1. Two Sum (#1)

Problem Statement

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Input Format: An array of integers nums and an integer target.

Output Format: An array of two integers representing the indices.

Examples:

- **Example 1:**
- Input: nums = [2,7,11,15], target = 9
- Output: [0,1]
- Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
- **Example 2:**

Input: nums = [3,2,4], target = 6

Output: [1,2]

- **Example 3:**

Input: nums = [3,3], target = 6

Output: [0,1]

Constraints:

- 2 <= nums.length <= 10^4
- -10^9 <= nums[i] <= 10^9
- -10^9 <= target <= 10^9
- Only one valid answer exists.

Answer Explanation

We use a hash map to store each number and its index as we iterate through the array. For each number, we calculate the complement (target - current number) and check if it exists in the map. If it does, we return the indices of the current number and its complement. If not, we add the current number and its index to the map. This approach ensures O(n) time complexity and O(n) space complexity, efficient for the given constraints.

Coding Solution in Golang

```
package main

import "fmt"

// twoSum returns indices of two numbers that add up to target.
// It uses a map to store numbers and their indices for O(1) lookups.
func twoSum(nums []int, target int) []int {
    m := make(map[int]int) // Map to store num -> index
    for i, num := range nums {
        complement := target - num
        if idx, ok := m[complement]; ok {
            return []int{idx, i} // Found the pair
        }
        m[num] = i // Add current num to map
    }
    return nil // Assumed always one solution, but return nil otherwise
}

func main() {
    // Example 1
    nums1 := []int{2, 7, 11, 15}
    target1 := 9
    fmt.Printf("Input: nums = %v, target = %d\n", nums1, target1)
    fmt.Printf("Output: %v\n\n", twoSum(nums1, target1))

    // Example 2
    nums2 := []int{3, 2, 4}
    target2 := 6
    fmt.Printf("Input: nums = %v, target = %d\n", nums2, target2)
    fmt.Printf("Output: %v\n\n", twoSum(nums2, target2))
```

```
    // Example 3
    nums3 := []int{3, 3}
    target3 := 6
    fmt.Printf("Input: nums = %v, target = %d\n", nums3, target3)
    fmt.Printf("Output: %v\n\n", twoSum(nums3, target3))
}
```

2. Valid Parentheses (#20)

Problem Statement

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

Input Format: A string s.

Output Format: A boolean (true if valid, false otherwise).

Examples:

- **Example 1:**

Input: s = "()"

Output: true

- **Example 2:**

Input: s = "()[]{}"

Output: true

- **Example 3:**

Input: s = "(]"

Output: false

Constraints:

- 1 <= s.length <= 10^4
- s consists of parentheses only '()[]{}'.

Answer Explanation

We use a stack to track opening brackets. As we iterate through the string, we push opening brackets onto the stack. For closing brackets, we pop the top of the stack and check if it matches the corresponding opening bracket. If the stack is empty when we encounter a closing bracket or there's a mismatch, the string is invalid. At the end, the stack should be empty for validity. This is O(n) time and O(n) space.

Coding Solution in Golang

```
package main

import "fmt"

// isValid checks if the parentheses in s are valid using a stack.
func isValid(s string) bool {
    stack := []rune{} // Stack to hold opening brackets
    m := map[rune]rune{ // Mapping closing to opening
        ')': '(',
        ']': '[',
        '}': '{',
    }
    for _, char := range s {
        if char == '(' || char == '[' || char == '{' {
            stack = append(stack, char) // Push opening
        } else {
            if len(stack) == 0 || stack[len(stack)-1] != m[char] {
                return false // Mismatch or empty stack
            }
            stack = stack[:len(stack)-1] // Pop
        }
    }
    return len(stack) == 0 // Valid if stack is empty
}
```

```
func main() {
   // Example 1
   s1 := "()"
   fmt.Printf("Input: s = \"%s\"\n", s1)
   fmt.Printf("Output: %t\n\n", isValid(s1))

   // Example 2
   s2 := "()[]{}"
   fmt.Printf("Input: s = \"%s\"\n", s2)
   fmt.Printf("Output: %t\n\n", isValid(s2))

   // Example 3
   s3 := "(]"
   fmt.Printf("Input: s = \"%s\"\n", s3)
   fmt.Printf("Output: %t\n\n", isValid(s3))
}
```

3. Merge Two Sorted Lists (#21)

Problem Statement

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

Input Format: Two heads of sorted linked lists (ListNode).

Output Format: Head of the merged sorted linked list.

Examples:

- **Example 1:**

Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

- **Example 2:**

Input: list1 = [], list2 = []

Output: []

- **Example 3:**

Input: list1 = [], list2 = [0]

Output: [0]

Constraints:

- The number of nodes in both lists is in the range [0, 50].
- -100 <= Node.val <= 100
- Both list1 and list2 are sorted in non-decreasing order.

Answer Explanation

We use a dummy node to build the merged list. We iterate through both lists, comparing node values and appending the smaller one to the merged list. Once one list is exhausted, we append the remaining nodes from the other list. This is O(m + n) time and O(1) space (excluding the output).

Coding Solution in Golang

```
package main

import "fmt"

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

// mergeTwoLists merges two sorted linked lists.
func mergeTwoLists(list1 *ListNode, list2 *ListNode) *ListNode {
   dummy := &ListNode{} // Dummy head
   current := dummy
   p1, p2 := list1, list2
   for p1 != nil && p2 != nil {
```

```
        if p1.Val <= p2.Val {
            current.Next = p1
            p1 = p1.Next
        } else {
            current.Next = p2
            p2 = p2.Next
        }
        current = current.Next
    }
    if p1 != nil {
        current.Next = p1 // Append remaining
    } else {
        current.Next = p2
    }
    return dummy.Next
}

// Helper to build list from slice
func buildList(nums []int) *ListNode {
    if len(nums) == 0 {
        return nil
    }
    head := &ListNode{Val: nums[0]}
    current := head
    for _, val := range nums[1:] {
        current.Next = &ListNode{Val: val}
        current = current.Next
    }
    return head
}

// Helper to print list
func printList(head *ListNode) {
    for head != nil {
        fmt.Printf("%d ", head.Val)
        head = head.Next
    }
    fmt.Println()
}

func main() {
```

```
    // Example 1
    list1_1 := buildList([]int{1, 2, 4})
    list2_1 := buildList([]int{1, 3, 4})
    fmt.Print("Input: list1 = [1,2,4], list2 = [1,3,4]\nOutput: ")
    printList(mergeTwoLists(list1_1, list2_1))

    // Example 2
    list1_2 := buildList([]int{})
    list2_2 := buildList([]int{})
    fmt.Print("Input: list1 = [], list2 = []\nOutput: ")
    printList(mergeTwoLists(list1_2, list2_2))

    // Example 3
    list1_3 := buildList([]int{})
    list2_3 := buildList([]int{0})
    fmt.Print("Input: list1 = [], list2 = [0]\nOutput: ")
    printList(mergeTwoLists(list1_3, list2_3))
}
```

4. Best Time to Buy and Sell Stock (#121)

Problem Statement

You are given an array prices where prices[i] is the price of a given stock on the i-th day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Input Format: An array of integers prices.

