

1. 41. 缺失的第一个正数
2. 129. 求根节点到叶节点数字之和
3. 43. 字符串相乘
4. 78. 子集
5. 32. 最长有效括号
6. 113. 路径总和 II
7. 101. 对称二叉树
8. 718. 最长重复子数组
9. 543. 二叉树的直径
10. 98. 验证二叉搜索树
11. 64. 最小路径和
12. 470. 用 Rand7() 实现 Rand10()
13. 112. 路径总和
14. 234. 回文链表
15. 48. 旋转图像
16. 169. 多数元素
17. 39. 组合总和
18. 165. 比较版本号
19. 221. 最大正方形
20. 226. 翻转二叉树
21. 34. 在排序数组中查找元素的第一个和最后一个位置

41. 缺失的第一个正数

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

```
func firstMissingPositive(nums []int) int {  
    n := len(nums)  
    for _, x := range nums {  
        for x > 0 && x <= n && nums[x-1] != x {  
            nums[x-1], x = x, nums[x-1]  
        }  
    }  
    for i, x := range nums {
```

```
        if x != i+1 {
            return i + 1
        }
    }
    return n + 1
}
```

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 sumNumbers(root *TreeNode) int {
    return dfs(root, 0)
}
func dfs(root *TreeNode, prevSum int) int {
    if root == nil {
        return 0
    }
    sum := prevSum*10 + root.Val
    if root.Left == nil && root.Right == nil {
        return sum
    }
    return dfs(root.Left, sum) + dfs(root.Right, sum)
}
```

43. 字符串相乘

竖式乘法

$$\begin{array}{r}
 \begin{array}{r}
 123 \\
 456 \\
 \hline
 61218 \\
 51015 \\
 4812 \\
 \hline
 413282718 \\
 \hline
 56088
 \end{array}
 \end{array}$$

```

func multiply(num1 string, num2 string) string {
    if num1 == "0" || num2 == "0" {
        return "0"
    }
    m, n := len(num1), len(num2)
    A := make([]int, m+n)
    for i := m - 1; i >= 0; i-- {
        x := int(num1[i] - '0')
        for j := n - 1; j >= 0; j-- {
            y := int(num2[j] - '0')
            A[i+j+1] += x * y // 竖式乘法，只累加但不进位
        }
    }
    for i := m + n - 1; i > 0; i-- {
        A[i-1] += A[i] / 10 // 进位
        A[i] %= 10         // 个位
    }
    res, i := "", 0
    if A[0] == 0 { // 去除前导0
        i = 1
    }
    for ; i < m+n; i++ {
        res += strconv.Itoa(A[i]) // 整数转字符串、拼接
    }
    return res
}

```

[参考视频](#)

78. 子集

```
func subsets(nums []int) (res [][]int) {
    n := len(nums)
    for mask := 0; mask < 1<<n; mask++ { // mask 从 000 到 111
        set := []int{}
        for i, v := range nums {
            if mask>>i&1 == 1 { // mask 的第 i 位是否为1
                set = append(set, v)
            }
        }
        res = append(res, set)
    }
    return
}
```

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

    dfs = func(i int) {
        if i == len(nums) { // 指针越界
            res = append(res, append([]int(nil), set...)) // 加入解集
            return // 结束当前的递归
        }
        set = append(set, nums[i]) //选择这个数
        dfs(i + 1) // 基于该选择, 继续往下递归, 考察下一个数
        set = set[:len(set)-1] // 上面的递归结束, 撤销该选择
        dfs(i + 1) // 不选这个数, 继续往下递归, 考察下一个数
    }

    dfs(0)
    return res
}
```

```
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
}
```

32. 最长有效括号

方法三：正序+逆序遍历,不需要额外的空间

```
func longestValidParentheses(s string) int {
    left, right, maxLength, n := 0, 0, 0, len(s)
    for i := 0; i < n; i++ { // 正序遍历
        if s[i] == '(' {
            left++
        } else {
            right++
        }
        if left == right {
            maxLength = max(maxLength, 2*left)
        } else if right > left {
            left, right = 0, 0
        }
    }
    left, right = 0, 0
    for i := n - 1; i >= 0; i-- { // 逆序遍历
        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 max(x, y int) int {
    if x > y {
        return x
    }
    return y
}
```

