prev.Next.Next != nil { // 至少有2个节点
        if prev.Next.Val == prev.Next.Next.Val { // 如果有重复数字的节点
            x := prev.Next.Val // 记录重复元素
            for prev.Next != nil && x == prev.Next.Val {
                prev.Next = prev.Next.Next // 删除重复元素
            }
```

```
        } else {
            prev = prev.Next // 向后扫描
        }
    }
    return dummy.Next
}
```

704. 二分查找

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

```
func binarySearchLeft(nums []int, target int) int {
    low, high := 0, len(nums)-1
    for low < high {
        mid := low + (high-low)>>1
        if nums[mid] > target {
            high = mid // 答案在 mid 左侧
        } else {
            low = mid + 1
        }
    }
    return nums[low] // low == high
}

func binarySearchRight(nums []int, target int) int {
    low, high := 0, len(nums)-1
    for low < high {
        mid := low + (high-low+1)>>1
        if nums[mid] <= target {
            low = mid // 答案在 mid 右侧
        } else {
            high = mid - 1
        }
    }
    return nums[low] // low == high
```

```
}

func search(nums []int, target int) int {
    if nums[0] == target {
        return 0
    }
    left, right := 0, len(nums)-1
    for left < right {
        mid := left + (right-left+1)>>1
        if nums[mid] == target {
            return mid
        }
        if nums[mid] <= target {
            left = mid
        } else {
            right = mid - 1
        }
    }
    return -1
}
func search1(nums []int, target int) int {
    left, right := 0, len(nums)
    for left < right {
        mid := left + (right-left)>>1
        if nums[mid] == target {
            return mid
        } else if nums[mid] < target {
            left = mid + 1
        } else {
            right = mid
        }
    }
    return -1
}
```

199. 二叉树的右视图

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func rightSideView(root *TreeNode) []int {
    res := []int{}
    var dfs func(*TreeNode, int)
    dfs = func(node *TreeNode, depth int) {
        if node == nil {
            return
```

```
        }
        if depth == len(res) {
            res = append(res, node.Val)
        }
        dfs(node.Right, depth+1) // 递归栈操作，先进后出
        dfs(node.Left, depth+1)
    }
    dfs(root, 0)
    return res
}
```

31. 下一个排列

思路

1. 我们需要将一个左边的「较小数」与一个右边的「较大数」交换，以能够让当前排列变大，从而得到下一个排列。
2. 同时我们要让这个「较小数」尽量靠右，而「较大数」尽可能小。当交换完成后，「较大数」右边的数需要按照升序重新排列。这样可以在保证新排列大于原来排列的情况下，使变大的幅度尽可能小。

- 从低位挑一个大一点的数，换掉前面的小一点的一个数，实现变大。
- 变大的幅度要尽量小。

像 [3,2,1] 递减的，没有下一个排列，因为大的已经尽量往前排了，没法更大。

```
func nextPermutation(nums []int) {
    i := len(nums) - 2                    // 从右向左遍历，i从倒数第二开始是为了nums[i+1]要存在
    for i >= 0 && nums[i] >= nums[i+1] { // 寻找第一个小于右邻居的数
        i--
    }
    if i >= 0 { // 这个数在数组中存在，从它身后挑一个数，和它换
        j := len(nums) - 1                  // 从最后一项，向左遍历
        for j >= 0 && nums[j] <= nums[i] { // 寻找第一个大于 nums[i] 的数
            j--
        }
        nums[i], nums[j] = nums[j], nums[i] // 两数交换，实现变大
    }
    // 如果 i = -1，说明是递减排列，如 3 2 1，没有下一排列，直接翻转为最小排列：1 2 3
    l, r := i+1, len(nums)-1
    for l < r { // i 右边的数进行翻转，使得变大的幅度小一些
        nums[l], nums[r] = nums[r], nums[l]
        l++
        r--
    }
}
```

4. 寻找两个正序数组的中位数

for example，a=[1 2 3 4 6 9]and, b=[1 1 5 6 9 10 11]，total numbers are 13， you should find the seventh number , int(7/2)=3, a[3]<b[3], so you don't need to consider a[0],a[1],a[2] because they can't be the seventh number. Then find the fourth number in the others numbers which don't include a[0]a[1]a[2]. just like this , decrease half of numbers every time .....

```
func findMedianSortedArrays(nums1 []int, nums2 []int) float64 {
    if l := len(nums1) + len(nums2); l%2 == 0 {
        return (findKth(nums1, nums2, l/2-1) + findKth(nums1, nums2, l/2))
/ 2.0
    } else {
        return findKth(nums1, nums2, l/2)
    }
}
func findKth(nums1, nums2 []int, k int) float64 {
    for {
        l1, l2 := len(nums1), len(nums2)
        m1, m2 := l1/2, l2/2
        if l1 == 0 {
            return float64(nums2[k])
        } else if l2 == 0 {
            return float64(nums1[k])
        } else if k == 0 {
            if n1, n2 := nums1[0], nums2[0]; n1 <= n2 {
                return float64(n1)
            } else {
                return float64(n2)
            }
        }
        if k <= m1+m2 {
            if nums1[m1] <= nums2[m2] {
                nums2 = nums2[:m2]
            } else {
                nums1 = nums1[:m1]
            }
        } else {
            if nums1[m1] <= nums2[m2] {
                nums1 = nums1[m1+1:]
                k -= m1 + 1
            } else {
                nums2 = nums2[m2+1:]
            }
        }
    }
}
```

复杂度分析

- 时间复杂度：O(log(m+n))，其中 m 和 n 分别是数组 nums1 和 nums2 的长度。初始时有 k=(m+n)/2 或 k=(m+n)/2+1，每一轮循环可以将查找范围减少一半，因此时间复杂度是 O(log(m+n))。

- 空间复杂度：O(1)。

```
func findMedianSortedArrays(nums1 []int, nums2 []int) float64 {
    l1, l2 := len(nums1), len(nums2)
    if l1 > l2 {
        return findMedianSortedArrays(nums2, nums1)
    }
    for start, end := 0, l1; ; {
        nums1Med := (start + end) / 2
        nums2Med := (l2+l1+1)/2 - nums1Med
        nums1Left, nums1Right, nums2Left, nums2Right := math.MinInt64,
            math.MaxInt64, math.MinInt64, math.MaxInt64
        if nums1Med != 0 {
            nums1Left = nums1[nums1Med-1]
        }
        if nums1Med != l1 {
            nums1Right = nums1[nums1Med]
        }
        if nums2Med != 0 {
            nums2Left = nums2[nums2Med-1]
        }
        if nums2Med != l2 {
            nums2Right = nums2[nums2Med]
        }
        if nums1Left > nums2Right {
            end = nums1Med - 1
        } else if nums2Left > nums1Right {
            start = nums1Med + 1
        } else {
            if (l1+l2)%2 == 1 {
                return math.Max(float64(nums1Left), float64(nums2Left))
            }
            return (math.Max(float64(nums1Left), float64(nums2Left)) +
                math.Min(float64(nums1Right), float64(nums2Right))) / 2
        }
    }
}
```

复杂度分析

- 时间复杂度：O(logmin(m,n)))，其中 m 和 n 分别是数组 nums1 和 nums2 的长度。查找的区间是 [0,m]，而该区间的长度在每次循环之后都会减少为原来的一半。所以，只需要执行 logm 次循环。由于每次循环中的操作次数是常数，所以时间复杂度为 O(logm)。由于我们可能需要交换 nums1 和 nums2 使得 m≤n，因此时间复杂度是 O(log -min(m,n)))。

- 空间复杂度：O(1)。

232. 用栈实现队列

```go
type MyQueue struct {
    inStack  []int
    outStack []int
}

func Constructor() MyQueue {
    return MyQueue{}
}

func (this *MyQueue) Push(x int) {
    this.inStack = append(this.inStack, x)
}

func (this *MyQueue) in2out() {
    for len(this.inStack) > 0 {
        this.outStack = append(this.outStack,
this.inStack[len(this.inStack)-1])
        this.inStack = this.inStack[:len(this.inStack)-1]
    }
}

func (this *MyQueue) Pop() int {
    if len(this.outStack) == 0 {
        this.in2out()
    }
    x := this.outStack[len(this.outStack)-1]
    this.outStack = this.outStack[:len(this.outStack)-1]
    return x
}

func (this *MyQueue) Peek() int {
    if len(this.outStack) == 0 {
        this.in2out()
    }
    return this.outStack[len(this.outStack)-1]
}

func (this *MyQueue) Empty() bool {
    return len(this.inStack) == 0 && len(this.outStack) == 0
}

/**
 * Your MyQueue object will be instantiated and called as such:
 * obj := Constructor();
 * obj.Push(x);
 * param_2 := obj.Pop();
 * param_3 := obj.Peek();
 * param_4 := obj.Empty();
 */
```

148. 排序链表

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func sortList(head *ListNode) *ListNode {
    if head == nil || head.Next == nil {
        return head
    }
    slow, fast := head, head.Next
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
    }
    mid := slow.Next
    slow.Next = nil
    left := sortList(head)
    right := sortList(mid)
    return mergeList(left, right)
}

func mergeList(l1, l2 *ListNode) *ListNode {
    dummy := &ListNode{}
    prev := dummy
    for l1 != nil && l2 != nil {
        if l1.Val < l2.Val {
            prev.Next = l1
            l1 = l1.Next
        } else {
            prev.Next = l2
            l2 = l2.Next
        }
        prev = prev.Next
    }
    if l1 == nil {
        prev.Next = l2
    }
    if l2 == nil {
        prev.Next = l1
    }
    return dummy.Next
}
```

69. x 的平方根

```go
func mySqrt(x int) int {
    left, right := 0, x
    res := -1
```

```
    for left <= right {
        mid := (left + right) >> 1
        if mid*mid <= x {
            res = mid
            left = mid + 1
        } else {
            right = mid - 1
        }
    }
    return res
}
```

8. 字符串转换整数 (atoi)