Output Format: An integer representing the maximum profit.

Examples:

- **Example 1:**

Input: prices = [7,1,5,3,6,4]

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

- **Example 2:**

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

Constraints:

- 1 <= prices.length <= 10^5
- 0 <= prices[i] <= 10^4

Answer Explanation

We track the minimum price seen so far and the maximum profit. Iterate through the prices, updating the min price if a lower one is found, and calculating the potential profit (current - min). Update max profit if higher. O(n) time, O(1) space.

Coding Solution in Golang

```
package main

import "fmt"

// maxProfit finds the maximum profit from buying and selling stock.
func maxProfit(prices []int) int {
  if len(prices) == 0 {
    return 0
  }
  minPrice := prices[0] // Track min price seen
  maxProf := 0        // Track max profit
  for _, price := range prices[1:] {
    if price < minPrice {
      minPrice = price // Update min
    } else {
      prof := price - minPrice
      if prof > maxProf {
        maxProf = prof // Update max profit
```

```
        }
      }
    }
    return maxProf
}

func main() {
    // Example 1
    prices1 := []int{7, 1, 5, 3, 6, 4}
    fmt.Printf("Input: prices = %v\n", prices1)
    fmt.Printf("Output: %d\n\n", maxProfit(prices1))

    // Example 2
    prices2 := []int{7, 6, 4, 3, 1}
    fmt.Printf("Input: prices = %v\n", prices2)
    fmt.Printf("Output: %d\n\n", maxProfit(prices2))
}
```

5. Valid Palindrome (#125)

Problem Statement

A phrase is a palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string s, return true if it is a palindrome, or false otherwise.

Input Format: A string s.

Output Format: A boolean.

Examples:

- **Example 1:**

Input: s = "A man, a plan, a canal: Panama"

Output: true

Explanation: "amanaplanacanalpanama" is a palindrome.

- **Example 2:**

Input: s = "race a car"

Output: false

Explanation: "raceacar" is not a palindrome.

- **Example 3:**

Input: s = " "

Output: true

Explanation: s is an empty string "" after removing non-alphanumeric characters.

Constraints:

- 1 <= s.length <= 2 * 10^5
- s consists only of printable ASCII characters.

Answer Explanation

Use two pointers from start and end, skipping non-alphanumeric characters, and compare lowercase versions. If mismatch, false. O(n) time, O(1) space.

Coding Solution in Golang

```
package main

import (
   "fmt"
   "unicode"
)

// isPalindrome checks if s is a palindrome ignoring non-alphanum and case.
func isPalindrome(s string) bool {
   left, right := 0, len(s)-1
   for left < right {
      // Skip non-alphanumeric from left
      for left < right && !unicode.IsLetter(rune(s[left])) && !unicode.IsDigit(rune(s[left])) {
         left++
      }
```

```
        // Skip non-alphanumeric from right
        for left < right && !unicode.IsLetter(rune(s[right])) && !unicode.IsDigit(rune(s[right]))
{
            right--
        }
        if left < right {
            if unicode.ToLower(rune(s[left])) != unicode.ToLower(rune(s[right])) {
                return false // Mismatch
            }
            left++
            right--
        }
    }
    return true
}

func main() {
    // Example 1
    s1 := "A man, a plan, a canal: Panama"
    fmt.Printf("Input: s = \"%s\"\n", s1)
    fmt.Printf("Output: %t\n\n", isPalindrome(s1))

    // Example 2
    s2 := "race a car"
    fmt.Printf("Input: s = \"%s\"\n", s2)
    fmt.Printf("Output: %t\n\n", isPalindrome(s2))

    // Example 3
    s3 := " "
    fmt.Printf("Input: s = \"%s\"\n", s3)
    fmt.Printf("Output: %t\n\n", isPalindrome(s3))
}
```

6. Linked List Cycle (#141)

Problem Statement

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer.

Return true if there is a cycle in the linked list. Otherwise, return false.

Input Format: Head of a linked list.

Output Format: Boolean.

Examples:

- **Example 1:**

Input: head = [3,2,0,-4], pos = 1 (cycle to index 1)

Output: true

- **Example 2:**

Input: head = [1,2], pos = 0

Output: true

- **Example 3:**

Input: head = [1], pos = -1 (no cycle)

Output: false

Constraints:

- The number of the nodes in the list is in the range [0, 10^4].
- -10^5 <= Node.val <= 10^5
- pos is -1 or a valid index in the linked-list.

Answer Explanation

Use Floyd's cycle detection with slow and fast pointers. Slow moves one step, fast two. If they meet, there's a cycle. O(n) time, O(1) space.

Coding Solution in Golang

```
package main

import "fmt"
```

```
// ListNode definition.
type ListNode struct {
   Val  int
   Next *ListNode
}

// hasCycle detects cycle using Floyd's algorithm.
func hasCycle(head *ListNode) bool {
   if head == nil {
      return false
   }
   slow, fast := head, head
   for fast != nil && fast.Next != nil {
      slow = slow.Next     // Slow: 1 step
      fast = fast.Next.Next // Fast: 2 steps
      if slow == fast {
         return true // Cycle detected
      }
   }
   return false
}

// Helper to build list with cycle (pos = -1 for no cycle)
func buildListWithCycle(nums []int, pos int) *ListNode {
   if len(nums) == 0 {
      return nil
   }
   head := &ListNode{Val: nums[0]}
   current := head
   var cycleNode *ListNode
   for i, val := range nums[1:] {
      current.Next = &ListNode{Val: val}
      current = current.Next
      if i+1 == pos {
         cycleNode = current
      }
   }
   if pos >= 0 {
      current.Next = cycleNode
   }
```

```
    return head
}

func main() {
    // Example 1
    head1 := buildListWithCycle([]int{3, 2, 0, -4}, 1)
    fmt.Println("Input: [3,2,0,-4] with cycle at pos 1")
    fmt.Printf("Output: %t\n\n", hasCycle(head1))

    // Example 2
    head2 := buildListWithCycle([]int{1, 2}, 0)
    fmt.Println("Input: [1,2] with cycle at pos 0")
    fmt.Printf("Output: %t\n\n", hasCycle(head2))

    // Example 3
    head3 := buildListWithCycle([]int{1}, -1)
    fmt.Println("Input: [1] with no cycle")
    fmt.Printf("Output: %t\n\n", hasCycle(head3))
}
```

7. Maximum Depth of Binary Tree (#104)

Problem Statement

Given the root of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Input Format: Root of binary tree.

