```go
// 1. Array Utilities
package main

// Reverse an array
func reverseArray(arr []int) []int {
    n := len(arr)
    result := make([]int, n)
    for i := 0; i < n; i++ {
        result[i] = arr[n-i-1]
    }
    return result
}

// Rotate array by k positions
func rotateArray(arr []int, k int) []int {
    n := len(arr)
    if n == 0 {
        return arr
    }
    k = k % n
    result := make([]int, n)
    for i := 0; i < n; i++ {
        result[(i+k)%n] = arr[i]
    }
    return result
}

// Find max and min
func findMaxMin(arr []int) (int, int) {
    if len(arr) == 0 {
        return 0, 0
    }
    max, min := arr[0], arr[0]
    for _, v := range arr {
        if v > max {
            max = v
        }
        if v < min {
            min = v
        }
    }
    return max, min
}

// Remove duplicates
func removeDuplicates(arr []int) []int {
    if len(arr) == 0 {
        return arr
    }
    seen := make(map[int]bool)
    result := []int{}
    for _, v := range arr {
        if !seen[v] {
            seen[v] = true
            result = append(result, v)
        }
    }
    return result
}

// Check if array is sorted
func isSorted(arr []int) bool {
    for i := 1; i < len(arr); i++ {
        if arr[i] < arr[i-1] {
            return false
        }
    }
    return true
}

// 2. String Utilities
// Reverse a string
func reverseString(s string) string {
    runes := []rune(s)
    for i, j := 0, len(runes)-1; i < j; i, j = i+1, j-1 {
        runes[i], runes[j] = runes[j], runes[i]
    }
    return string(runes)
}

// Check palindrome
func isPalindrome(s string) bool {
    runes := []rune(s)
    for i, j := 0, len(runes)-1; i < j; i, j = i+1, j-1 {
        if runes[i] != runes[j] {
            return false
        }
    }
    return true
}

// Count character frequency
func charFrequency(s string) map[rune]int {
    freq := make(map[rune]int)
    for _, ch := range s {
        freq[ch]++
    }
    return freq
}

// Check anagram
func isAnagram(s1, s2 string) bool {
    if len(s1) != len(s2) {
        return false
    }
    freq1 := charFrequency(s1)
    freq2 := charFrequency(s2)
    for k, v := range freq1 {
        if freq2[k] != v {
            return false
        }
    }
    return true
}

// Find first non-repeating character
func firstNonRepeating(s string) rune {
    freq := charFrequency(s)
    for _, ch := range s {
        if freq[ch] == 1 {
```

```
            return ch
        }
    }
    return 0
}

// 3. Searching Algorithms
// Linear search
func linearSearch(arr []int, target int) int {
    for i, v := range arr {
        if v == target {
            return i
        }
    }
    return -1
}

// Binary search
func binarySearch(arr []int, target int) int {
    left, right := 0, len(arr)-1
    for left <= right {
        mid := left + (right-left)/2
        if arr[mid] == target {
            return mid
        }
        if arr[mid] < target {
            left = mid + 1
        } else {
            right = mid - 1
        }
    }
    return -1
}

// First occurrence in sorted array
func firstOccurrence(arr []int, target int) int {
    left, right := 0, len(arr)-1
    result := -1
    for left <= right {
        mid := left + (right-left)/2
        if arr[mid] == target {
            result = mid
            right = mid - 1
        } else if arr[mid] < target {
            left = mid + 1
        } else {
            right = mid - 1
        }
    }
    return result
}

// Last occurrence in sorted array
func lastOccurrence(arr []int, target int) int {
    left, right := 0, len(arr)-1
    result := -1
    for left <= right {
        mid := left + (right-left)/2
        if arr[mid] == target {
            result = mid
            left = mid + 1
        } else if arr[mid] < target {
            left = mid + 1
        } else {
            right = mid - 1
        }
    }
    return result
}

// Count occurrences in sorted array
func countOccurrences(arr []int, target int) int {
    first := firstOccurrence(arr, target)
    if first == -1 {
        return 0
    }
    last := lastOccurrence(arr, target)
    return last - first + 1
}

// 4. Sorting Helpers
// Custom sort using key
type Sortable struct {
    Value int
    Key   int
}

func customSort(arr []Sortable) []Sortable {