```
func myAtoi(s string) int {
    abs, sign, i, n := 0, 1, 0, len(s)
    //丢弃无用的前导空格
    for i < n && s[i] == ' ' {
        i++
    }
    //标记正负号
    if i < n {
        if s[i] == '-' {
            sign = -1
            i++
        } else if s[i] == '+' {
            sign = 1
            i++
        }
    }
    for i < n && s[i] >= '0' && s[i] <= '9' {
        abs = 10*abs + int(s[i]-'0')  //字节 byte '0' == 48
        if sign*abs < math.MinInt32 { //整数超过 32 位有符号整数范围
            return math.MinInt32
        } else if sign*abs > math.MaxInt32 {
            return math.MaxInt32
        }
        i++
    }
    return sign * abs
}
```

22. 括号生成

```
func generateParenthesis(n int) []string {
    res := []string{}
    var dfs func(int, int, string)

    dfs = func(lRemain, rRemain int, str string) { // 左右括号所剩的数量, str
```

```
是当前构建的字符串
        if 2*n == len(str) { // 字符串构建完成
            res = append(res, str) // 加入解集
            return                 // 结束当前递归分支
        }
        if 0 < lRemain { // 只要左括号有剩，就可以选它，然后继续做选择（递归）
            dfs(lRemain-1, rRemain, str+"(")
        }
        if lRemain < rRemain { // 右括号比左括号剩的多，才能选右括号
            dfs(lRemain, rRemain-1, str+")") // 然后继续做选择（递归）
        }
    }

    dfs(n, n, "") // 递归的入口，剩余数量都是n，初始字符串是空串
    return res
}
```

参考链接

70. 爬楼梯

```
func climbStairs0(n int) int {
    prev, curr := 1, 1
    for ; n-1 > 0; n-- {
        sum := prev + curr
        prev = curr
        curr = sum
    }
    return curr
}

func climbStairs(n int) int {
    prev, curr := 1, 1
    for ; n-1 > 0; n-- {
        prev, curr = curr, prev+curr
    }
    return curr
}
```

```
func climbStairs(n int) int {
    prev, curr := 1, 1
    for ; n > 0; n-- {
        next := prev + curr
        prev = curr
        curr = next
    }
    return prev
}
```

```go
func climbStairs1(n int) int {
    prev, curr := 1, 1
    for ; n > 0; n-- {
        prev, curr = curr, prev+curr
    }
    return prev
}
```

```go
// Time Limit Exceeded
func climbStairs(n int) int {
    if n <= 2 {
        return n
    }
    return climbStairs(n-1) + climbStairs(n-2)
}
```

方法一：动态规划/滚动数组 (斐波那契数列)

思路和算法

我们用 f(x) 表示爬到第 x 级台阶的方案数，考虑最后一步可能跨了一级台阶，也可能跨了两级台阶，所以我们可以列出如下式子：

f(x) = f(x - 1) + f(x - 2)

f(x) 只和 f(x−1) 与 f(x−2) 有关，所以我们可以用「滚动数组思想」把空间复杂度优化成 O(1)。

它意味着爬到第 x 级台阶的方案数是爬到第 x−1 级台阶的方案数和爬到第 x−2 级台阶的方案数的和。 很好理解，因为每次只能爬 1 级或 2 级，所以 f(x) 只能从 f(x−1) 和 f(x−2) 转移过来，而这里要统计方案总数，我们就需要对这两项的贡献求和。

以上是动态规划的转移方程，下面我们来讨论边界条件。

- 我们是从第 0 级开始爬的，所以从第 0 级爬到第 0 级我们可以看作只有一种方案，即 f(0)=1；
- 从第 0 级到第 1 级也只有一种方案，即爬一级，f(1)=1。

这两个作为边界条件就可以继续向后推导出第 n 级的正确结果。

```go
func climbStairs(n int) int {
    p, q, r := 0, 0, 1
    for i := 1; i <= n; i++ {
        p = q
        q = r
        r = p + q
    }
    return r
}
```

```
func climbStairs(n int) int {
    p, q, r := 0, 1, 1
    for i := 2; i <= n; i++ {
        p = q
        q = r
        r = p + q
    }
    return r
}
```

 参考LeetCode官方图解

方法二：动态规划

解题思路

- 简单的 DP，经典的爬楼梯问题。一个楼梯可以由 n-1 和 n-2 的楼梯爬上来。
- 这一题求解的值就是斐波那契数列。

```
func climbStairs(n int) int {
    dp := make([]int, n+1)
    dp[0], dp[1] = 1, 1
    for i := 2; i <= n; i++ {
        dp[i] = dp[i-1] + dp[i-2]
    }
    return dp[n]
}
```

```
func climbStairs(n int) int {
    dp := make([]int, n+1)
    dp[0] = 1 // 从第0级到第0级可以看作有1种方案
    dp[1] = 1 // 从第0级到第1级有1种方案
    for i := 2; i <= n; i++ {
        // 爬到第 i 阶楼梯的方案数 = 爬到第 i-1 阶的方案数 + 爬到第 i-2 阶的方案数
        dp[i] = dp[i-1] + dp[i-2]
    }
    return dp[n]
}
```

压缩空间，优化

dp[i] 只与过去的两项：dp[i-1] 和 dp[i-2] 有关，没有必要存下所有计算过的 dp 项。用两个变量去存这两个过去的状态就好。

```
func climbStairs(n int) int {
    prev, curr := 1, 1
```

```
    for i := 2; i <= n; i++ {
        next := curr
        curr += prev
        prev = next
    }
    return curr
}
```

2. 两数相加

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func addTwoNumbers(l1 *ListNode, l2 *ListNode) *ListNode {
    dummy := new(ListNode)
    curr, carry := dummy, 0
    for l1 != nil || l2 != nil || carry != 0 {
        curr.Next = new(ListNode) // 新建节点
        curr = curr.Next
        if l1 != nil { // 如果指针不为空
            carry += l1.Val // 将指针向的值加入 carry
            l1 = l1.Next    // 向后扫描
        }
        if l2 != nil {
            carry += l2.Val
            l2 = l2.Next
        }
        curr.Val = carry % 10 // 取个位
        carry /= 10           // 取十位
    }
    return dummy.Next
}
```

165. 比较版本号

```
func compareVersion(version1 string, version2 string) int {
    i, j, m, n := 0, 0, len(version1), len(version2)
    for i < m || j < n {
        x, y := 0, 0
        for i < m && version1[i] != '.' {
            x = 10*x + int(version1[i]-'0')
            i++
        }
        i++ // 跳过 "."
        for j < n && version2[j] != '.' {
```

```
            y = 10*y + int(version2[j]-'0') // 字符转整数
            j++
        }
        j++ // 跳过 "."
        if x < y {
            return -1
        }
        if x > y {
            return 1
        }
    }
    return 0
}
```

```
func compareVersion(version1 string, version2 string) int {
    v1 := strings.Split(version1, ".")            // [1 01]
    v2 := strings.Split(version2, ".")            // [1 001]
    for i := 0; i < len(v1) || i < len(v2); i++ { // x == y 跳过，扫描下一位
        x, y := 0, 0
        if i < len(v1) {
            x, _ = strconv.Atoi(v1[i]) // 字符转整数
        }
        if i < len(v2) {
            y, _ = strconv.Atoi(v2[i])
        }
        if x < y {
            return -1
        }
        if x > y {
            return 1
        }
    }
    return 0
}
```

239. 滑动窗口最大值

方法一 暴力解法 O(nk)

```
func maxSlidingWindow(nums []int, k int) []int {
    res, n := make([]int, 0, k), len(nums)
    if n == 0 {
        return make([]int, 0)
    }
    for i := 0; i <= n-k; i++ {
        max := nums[i]
        for j := 1; j < k; j++ {
            if max < nums[i+j] {
                max = nums[i+j]
```

```
            }
        }
        res = append(res, max)
    }
    return res
}
```

Time Limit Exceeded 50/61 cases passed (N/A)

方法二 双端队列 Deque

最优的解法是用双端队列，队列的一边永远都存的是窗口的最大值，队列的另外一个边存的是比最大值小的值。队列中最大值左边的所有值都出队。在保证了双端队列的一边即是最大值以后，

- 时间复杂度是 O(n)
- 空间复杂度是 O(K)

```
func maxSlidingWindow(nums []int, k int) (res []int) {
    q := []int{} // 动态维护单调递减队列，存储 nums 的索引
    for i, v := range nums {
        if i >= k && q[0] <= i-k { // 队满
            q = q[1:] // 删除队头
        }
        for len(q) > 0 && nums[q[len(q)-1]] <= v { // 队尾元素小于等于当前元素
            q = q[:len(q)-1] // 删除队尾
        }
        q = append(q, i) // 存储当前索引
        if i >= k-1 {    // 首次队满
            res = append(res, nums[q[0]]) // 队头存储 nums 的最大值的索引
        }
    }
    return
}
```

```
func maxSlidingWindow(nums []int, k int) []int {
    q, res := []int{}, []int{}
    for i := 0; i < len(nums); i++ {
        if len(q) > 0 && i-k+1 > q[0] {
            q = q[1:] //窗口满了，删除队头
        }
        for len(q) > 0 && nums[q[len(q)-1]] <= nums[i] {
            q = q[:len(q)-1] //队尾小于当前元素，删除队尾
        }
        q = append(q, i)
        if i >= k-1 { //窗口大小大于等于 k
            res = append(res, nums[q[0]])
        }
```

```
    }
    return res
}
```

41. 缺失的第一个正数

```
func firstMissingPositive(nums []int) int {
    n := len(nums)
    for i := 0; i < n; i++ {
        for nums[i] > 0 && nums[i] <= n && nums[nums[i]-1] != nums[i] {
            nums[nums[i]-1], nums[i] = nums[i], nums[nums[i]-1] // 将
nums[i]-1 放在 nums[i]
        }
    }
    for i := 0; i < n; i++ {
        if nums[i] != i+1 { // 如果 i 位置的数不是 i+1
            return i + 1
        }
    }
    return n + 1
}
```

```
func firstMissingPositive(nums []int) int {
    hash := make(map[int]int, len(nums))
    for _, v := range nums {
        hash[v] = v
    }
    for i := 1; i < len(nums)+1; i++ {
        if _, ok := hash[i]; !ok {
            return i
        }
    }
    return len(nums) + 1
}
```

剑指 Offer 22. 链表中倒数第k个节点

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func trainingPlan1(head *ListNode, cnt int) *ListNode {
    slow, fast := head, head
```