Output Format: Integer depth.

Examples:

- **Example 1:**

Input: root = [3,9,20,null,null,15,7]

Output: 3

- **Example 2:**

Input: root = [1,null,2]

Output: 2

Constraints:

- The number of nodes in the tree is in the range [0, 10^4].
- -100 <= Node.val <= 100

Answer Explanation

Use recursion to calculate the max depth of left and right subtrees, then add 1 for the current node. Base case: null node has depth 0. O(n) time, O(h) space (height h).

Coding Solution in Golang

```
package main

import "fmt"

// TreeNode definition.
type TreeNode struct {
   Val   int
   Left  *TreeNode
   Right *TreeNode
}

// maxDepth computes the maximum depth of the binary tree.
func maxDepth(root *TreeNode) int {
   if root == nil {
      return 0 // Base case
   }
   leftDepth := maxDepth(root.Left)
   rightDepth := maxDepth(root.Right)
   if leftDepth > rightDepth {
      return leftDepth + 1
   }
   return rightDepth + 1
}
```

```
// Helper to build tree (level order, null as nil)
func buildTree(vals []interface{}) *TreeNode {
    if len(vals) == 0 || vals[0] == nil {
        return nil
    }
    root := &TreeNode{Val: vals[0].(int)}
    queue := []*TreeNode{root}
    i := 1
    for len(queue) > 0 && i < len(vals) {
        current := queue[0]
        queue = queue[1:]
        if i < len(vals) && vals[i] != nil {
            current.Left = &TreeNode{Val: vals[i].(int)}
            queue = append(queue, current.Left)
        }
        i++
        if i < len(vals) && vals[i] != nil {
            current.Right = &TreeNode{Val: vals[i].(int)}
            queue = append(queue, current.Right)
        }
        i++
    }
    return root
}

func main() {
    // Example 1
    vals1 := []interface{}{3, 9, 20, nil, nil, 15, 7}
    root1 := buildTree(vals1)
    fmt.Println("Input: [3,9,20,null,null,15,7]")
    fmt.Printf("Output: %d\n\n", maxDepth(root1))

    // Example 2
    vals2 := []interface{}{1, nil, 2}
    root2 := buildTree(vals2)
    fmt.Println("Input: [1,null,2]")
    fmt.Printf("Output: %d\n\n", maxDepth(root2))
}
```

8. Binary Search (#704)

Problem Statement

Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.

You must write an algorithm with O(log n) runtime complexity.

Input Format: Sorted array nums, integer target.

Output Format: Index or -1.

Examples:

- **Example 1:**

Input: nums = [-1,0,3,5,9,12], target = 9

Output: 4

- **Example 2:**

Input: nums = [-1,0,3,5,9,12], target = 2

Output: -1

Constraints:

- 1 <= nums.length <= 10^4
- -10^4 < nums[i], target < 10^4
- All the integers in nums are unique.
- nums is sorted in ascending order.

Answer Explanation

Use binary search: Set low and high pointers, find mid, compare with target, adjust pointers. Continue until found or low > high. O(log n) time, O(1) space.

Coding Solution in Golang

package main

```
import "fmt"

// search performs binary search on sorted nums for target.
func search(nums []int, target int) int {
   low, high := 0, len(nums)-1
   for low <= high {
      mid := low + (high-low)/2 // Avoid overflow
      if nums[mid] == target {
         return mid // Found
      } else if nums[mid] < target {
         low = mid + 1 // Search right
      } else {
         high = mid - 1 // Search left
      }
   }
   return -1 // Not found
}

func main() {
   // Example 1
   nums1 := []int{-1, 0, 3, 5, 9, 12}
   target1 := 9
   fmt.Printf("Input: nums = %v, target = %d\n", nums1, target1)
   fmt.Printf("Output: %d\n\n", search(nums1, target1))

   // Example 2
   target2 := 2
   fmt.Printf("Input: nums = %v, target = %d\n", nums1, target2)
   fmt.Printf("Output: %d\n\n", search(nums1, target2))
}
```

9. Flood Fill (#733)

Problem Statement

An image is represented by an m x n integer grid image where image[i][j] represents the pixel value of the image.

You are also given three integers sr, sc, and color. You should perform a flood fill on the image starting from the pixel image[sr][sc].

To perform a flood fill, consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color), and so on. Replace the color of all of the aforementioned pixels with color.

Return the modified image after performing the flood fill.

Input Format: 2D array image, integers sr, sc, color.

Output Format: Modified 2D array.

Examples:

- **Example 1:**

Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2

Output: [[2,2,2],[2,2,0],[2,0,1]]

- **Example 2:**

Input: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0

Output: [[0,0,0],[0,0,0]]

Constraints:

- m == image.length
- n == image[i].length
- 1 <= m, n <= 50
- 0 <= image[i][j], color <= 65535
- 0 <= sr < m
- 0 <= sc < n

Answer Explanation

Use DFS or BFS to visit connected pixels of the same color, changing them to the new color. Start from (sr, sc), avoid revisiting by checking color. O(m*n) time, O(m*n) space for recursion/stack.

Coding Solution in Golang

```
package main

import "fmt"

// floodFill performs flood fill starting from (sr, sc) with new color.
func floodFill(image [][]int, sr int, sc int, color int) [][]int {
   if image[sr][sc] == color {
      return image // No change needed
   }
   rows, cols := len(image), len(image[0])
   oldColor := image[sr][sc]
   var dfs func(r, c int)
   dfs = func(r, c int) {
      if r < 0 || r >= rows || c < 0 || c >= cols || image[r][c] != oldColor {
         return // Out of bounds or wrong color
      }
      image[r][c] = color // Change color
      dfs(r-1, c) // Up
      dfs(r+1, c) // Down
      dfs(r, c-1) // Left
      dfs(r, c+1) // Right
   }
   dfs(sr, sc)
   return image
}

// Helper to print 2D array
func printImage(image [][]int) {
   for _, row := range image {
      fmt.Println(row)
   }
}

func main() {
   // Example 1
   image1 := [][]int{{1, 1, 1}, {1, 1, 0}, {1, 0, 1}}
   sr1, sc1, color1 := 1, 1, 2
   fmt.Printf("Input: image = %v, sr = %d, sc = %d, color = %d\n", image1, sr1, sc1,
color1)
```

```
    fmt.Println("Output:")
    printImage(floodFill(image1, sr1, sc1, color1))
    fmt.Println()

    // Example 2
    image2 := [][]int{{0, 0, 0}, {0, 0, 0}}
    sr2, sc2, color2 := 0, 0, 0
    fmt.Printf("Input: image = %v, sr = %d, sc = %d, color = %d\n", image2, sr2, sc2,
color2)
    fmt.Println("Output:")
    printImage(floodFill(image2, sr2, sc2, color2))
}
```

10. Lowest Common Ancestor of a BST (#235)

Problem Statement

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).