    for i := 0; i < len(arr); i++ {
        for j := i + 1; j < len(arr); j++ {
            if arr[j].Key < arr[i].Key {
                arr[i], arr[j] = arr[j], arr[i]
            }
        }
    }
    return arr
}

// Sort by frequency
func sortByFrequency(arr []int) []int {
    freq := make(map[int]int)
    for _, v := range arr {
        freq[v]++
    }

    // Convert to slice for sorting
    type item struct {
        value int
        count int
    }

    items := make([]item, 0, len(freq))
    for k, v := range freq {
        items = append(items, item{k, v})
    }

    // Bubble sort by frequency then value
    for i := 0; i < len(items); i++ {
        for j := i + 1; j < len(items); j++ {
            if items[j].count > items[i].count ||
```

```
            (items[j].count == items[i].count &&
items[j].value < items[i].value) {
                items[i], items[j] = items[j], items[i]
            }
        }
    }

    // Build result
    result := make([]int, 0, len(arr))
    for _, it := range items {
        for i := 0; i < it.count; i++ {
            result = append(result, it.value)
        }
    }
    return result
}

// Sort strings by length
func sortStringsByLength(strs []string) []string {
    for i := 0; i < len(strs); i++ {
        for j := i + 1; j < len(strs); j++ {
            if len(strs[j]) < len(strs[i]) {
                strs[i], strs[j] = strs[j], strs[i]
            }
        }
    }
    return strs
}

// Kth smallest element
func kthSmallest(arr []int, k int) int {
    if k <= 0 || k > len(arr) {
        return -1
    }
    // Simple bubble sort for demonstration
    for i := 0; i < len(arr); i++ {
        for j := i + 1; j < len(arr); j++ {
            if arr[j] < arr[i] {
                arr[i], arr[j] = arr[j], arr[i]
            }
        }
    }
    return arr[k-1]
}

// 5. Two Pointers Patterns
// Pair sum in sorted array
func pairSumSorted(arr []int, target int) [][]int {
    result := [][]int{}
    left, right := 0, len(arr)-1
    for left < right {
        sum := arr[left] + arr[right]
        if sum == target {
            result = append(result, []int{arr[left], arr[right]})
            left++
            right--
        } else if sum < target {
            left++
        } else {
            right--
        }
    }
    return result
}

// Remove duplicates in-place
func removeDuplicatesInPlace(arr []int) []int {
    if len(arr) == 0 {
        return arr
    }
    j := 1
    for i := 1; i < len(arr); i++ {
        if arr[i] != arr[i-1] {
            arr[j] = arr[i]
            j++
        }
    }
    return arr[:j]
}

// Reverse vowels
func reverseVowels(s string) string {
    vowels := map[byte]bool{
        'a': true, 'e': true, 'i': true, 'o': true, 'u': true,
        'A': true, 'E': true, 'I': true, 'O': true, 'U': true,
    }
    bytes := []byte(s)
    left, right := 0, len(bytes)-1
    for left < right {
        for left < right && !vowels[bytes[left]] {
            left++
        }
        for left < right && !vowels[bytes[right]] {
            right--
        }
        bytes[left], bytes[right] = bytes[right], bytes[left]
        left++
        right--
    }
    return string(bytes)
}

// Merge two sorted arrays
func mergeSortedArrays(arr1, arr2 []int) []int {
    result := make([]int, len(arr1)+len(arr2))
    i, j, k := 0, 0, 0
    for i < len(arr1) && j < len(arr2) {
        if arr1[i] <= arr2[j] {
            result[k] = arr1[i]
            i++
        } else {
            result[k] = arr2[j]
            j++
        }
        k++
    }
    for i < len(arr1) {
        result[k] = arr1[i]
        i++
        k++
```

```
    }
    for j < len(arr2) {
        result[k] = arr2[j]
        j++
        k++
    }
    return result
}

// Check palindrome ignoring non-alphanumeric
func isAlphanumeric(ch byte) bool {
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <=
'Z') || (ch >= '0' && ch <= '9')
}

func isPalindromeAlphaNum(s string) bool {
    bytes := []byte(s)
    left, right := 0, len(bytes)-1
    for left < right {
        for left < right && !isAlphanumeric(bytes[left]) {
            left++
        }
        for left < right && !isAlphanumeric(bytes[right]) {
            right--
        }
        lowerLeft := bytes[left]