```
    for fast != nil {
        if cnt >0 {
            fast = fast.Next
            cnt--
        } else {
            slow = slow.Next
            fast = fast.Next
        }
    }
    return slow
}

func trainingPlan(head *ListNode, cnt int) *ListNode {
    slow, fast := head, head
    for i := 0; fast != nil; i++ {
        if i>= cnt{
            slow = slow.Next
        }
        fast = fast.Next
    }
    return slow
}
```

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func getKthFromEnd(head *ListNode, k int) *ListNode {
    slow, fast := head, head
    for i := 0; fast != nil; i++ {
        if i >= k {
            slow = slow.Next
        }
        fast = fast.Next
    }
    return slow
}
```

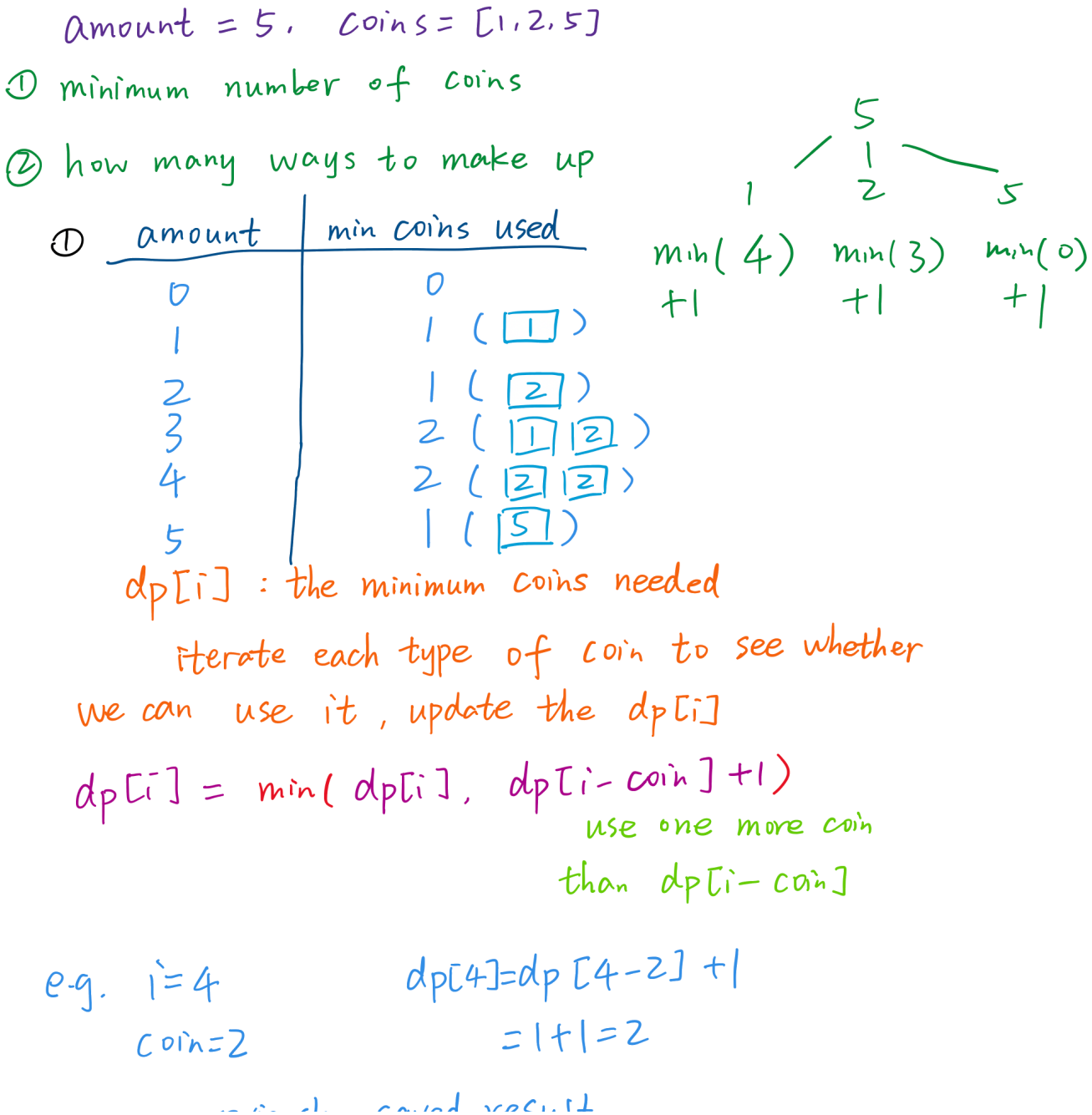
```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func getKthFromEnd(head *ListNode, k int) *ListNode {
```

```
    slow, fast := head, head
    for fast != nil {
        if k > 0 {
            fast = fast.Next
            k--
        } else {
            slow = slow.Next
            fast = fast.Next
        }
    }
    return slow
}
```

322. 零钱兑换

Coin Change

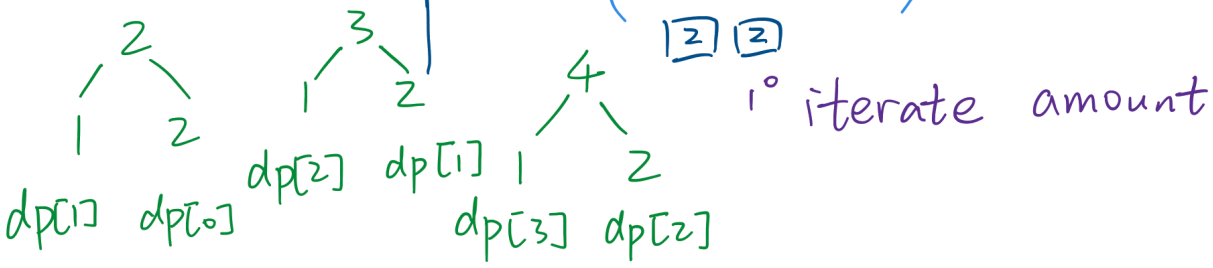
2021年1月25日星期一 19:25



use previously saved result

②

amount	ways
1	1 (1)
2	2 (2 / 1 1)
3	2 (1 2 / 1 1 1)
4	3 (1 1 1 1 / 1 1 2 / 2 2)



1° iterate amount

2° iterate coins

coin	amount	dp[0]=1
1	dp[1] = 1 dp[2] += dp[1] = 1	dp[3] += dp[2] = 1 dp[4] += dp[3] = 1 dp[5] += dp[4] = 1
2	dp[1] dp[2] += dp[0] = 2	dp[3] += dp[1] = 2 dp[4] += dp[2] = 3 dp[5] += dp[3] = 3
5	dp[1] dp[2] dp[3] dp[4] dp[5] += dp[0] = 4	

322. 零钱兑换 minmum number of coins

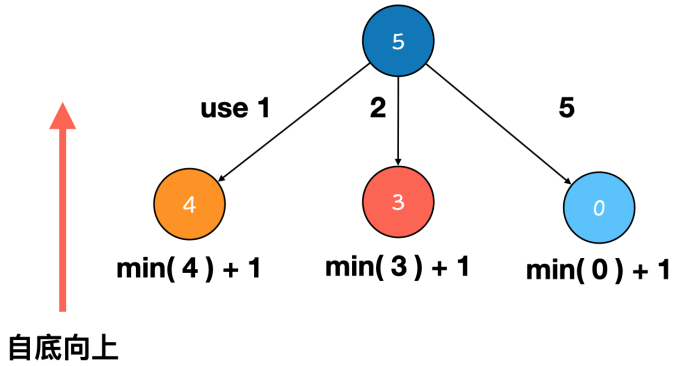
amount = 5 coins = [1,2,5]

凑出5等价于使用1、2、5硬币后，凑出4、3、0取最小值加 1

iterate amount

amount	min coins used
0	0
1	1 (1)
2	1 (2)
3	2 (1, 2)
4	2 (2, 2)
5	1 (5)

dp[i]：the min coins needed

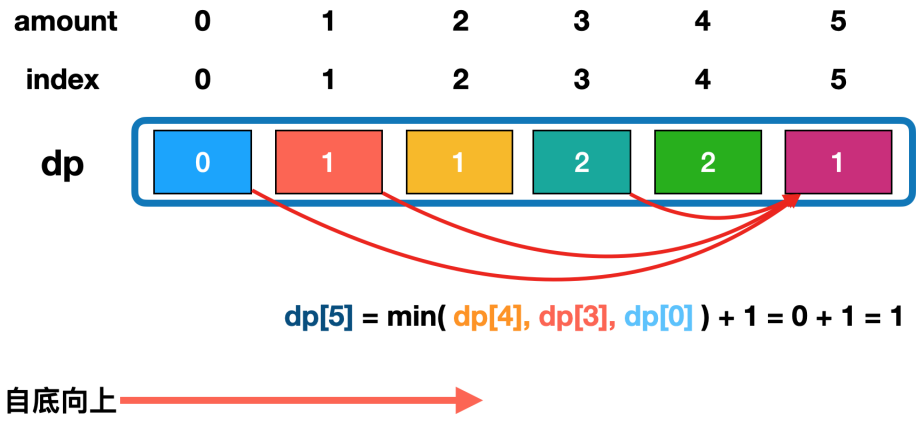


dp[i] = min(dp[i], dp[i-coin] + 1)

dp[5] = min(dp[4], dp[3], dp[0]) + 1 = 0 + 1 = 1

amount = 5 coins = [1,2,5]

dp[i]：the min coins needed



iterate amount

```
func coinChange(coins []int, amount int) int {
    dp := make([]int, amount+1) // 需要的最小硬币数量
    dp[0] = 0                   // 无法组成0的硬币
    for i := 1; i <= amount; i++ {
        dp[i] = amount + 1 //初始化需要最小硬币数量为最大值（不可能取到）
    }
    for i := 1; i <= amount; i++ { // 自底向上，遍历所有状态
        for _, coin := range coins { //求所有选择的最小值
min(dp[4],dp[3],dp[0])+1
```

```
            if i-coin >= 0 {
                dp[i] = min(dp[i], dp[i-coin]+1)
            }
        }
    }
    if amount < dp[amount] { // 不可能兑换成比自己更大的硬币
        return -1
    }
    return dp[amount]
}
```

