0101-0200

101. 对称二叉树

题目

```
给定一个二叉树，检查它是否是镜像对称的。
例如，二叉树 [1,2,2,3,4,4,3] 是对称的。
    1
   / \
  2   2
```

```
 / \ / \
3  4 4  3

但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的:
    1
   / \
  2   2
   \   \
   3    3

说明:
如果你可以运用递归和迭代两种方法解决这个问题，会很加分。
```

解答思路

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	O(n)	O(n)
02	迭代	O(n)	O(n)

```go
// 递归
func isSymmetric(root *TreeNode) bool {
  if root == nil {
    return true
  }
  return recur(root.Left, root.Right)
}

func recur(left, right *TreeNode) bool {
  if left == nil && right == nil {
    return true
  }
  if left == nil || right == nil {
    return false
  }

  return left.Val == right.Val &&
    recur(left.Left, right.Right) &&
    recur(left.Right, right.Left)
}
// 迭代
func isSymmetric(root *TreeNode) bool {
  leftQ := make([]*TreeNode, 0)
  rightQ := make([]*TreeNode, 0)
  leftQ = append(leftQ, root)
  rightQ = append(rightQ, root)

```

```
  for len(leftQ) != 0 && len(rightQ) != 0 {
    leftCur, rightCur := leftQ[0], rightQ[0]
    leftQ, rightQ = leftQ[1:], rightQ[1:]

    if leftCur == nil && rightCur == nil {
      continue
    } else if leftCur != nil && rightCur != nil && leftCur.Val == rightCur.Val {
      leftQ = append(leftQ, leftCur.Left, leftCur.Right)
      rightQ = append(rightQ, rightCur.Right, rightCur.Left)
    } else {
      return false
    }
  }

  if len(leftQ) == 0 && len(rightQ) == 0 {
    return true
  } else {
    return false
  }
}
```

104.二叉树的最大深度

题目

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	O(n)	O(log(n))
02	迭代	O(n)	O(n)

解答思路

```
// 递归
func maxDepth(root *TreeNode) int {
  if root == nil {
    return 0
  }
  left := maxDepth(root.Left)
  right := maxDepth(root.Right)

  return max(left, right) + 1
}

func max(a, b int) int {
  if a > b {
```

```
    return a
  }
  return b
}


// 迭代
func maxDepth(root *TreeNode) int {
  if root == nil {
    return 0
  }
  queue := make([]*TreeNode,0)
  queue = append(queue,root)
  depth := 0

  for len(queue) > 0{
    length := len(queue)

    for i := 0; i < length; i++{
      node := queue[0]
      queue = queue[1:]
      if node.Left != nil{
        queue = append(queue,node.Left)
      }
      if node.Right != nil{
        queue = append(queue,node.Right)
      }
    }
    depth++
  }
  return depth
}
```

107.二叉树的层次遍历II

题目

```
给定一个二叉树，返回其节点值自底向上的层次遍历。 （即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

例如：
给定二叉树 [3,9,20,null,null,15,7],
    3
   / \
  9  20
    /  \
   15   7