Input Format: Root of BST, nodes p and q.

Output Format: LCA node.

Examples:

- **Example 1:**

Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

Output: 6

- **Example 2:**

Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4

Output: 2

Constraints:

- The number of nodes in the tree is in the range [2, 10^5].
- -10^9 <= Node.val <= 10^9
- All Node.val are unique.
- p != q
- p and q will exist in the BST.

Answer Explanation

Traverse from root. If both p and q are less than current, go left; if greater, go right; else, current is LCA. O(h) time, O(1) space.

Coding Solution in Golang

```
package main

import "fmt"

// TreeNode definition.
type TreeNode struct {
   Val   int
   Left  *TreeNode
   Right *TreeNode
}

// lowestCommonAncestor finds LCA in BST.
func lowestCommonAncestor(root, p, q *TreeNode) *TreeNode {
   current := root
   for current != nil {
      if p.Val < current.Val && q.Val < current.Val {
         current = current.Left // Both left
      } else if p.Val > current.Val && q.Val > current.Val {
         current = current.Right // Both right
      } else {
         return current // Split or one is current
      }
   }
   return nil // Should not happen
}
```

```
// Helper to build tree (similar to before, but for BST assume input is level order)
func buildTree(vals []interface{}) *TreeNode {
    // Note: For simplicity, assuming the input represents a valid BST in level order.
    if len(vals) == 0 || vals[0] == nil {
        return nil
    }
    root := &TreeNode{Val: vals[0].(int)}
    queue := []*TreeNode{root}
    i := 1
    for len(queue) > 0 && i < len(vals) {
        current := queue[0]
        queue = queue[1:]
        if i < len(vals) && vals[i] != nil {
            current.Left = &TreeNode{Val: vals[i].(int)}
            queue = append(queue, current.Left)
        }
        i++
        if i < len(vals) && vals[i] != nil {
            current.Right = &TreeNode{Val: vals[i].(int)}
            queue = append(queue, current.Right)
        }
        i++
    }
    return root
}

// Find node by value (for p and q)
func findNode(root *TreeNode, val int) *TreeNode {
    if root == nil {
        return nil
    }
    if root.Val == val {
        return root
    }
    if val < root.Val {
        return findNode(root.Left, val)
    }
    return findNode(root.Right, val)
}

func main() {
```

```
    // Example 1
    vals1 := []interface{}{6, 2, 8, 0, 4, 7, 9, nil, nil, 3, 5}
    root1 := buildTree(vals1)
    p1 := findNode(root1, 2)
    q1 := findNode(root1, 8)
    lca1 := lowestCommonAncestor(root1, p1, q1)
    fmt.Println("Input: root = [6,2,8,0,4,7,9,null,null,3,5], p=2, q=8")
    fmt.Printf("Output: %d\n\n", lca1.Val)

    // Example 2
    p2 := findNode(root1, 2)
    q2 := findNode(root1, 4)
    lca2 := lowestCommonAncestor(root1, p2, q2)
    fmt.Println("Input: root = [6,2,8,0,4,7,9,null,null,3,5], p=2, q=4")
    fmt.Printf("Output: %d\n\n", lca2.Val)
}
```

11. Balanced Binary Tree (#110)

Problem Statement

Given a binary tree, determine if it is height-balanced.

A height-balanced binary tree is a binary tree in which the depth of the two subtrees of every node never differs by more than one.

Input Format: Root of binary tree.

Output Format: Boolean.

Examples:

- **Example 1:**

Input: root = [3,9,20,null,null,15,7]

Output: true

- **Example 2:**

Input: root = [1,2,2,3,3,null,null,4,4]

Output: false

- **Example 3:**

Input: root = []

Output: true

Constraints:

- The number of nodes in the tree is in the range [0, 5000].
- -10^4 <= Node.val <= 10^4

Answer Explanation

Recursively calculate height of subtrees. If difference >1 or subtrees not balanced, false. Use -1 to indicate unbalanced. O(n) time, O(h) space.

Coding Solution in Golang

```
package main

import "fmt"

// TreeNode definition.
type TreeNode struct {
   Val   int
   Left  *TreeNode
   Right *TreeNode
}

// isBalanced checks if the tree is height-balanced.
func isBalanced(root *TreeNode) bool {
   var getHeight func(node *TreeNode) int
   getHeight = func(node *TreeNode) int {
      if node == nil {
         return 0
      }
      left := getHeight(node.Left)
      right := getHeight(node.Right)
      if left == -1 || right == -1 || abs(left-right) > 1 {
         return -1 // Unbalanced
```

```
        }
        if left > right {
            return left + 1
        }
        return right + 1
    }
    return getHeight(root) != -1
}

// abs helper
func abs(x int) int {
    if x < 0 {
        return -x
    }
    return x
}

// buildTree from earlier
func buildTree(vals []interface{}) *TreeNode {
    if len(vals) == 0 || vals[0] == nil {
        return nil
    }
    root := &TreeNode{Val: vals[0].(int)}
    queue := []*TreeNode{root}
    i := 1
    for len(queue) > 0 && i < len(vals) {
        current := queue[0]
        queue = queue[1:]
        if i < len(vals) && vals[i] != nil {
            current.Left = &TreeNode{Val: vals[i].(int)}
            queue = append(queue, current.Left)
        }
        i++
        if i < len(vals) && vals[i] != nil {
            current.Right = &TreeNode{Val: vals[i].(int)}
            queue = append(queue, current.Right)
        }
        i++
    }
    return root
}
```

```
func main() {
   // Example 1
   vals1 := []interface{}{3, 9, 20, nil, nil, 15, 7}
   root1 := buildTree(vals1)
   fmt.Println("Input: [3,9,20,null,null,15,7]")
   fmt.Printf("Output: %t\n\n", isBalanced(root1))

   // Example 2
   vals2 := []interface{}{1, 2, 2, 3, 3, nil, nil, 4, 4}
   root2 := buildTree(vals2)
   fmt.Println("Input: [1,2,2,3,3,null,null,4,4]")
   fmt.Printf("Output: %t\n\n", isBalanced(root2))

   // Example 3
   root3 := buildTree([]interface{}{})
   fmt.Println("Input: []")
   fmt.Printf("Output: %t\n\n", isBalanced(root3))
}
```

12. Implement Queue using Stacks (#232)

Problem Statement

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, pop, peek, and empty).

Implement the MyQueue class:

- void push(int x) Pushes element x to the back of the queue.
- int pop() Removes the element from the front of the queue and returns it.
- int peek() Returns the element at the front of the queue.
- boolean empty() Returns true if the queue is empty, false otherwise.