518. 零钱兑换 II

518. 零钱兑换 II

amount = 5 coins = [1,2,5]

iterate coins

coins	amount dp[0] = 1 dp[i]：面额为i有多少种兑换方法
1	dp[1] = 1 dp[3] += dp[2] = 1 dp[4] += dp[3] = 1 dp[2]=dp[2]+dp[2-1]=0+1=1 dp[5] += dp[4] = 1
2	dp[1] = 1 dp[3]=dp[3]+dp[1]=1+1=2 dp[4]=dp[4]+dp[2]=1+2=3 dp[2]=dp[2]+dp[2-2]=1+1=2 dp[5]=dp[5]+dp[3] = 1+2=3
5	dp[1] dp[3] dp[2] dp[4] dp[5]=dp[5]+dp[0] = 3+1=4

iterate coins

```
func change(amount int, coins []int) int {
    dp := make([]int, amount+1) // dp[x] 表示金额之和等于 xx 的硬币组合数
    dp[0] = 1                   // 当不选取任何硬币时，金额之和才为 0，只有 1 种硬
币组合
    for _, coin := range coins {
        for i := coin; i <= amount; i++ {
            // 如果存在一种硬币组合的金额之和等于 i - coin，则在该硬币组合中增加一个
面额为 coin 的硬币，
            dp[i] += dp[i-coin] // 即可得到一种金额之和等于 i 的硬币组合。
        }
    }
    return dp[amount]
}
```

参考视频

代码

76. 最小覆盖子串

```
func minWindow(s string, t string) string {
    need := make(map[byte]int)
    for i := range t {
        need[t[i]]++
    }
    start, end, count, i := 0, -1, len(t), 0
    for j := 0; j < len(s); j++ {
        if need[s[j]] > 0 { //如果t中存在字符 s[j]，减少计数器
            count--
        }
        need[s[j]]--    //减少s[j]，如果字符s[j]在t中不存在，need[s[j]]置为负数
        if count == 0 { //找到有效的窗口后，开始移动以查找较小的窗口
            for i < j && need[s[i]] < 0 { //指针未越界且 字符s[i]在t中不存在
                need[s[i]]++ //移除t中不存在的字符 s[i]
                i++          // 左移窗口
            }
            if end == -1 || j-i < end-start {
                start, end = i, j
            }
            need[s[i]]++ //移除t中存在的字符 s[i]
            count++
            i++ //缩小窗口
        }
    }
    if end < start {
        return ""
    }
    return s[start : end+1]
}
```

78. 子集

```
func subsets(nums []int) (res [][]int) {
    n := len(nums)
    for mask := 0; mask < 1<<n; mask++ { // 000 -> 111  0 -> 2^3-1
        set := []int{}
        for i, v := range nums {
            if mask>>i&1 > 0 { // 如果 mask 第i位是1
                set = append(set, v) // 选取第i位下标指向的数
            }
        }
        res = append(res, set)
    }
```

```
    return
}
```

参考

```
func subsets(nums []int) [][]int {
    res, set := [][]int{}, []int{}
    var dfs func(int)

    dfs = func(i int) {
        res = append(res, append([]int(nil), set...)) // 调用子递归前，加入解集
        for j := i; j < len(nums); j++ {               // 枚举出所有可选的数
            set = append(set, nums[j]) // 选这个数
            dfs(j + 1)                 // 基于选这个数，继续递归，传入的j+1，不是i+1
            set = set[:len(set)-1]     // 撤销选这个数
        }
    }

    dfs(0)
    return res
}
```

105. 从前序与中序遍历序列构造二叉树

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func buildTree(preorder []int, inorder []int) *TreeNode {
    if len(preorder) == 0 {
        return nil
    }
    root := &TreeNode{Val: preorder[0]}
    for pos, node_val := range inorder {
        if node_val == root.Val {
            root.Left = buildTree(preorder[1:pos+1], inorder[:pos])   // 左子树的前序和中序遍历结果
            root.Right = buildTree(preorder[pos+1:], inorder[pos+1:]) // 右子树的前序和中序遍历结果
        }
    }
    return root
}
```

43. 字符串相乘

```
func multiply(num1 string, num2 string) string {
    if num1 == "0" || num2 == "0" {
        return "0"
    }
    b1, b2, tmp := []byte(num1), []byte(num2), make([]int,
len(num1)+len(num2))
    for i := len(b1) - 1; i >= 0; i-- {
        for j := len(b2) - 1; j >= 0; j-- {
            tmp[i+j+1] += int(b1[i]-'0') * int(b2[j]-'0')
        }
    }
    for i := len(tmp) - 1; i > 0; i-- {
        tmp[i-1] += tmp[i] / 10 // 进位
        tmp[i] %= 10            // 存储个位
    }
    if tmp[0] == 0 {
        tmp = tmp[1:] // 去除前导0
    }
    res := make([]byte, len(tmp))
    for i := 0; i < len(tmp); i++ {
        res[i] = '0' + byte(tmp[i]) // 整数转字节
    }
    return string(res) // 字节强转为字符串
}
```

思路清晰，效率低

```
func multiply(num1 string, num2 string) string {
    if num1 == "0" || num2 == "0" {
        return "0"
    }
    tmp := make([]int, len(num1)+len(num2))
    for i := len(num1) - 1; i >= 0; i-- {
        for j := len(num2) - 1; j >= 0; j-- {
            sum := int(num1[i]-'0')*int(num2[j]-'0') + tmp[i+j+1]
            tmp[i+j+1] = sum % 10 // 进位
            tmp[i+j] += sum / 10  // 个位
        }
    }
    res := ""
    for i, v := range tmp {
        if i == 0 && v == 0 {
            continue
        }
        res += string(v + '0') // 字符串拼接：2次内存拷贝（不是零时）
    }
```

```
    return res
}
```

32. 最长有效括号

```
func longestValidParentheses(s string) int {
    left, right, maxLength := 0, 0, 0
    for i := 0; i < len(s); i++ { // 从左向右遍历
        if s[i] == '(' {
            left++
        } else {
            right++
        }
        if left == right {
            maxLength = max(maxLength, 2*left)
        } else if left < right {
            left, right = 0, 0
        }
    }
    left, right = 0, 0                  // 重置
    for i := len(s) - 1; i >= 0; i-- { // 从右向左遍历
        if s[i] == '(' {
            left++
        } else {
            right++
        }
        if left == right {
            maxLength = max(maxLength, 2*left)
        } else if right < left {
            left, right = 0, 0
        }
    }
    return maxLength
}
```

```
func longestValidParentheses(s string) int {
    stack := []int{-1} // 为了相减后直接得到结果凑的，例如：1-(-1)=2
    res := 0
    for i := 0; i < len(s); i++ {
        if s[i] == '(' { // 如果是左括号，入栈
            stack = append(stack, i)
        } else { // 如果是右括号，出栈
            stack = stack[:len(stack)-1]
            if len(stack) == 0 { // 如果栈匹配后为空，继续入栈标记起点
                stack = append(stack, i)
            } else { // 栈不为空，最大长度等于索引的差值取最大值
                res = max(res, i-stack[len(stack)-1])
            }
        }
```

```
    }
    return res
}
```

155. 最小栈

```
type MinStack struct {
    stack    []int
    minStack []int
}

func Constructor() MinStack {
    return MinStack{
        stack:    []int{},
        minStack: []int{math.MaxInt64},
    }
}

func (this *MinStack) Push(val int) {
    this.stack = append(this.stack, val)
    minStackTop := this.minStack[len(this.minStack)-1]
    this.minStack = append(this.minStack, min(minStackTop, val))
}

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

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

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

func min(x, y int) int {
    if x < y {
        return x
    }
    return y
}

/**
 * Your MinStack object will be instantiated and called as such:
 * obj := Constructor();
 * obj.Push(val);
 * obj.Pop();
 * param_3 := obj.Top();
```

```
 * param_4 := obj.GetMin();
 */
```

151. 翻转字符串里的单词

```
func reverseWords(s string) string {
    slice := strings.Fields(s)           // ["the", "sky", "is", "blue"]
    var reverse func([]string, int, int) // ["blue", "is", "sky", "the"]

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

    reverse(slice, 0, len(slice)-1)
    return strings.Join(slice, " ") // "blue is sky the"
}
```

```
func reverseWords(s string) string {
    str := strings.Fields(s)      // ["the", "sky", "is", "blue"]
    reverse(&str, 0, len(str)-1)  // ["blue", "is", "sky", "the"]
    return strings.Join(str, " ") // "blue is sky the"
}

func reverse(p *[]string, i, j int) { // 值传递
    for i < j {
        (*p)[i], (*p)[j] = (*p)[j], (*p)[i]
        i++
        j--
    }
}
```

129. 求根节点到叶节点数字之和

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func dfs(root *TreeNode, preSum int) int {
```

```
    if root == nil {
        return 0
    }
    sum := 10*preSum + root.Val
    if root.Left == nil && root.Right == nil {
        return sum
    }
    return dfs(root.Left, sum) + dfs(root.Right, sum)
}

func sumNumbers(root *TreeNode) int {
    return dfs(root, 0)
}
```

104. 二叉树的最大深度

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func maxDepth(root *TreeNode) int {
    if root == nil {
        return 0
    }
    return max(maxDepth(root.Left), maxDepth(root.Right)) + 1
}
```

101. 对称二叉树

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */

func isMirror(left, right *TreeNode) bool {
    if left == nil && right == nil { // 左右子树同时越过叶子节点，自顶向下的节点都
对称
        return true
    }
    if left == nil || right == nil { // 只有一个越过叶子节点，不对称
        return false
```

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

func isSymmetric(root *TreeNode) bool {
    return isMirror(root, root)
}
```

144. 二叉树的前序遍历

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func preorderTraversal(root *TreeNode) []int {
    var res []int
    var dfs func(*TreeNode)

    dfs = func(node *TreeNode) {
        if node != nil {
            res = append(res, node.Val)
            dfs(node.Left)
            dfs(node.Right)
        }
    }

    dfs(root)
    return res
}
```

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func preorder(root *TreeNode, res *[]int) {
    if root == nil {
        return
    }
    *res = append(*res, root.Val)
    preorder(root.Left, res)
```