```

```
返回其自底向上的层次遍历为：
[
  [15,7],
  [9,20],
  [3]
]
```

解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	O(n)	O(n)
02	迭代	O(n)	O(n)

```go
// 迭代
func levelOrderBottom(root *TreeNode) [][]int {
  if root == nil {
    return nil
  }
  queue := make([]*TreeNode,0)
  out := make([][]int,0)
  queue = append(queue, root)

  for len(queue) != 0 {
    l := len(queue)
    arr := make([]int,0)
    for i := 0; i < l; i++ {
      pop := queue[i]
      arr = append(arr, pop.Val)
      if pop.Left != nil {
        queue = append(queue, pop.Left)
      }
      if pop.Right != nil {
        queue = append(queue, pop.Right)
      }
    }
    out = append(out, arr)
    queue = queue[l:]
  }

  out2 := make([][]int, len(out))
  for i := 0; i < len(out); i++ {
    out2[len(out)-1-i] = out[i]
  }

  return out2
}
```

```go

// 递归
func levelOrderBottom(root *TreeNode) [][]int {
  result := make([][]int, 0)
  level := 0
  if root == nil {
    return result
  }

  orderBottom(root, &result, level)

  left, right := 0, len(result)-1
  for left < right {
    result[left], result[right] = result[right], result[left]
    left++
    right--
  }
  return result
}

func orderBottom(root *TreeNode, result *[][]int, level int) {
  if root == nil {
    return
  }
  if len(*result) > level {
    fmt.Println(level, result, root.Val)
    (*result)[level] = append((*result)[level], root.Val)
  } else {
    *result = append(*result, []int{root.Val})
  }
  orderBottom(root.Left, result, level+1)
  orderBottom(root.Right, result, level+1)
}
```

108.将有序数组转换为二叉搜索树

题目

```
将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

示例：

给定有序数组：[-10,-3,0,5,9],

一个可能的答案是：[0,-3,9,-10,null,5]，它可以表示下面这个高度平衡二叉搜索树：
```

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

解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	O(n)	O(log(n))
02	迭代	O(n)	O(n)

```go
// 递归
func sortedArrayToBST(nums []int) *TreeNode {
  if len(nums) == 0 {
    return nil
  }

  mid := len(nums) / 2

  return &TreeNode{
    Val:   nums[mid],
    Left:  sortedArrayToBST(nums[:mid]),
    Right: sortedArrayToBST(nums[mid+1:]),
  }
}

// 迭代
type MyTreeNode struct {
  root  *TreeNode
  start int
  end   int
}

func sortedArrayToBST(nums []int) *TreeNode {
  if len(nums) == 0 {
    return nil
  }

  queue := make([]MyTreeNode, 0)
  root := &TreeNode{Val: 0}
  queue = append(queue, MyTreeNode{root, 0, len(nums)})
  for len(queue) > 0 {
    myRoot := queue[0]
    queue = queue[1:]
```

```
    start := myRoot.start
    end := myRoot.end
    mid := (start + end) / 2
    curRoot := myRoot.root
    curRoot.Val = nums[mid]
    if start < mid {
      curRoot.Left = &TreeNode{Val: 0}
      queue = append(queue, MyTreeNode{curRoot.Left, start, mid})
    }
    if mid+1 < end {
      curRoot.Right = &TreeNode{Val: 0}
      queue = append(queue, MyTreeNode{curRoot.Right, mid + 1, end})
    }
  }
  return root
}
```

110.平衡二叉树(缺少多种思路)

题目

```
给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

    一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过1。

示例 1:

给定二叉树 [3,9,20,null,null,15,7]

    3
   / \
  9  20
    /  \
   15   7

返回 true 。

示例 2:

给定二叉树 [1,2,2,3,3,null,null,4,4]

       1
      / \
     2   2
    / \
   3   3
```

```
  / \
 4   4

返回 false 。
```

解题思路

No.	思路	时间复杂度	空间复杂度
01	递归	O(n)	O(log(n))

```go
func isBalanced(root *TreeNode) bool {
  _, isBalanced := recur(root)
  return isBalanced

}

func recur(root *TreeNode) (int, bool) {
  if root == nil {
    return 0, true
  }

  leftDepth, leftIsBalanced := recur(root.Left)
  if leftIsBalanced == false{
    return 0,false
  }
  rightDepth, rightIsBalanced := recur(root.Right)
  if rightIsBalanced == false{
    return 0,false
  }

  if -1 <= leftDepth-rightDepth &&
    leftDepth-rightDepth <= 1 {
    return max(leftDepth, rightDepth) + 1, true
  }
  return 0, false
}

func max(a, b int) int {
  if a > b {
    return a
  }
  return b
}
```