[参考](#)

方法二：栈

```

func longestValidParentheses(s string) int {
    stack, res, n := []int{-1}, 0, len(s)
    for i := 0; i < n; i++ {
        if s[i] == '(' {
            stack = append(stack, i) // 入栈
        } else {
            stack = stack[:len(stack)-1] // 出栈
            if len(stack) == 0 {
                stack = append(stack, i) // 入栈
            } else {
                res = max(res, i-stack[len(stack)-1])
            }
        }
    }
    return res
}

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

```

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

```

```

    dfs(root, targetSum)
    return res
}

```

101. 对称二叉树

```

/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func isSymmetric(root *TreeNode) bool {
    return isMirror(root, root)
}
func isMirror(left, right *TreeNode) bool {
    if left == nil && right == nil {
        return true
    }
    if left == nil || right == nil {
        return false
    }
    return left.Val == right.Val && isMirror(left.Left, right.Right) &&
isMirror(left.Right, right.Left)
}

```

```

/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func isSymmetric(root *TreeNode) bool {
    q := []*TreeNode{root, root}
    for len(q) > 0 {
        left, right := q[0], q[1]
        q = q[2:]
        if left == nil && right == nil {
            continue
        }
        if left == nil || right == nil {
            return false
        }
        if left.Val != right.Val {
            return false
        }
    }
}

```

```

    }
    q = append(q, left.Left)
    q = append(q, right.Right)

    q = append(q, left.Right)
    q = append(q, right.Left)
}
return true
}

```

718. 最长重复子数组

```

func findLength(nums1 []int, nums2 []int) int {
    m, n, res := len(nums1), len(nums2), 0
    dp := make([][]int, m+1) // 初始化整个dp矩阵，每个值为0
    for i := 0; i < m+1; i++ {
        dp[i] = make([]int, n+1)
    } // i=0或j=0的base case，初始化时已经包括
    for i := 1; i <= m; i++ { // 从1开始遍历
        for j := 1; j <= n; j++ { // 从1开始遍历
            if nums1[i-1] == nums2[j-1] {
                dp[i][j] = dp[i-1][j-1] + 1
            } // A[i-1] != B[j-1]的情况，初始化时已包括了
            if dp[i][j] > res {
                res = dp[i][j]
            }
        }
    }
    return res
}

```

```

func findLength(A []int, B []int) int {
    m, n := len(A), len(B)
    dp, res := make([]int, m+1), 0
    for i := 1; i < m+1; i++ { // 从上到下遍历行
        for j := n; j >= 1; j-- { // 从右到左遍历列
            if A[i-1] == B[j-1] {
                dp[j] = dp[j-1] + 1 // 从右上角开始计算dp[j]
            } else {
                dp[j] = 0
            }
            if dp[j] > res {
                res = dp[j]
            }
        }
    }
    return res
}

```


543. 二叉树的直径

```

/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func diameterOfBinaryTree(root *TreeNode) int {
    res := 1
    var depth func(*TreeNode) int

    depth = func(node *TreeNode) int {
        if node == nil {
            return 0
        }
        left, right := depth(node.Left), depth(node.Right) // 左右子树最大深度
        res = max(res, left+right+1) // max(最大直径, 当前节点的直径)
        return max(left, right) + 1 // 返回该节点为根的子树的深度
    }
    depth(root)
    return res - 1
}

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

```

98. 验证二叉搜索树

```

/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func isValidBST(root *TreeNode) bool {
    return dfs(root, math.MinInt64, math.MaxInt64)
}

func dfs(node *TreeNode, lower, upper int) bool {
    if node == nil {

```

```

        return true
    }
    if node.Val <= lower || node.Val >= upper {
        return false
    }
    return dfs(node.Left, lower, node.Val) && dfs(node.Right, node.Val,
upper)
}

```