```
    preorder(root.Right, res)
}

func preorderTraversal(root *TreeNode) []int {
    var ans []int
    preorder(root, &ans)
    return ans
}
```

110. 平衡二叉树

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
// 自底向上的递归
func isBalanced(root *TreeNode) bool {
    return height(root) >= 0
}

func height(root *TreeNode) int {
    if root == nil {
        return 0
    } // 自底向上递归的做法类似于后序遍历，对于当前遍历到的节点，
    // 先递归地判断其左右子树是否平衡，再判断以当前节点为根的子树是否平衡。
    leftHeight := height(root.Left)
    rightHeight := height(root.Right)
    if leftHeight < 0 || rightHeight < 0 || abs(leftHeight-rightHeight) >
1 {
        return -1 // 如果左右子树是不平衡的，返回 -1
    }
    return max(leftHeight, rightHeight) + 1 // 如果一棵子树是平衡的，则返回其高
度
}

func abs(x int) int {
    if x < 0 {
        return -1 * x
    }
    return x
}

// 时间复杂度：O(n)，其中 n 是二叉树中的节点个数。使用自底向上的递归，每个节点的计算高度
和判断是否平衡都只需要处理一次，最坏情况下需要遍历二叉树中的所有节点，因此时间复杂度是
O(n)。

// 空间复杂度：O(n)，其中 n 是二叉树中的节点个数。空间复杂度主要取决于递归调用的层数，递
```

归调用的层数不会超过 n。

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
// 方法一：自顶向下的递归
func isBalanced(root *TreeNode) bool {
    if root == nil {
        return true
    } // 自顶向下的递归做法类似于二叉树的前序遍历
    leftHeight := depth(root.Left) // 计算左/右子树的高度
    rightHeight := depth(root.Right)
    // 如果左右子树的高度差是否不超过 1，再分别递归地遍历左右子节点，并判断左子树和右子树是否平衡。
    return abs(leftHeight-rightHeight) <= 1 && isBalanced(root.Left) && isBalanced(root.Right)
}
func depth(root *TreeNode) int {
    if root == nil {
        return 0
    }
    return max(depth(root.Left), depth(root.Right)) + 1
}

func abs(x int) int {
    if x < 0 {
        return -1 * x
    }
    return x
}


// 时间复杂度：O(n^2)，其中 n 是二叉树中的节点个数。 最坏情况下，二叉树是满二叉树，需要遍历二叉树中的所有节点，时间复杂度是 O(n)。
//                   对于节点 p，如果它的高度是 d，则 height(p) 最多会被调用 d 次（即遍历到它的每一个祖先节点时）。
//                   对于平均的情况，一棵树的高度 hhh 满足 O(h)=O(logn)，因为 d≤h，所以总时间复杂度为 O(nlogn)。
//                   对于最坏的情况，二叉树形成链式结构，高度为 O(n)，此时总时间复杂度为 O(n^2)。

// 空间复杂度：O(n)，其中 n 是二叉树中的节点个数。空间复杂度主要取决于递归调用的层数，递归调用的层数不会超过 n。
```

参考

39. 组合总和

```
func combinationSum(candidates []int, target int) (ans [][]int) {
    comb := []int{}
    var dfs func(int, int)

    dfs = func(target int, idx int) {
        if idx == len(candidates) {
            return
        }
        if target == 0 {
            ans = append(ans, append([]int(nil), comb...))
            return
        }
        // 直接跳过
        dfs(target, idx+1)
        // 选择当前数
        if target-candidates[idx] >= 0 {
            comb = append(comb, candidates[idx])
            dfs(target-candidates[idx], idx) // 可以重复选取，idx不变
            comb = comb[:len(comb)-1]        // 回溯
        }
    }

    dfs(target, 0)
    return
}
```

```
// 剪枝优化1
func combinationSum(candidates []int, target int) (ans [][]int) {
    comb := []int{}
    var dfs func(int, int)

    dfs = func(target int, idx int) {
        if target <= 0 {
            if target == 0 { // 找到一组正确组合
                ans = append(ans, append([]int(nil), comb...)) // 将当前组合
加入解集
            }
            return // 结束当前递归
        }
        // 选择当前数
        for i := idx; i < len(candidates); i++ { // 枚举当前可选的数，从index
开始
            comb = append(comb, candidates[i]) // 选这个数,基于此，继续选择，传
i，下次就不会选到i左边的数
            dfs(target-candidates[i], i)       // 注意这里迭代的时候 index 依
旧不变，因为一个元素可以取多次
```

```
            comb = comb[:len(comb)-1]           // 撤销选择，回到选择
candidates[i]之前的状态，继续尝试选同层右边的数
        }
    }

    dfs(target, 0)
    return
}
```

```
// 剪枝优化2
func combinationSum(candidates []int, target int) (ans [][]int) {
    comb := []int{}
    sort.Ints(candidates)
    var dfs func(int, int)

    dfs = func(target int, idx int) {
        if target <= 0 {
            if target == 0 { // 找到一组正确组合
                ans = append(ans, append([]int(nil), comb...)) // 将当前组合
加入解集
            }
            return // 结束当前递归
        }
        // 选择当前数
        for i := idx; i < len(candidates); i++ { // 枚举当前可选的数，从index
开始
            if candidates[i] > target {
                break
            }
            comb = append(comb, candidates[i]) // 选这个数,基于此，继续选择，传
i，下次就不会选到i左边的数
            dfs(target-candidates[i], i)        // 注意这里迭代的时候 index 依
旧不变，因为一个元素可以取多次
            comb = comb[:len(comb)-1]           // 撤销选择，回到选择
candidates[i]之前的状态，继续尝试选同层右边的数
        }
    }

    dfs(target, 0)
    return
}
```

543. 二叉树的直径

一条路径的长度为该路径经过的节点数减一，所以求直径（即求路径长度的最大值）等效于求路径经过节点数的最大值减一

```
/**
 * Definition for a binary tree node.
```

```
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func diameterOfBinaryTree(root *TreeNode) int {
    res := 0
    var depth func(*TreeNode) int

    depth = func(root *TreeNode) int {
        if root == nil {
            return 0
        }
        left := depth(root.Left)
        right := depth(root.Right)
        res = max(res, left+right+1)
        return max(left, right) + 1
    }

    depth(root)
    return res - 1
}
```

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func diameterOfBinaryTree(root *TreeNode) int {
    res := 0
    depth(root, &res)
    return res
}

func depth(root *TreeNode, res *int) int {
    if root == nil {
        return 0
    }
    left := depth(root.Left, res)   // 左儿子为根的子树的深度
    right := depth(root.Right, res) // 右儿子为根的子树的深度
    *res = max(*res, left+right)    // 如果当前路径和大于历史路径和，更新结果
    return max(left, right) + 1     // 返回该节点为根的子树的深度
}
```

470. 用 Rand7() 实现 Rand10()

```go
func rand10() int {
    for {
        row, col := rand7(), rand7()
        idx := (row-1)*7 + col // 42+7=49   [1,49]
        if idx <= 40 {         // 只使用小于等于40的数
            return 1 + (idx-1)%10 // [1,10]
        }
    }
}
```

```go
func rand10() int {
    for {
        row, col := rand7(), rand7()
        idx := (row-1)*7 + col // [0,49]
        if idx <= 40 {
            return 1 + (idx-1)%10
        }
    }
}
```

```go
func rand10() int {
    rand40 := 40
    for rand40 >= 40 {
        rand40 = (rand7()-1)*7 + rand7() - 1
    }
    return rand40%10 + 1
}
```

48. 旋转图像

```go
/*
    旋转图像    =>  1. 水平翻转     =>      2. 主对角线翻转

1   2  3  4       1   5  9  13         13  9  5  1
5   6  7  8   =>  2   6  10 14   =>    14  10 6  2
9  10 11 12       3   7  11 15         15  11 7  3
13 14 15 16       4   8  12 16         16  12 8  4
*/
func rotate(matrix [][]int) {
    m := len(matrix)
    // 水平翻转
    for i := 0; i < m>>1; i++ {
        matrix[i], matrix[m-1-i] = matrix[m-1-i], matrix[i]
    }
    // 主对角线翻转
    for i := 0; i < m; i++ {
```

```
            for j := 0; j < i; j++ {
                matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
            }
        }
    }
```

98. 验证二叉搜索树

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func helper(root *TreeNode, lower, upper int) bool {
    if root == nil {
        return true
    }
    return lower < root.Val && root.Val < upper &&
        helper(root.Left, lower, root.Val) && helper(root.Right, root.Val,
upper)
}

func isValidBST(root *TreeNode) bool {
    return helper(root, math.MinInt64, math.MaxInt64)
}

func helpe2(root *TreeNode, lower, upper int) bool {
    if root == nil {
        return true
    }
    if root.Val <= lower || root.Val >= upper { //  == 解决 [2,2,2] 应返回
false
        return false
    }
    return helper(root.Left, lower, root.Val) && helper(root.Right,
root.Val, upper)
}
```

394. 字符串解码

外层的先等等，把内层的解决了再和你连线

方法1 栈解

- 外层的解码需要等待内层解码的结果。先扫描的字符还用不上，但不能忘了它们。
- 我们准备由内到外，层层解决[]，需要保持对字符的记忆，于是用栈。

入栈和出栈的时机

入栈时机：遇到[。意味着要解决内部的人了，外部的数字和字母，去栈里等。

- 当遇到[，已经扫描的数字就是“倍数”，入栈暂存
- 当遇到[，已经扫描的字母也入栈等待，括号里的解码完了，一起参与构建字符串。

出栈时机：遇到]。内层的扫描完了，栈顶元素可以出栈了，共同参与子串的构建。

- 栈顶就是最近遇到的“倍数”和字母

```
func decodeString(s string) string {
    numStack := []int{}       // 存倍数的栈
    strStack := []string{}    // 存待拼接的str的栈
    num := 0                  // 倍数的“搬运工”
    res := ""                 // 字符串的“搬运工”
    for _, char := range s { // 逐字符扫描
        if char >= '0' && char <= '9' { // 遇到数字
            n, _ := strconv.Atoi(string(char))
            num = 10*num + n // 算出倍数
        } else if char == '[' { // 遇到 [
            strStack = append(strStack, res) // res串入栈
            res = ""                         // 入栈后清零
            numStack = append(numStack, num) // 倍数num进入栈等待
            num = 0                          // 入栈后清零
        } else if char == ']' { // 遇到 ]，两个栈的栈顶出栈
            count := numStack[len(numStack)-1] // 获取拷贝次数
            numStack = numStack[:len(numStack)-1]
            preStr := strStack[len(strStack)-1]
            strStack = strStack[:len(strStack)-1]
            res = string(preStr) + strings.Repeat(res, count) // 构建子串 =
外层 + 内部重复
        } else {
            res += string(char) // 遇到字母，追加给res串
        }
    }
    return res
}
```

参考

34. 在排序数组中查找元素的第一个和最后一个位置