111.二叉树的最小深度

题目

```
给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。
说明：叶子节点是指没有子节点的节点。

示例：
给定二叉树 [3,9,20,null,null,15,7],
    3
   / \
  9  20
    /  \
   15   7
返回它的最小深度  2.
```

解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	O(n)	O(log(n))
02	广度优先	O(n)	O(n)

```go
// 递归
func minDepth(root *TreeNode) int {
  if root == nil {
    return 0
  } else if root.Left == nil {
    return 1 + minDepth(root.Right)
  } else if root.Right == nil {
    return 1 + minDepth(root.Left)
  } else {
    return 1 + min(minDepth(root.Left), minDepth(root.Right))
  }
}

func min(a, b int) int {
  if a < b {
    return a
  }
  return b
}

// 广度优先搜索
func minDepth(root *TreeNode) int {
  if root == nil{
```

```
    return 0
  }

  list := make([]*TreeNode,0)
  list = append(list,root)
  depth := 1

  for len(list) > 0{
    length := len(list)
    for i := 0; i < length; i++{
      node := list[0]
      list = list[1:]
      if node.Left == nil && node.Right == nil{
        return depth
      }
      if node.Left != nil{
        list = append(list,node.Left)
      }
      if node.Right != nil{
        list = append(list,node.Right)
      }
    }
    depth++
  }
  return depth
}
```

112.路径总和

题目

给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

说明：叶子节点是指没有子节点的节点。

示例：

给定如下二叉树，以及目标和 sum = 22，

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \      \
        7    2      1
```

返回 true, 因为存在目标和为 22 的根节点到叶子节点的路径 5->4->11->2。

解题思路

No.	思路	时间复杂度	空间复杂度
01(最优)	递归	O(n)	O(log(n))
02	迭代	O(n)	O(n)

```golang
// 递归
func hasPathSum(root *TreeNode, sum int) bool {
  if root == nil {
    return false
  }
  sum = sum - root.Val
  if root.Left == nil && root.Right == nil {
    return sum == 0
  }
  return hasPathSum(root.Left, sum) || hasPathSum(root.Right, sum)
}

// 迭代
func hasPathSum(root *TreeNode, sum int) bool {
  if root == nil {
    return false
  }
  list1 := list.New()
  list2 := list.New()

  list1.PushFront(root)
  list2.PushFront(sum - root.Val)
  for list1.Len() > 0 {
    length := list1.Len()

    for i := 0; i < length; i++ {
      node := list1.Remove(list1.Back()).(*TreeNode)
      currentSum := list2.Remove(list2.Back()).(int)
      if node.Left == nil && node.Right == nil && currentSum == 0 {
        return true
      }
      if node.Left != nil {
        list1.PushFront(node.Left)
        list2.PushFront(currentSum - node.Left.Val)
      }
      if node.Right != nil {
        list1.PushFront(node.Right)
        list2.PushFront(currentSum - node.Right.Val)
      }
    }
  }
  return false
}
```

118.杨辉三角

题目

```
给定一个非负整数 numRows，生成杨辉三角的前 numRows 行。
在杨辉三角中，每个数是它左上方和右上方的数的和。

示例：
输入：5
输出：
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

解题思路

No.	思路	时间复杂度	空间复杂度
01	动态规划	O(n^2)	O(n^2)
02(最优)	递推	O(n^2)	O(n^2)

```go
// 动态规划
func generate(numRows int) [][]int {
  var result [][]int
  for i := 0; i < numRows; i++ {
    var row []int
    for j := 0; j <= i; j++ {
      tmp := 1
      if j == 0 || j == i {

      } else {
        tmp = result[i-1][j-1] + result[i-1][j]
      }
      row = append(row, tmp)
    }
    result = append(result, row)
  }
  return result
}

// 递推
```

```go
func generate(numRows int) [][]int {
  res := make([][]int, 0)
  if numRows == 0 {
    return res
  }

  res = append(res, []int{1})
  if numRows == 1 {
    return res
  }

  for i := 1; i < numRows; i++ {
    res = append(res, genNext(res[i-1]))
  }
  return res
}

func genNext(p []int) []int {
  res := make([]int, 1, len(p)+1)
  res = append(res, p...)

  for i := 0; i < len(res)-1; i++ {
    res[i] = res[i] + res[i+1]
  }
  return res
}
```