        lowerRight := bytes[right]
        if lowerLeft >= 'A' && lowerLeft <= 'Z' {
            lowerLeft += 32
        }
        if lowerRight >= 'A' && lowerRight <= 'Z' {
            lowerRight += 32
        }
        if lowerLeft != lowerRight {
            return false
        }
        left++
        right--
    }
    return true
}

// 6. Sliding Window Patterns
// Maximum sum subarray of size k
func maxSumSubarray(arr []int, k int) int {
    if len(arr) < k {
        return 0
    }
    maxSum := 0
    windowSum := 0
    for i := 0; i < k; i++ {
        windowSum += arr[i]
    }
    maxSum = windowSum
    for i := k; i < len(arr); i++ {
        windowSum += arr[i] - arr[i-k]
        if windowSum > maxSum {
            maxSum = windowSum
        }
    }
    return maxSum
}

// Longest substring without repeating characters
func longestUniqueSubstring(s string) int {
    lastSeen := make(map[byte]int)
    maxLen := 0
    start := 0
    for i := 0; i < len(s); i++ {
        if idx, found := lastSeen[s[i]]; found && idx >= start
{
            start = idx + 1
        }
        lastSeen[s[i]] = i
        if i-start+1 > maxLen {
            maxLen = i - start + 1
        }
    }
    return maxLen
}

// Longest substring with k distinct characters
func longestSubstringKDistinct(s string, k int) int {
    if k == 0 {
        return 0
    }
    freq := make(map[byte]int)
    maxLen := 0
    left := 0
    for right := 0; right < len(s); right++ {
        freq[s[right]]++
        for len(freq) > k {
            freq[s[left]]--
            if freq[s[left]] == 0 {
                delete(freq, s[left])
            }
            left++
        }
        if right-left+1 > maxLen {
            maxLen = right - left + 1
        }
    }
    return maxLen
}

// Count subarrays with given sum
func countSubarraysWithSum(arr []int, target int) int {
    count := 0
    for start := 0; start < len(arr); start++ {
        sum := 0
        for end := start; end < len(arr); end++ {
            sum += arr[end]
            if sum == target {
                count++
            }
        }
    }
    return count
}
```

```go
// Minimum window substring
func minWindow(s, t string) string {
    if len(s) == 0 || len(t) == 0 {
        return ""
    }

    targetFreq := make(map[byte]int)
    windowFreq := make(map[byte]int)
    for i := 0; i < len(t); i++ {
        targetFreq[t[i]]++
    }

    required := len(targetFreq)
    formed := 0
    left, right := 0, 0
    minLen := len(s) + 1
    minStart := 0

    for right < len(s) {
        ch := s[right]
        windowFreq[ch]++

        if targetFreq[ch] > 0 && windowFreq[ch] ==
targetFreq[ch] {
            formed++
        }

        for left <= right && formed == required {
            if right-left+1 < minLen {
                minLen = right - left + 1
                minStart = left
            }

            leftCh := s[left]
            windowFreq[leftCh]--
            if targetFreq[leftCh] > 0 && windowFreq[leftCh] <
targetFreq[leftCh] {
                formed--
            }
            left++
        }
        right++
    }

    if minLen == len(s)+1 {
        return ""
    }
    return s[minStart : minStart+minLen]
}

// 7. Hashing / Dictionary Usage
// Two sum
func twoSum(nums []int, target int) []int {
    seen := make(map[int]int)
    for i, num := range nums {
        complement := target - num
        if idx, found := seen[complement]; found {
            return []int{idx, i}
        }
        seen[num] = i
    }
    return nil
}

// Group anagrams
func groupAnagrams(strs []string) [][]string {
    groups := make(map[string][]string)
    for _, s := range strs {
        // Create frequency signature
        freq := make([]int, 26)
        for _, ch := range s {
            freq[ch-'a']++
        }
        // Convert to string key
        key := ""
        for _, count := range freq {
            key += string(count) + ","
        }
        groups[key] = append(groups[key], s)
    }

    result := make([][]string, 0, len(groups))
    for _, group := range groups {
        result = append(result, group)
    }
    return result
}

// Longest consecutive sequence
func longestConsecutive(nums []int) int {
    if len(nums) == 0 {
        return 0