```
func searchRange(nums []int, target int) []int {
    start, end := findFirst(nums, target), findLast(nums, target)
```

```
    return []int{start, end}
}

func findFirst(nums []int, target int) int {
    low, high, start := 0, len(nums)-1, -1
    for low <= high {
        mid := low + (high-low)>>1
        if nums[mid] >= target {
            high = mid - 1
        } else {
            low = mid + 1
        }
        if nums[mid] == target {
            start = mid
        }
    }
    return start
}

func findLast(nums []int, target int) int {
    low, high, end := 0, len(nums)-1, -1
    for low <= high {
        mid := low + (high-low)>>1
        if nums[mid] <= target {
            low = mid + 1
        } else {
            high = mid - 1
        }
        if nums[mid] == target {
            end = mid
        }
    }
    return end
}
```

```
func searchRange(nums []int, target int) []int {
    start, end := findFirst(nums, target), findLast(nums, target)
    return []int{start, end}
}
// 二分查找第一个与 target 相等的元素，时间复杂度 O(logn)
func findFirst(nums []int, target int) int {
    low, high := 0, len(nums)-1
    for low <= high {
        mid := low + (high-low)>>1
        if nums[mid] < target {
            low = mid + 1
        } else if nums[mid] > target {
            high = mid - 1
        } else {
            if mid == 0 || nums[mid-1] != target { // 找到第一个与 target 相
等的元素
```

```
                return mid
            }
            high = mid - 1
        }
    }
    return -1
}
// 二分查找最后一个与 target 相等的元素，时间复杂度 O(logn)
func findLast(nums []int, target int) int {
    low, high := 0, len(nums)-1
    for low <= high {
        mid := low + (high-low)>>1
        if nums[mid] < target {
            low = mid + 1
        } else if nums[mid] > target {
            high = mid - 1
        } else {
            if mid == len(nums)-1 || nums[mid+1] != target { // 找到最后一个
与 target 相等的元素
                return mid
            }
            low = mid + 1
        }
    }
    return -1
}
```

113. 路径总和 II

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func pathSum(root *TreeNode, targetSum int) (res [][]int) {
    var dfs func(*TreeNode, int)
    var path []int

    dfs = func(node *TreeNode, left int) {
        if node == nil {
            return
        }
        left -= node.Val
        path = append(path, node.Val)
        defer func() {
            path = path[:len(path)-1]
        }()
        if node.Left == nil && node.Right == nil && left == 0 {
```

```
            res = append(res, append([]int(nil), path...))
            return
        }
        dfs(node.Left, left)
        dfs(node.Right, left)
    }

    dfs(root, targetSum)
    return
}
```

240. 搜索二维矩阵 II

```
func searchMatrix(matrix [][]int, target int) bool {
    x, y := 0, len(matrix[0])-1 // 从右上角开始遍历
    for y >= 0 && x < len(matrix) {
        if matrix[x][y] == target {
            return true
        }
        if matrix[x][y] > target {
            y--
        } else {
            x++
        }
    }
    return false
}
```

```
// 暴力
func searchMatrix(matrix [][]int, target int) bool {
    for _, row := range matrix {
        for _, v := range row {
            if v == target {
                return true
            }
        }
    }
    return false
}
```

64. 最小路径和

```
func minPathSum(grid [][]int) int {
    m, n := len(grid), len(grid[0]) // m 行 n 列
    for i := 1; i < m; i++ {
        grid[i][0] += grid[i-1][0] // 第0列 累加和
```

```
    }
    for j := 1; j < n; j++ {
        grid[0][j] += grid[0][j-1] // 第0行 累加和
    }
    for i := 1; i < m; i++ {
        for j := 1; j < n; j++ {
            grid[i][j] += min(grid[i-1][j], grid[i][j-1]) // 最小路径和
        }
    }
    return grid[m-1][n-1]
}
```

221. 最大正方形

```
func maximalSquare(matrix [][]byte) int {
    m, n, maxSide := len(matrix), len(matrix[0]), 0
    dp := make([][]int, m+1)
    for i := 0; i < m+1; i++ {
        dp[i] = make([]int, n+1)
    }
    for i := 1; i <= m; i++ {
        for j := 1; j <= n; j++ {
            if matrix[i-1][j-1] == '1' {
                dp[i][j] = min(dp[i-1][j-1], min(dp[i-1][j], dp[i][j-1]))
+ 1
            } else {
                dp[i][j] = 0
            }
            maxSide = max(maxSide, dp[i][j])
        }
    }
    return maxSide * maxSide
}
```

参考题解

162. 寻找峰值

二分查找优化

```
func findPeakElement(nums []int) int {
    low, high := 0, len(nums)-1
    for low < high {
        mid := low + (high-low)>>1
        if nums[mid] > nums[mid+1] { // 如果 mid 较大，则左侧存在峰值，high = m
            high = mid
        } else { // 如果 mid + 1 较大，则右侧存在峰值，low = mid + 1
            low = mid + 1
```

```
            }
        }
        return low // low == high
    }
```

- 时间复杂度：O(logn)，其中 nnn 是数组 nums 的长度。
- 空间复杂度：O(1)。

14. 最长公共前缀

方法一：纵向扫描 从前往后遍历所有字符串的每一列，比较相同列上的字符是否相同，

- 如果相同则继续对下一列进行比较;
- 如果不相同则当前列不再属于公共前缀，当前列之前的部分为最长公共前缀。

```go
func longestCommonPrefix(strs []string) string {
    if len(strs) == 0 {
        return ""
    }
    for i := 0; i < len(strs[0]); i++ {
        for j := 1; j < len(strs); j++ {
            if i == len(strs[j]) || strs[j][i] != strs[0][i] {
                return strs[0][:i]
            }
        }
    }
    return strs[0]
}
```

```go
func longestCommonPrefix(strs []string) string {
    prefix := strs[0]
    for i := 1; i < len(strs); i++ {
        for j := 0; j < len(prefix); j++ {
            if len(strs[i]) <= j || strs[i][j] != prefix[j] {
                prefix = prefix[0:j]
                break // 如果不中断，j++后会越界
            }
        }
    }
    return prefix
}
```

128. 最长连续序列

方法一：哈希表

```
func longestConsecutive(nums []int) int {
    numSet, longgest := map[int]bool{}, 0
    for _, num := range nums {
        numSet[num] = true // 标记 nums 数组中所有元素都存在
    }
    for _, num := range nums {
        if !numSet[num-1] { // 如果 num 没有前驱数（左邻居）num-1
            currNum, currLonggest := num, 1
            for numSet[currNum+1] { // 枚举数组中的每个数 x，考虑以其为起点，不断
尝试匹配 x+1,x+2,… 是否存在
                currNum++      // 枚举连续的下一个数
                currLonggest++ // 当前最长连续长度递增
            }
            if longgest < currLonggest { // 计算最大长度
                longgest = currLonggest
            }
        }
    }
    return longgest
}
```

- 时间复杂度：O(n)，其中 n 为数组的长度。具体分析已在上面正文中给出。
- 空间复杂度：O(n)。哈希表存储数组中所有的数需要 O(n) 的空间。

参考地址

```
func longestConsecutive(nums []int) (res int) {
    numSet := make(map[int]bool, len(nums))
    for _, num := range nums {
        numSet[num] = true
    }
    for num, _ := range numSet {
        if !numSet[num-1] {
            x := num + 1
            for numSet[x] {
                x++
            }
            if res < x-num {
                res = x - num
            }
        }
    }
    return
}
```

234. 回文链表

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func isPalindrome(head *ListNode) bool {
    vals := []int{}
    for head != nil {
        vals = append(vals, head.Val)
        head = head.Next
    }
    start, end := 0, len(vals)-1
    for start < end {
        if vals[start] != vals[end] {
            return false
        }
        start++
        end--
    }
    return true
}
```

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func isPalindrome(head *ListNode) bool {
    if head == nil || head.Next == nil {
        return true
    }
    slow, fast := head, head.Next
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
    }
    curr := slow.Next
    slow.Next = nil

    var head2 *ListNode
    for curr != nil {
        temp := curr.Next
        curr.Next = head2
        head2 = curr
        curr = temp
    }
```

```
    for head != nil && head2 != nil {
        if head.Val != head2.Val {
            return false
        }
        head = head.Next
        head2 = head2.Next
    }
    return true
}
```

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func isPalindrome(head *ListNode) bool {
    slice := []int{}
    for ; head != nil; head = head.Next {
        slice = append(slice, head.Val)
    }
    for i, j := 0, len(slice)-1; i < j; {
        if slice[i] != slice[j] {
            return false
        }
        i++
        j--
    }
    return true
}
```

112. 路径总和

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func hasPathSum(root *TreeNode, targetSum int) bool {
    if root == nil { // 空树不存在根节点到叶子节点的路径。
        return false
    }
    if root.Left == nil && root.Right == nil { // 如果当前节点是叶子节点
        return targetSum-root.Val == 0 // 路径和等于 targetSum 返回 true
    }
```

```
    return hasPathSum(root.Left, targetSum-root.Val) ||
hasPathSum(root.Right, targetSum-root.Val)
}
```

662. 二叉树最大宽度

方法一：广度优先搜索

思路

1. 此题求二叉树所有层的最大宽度，比较直观的方法是求出每一层的宽度，然后求出最大值。
2. 求每一层的宽度时，因为两端点间的 null 节点也需要计入宽度，因此可以对节点进行编号。
3. 一个编号为 index 的左子节点的编号记为 2*index*，*右子节点的编号记为* 2index + 1,
4. 计算每层宽度时，用每层节点的最大编号减去最小编号再加 1 即为宽度。