Input Format: Series of operations.

Output Format: Results of operations.

Examples:

- MyQueue myQueue = new MyQueue();

```
myQueue.push(1);
myQueue.push(2);
myQueue.peek(); // return 1
myQueue.pop(); // return 1
myQueue.empty(); // return false
```

Constraints:

- 1 <= x <= 9
- At most 100 calls will be made to push, pop, peek, and empty.
- All the calls to pop and peek are valid.

Answer Explanation

Use two stacks: input for push, output for pop/peek. When output is empty, transfer from input by popping and pushing to output (reverses order). Push O(1), pop/peek amortized O(1).

Coding Solution in Golang

```
package main

import "fmt"

// MyQueue implements queue using two stacks.
type MyQueue struct {
  input  []int // For push
  output []int // For pop/peek
}

func Constructor() MyQueue {
  return MyQueue{}
}

// push adds x to back.
func (q *MyQueue) Push(x int) {
  q.input = append(q.input, x)
}

// pop removes front and returns it.
```

```
func (q *MyQueue) Pop() int {
   q.Peek() // Ensure output has front
   if len(q.output) == 0 {
      return -1 // Invalid, but per constraints valid
   }
   front := q.output[len(q.output)-1]
   q.output = q.output[:len(q.output)-1]
   return front
}

// peek returns front.
func (q *MyQueue) Peek() int {
   if len(q.output) == 0 {
      for len(q.input) > 0 {
         q.output = append(q.output, q.input[len(q.input)-1])
         q.input = q.input[:len(q.input)-1]
      }
   }
   if len(q.output) == 0 {
      return -1 // Empty
   }
   return q.output[len(q.output)-1]
}

// empty checks if queue is empty.
func (q *MyQueue) Empty() bool {
   return len(q.input) == 0 && len(q.output) == 0
}

func main() {
   q := Constructor()
   q.Push(1)
   q.Push(2)
   fmt.Printf("Peek: %d\n", q.Peek())   // 1
   fmt.Printf("Pop: %d\n", q.Pop())     // 1
   fmt.Printf("Empty: %t\n", q.Empty()) // false
}
```

13. Min Stack (#155)

Problem Statement

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.

Input Format: Series of operations.

Output Format: Results.

Examples:

- MinStack minStack = new MinStack();

```
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top(); // return 0
minStack.getMin(); // return -2
```

Constraints:

- -2^31 <= val <= 2^31 - 1
- Methods pop, top and getMin operations will always be called on non-empty stacks.
- At most 3 * 10^4 calls will be made to push, pop, top, and getMin.

Answer Explanation

Use two stacks: one for values, one for mins. On push, push val to stack, and push min(current min, val) to min stack. Pop pops both. Top from stack, min from min stack top. O(1) for all, O(n) space.

Coding Solution in Golang

```
package main

import "fmt"

// MinStack supports min in O(1).
type MinStack struct {
   stack []int // Values
   min   []int // Mins
}

func Constructor() MinStack {
   return MinStack{}
}

// push adds val.
func (s *MinStack) Push(val int) {
   s.stack = append(s.stack, val)
   if len(s.min) == 0 || val <= s.min[len(s.min)-1] {
      s.min = append(s.min, val) // New min
   } else {
      s.min = append(s.min, s.min[len(s.min)-1]) // Current min
   }
}

// pop removes top.
func (s *MinStack) Pop() {
   if len(s.stack) > 0 {
      s.stack = s.stack[:len(s.stack)-1]
      s.min = s.min[:len(s.min)-1]
   }
}

// top returns top.
func (s *MinStack) Top() int {
   if len(s.stack) == 0 {
      return -1 // Invalid
   }
   return s.stack[len(s.stack)-1]
}
```

```
// getMin returns min.
func (s *MinStack) GetMin() int {
   if len(s.min) == 0 {
      return -1 // Invalid
   }
   return s.min[len(s.min)-1]
}

func main() {
   ms := Constructor()
   ms.Push(-2)
   ms.Push(0)
   ms.Push(-3)
   fmt.Printf("GetMin: %d\n", ms.GetMin()) // -3
   ms.Pop()
   fmt.Printf("Top: %d\n", ms.Top())      // 0
   fmt.Printf("GetMin: %d\n", ms.GetMin()) // -2
}
```

14. Invert Binary Tree (#226)

Problem Statement

Given the root of a binary tree, invert the tree, and return its root.

Input Format: Root of binary tree.

Output Format: Inverted root.

Examples:

- **Example 1:**

Input: root = [4,2,7,1,3,6,9]

Output: [4,7,2,9,6,3,1]

- **Example 2:**

Input: root = [2,1,3]

Output: [2,3,1]

- **Example 3:**

Input: root = []

Output: []

Constraints:

- The number of nodes in the tree is in the range [0, 100].
- -100 <= Node.val <= 100

Answer Explanation

Recursively swap left and right subtrees for each node. Base case null. O(n) time, O(h) space.

Coding Solution in Golang

```
package main

import "fmt"

// TreeNode definition.
type TreeNode struct {
   Val   int
   Left  *TreeNode
   Right *TreeNode
}

// invertTree inverts the binary tree.
func invertTree(root *TreeNode) *TreeNode {
   if root == nil {
      return nil
   }
   root.Left, root.Right = root.Right, root.Left // Swap
   invertTree(root.Left)
   invertTree(root.Right)
   return root
}
```

```
// buildTree from earlier
func buildTree(vals []interface{}) *TreeNode {
    if len(vals) == 0 || vals[0] == nil {
        return nil
    }
    root := &TreeNode{Val: vals[0].(int)}
    queue := []*TreeNode{root}
    i := 1
    for len(queue) > 0 && i < len(vals) {
        current := queue[0]
        queue = queue[1:]
        if i < len(vals) && vals[i] != nil {
            current.Left = &TreeNode{Val: vals[i].(int)}
            queue = append(queue, current.Left)
        }
        i++
        if i < len(vals) && vals[i] != nil {
            current.Right = &TreeNode{Val: vals[i].(int)}
            queue = append(queue, current.Right)
        }
        i++
    }
    return root
}

// printTree level order
func printTree(root *TreeNode) {
    if root == nil {
        fmt.Println("[]")
        return
    }
    queue := []*TreeNode{root}
    vals := []int{}
    for len(queue) > 0 {
        current := queue[0]
        queue = queue[1:]
        vals = append(vals, current.Val)
        if current.Left != nil {
            queue = append(queue, current.Left)
        }
        if current.Right != nil {
```

```
            queue = append(queue, current.Right)
        }
    }
    fmt.Println(vals)
}

func main() {
    // Example 1
    vals1 := []interface{}{4, 2, 7, 1, 3, 6, 9}
    root1 := buildTree(vals1)
    fmt.Println("Input: [4,2,7,1,3,6,9]")
    inverted1 := invertTree(root1)
    fmt.Print("Output: ")
    printTree(inverted1)
    fmt.Println()

    // Example 2
    vals2 := []interface{}{2, 1, 3}
    root2 := buildTree(vals2)
    fmt.Println("Input: [2,1,3]")
    inverted2 := invertTree(root2)
    fmt.Print("Output: ")
    printTree(inverted2)
    fmt.Println()

    // Example 3
    root3 := buildTree([]interface{}{})
    fmt.Println("Input: []")
    inverted3 := invertTree(root3)
    fmt.Print("Output: ")
    printTree(inverted3)
}
```

Medium Problems

15. Product of Array Except Self (#238)

Problem Statement

Given an integer array nums, return an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i].