    }

    numSet := make(map[int]bool)
    for _, num := range nums {
        numSet[num] = true
    }

    longest := 0
    for num := range numSet {
        if !numSet[num-1] { // Start of sequence
            current := num
            length := 1
            for numSet[current+1] {
                current++
                length++
            }
            if length > longest {
                longest = length
            }
        }
    }
    return longest
}

// Find duplicates
func findDuplicates(nums []int) []int {
    seen := make(map[int]bool)
    duplicates := []int{}
```

```
    for _, num := range nums {
        if seen[num] {
            duplicates = append(duplicates, num)
        } else {
            seen[num] = true
        }
    }
    return duplicates
}

// 8. Stack Utilities
type Stack []int

func (s *Stack) Push(x int) {
    *s = append(*s, x)
}

func (s *Stack) Pop() int {
    if len(*s) == 0 {
        return -1
    }
    x := (*s)[len(*s)-1]
    *s = (*s)[:len(*s)-1]
    return x
}

func (s *Stack) Peek() int {
    if len(*s) == 0 {
        return -1
    }
    return (*s)[len(*s)-1]
}

func (s *Stack) IsEmpty() bool {
    return len(*s) == 0
}

// Valid parentheses
func isValidParentheses(s string) bool {
    stack := Stack{}
    pairs := map[byte]byte{
        ')': '(',
        ']': '[',
        '}': '{',
    }

    for i := 0; i < len(s); i++ {
        ch := s[i]
        switch ch {
        case '(', '[', '{':
            stack.Push(int(ch))
        case ')', ']', '}':
            if stack.IsEmpty() || stack.Peek() != int(pairs[ch]) {
                return false
            }
            stack.Pop()
        }
    }
    return stack.IsEmpty()
}

// Next greater element
func nextGreaterElement(nums []int) []int {
    result := make([]int, len(nums))
    stack := Stack{}

    for i := len(nums) - 1; i >= 0; i-- {
        for !stack.IsEmpty() && stack.Peek() <= nums[i] {
            stack.Pop()
        }
        if stack.IsEmpty() {
            result[i] = -1
        } else {
            result[i] = stack.Peek()
        }
        stack.Push(nums[i])
    }
    return result
}

// Previous smaller element
func previousSmallerElement(nums []int) []int {
    result := make([]int, len(nums))
    stack := Stack{}

    for i := 0; i < len(nums); i++ {
        for !stack.IsEmpty() && stack.Peek() >= nums[i] {
            stack.Pop()
        }
        if stack.IsEmpty() {
            result[i] = -1
        } else {
            result[i] = stack.Peek()
        }
        stack.Push(nums[i])
    }
    return result
}

// Evaluate postfix expression
func evaluatePostfix(tokens []string) int {
    stack := []int{}
    for _, token := range tokens {
        if token == "+" || token == "-" || token == "*" || token
== "/" {
            b := stack[len(stack)-1]
            a := stack[len(stack)-2]
            stack = stack[:len(stack)-2]

            var result int
            switch token {
            case "+":
                result = a + b
            case "-":
                result = a - b
            case "*":
                result = a * b
            case "/":
                result = a / b
            }
```

```
            stack = append(stack, result)
        } else {
            // Convert string to int (simplified)
            num := 0
            for _, ch := range token {
                num = num*10 + int(ch-'0')
            }
            stack = append(stack, num)
        }
    }
    return stack[0]
}

// Remove adjacent duplicates
func removeAdjacentDuplicates(s string) string {
    stack := []byte{}
    for i := 0; i < len(s); i++ {
        if len(stack) > 0 && stack[len(stack)-1] == s[i] {
            stack = stack[:len(stack)-1]
        } else {
            stack = append(stack, s[i])
        }
    }
    return string(stack)
}

// 9. Queue & Deque
// Simple queue implementation
type Queue []int

func (q *Queue) Enqueue(x int) {
    *q = append(*q, x)
}

func (q *Queue) Dequeue() int {
    if len(*q) == 0 {
        return -1
    }
    x := (*q)[0]
    *q = (*q)[1:]
    return x
}

func (q *Queue) IsEmpty() bool {
    return len(*q) == 0
}

// Simple deque implementation
type Deque []int

func (d *Deque) PushFront(x int) {
    *d = append([]int{x}, *d...)