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */

type pair struct {
    node  *TreeNode
    index int
}

func widthOfBinaryTree(root *TreeNode) int {
    res := 0
    q := []pair{{root, 1}}
    for q != nil {
        res = max(res, q[len(q)-1].index-q[0].index+1)
        temp := q
        q = nil
        for _, p := range temp {
            if p.node.Left != nil {
                q = append(q, pair{p.node.Left, p.index * 2})
            }
            if p.node.Right != nil {
                q = append(q, pair{p.node.Right, p.index*2 + 1})
            }
        }
    }
    return res
}
```

复杂度分析

- 时间复杂度：O(n)，其中 n 是二叉树的节点个数。需要遍历所有节点。
- 空间复杂度：O(n)。广度优先搜索的空间复杂度最多为 O(n)。

方法二：深度优先搜索

思路

仍然按照上述方法编号，可以用深度优先搜索来遍历。

1. 遍历时如果是先访问左子节点，再访问右子节点，每一层最先访问到的节点会是最左边的节点，即每一层编号的最小值，需要记录下来进行后续的比较。
2. 一次深度优先搜索中，需要当前节点到当前行最左边节点的宽度，以及对子节点进行深度优先搜索，求出最大宽度，并返回最大宽度。

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */

func widthOfBinaryTree(root *TreeNode) int {
    levelMin := map[int]int{} // 每一层编号的最小值
    var dfs func(*TreeNode, int, int) int

    dfs = func(node *TreeNode, depth, index int) int {
        if node == nil {
            return 0
        }
        if _, ok := levelMin[depth]; !ok {
            levelMin[depth] = index // 每一层最先访问到的节点会是最左边的节点，即
每一层编号的最小值
        }
        return max(index-levelMin[depth]+1, max(dfs(node.Left, depth+1,
index*2), dfs(node.Right, depth+1, index*2+1)))
    }
    return dfs(root, 1, 1)
}
```

复杂度分析

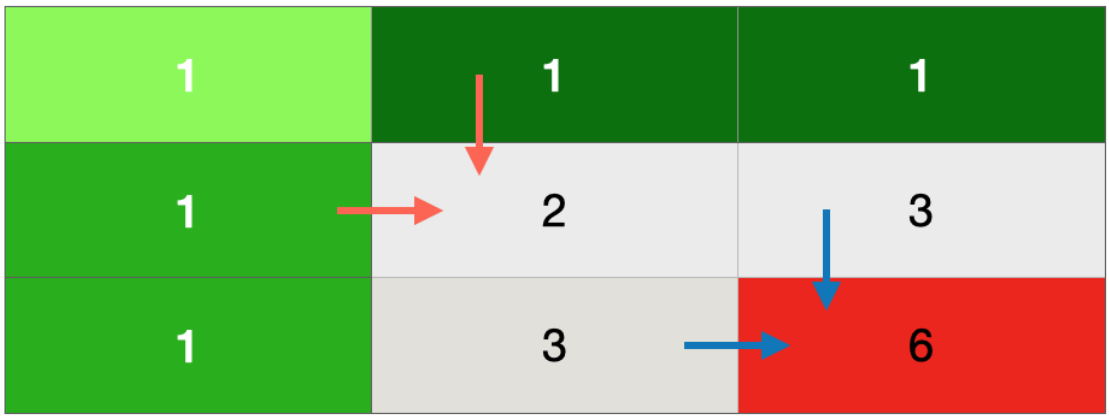
- 时间复杂度：O(n)，其中 n 是二叉树的节点个数。需要遍历所有节点。
- 空间复杂度：O(n)。递归的深度最多为 O(n)。

169. 多数元素

```go
func majorityElement(nums []int) int {
    major, vote := -1, 0
    for _, num := range nums {
        if vote == 0 { // 如果票数等于0，重新赋值，抵消掉非众数
            major = num
        }
        if major == num { // 如果众数 major 和 num 相等,票数自增1
            vote++
        } else { // 不相等,票数自减1
            vote--
        }
    }
    return major
}
```

62. 不同路径

方法一：动态规划



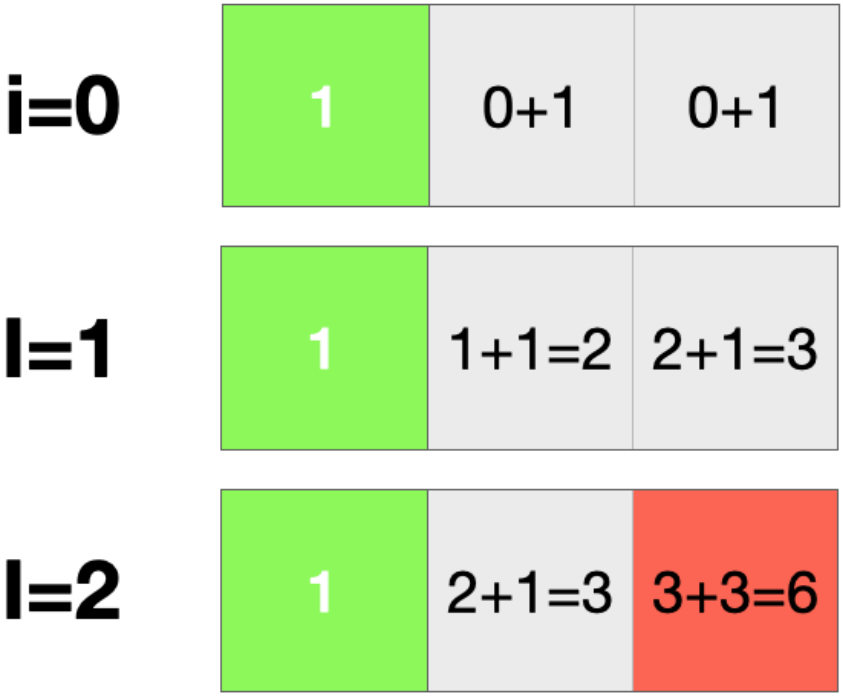

```
// 每一格的路径由其上一格和左一格决定
dp[i][j] = dp[i-1][j] + dp[i][j-1]
```

```go
func uniquePaths(m int, n int) int {
    dp := make([][]int, m) // 定义二维数组
    for i := 0; i < m; i++ {
        dp[i] = make([]int, n)
    }
    for i := 0; i < m; i++ {
```

```
            for j := 0; j < n; j++ {
                if i == 0 || j == 0 {
                    dp[i][j] = 1 // 初始化二维数组的第0行 第0列等于1
                    continue
                }
                dp[i][j] = dp[i-1][j] + dp[i][j-1] // 每一格的路径由其上一格和左一格
决定
            }
        }
        return dp[m-1][n-1]
}
```

方法二：滚动数组 优化空间 O(n)

dp[i][j] 仅与第 i 行和第 i−1 行的状态有关，因此我们可以使用滚动数组代替代码中的二维数组，使空间复杂度降低为 O(n)。




```
// 将自身与上一格相加得到右一格
dp[j] += dp[j-1]
```

- 只用长度为 n 的列表记录路径（纵向）

- 将自身与上一格相加得到右一格

```
func uniquePaths(m int, n int) int {
    dp := make([]int, n)
    for i := 0; i < m; i++ {
        for j := 0; j < n; j++ {
            if j == 0 { // 初始化：到达起点只有一种走法
                dp[j] = 1
                continue
            }
            dp[j] += dp[j-1] // 将自身与上一格相加得到右一格
        }
    }
    return dp[n-1]
}
```

参考官方视频题解

179. 最大数

```
func largestNumber(nums []int) string {
    sort.Slice(nums, func(i, j int) bool {
        x, y := nums[i], nums[j]
        sx, sy := 10, 10
        for sx <= x {
            sx *= 10
        }
        for sy <= y {
            sy *= 10
        }
        return sy*x+y > sx*y+x
    })
    if nums[0] == 0 {
        return "0"
    }
    res := []byte{}
    for _, x := range nums {
        res = append(res, strconv.Itoa(x)...)
    }
    return string(res)
}
```

```
func largestNumber(nums []int) string {
    if len(nums) == 0 {
        return ""
    }
    res := ""
    s := numToString(nums)
```

```
    quickSortString(s, 0, len(s)-1)
    for _, str := range s {
        if res == "0" && str == "0" {
            continue
        }
        res += str
    }
    return res
}

func numToString(nums []int) []string {
    s := make([]string, 0)
    for i := 0; i < len(nums); i++ {
        s = append(s, strconv.Itoa(nums[i]))
    }
    return s
}

func quickSortString(s []string, start, end int) {
    if start <= end {
        piv_pos := partition(s, start, end)
        quickSortString(s, start, piv_pos-1)
        quickSortString(s, piv_pos+1, end)
    }
}

func partition(s []string, start, end int) int {
    i, x := start, s[end]
    for j := start; j < end; j++ {
        sjx, xsj := s[j]+x, x+s[j]
        if sjx > xsj {
            s[i], s[j] = s[j], s[i]
            i++
        }
    }
    s[i], s[end] = s[end], s[i]
    return i
}
```

718. 最长重复子数组

```
func findLength(nums1 []int, nums2 []int) (res int) {
    m, n := len(nums1), len(nums2)
    dp := make([][]int, m+1)
    for i := 0; i < m+1; i++ {
        dp[i] = make([]int, n+1)
    }
    for i := 1; i <= m; i++ {
        for j := 1; j <= n; j++ {
            if nums1[i-1] == nums2[j-1] {
                dp[i][j] = dp[i-1][j-1] + 1
```

```
            }
            if res < dp[i][j] {
                res = dp[i][j]
            }
        }
    }
    return
}
```

```
func findLength(nums1 []int, nums2 []int) (res int) {
    m, n := len(nums1), len(nums2)
    dp := make([]int, m+1)
    for i := 1; i <= m; i++ {
        for j := n; j >= 1; j-- {
            if nums1[i-1] == nums2[j-1] {
                dp[j] = dp[j-1] + 1
            } else {
                dp[j] = 0
            }
            if res < dp[j] {
                res = dp[j]
            }
        }
    }
    return
}
```

参考题解

227. 基本计算器 II

先进行所有乘除运算，并将这些乘除运算后的整数值放回原表达式的相应位置，则随后整个表达式的值，就等于一系列整数加减后的值。 - 对于乘除号后的数字，可以直接与栈顶元素计算，并替换栈顶元素为计算后的结果； - 对于加减号后的数字，将其直接压入栈中。