The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

Input Format: Array nums.

Output Format: Array of products.

Examples:

- **Example 1:**

Input: nums = [1,2,3,4]

Output: [24,12,8,6]

- **Example 2:**

Input: nums = [-1,1,0,-3,3]

Output: [0,0,9,0,0]

Constraints:

- 2 <= nums.length <= 10^5
- -30 <= nums[i] <= 30
- The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

Answer Explanation

Use two passes: first for left products, second for right products multiplied to left. Handles zeros implicitly. O(n) time, O(n) space (can optimize to O(1) with output array).

Coding Solution in Golang

```
package main

import "fmt"

// productExceptSelf computes product except self without division.
func productExceptSelf(nums []int) []int {
   n := len(nums)
   answer := make([]int, n)
   // Left products
   left := 1
   for i := 0; i < n; i++ {
      answer[i] = left
      left *= nums[i]
   }
   // Right products multiplied
   right := 1
   for i := n - 1; i >= 0; i-- {
      answer[i] *= right
      right *= nums[i]
   }
   return answer
}

func main() {
   // Example 1
   nums1 := []int{1, 2, 3, 4}
   fmt.Printf("Input: %v\n", nums1)
   fmt.Printf("Output: %v\n\n", productExceptSelf(nums1))

   // Example 2
   nums2 := []int{-1, 1, 0, -3, 3}
   fmt.Printf("Input: %v\n", nums2)
   fmt.Printf("Output: %v\n\n", productExceptSelf(nums2))
}
```

16. Maximum Subarray (Kadane's) (#53)

Problem Statement

Given an integer array nums, find the subarray with the largest sum, and return its sum.

Input Format: Array nums.

Output Format: Integer max sum.

Examples:

- **Example 1:**

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: The subarray [4,-1,2,1] has the largest sum 6.

- **Example 2:**

Input: nums = [1]

Output: 1

- **Example 3:**

Input: nums = [5,4,-1,7,8]

Output: 23

Constraints:

- 1 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4

Answer Explanation

Kadane's algorithm: Track current sum, reset if negative. Update max sum. O(n) time, O(1) space.

Coding Solution in Golang

```
package main

import "fmt"

// maxSubArray finds max subarray sum using Kadane's.
func maxSubArray(nums []int) int {
   if len(nums) == 0 {
      return 0
   }
   maxSum := nums[0]
   current := nums[0]
   for _, num := range nums[1:] {
      if current < 0 {
         current = num // Reset if negative
      } else {
         current += num
      }
      if current > maxSum {
         maxSum = current
      }
   }
   return maxSum
}

func main() {
   // Example 1
   nums1 := []int{-2, 1, -3, 4, -1, 2, 1, -5, 4}
   fmt.Printf("Input: %v\n", nums1)
   fmt.Printf("Output: %d\n\n", maxSubArray(nums1))

   // Example 2
   nums2 := []int{1}
   fmt.Printf("Input: %v\n", nums2)
   fmt.Printf("Output: %d\n\n", maxSubArray(nums2))

   // Example 3
   nums3 := []int{5, 4, -1, 7, 8}
   fmt.Printf("Input: %v\n", nums3)
   fmt.Printf("Output: %d\n\n", maxSubArray(nums3))
```

}

17. 3Sum (#15)

Problem Statement

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

Input Format: Array nums.

Output Format: 2D array of triplets.

Examples:

- **Example 1:**

Input: nums = [-1,0,1,2,-1,-4]

Output: [[-1,-1,2],[-1,0,1]]

- **Example 2:**

Input: nums = [0,1,1]

Output: []

- **Example 3:**

Input: nums = [0,0,0]

Output: [[0,0,0]]

Constraints:

- 0 <= nums.length <= 3000
- -10^5 <= nums[i] <= 10^5

Answer Explanation

Sort the array. For each i, use two pointers for j and k to find sum = -nums[i]. Skip duplicates. O(n^2) time, O(1) space (excluding output).

Coding Solution in Golang

```
package main

import (
   "fmt"
   "sort"
)

// threeSum finds unique triplets that sum to 0.
func threeSum(nums []int) [][]int {
   sort.Ints(nums) // Sort
   result := [][]int{}
   n := len(nums)
   for i := 0; i < n-2; i++ {
      if i > 0 && nums[i] == nums[i-1] {
         continue // Skip duplicate i
      }
      left, right := i+1, n-1
      for left < right {
         sum := nums[i] + nums[left] + nums[right]
         if sum == 0 {
            result = append(result, []int{nums[i], nums[left], nums[right]})
            left++
            right--
            // Skip duplicates
            for left < right && nums[left] == nums[left-1] {
               left++
            }
            for left < right && nums[right] == nums[right+1] {
               right--
            }
         } else if sum < 0 {
            left++
         } else {
            right--
```

```
        }
      }
    }
    return result
}

func main() {
    // Example 1
    nums1 := []int{-1, 0, 1, 2, -1, -4}
    fmt.Printf("Input: %v\n", nums1)
    fmt.Printf("Output: %v\n\n", threeSum(nums1))

    // Example 2
    nums2 := []int{0, 1, 1}
    fmt.Printf("Input: %v\n", nums2)
    fmt.Printf("Output: %v\n\n", threeSum(nums2))

    // Example 3
    nums3 := []int{0, 0, 0}
    fmt.Printf("Input: %v\n", nums3)
    fmt.Printf("Output: %v\n\n", threeSum(nums3))
}
```

18. Container With Most Water (#11)

Problem Statement

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return the maximum amount of water a container can store.

Input Format: Array height.

Output Format: Integer max area.

Examples:

- **Example 1:**

Input: height = [1,8,6,2,5,4,8,3,7]

Output: 49

- **Example 2:**

Input: height = [1,1]

Output: 1

Constraints:

- n == height.length
- 2 <= n <= 10^5
- 0 <= height[i] <= 10^4

Answer Explanation

Use two pointers from ends. Calculate area = min(height[l], height[r]) * (r - l). Move the smaller height pointer. O(n) time, O(1) space.