}

func (d *Deque) PushBack(x int) {
    *d = append(*d, x)
}

func (d *Deque) PopFront() int {
    if len(*d) == 0 {
        return -1
    }
    x := (*d)[0]
    *d = (*d)[1:]
    return x
}

func (d *Deque) PopBack() int {
    if len(*d) == 0 {
        return -1
    }
    x := (*d)[len(*d)-1]
    *d = (*d)[:len(*d)-1]
    return x
}

// Sliding window maximum
func slidingWindowMaximum(nums []int, k int) []int {
    if len(nums) == 0 {
        return []int{}
    }

    result := make([]int, len(nums)-k+1)
    deque := Deque{}

    for i := 0; i < len(nums); i++ {
        // Remove elements outside window
        if len(deque) > 0 && deque[0] == i-k {
            deque.PopFront()
        }

        // Remove smaller elements
        for len(deque) > 0 && nums[deque[len(deque)-1]] <=
nums[i] {
            deque.PopBack()
        }

        deque.PushBack(i)

        // Add to result
        if i >= k-1 {
            result[i-k+1] = nums[deque[0]]
        }
    }
    return result
}

// First negative number in window
func firstNegativeInWindow(arr []int, k int) []int {
    if len(arr) < k {
        return []int{}
    }

    result := make([]int, len(arr)-k+1)
    deque := Deque{}

    for i := 0; i < len(arr); i++ {
        // Remove elements outside window
        if len(deque) > 0 && deque[0] == i-k {
            deque.PopFront()
```

```
        }

        // Add negative numbers
        if arr[i] < 0 {
            deque.PushBack(i)
        }

        // Add to result
        if i >= k-1 {
            if len(deque) > 0 {
                result[i-k+1] = arr[deque[0]]
            } else {
                result[i-k+1] = 0
            }
        }
    }
    return result
}

// Generate binary numbers
func generateBinaryNumbers(n int) []string {
    result := make([]string, n)
    queue := []string{"1"}

    for i := 0; i < n; i++ {
        current := queue[0]
        queue = queue[1:]
        result[i] = current

        queue = append(queue, current+"0")
        queue = append(queue, current+"1")
    }
    return result
}

// 10. Recursion Basics
// Factorial
func factorial(n int) int {
    if n <= 1 {
        return 1
    }
    return n * factorial(n-1)
}

// Fibonacci
func fibonacci(n int) int {
    if n <= 1 {
        return n
    }
    return fibonacci(n-1) + fibonacci(n-2)
}

// Reverse string recursively
func reverseStringRecursive(s string) string {
    if len(s) <= 1 {
        return s
    }
    return reverseStringRecursive(s[1:]) + string(s[0])
}

// Power function
func power(x float64, n int) float64 {
    if n == 0 {
        return 1
    }
    if n < 0 {
        return 1 / power(x, -n)
    }
    return x * power(x, n-1)
}

// 11. Backtracking Patterns
// Generate subsets
func subsets(nums []int) [][]int {
    result := [][]int{}
    backtrackSubsets(nums, 0, []int{}, &result)
    return result
}

func backtrackSubsets(nums []int, start int, current []int,
result *[][]int) {
    temp := make([]int, len(current))
    copy(temp, current)
    *result = append(*result, temp)

    for i := start; i < len(nums); i++ {
        current = append(current, nums[i])
        backtrackSubsets(nums, i+1, current, result)
        current = current[:len(current)-1]
    }
}

// Generate permutations
func permutations(nums []int) [][]int {
    result := [][]int{}
    backtrackPermutations(nums, 0, &result)
    return result
}

func backtrackPermutations(nums []int, start int, result *[]
[]int) {
    if start == len(nums) {