```

/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func isValidBST(root *TreeNode) bool {
    stack, inorder := []*TreeNode{}, math.MinInt64
    for root != nil || len(stack) > 0 {
        for ; root != nil; root = root.Left {
            stack = append(stack, root)
        }
        root = stack[len(stack)-1]
        stack = stack[:len(stack)-1]
        if root.Val <= inorder {
            return false
        }
        inorder = root.Val
        root = root.Right
    }
    return true
}

```

64. 最小路径和

方法一：原地 DP，无辅助空间

```

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

```

470. 用 Rand7() 实现 Rand10()

```

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

```

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)
}

```

234. 回文链表

```

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

```

```

/**
 * 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
    var prev *ListNode
    for fast != nil && fast.Next != nil {
        prev = slow
        slow = slow.Next
        fast = fast.Next.Next
    }
    prev.Next = nil
    head2 := new(ListNode)
    for slow != nil {
        next := slow.Next
        slow.Next = head2
        head2 = slow
        slow = next
    }
    for head != nil && head2 != nil {

```

```

        if head.Val != head2.Val {
            return false
        }
        head = head.Next
        head2 = head2.Next
    }
    return true
}

```

48. 旋转图像

```

func rotate(matrix [][]int) {
    n := len(matrix)
    for i := 0; i < n/2; i++ { // 水平翻转：枚举矩阵上半部分的元素，和下半部分的元素进行交换
        matrix[i], matrix[n-1-i] = matrix[n-1-i], matrix[i]
    }
    for i := 0; i < n; i++ {
        for j := 0; j < i; j++ { // 主对角线翻转：枚举对角线左侧的元素，和右侧的元素进行交换
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
        }
    }
}

```

169. 多数元素

```

func majorityElement(nums []int) int {
    m, vote := 0, 0
    for _, x := range nums {
        if vote == 0 {
            m = x
        }
        if m == x { // x 等于众数
            vote++
        } else {
            vote-- // 不相等，删除
        }
    }
    return m
}

```

39. 组合总和

方法一：搜索回溯

参考

- x: 当前组合和之前生成的组合重复了。
- Δ: 当前求和 > target, 不能选下去了, 返回。
- ○: 求和正好 == target, 加入解集, 并返回。



```

        if target <= 0 {
            if target == 0 { // 找到一组正确组合
                res = append(res, append([]int(nil), comb...)) // 将当前组合
加入解集
            }
            return // 结束当前递归
        }
    }

```

不产生重复组合怎么限制（剪枝）？

如图，只要限制下一次选择的起点，是基于本次的选择，这样下一次就不会选到本次选择同层左边的数。即通过控制 for 遍历的起点，去掉会产生重复组合的选项。

```

        for i := index; 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
        }
    }

```

注意，子递归传了 i 而不是 i+1，因为元素可以重复选入集合，如果传 i+1 就不重复了。

```

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

    dfs = func(target, index int) {
        if target <= 0 {
            if target == 0 { // 找到一组正确组合
                res = append(res, append([]int(nil), comb...)) // 将当前组合
加入解集
            }
            return // 结束当前递归
        }
        for i := index; 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 res
}
```

```
func combinationSum(candidates []int, target int) (res [][]int) {
    path := []int{}
    sort.Ints(candidates)
    var dfs func(int, int)

    dfs = func(target, index int) {
        if target <= 0 {
            if target == 0 {
                res = append(res, append([]int(nil), path...))
            }
            return
        }
        for i := index; i < len(candidates); i++ { // 枚举当前可选的数, 从
index开始
            if candidates[i] > target { // 剪枝优化
                break
            }
            path = append(path, 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
            path = path[:len(path)-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
}
```

参考

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 := 0
        for ; i < m && version1[i] != '.'; i++ {
            x = x*10 + int(version1[i]-'0')
        }
        i++ // 跳过点号
        y := 0
        for ; j < n && version2[j] != '.'; j++ {
            y = y*10 + int(version2[j]-'0')
        }
    }
}
```