- 加号：将数字压入栈；
- 减号：将数字的相反数压入栈；
- 乘除号：计算数字与栈顶元素，并将栈顶元素替换为计算结果。

```
func calculate(s string) (res int) {
    // 对于乘除号后的数字，可以直接与栈顶元素计算，并替换栈顶元素为计算后的结果。
    // 对于加减号后的数字，将其直接压入栈中；
    stack := []int{}
    preSign := '+' // 记录每个数字之前的运算符
    num := 0
    for i, ch := range s {
        isDight := '0' <= ch && ch <= '9'
        if isDight {
            num = num*10 + int(ch-'0') // 字符串转数字
```

```
        }
        if !isDight && ch != ' ' || i == len(s)-1 {
            switch preSign {
            case '+':
                stack = append(stack, num) // 加号：将数字压入栈
            case '-':
                stack = append(stack, -num) // 减号：将数字的相反数压入栈
            case '*':
                stack[len(stack)-1] *= num // 乘号：数字乘以栈顶元素，并将栈顶元素替换为计算结果
            default:
                stack[len(stack)-1] /= num // 除号：数字除以栈顶元素，并将栈顶元素替换为计算结果
            }
            preSign = ch
            num = 0
        }
    }

    for _, v := range stack {
        res += v // 将栈中元素累加
    }
    return
}
```

122. 买卖股票的最佳时机 II

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

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

198. 打家劫舍

解法一 模拟

```go
func rob(nums []int) int {
    a, b := 0, 0
    for i, v := range nums {
        if i%2 == 0 {
            a = max(b, a+v) // a 对于偶数位上的最大值的记录
        } else {
            b = max(a, b+v) // b 对于奇数位上的最大值的记录
        }
    }
    return max(a, b)
}
```

解法二 动态规划

```go
func rob(nums []int) int {
    n := len(nums)
    if n == 0 {
        return 0
    }
    if n == 1 {
        return nums[0]
    }
    dp := make([]int, n)
    dp[0] = nums[0]                // 只有一间房屋，则偷窃该房屋
    dp[1] = max(nums[0], nums[1]) // 只有两间房屋，选择其中金额较高的房屋进行偷窃
    for i := 2; i < n; i++ {
        dp[i] = max(dp[i-2]+nums[i], dp[i-1]) // dp[i] 前i间房屋能偷窃到的最高总金额 = max(抢第i间房子，不抢第i件房子)
    }
    return dp[n-1]
}
```

解法三：滚动数组（优化空间）

```go
func rob(nums []int) int {
    preMax, curMax := 0, 0
    for i := 0; i < len(nums); i++ {
        // preMax, curMax = curMax, max(preMax+nums[i], curMax)
        tmp := curMax
        curMax = max(preMax+nums[i], curMax)
        preMax = tmp
    }
    return curMax
}
```

152. 乘积最大子数组

```go
func maxProduct(nums []int) int {
    maxF, minF, res := nums[0], nums[0], nums[0]
    for i := 1; i < len(nums); i++ {
        if nums[i] < 0 { // 如果 nums[i] 是负数
            maxF, minF = minF, maxF // 交换最大值与最小值
        }
        maxF = max(nums[i], maxF*nums[i])
        minF = min(nums[i], minF*nums[i])
        res = max(res, maxF)
    }
    return res
}
```

```go
func maxProduct(nums []int) int {
    maxF, minF, res := nums[0], nums[0], nums[0]
    for i := 1; i < len(nums); i++ {
        mx, mn := maxF, minF
        maxF = max(mx*nums[i], max(mn*nums[i], nums[i]))
        minF = min(mn*nums[i], min(mx*nums[i], nums[i]))
        res = max(res, maxF)
    }
    return res
}
```

83. 删除排序链表中的重复元素

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func deleteDuplicates(head *ListNode) *ListNode {
    if head == nil {
        return head
    }
    curr := head
    for curr.Next != nil {
        if curr.Val == curr.Next.Val { // 如果当前当前节点值等于下一个节点值
            curr.Next = curr.Next.Next // 删除重复
        } else { // 无重复
            curr = curr.Next // 继续向后扫描
        }
    }
```

```
    return head
}
```

695. 岛屿的最大面积

```
func maxAreaOfIsland(grid [][]int) int {
    maxArea := 0
    for i := 0; i < len(grid); i++ {
        for j := 0; j < len(grid[0]); j++ {
            if grid[i][j] == 1 {
                maxArea = max(maxArea, dfs(grid, i, j))
            }
        }
    }
    return maxArea
}

func dfs(grid [][]int, i, j int) int {
    if i < 0 || j < 0 || i >= len(grid) || j >= len(grid[0]) || grid[i][j]
== 0 {
        return 0
    }
    area := 1      // 岛屿的面积至少为1
    grid[i][j] = 0 // 已扫描，标记为海洋，防止重复扫描
    area += dfs(grid, i-1, j)
    area += dfs(grid, i+1, j)
    area += dfs(grid, i, j-1)
    area += dfs(grid, i, j+1)
    return area
}

func max(x, y int) int {
    if x > y {
        return x
    }
    return y
}
```

226. 翻转二叉树

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func invertTree(root *TreeNode) *TreeNode {
```

```
    if root == nil {
        return nil
    }
    invertTree(root.Left)                              // 翻转左子树
    invertTree(root.Right)                             // 翻转右子树（入栈：压栈压到
底部）
    root.Left, root.Right = root.Right, root.Left // 交换（出栈：自底向上）
    return root
}
```

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func invertTree(root *TreeNode) *TreeNode {
    if root == nil {
        return nil
    }
    root.Left, root.Right = root.Right, root.Left // 交换左右子树
    invertTree(root.Left)                              // 翻转左子树
    invertTree(root.Right)                             // 翻转右子树
    return root
}
```

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func invertTree(root *TreeNode) *TreeNode {
    if root == nil {
        return nil
    }
    q := []*TreeNode{root}
    for len(q) > 0 {
        node := q[0]                                       // 取队首
        q = q[1:]                                          // 队首元素出队
        node.Left, node.Right = node.Right, node.Left // 翻转左右子树
        if node.Left != nil {
            q = append(q, node.Left)
        }
        if node.Right != nil {
```

```
            q = append(q, node.Right)
        }
    }
    return root
}
```

139. 单词拆分

560. 和为 K 的子数组

209. 长度最小的子数组

补充题6. 手撕堆排序 912. 排序数组

```
func sortArray(nums []int) []int {
    heapSort(nums)
    return nums
}

func heapSort(nums []int) {
    heapSize := len(nums)
    buildMaxHeap(nums, heapSize)
    for i := heapSize - 1; i >= 1; i-- {
        nums[0], nums[i] = nums[i], nums[0]
        heapSize--
        maxHeapify(nums, 0, heapSize)
    }
}

func buildMaxHeap(nums []int, heapSize int) {
    for i := heapSize >> 1; i >= 0; i-- {
        maxHeapify(nums, i, heapSize)
    }
}

func maxHeapify(nums []int, i, heapSize int) {
    for i<<1+1 < heapSize {
        lson, rson, large := i<<1+1, i<<1+2, i
        if lson < heapSize && nums[large] < nums[lson] {
            large = lson
        }
        for rson < heapSize && nums[large] < nums[rson] {
            large = rson
        }
        if large != i {
            nums[i], nums[large] = nums[large], nums[i]
            i = large
        } else {
            break
        }
    }
```

```
}

func maxHeapify1(nums []int, i, heapSize int) {
    lson, rson, large := i<<1+1, i<<1+2, i
    if lson < heapSize && nums[large] < nums[lson] {
        large = lson
    }
    if rson < heapSize && nums[large] < nums[rson] {
        large = rson
    }
    if large != i {
        nums[i], nums[large] = nums[large], nums[i]
        maxHeapify(nums, large, heapSize)
    }
}
```

24. 两两交换链表中的节点

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func swapPairs(head *ListNode) *ListNode {
    dummy := &ListNode{Next: head}
    temp := dummy
    for temp.Next != nil && temp.Next.Next != nil {
        node1, node2 := temp.Next, temp.Next.Next
        temp.Next = node2       // 头插
        node1.Next = node2.Next // 连接后继
        node2.Next = node1      // 交换
        temp = node1            // 交换下一对
    }
    return dummy.Next
}

// 递归
func swapPairs1(head *ListNode) *ListNode {
    if head == nil || head.Next == nil {
        return head
    }
    newHead := head.Next
    head.Next = swapPairs(newHead.Next) // 将其余节点进行两两交换，交换后的新的头节点为 head 的下一个节点。
    newHead.Next = head                 // 节点交换
    return newHead
}
```

参考官方题解

224. 基本计算器

解题思路：

- stack 记录括号前的正负
- sign 记录数字前的正负

3-(2-1)
=3-2+1
=1+1 =2

想不通的问题，调试，看运行过程中参数变化

```
func calculate(s string) (res int) {
    sign := 1          // 记录当前数的正负号
    stack := []int{1} // - ( 记录括号前的正负号（-1表示负号、1表示正号）
    n := len(s)
    for i := 0; i < n; {
        switch s[i] {
        case ' ':
            i++
        case '+':
            sign = stack[len(stack)-1] // 加号：计算去括号后最终的正负
            i++
        case '-':
            sign = -stack[len(stack)-1] // 减号：计算去括号后最终的正负
            i++
        case '(':
            stack = append(stack, sign) // 加括号（入栈），记录括号前的正负
            i++
        case ')':
            stack = stack[:len(stack)-1] // 去括号（出栈）
            i++
        default:
            num := 0
            for ; i < n && '0' <= s[i] && s[i] <= '9'; i++ {
                num = num*10 + int(s[i]-'0') // 统计 0~9 组成的数字
            }
            res += sign * num // 去括号后所有数字带正负号累加到结果
        }
    }
    return
}
```

```
func calculate(s string) int {
    stack, res, num, sign := []int{}, 0, 0, 1
    for i := 0; i < len(s); i++ {
        if s[i] >= '0' && s[i] <= '9' {
```

```
            num = num*10 + int(s[i]-'0')
        } else if s[i] == '+' {
            res += sign * num
            num = 0
            sign = 1
        } else if s[i] == '-' {
            res += sign * num
            num = 0
            sign = -1
        } else if s[i] == '(' {
            stack = append(stack, res) //将前一个结果和符号压入栈中
            stack = append(stack, sign)
            res = 0 //将结果设置为0，只需在括号内计算新结果。
            sign = 1
        } else if s[i] == ')' {
            res += sign * num
            num = 0
            res *= stack[len(stack)-1]
            res += stack[len(stack)-2]
            stack = stack[:len(stack)-2]
        }
    }
    if num != 0 {
        res += sign * num
    }
    return res
}
```