119.杨辉三角II

题目

```
给定一个非负索引 k，其中 k ≤ 33，返回杨辉三角的第 k 行。
在杨辉三角中，每个数是它左上方和右上方的数的和。

示例:
输入: 3
输出: [1,3,3,1]

进阶:
你可以优化你的算法到 O(k) 空间复杂度吗?
```

解题思路

No.	思路	时间复杂度	空间复杂度
01	动态规划	O(n^2)	O(n^2)
02	递推	O(n^2)	O(n)
03(最优)	二项式定理	O(n)	O(n)

```go
// 动态规划
func getRow(rowIndex int) []int {
  var result [][]int
  for i := 0; i < rowIndex+1; i++ {
    var row []int
    for j := 0; j <= i; j++ {
      tmp := 1
      if j == 0 || j == i {

      } else {
        tmp = result[i-1][j-1] + result[i-1][j]
      }
      row = append(row, tmp)
    }
    result = append(result, row)
  }
  return result[rowIndex]
}

// 递推
func getRow(rowIndex int) []int {
  res := make([]int,1,rowIndex+1)
  res[0] = 1
  if rowIndex == 0{
    return res
  }

  for i := 0; i < rowIndex; i++{
    res = append(res,1)
    for j := len(res) -2 ; j > 0; j--{
      res[j] = res[j] + res[j-1]
    }

  }
  return res
}

// 二项式定理
func getRow(rowIndex int) []int {
  res := make([]int,rowIndex+1)
  res[0] = 1
```

```
  if rowIndex == 0{
    return res
  }

  // 公式
  // C(n,k) = n! /(k! * (n-k)!)
  // C(n,k) = (n-k+1)/k * C(n,k-1)
  for i := 1; i <= rowIndex; i++{
    res[i] = res[i-1] * (rowIndex-i+1)/i
  }
  return res
}
```

121.买卖股票的最佳时机

题目

```
给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。

注意你不能在买入股票前卖出股票。

示例 1:

输入: [7,1,5,3,6,4]
输出: 5
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格。

示例 2:

输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。
```

解题思路

No.	思路	时间复杂度	空间复杂度
01	暴力法	O(n^2)	O(1)
02(最优)	动态规划(从前到后) 最大利润=max{前一天最大利润, 今天的价格 - 之前最低价格}	O(n)	O(1)
03	动态规划(从后到前)	O(n)	O(1)

```go
// 暴力法
func maxProfit(prices []int) int {
  max := 0
  length := len(prices)

  for i := 0; i < length-1 ; i++{
    for j := i+1; j <= length-1; j++{
      if prices[j] - prices[i] > max{
        max = prices[j] - prices[i]
      }
    }
  }
  return max
}

// 动态规划(从前到后)
func maxProfit(prices []int) int {
  if len(prices) < 2 {
    return 0
  }

  min := prices[0]
  profit := 0

  for i := 1; i < len(prices); i++ {
    if prices[i] < min {
      min = prices[i]
    }
    if profit < prices[i]-min {
      profit = prices[i] - min
    }
  }
  return profit
}


// 动态规划(从后到前)
func maxProfit(prices []int) int {
```

```
  if len(prices) < 2 {
    return 0
  }

  max := 0
  profit := 0

  for i := len(prices) - 1; i >= 0; i-- {
    if max < prices[i] {
      max = prices[i]
    }
    if profit < max-prices[i] {
      profit = max - prices[i]
    }
  }

  return profit
}
```