Coding Solution in Golang

```
package main

import "fmt"

// maxArea finds max container area.
func maxArea(height []int) int {
   left, right := 0, len(height)-1
   maxA := 0
   for left < right {
      h := min(height[left], height[right])
      area := h * (right - left)
      if area > maxA {
         maxA = area
      }
      if height[left] < height[right] {
         left++
      } else {
         right--
```

```
        }
    }
    return maxA
}

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

func main() {
    // Example 1
    height1 := []int{1, 8, 6, 2, 5, 4, 8, 3, 7}
    fmt.Printf("Input: %v\n", height1)
    fmt.Printf("Output: %d\n\n", maxArea(height1))

    // Example 2
    height2 := []int{1, 1}
    fmt.Printf("Input: %v\n", height2)
    fmt.Printf("Output: %d\n\n", maxArea(height2))
}
```

19. Longest Substring Without Repeating Characters (#3)

Problem Statement

Given a string s, find the length of the longest substring without repeating characters.

Input Format: String s.

Output Format: Integer length.

Examples:

- **Example 1:**

Input: s = "abcabcbb"

Output: 3

Explanation: The answer is "abc", with the length of 3.

- **Example 2:**

Input: s = "bbbbb"

Output: 1

- **Example 3:**

Input: s = "pwwkew"

Output: 3

Constraints:

- 0 <= s.length <= 5 * 10^4
- s consists of English letters, digits, symbols and spaces.

Answer Explanation

Use sliding window with set for characters. Expand right, if duplicate, remove left until no duplicate. Track max length. O(n) time, O(min(n, charset)) space.

Coding Solution in Golang

```
package main

import "fmt"

// lengthOfLongestSubstring finds longest non-repeating substring length.
func lengthOfLongestSubstring(s string) int {
   charSet := make(map[rune]bool)
   left := 0
   maxLen := 0
   for right, char := range s {
      for charSet[char] {
         delete(charSet, rune(s[left]))
         left++
      }
      charSet[char] = true
```

```
        if right-left+1 > maxLen {
            maxLen = right - left + 1
        }
    }
    return maxLen
}

func main() {
    // Example 1
    s1 := "abcabcbb"
    fmt.Printf("Input: \"%s\"\n", s1)
    fmt.Printf("Output: %d\n\n", lengthOfLongestSubstring(s1))

    // Example 2
    s2 := "bbbbb"
    fmt.Printf("Input: \"%s\"\n", s2)
    fmt.Printf("Output: %d\n\n", lengthOfLongestSubstring(s2))

    // Example 3
    s3 := "pwwkew"
    fmt.Printf("Input: \"%s\"\n", s3)
    fmt.Printf("Output: %d\n\n", lengthOfLongestSubstring(s3))
}
```

20. Top K Frequent Elements (#347)

Problem Statement

Given an integer array nums and an integer k, return the k most frequent elements. You may return the answer in any order.

Input Format: Array nums, integer k.

Output Format: Array of k elements.

Examples:

- **Example 1:**

Input: nums = [1,1,1,2,2,3], k = 2

Output: [1,2]

- **Example 2:**

Input: nums = [1], k = 1

Output: [1]

Constraints:

- 1 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4
- k is in the range [1, the number of unique elements in the array].
- It is guaranteed that the answer is unique.

Answer Explanation

Use map for frequency. Use bucket sort: array of lists where index is frequency. Collect from high to low until k. O(n) time, O(n) space.

Coding Solution in Golang

```
package main

import "fmt"

// topKFrequent returns k most frequent elements.
func topKFrequent(nums []int, k int) []int {
   freq := make(map[int]int)
   for _, num := range nums {
      freq[num]++
   }
   buckets := make([][]int, len(nums)+1)
   for num, f := range freq {
      buckets[f] = append(buckets[f], num)
   }
   result := []int{}
   for i := len(buckets) - 1; i >= 0 && len(result) < k; i-- {
      for _, num := range buckets[i] {
         result = append(result, num)
         if len(result) == k {
            return result
```

```
        }
      }
    }
    return result
}

func main() {
    // Example 1
    nums1 := []int{1, 1, 1, 2, 2, 3}
    k1 := 2
    fmt.Printf("Input: nums = %v, k = %d\n", nums1, k1)
    fmt.Printf("Output: %v\n\n", topKFrequent(nums1, k1))

    // Example 2
    nums2 := []int{1}
    k2 := 1
    fmt.Printf("Input: nums = %v, k = %d\n", nums2, k2)
    fmt.Printf("Output: %v\n\n", topKFrequent(nums2, k2))
}
```

21. Number of Islands (#200)

Problem Statement

Given a 2D binary grid grid which represents a map of '1's (land) and '0's (water), return the number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Input Format: 2D array grid.

Output Format: Integer number of islands.

Examples:

- **Example 1:**

Input: grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]

Output: 1

- **Example 2:**

Input: grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]

]

Output: 3

Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 300
- grid[i][j] is '0' or '1'.

Answer Explanation

Iterate through grid. When '1' found, increment count and DFS/BFS to mark all connected '1's as '0'. O(m*n) time, O(m*n) space for recursion.

Coding Solution in Golang

package main

import "fmt"

```
// numIslands counts number of islands.
func numIslands(grid [][]byte) int {
    if len(grid) == 0 {
        return 0
    }
    rows, cols := len(grid), len(grid[0])
    count := 0
    var dfs func(r, c int)
    dfs = func(r, c int) {
        if r < 0 || r >= rows || c < 0 || c >= cols || grid[r][c] == '0' {
            return
        }
        grid[r][c] = '0' // Mark visited
        dfs(r-1, c)
        dfs(r+1, c)
        dfs(r, c-1)
        dfs(r, c+1)
    }
    for r := 0; r < rows; r++ {
        for c := 0; c < cols; c++ {
            if grid[r][c] == '1' {
                count++
                dfs(r, c)
            }
        }
    }
    return count
}

func main() {
    // Example 1
    grid1 := [][]byte{
        {'1', '1', '1', '1', '0'},
        {'1', '1', '0', '1', '0'},
        {'1', '1', '0', '0', '0'},
        {'0', '0', '0', '0', '0'},
    }
    fmt.Println("Input: grid with 1 island")
    fmt.Printf("Output: %d\n\n", numIslands(grid1))
```

```
    // Example 2
    grid2 := [][]byte{
        {'1', '1', '0', '0', '0'},
        {'1', '1', '0', '0', '0'},
        {'0', '0', '1', '0', '0'},
        {'0', '0', '0', '1', '1'},
    }
    fmt.Println("Input: grid with 3 islands")
    fmt.Printf("Output: %d\n\n", numIslands(grid2))
}
```

22. Clone Graph (#133)

Problem Statement

Given a reference of a node in a connected undirected graph.