```

    }
    j++ // 跳过点号
    if x > y {
        return 1
    }
    if x < y {
        return -1
    }
}
return 0
}

```

221. 最大正方形

方法一：动态规划

方法一虽然直观，但是时间复杂度太高，有没有办法降低时间复杂度呢？可以使用动态规划降低时间复杂度。我们用 $dp(i,j)$ 表示以 (i,j) 为右下角，且只包含 1 的正方形的边长最大值。如果我们能计算出所有 $dp(i,j)$ 的值，那么其中的最大值即为矩阵中只包含 1 的正方形的边长最大值，其平方即为最大正方形的面积。那么如何计算 dp 中的每个元素值呢？对于每个位置 (i,j) ，检查在矩阵中该位置的值：

- 如果该位置的值是 0，则 $dp(i,j)=0$ ，因为当前位置不可能在由 1 组成的正方形中；
- 如果该位置的值是 1，则 $dp(i,j)$ 的值由其上方、左方和左上方的三个相邻位置的 dp 值决定。

具体而言，当前位置的元素值等于三个相邻位置的元素中的最小值加 1，状态转移方程如下：

$$dp(i,j)=\min(dp(i-1,j),dp(i-1,j-1),dp(i,j-1))+1$$

如果读者对这个状态转移方程感到不解，可以参考 1277. 统计全为 1 的正方形子矩阵的官方题解，其中给出了详细的证明。此外，还需要考虑边界条件。如果 i 和 j 中至少有一个为 0，则以位置 (i,j) 为右下角的最大正方形的边长只能是 1，因此 $dp(i,j)=1$ 。

以下用一个例子具体说明。原始矩阵如下。

```

0 1 1 1 0
1 1 1 1 0
0 1 1 1 1
0 1 1 1 1
0 0 1 1 1

```

对应的 dp 值如下。

```

0 1 1 1 0
1 1 2 2 0
0 1 2 3 1

```

```
0 1 2 3 2
0 0 1 2 3
```

原始矩阵

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 |
| 2 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 1 | 1 | 1 |

3×3 表示 $dp[2][3]$

2×2 表示 $dp[3][4]$

1×1 表示 $dp[4][2]$

dp

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 2 | 2 | 0 |
| 2 | 0 | 1 | 2 | 3 | 1 |
| 3 | 0 | 1 | 2 | 3 | 2 |
| 4 | 0 | 0 | 1 | 2 | 3 |

$$dp(2, 3) = \min(dp(1, 3), dp(1, 2), dp(2, 2)) + 1 = 3$$

$$dp(3, 4) = \min(dp(2, 4), dp(2, 3), dp(3, 3)) + 1 = 2$$

$$dp(4, 2) = \min(dp(3, 2), dp(3, 1), dp(4, 1)) + 1 = 1$$

```
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(min(dp[i-1][j], dp[i][j-1]), dp[i-1][j-1])
+ 1
            } else {
                dp[i][j] = 0
            }
            if dp[i][j] > maxSide {
                maxSide = dp[i][j]
            }
        }
    }
    return maxSide * maxSide
}

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

[参考](#)

226. 翻转二叉树

[←](#) 推文**Max Howell**

@mxcl

...

Google: 90% of our engineers use the software you wrote (Homebrew), but you can't invert a binary tree on a whiteboard so fuck off.

[翻译推文](#)

上午1:07 · 2015年6月11日 · Twitter Web Client

7,589 转推 **426** 引用推文 **1.3万** 喜欢次数

```
/**
 * 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
}

```

```

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

```

```
* }
*/
func invertTree(root *TreeNode) *TreeNode {
    if root != nil {
        root.Left, root.Right = root.Right, root.Left
        invertTree(root.Left)
        invertTree(root.Right)
    }
    return root
}
```

34. 在排序数组中查找元素的第一个和最后一个位置

方法一

```
func searchRange(nums []int, target int) []int {
    first, last := findFirst(nums, target), findLast(nums, target)
    return []int{first, last}
}

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

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

方法二

```
func searchRange(nums []int, target int) []int {
    first, last := findFirst(nums, target), findLast(nums, target)
    return []int{first, last}
}

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 {
                return mid
            }
            high = mid - 1
        }
    }
    return -1
}

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 {
                return mid
            }
            low = mid + 1
        }
    }
    return -1
}
```