Return a deep copy (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

Input Format: Node in graph.

Output Format: Cloned node.

Examples:

Assume graph with nodes 1-4 connected as adj list.

Constraints:

- The number of nodes in the graph is in the range [1, 100].
- 1 <= Node.val <= 100
- Node.val is unique for each node.
- There are no repeated edges and no self-loops in the graph.
- The Graph is connected and all nodes can be visited starting from the given node.

Answer Explanation

Use DFS or BFS with map to track cloned nodes. Clone node, recurse on neighbors. O(n + e) time, O(n) space.

Coding Solution in Golang

```
package main

import "fmt"

// Node definition.
type Node struct {
   Val      int
   Neighbors []*Node
}

// cloneGraph clones the graph.
func cloneGraph(node *Node) *Node {
   if node == nil {
      return nil
   }
   visited := make(map[*Node]*Node)
   var dfs func(n *Node) *Node
   dfs = func(n *Node) *Node {
      if clone, ok := visited[n]; ok {
         return clone // Already cloned
      }
      clone := &Node{Val: n.Val}
      visited[n] = clone
      for _, neigh := range n.Neighbors {
         clone.Neighbors = append(clone.Neighbors, dfs(neigh))
      }
      return clone
   }
   return dfs(node)
}

// Helper to build graph from adj list
func buildGraph(adj [][]int) *Node {
   if len(adj) == 0 {
```

```
        return nil
    }
    nodes := make([]*Node, len(adj))
    for i := range nodes {
        nodes[i] = &Node{Val: i + 1} // Assume 1-indexed
    }
    for i, neighbors := range adj {
        for _, neigh := range neighbors {
            nodes[i].Neighbors = append(nodes[i].Neighbors, nodes[neigh-1])
        }
    }
    return nodes[0]
}

// Print graph (simple val list)
func printGraph(node *Node) {
    if node == nil {
        fmt.Println("nil")
        return
    }
    visited := make(map[*Node]bool)
    var dfs func(n *Node)
    dfs = func(n *Node) {
        if visited[n] {
            return
        }
        visited[n] = true
        fmt.Printf("Node %d neighbors: ", n.Val)
        for _, neigh := range n.Neighbors {
            fmt.Printf("%d ", neigh.Val)
        }
        fmt.Println()
        for _, neigh := range n.Neighbors {
            dfs(neigh)
        }
    }
    dfs(node)
}

func main() {
    // Example graph: [[2,4],[1,3],[2,4],[1,3]] nodes 1-4
```

```
    adj := [][]int{{2, 4}, {1, 3}, {2, 4}, {1, 3}}
    graph := buildGraph(adj)
    fmt.Println("Original graph:")
    printGraph(graph)
    fmt.Println()

    cloned := cloneGraph(graph)
    fmt.Println("Cloned graph:")
    printGraph(cloned)
}
```

23. House Robber (#198)

Problem Statement

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Input Format: Array nums.

Output Format: Integer max amount.

Examples:

- **Example 1:**

Input: nums = [1,2,3,1]

Output: 4

Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).

- **Example 2:**

Input: nums = [2,7,9,3,1]

Output: 12

Explanation: Rob house 1 (2), house 3 (9) and house 5 (1).

Constraints:

- 0 <= nums.length <= 100
- 0 <= nums[i] <= 400

Answer Explanation

DP: dp[i] = max(dp[i-1], dp[i-2] + nums[i]). Use variables for prev and curr. O(n) time, O(1) space.

Coding Solution in Golang

```
package main

import "fmt"

// rob finds max amount without robbing adjacent.
func rob(nums []int) int {
   n := len(nums)
   if n == 0 {
      return 0
   }
   if n == 1 {
      return nums[0]
   }
   prev2 := nums[0] // dp[i-2]
   prev1 := max(nums[0], nums[1]) // dp[i-1]
   for i := 2; i < n; i++ {
      current := max(prev1, prev2+nums[i])
      prev2 = prev1
      prev1 = current
   }
   return prev1
}

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

```
        return a
    }
    return b
}

func main() {
    // Example 1
    nums1 := []int{1, 2, 3, 1}
    fmt.Printf("Input: %v\n", nums1)
    fmt.Printf("Output: %d\n\n", rob(nums1))

    // Example 2
    nums2 := []int{2, 7, 9, 3, 1}
    fmt.Printf("Input: %v\n", nums2)
    fmt.Printf("Output: %d\n\n", rob(nums2))
}
```

24. Course Schedule (#207)

Problem Statement

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

Return true if you can finish all courses. Otherwise, return false.

Input Format: Integer numCourses, 2D array prerequisites.

Output Format: Boolean.

Examples:

- **Example 1:**

Input: numCourses = 2, prerequisites = [[1,0]]

Output: true

- **Example 2:**

Input: numCourses = 2, prerequisites = [[1,0],[0,1]]

Output: false

Constraints:

- 1 <= numCourses <= 2000
- 0 <= prerequisites.length <= 5000
- prerequisites[i].length == 2
- 0 <= ai, bi < numCourses
- All the pairs prerequisites[i] are unique.

Answer Explanation

Model as graph, detect cycle using DFS or Kahn's algorithm (topological sort). If cycle, false. Here use DFS with colors: 0 not visited, 1 visiting, 2 visited. O(n + e) time, O(n + e) space.

Coding Solution in Golang

```
package main

import "fmt"

// canFinish checks if courses can be finished without cycle.
func canFinish(numCourses int, prerequisites [][]int) bool {
    graph := make([][]int, numCourses)
    for _, pre := range prerequisites {
        graph[pre[1]] = append(graph[pre[1]], pre[0]) // bi -> ai
    }
    visit := make([]int, numCourses) // 0: not, 1: visiting, 2: visited
    var dfs func(course int) bool
    dfs = func(course int) bool {
        if visit[course] == 1 {
            return false // Cycle
        }
        if visit[course] == 2 {
            return true // Done
        }
        visit[course] = 1
        for _, next := range graph[course] {
            if !dfs(next) {
```

```
            return false
        }
      }
      visit[course] = 2
      return true
   }
   for i := 0; i < numCourses; i++ {
      if !dfs(i) {
         return false
      }
   }
   return true
}

func main() {
   // Example 1
   num1 := 2
   pre1 := [][]int{{1, 0}}
   fmt.Printf("Input: numCourses = %d, prerequisites = %v\n", num1, pre1)
   fmt.Printf("Output: %t\n\n", canFinish(num1, pre1))

   // Example 2
   num2 := 2
   pre2 := [][]int{{1, 0}, {0, 1}}
   fmt.Printf("Input: numCourses = %d, prerequisites = %v\n", num2, pre2)
   fmt.Printf("Output: %t\n\n", canFinish(num2, pre2))
}
```