        temp := make([]int, len(nums))
        copy(temp, nums)
        *result = append(*result, temp)
        return
    }

    for i := start; i < len(nums); i++ {
        nums[start], nums[i] = nums[i], nums[start]
        backtrackPermutations(nums, start+1, result)
        nums[start], nums[i] = nums[i], nums[start]
    }
}

// Combination sum
func combinationSum(candidates []int, target int) [][]int {
    result := [][]int{}
    backtrackCombinationSum(candidates, target, 0, []int{},
&result)
```

```
    return result
}

func backtrackCombinationSum(candidates []int, target,
start int, current []int, result *[][]int) {
    if target < 0 {
        return
    }
    if target == 0 {
        temp := make([]int, len(current))
        copy(temp, current)
        *result = append(*result, temp)
        return
    }

    for i := start; i < len(candidates); i++ {
        current = append(current, candidates[i])
        backtrackCombinationSum(candidates, target-
candidates[i], i, current, result)
        current = current[:len(current)-1]
    }
}

// 12. Binary Search Patterns
// Lower bound
func lowerBound(arr []int, target int) int {
    left, right := 0, len(arr)-1
    result := len(arr)
    for left <= right {
        mid := left + (right-left)/2
        if arr[mid] >= target {
            result = mid
            right = mid - 1
        } else {
            left = mid + 1
        }
    }
    return result
}

// Upper bound
func upperBound(arr []int, target int) int {
    left, right := 0, len(arr)-1
    result := len(arr)
    for left <= right {
        mid := left + (right-left)/2
        if arr[mid] > target {
            result = mid
            right = mid - 1
        } else {
            left = mid + 1
        }
    }
    return result
}

// Search in rotated array
func searchRotated(nums []int, target int) int {
    left, right := 0, len(nums)-1
    for left <= right {
        mid := left + (right-left)/2
        if nums[mid] == target {
            return mid
        }

        if nums[left] <= nums[mid] {
            if nums[left] <= target && target < nums[mid] {
                right = mid - 1
            } else {
                left = mid + 1
            }
        } else {
            if nums[mid] < target && target <= nums[right] {
                left = mid + 1
            } else {
                right = mid - 1
            }
        }
    }
    return -1
}

// Find peak element
func findPeakElement(nums []int) int {
    left, right := 0, len(nums)-1
    for left < right {
        mid := left + (right-left)/2
        if nums[mid] > nums[mid+1] {
            right = mid
        } else {
            left = mid + 1
        }
    }
    return left
}

// Find minimum in rotated array
func findMinRotated(nums []int) int {
    left, right := 0, len(nums)-1
    for left < right {
        mid := left + (right-left)/2
        if nums[mid] > nums[right] {
            left = mid + 1
        } else {
            right = mid
        }
    }
    return nums[left]
}

// 13. Heap / Priority Queue
// Simple max heap implementation
type MaxHeap []int

func (h MaxHeap) Len() int           { return len(h) }
func (h MaxHeap) Less(i, j int) bool { return h[i] > h[j] }
func (h MaxHeap) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }

func (h *MaxHeap) Push(x interface{}) {
    *h = append(*h, x.(int))
```

```
}

func (h *MaxHeap) Pop() interface{} {
    old := *h
    n := len(old)
    x := old[n-1]
    *h = old[:n-1]
    return x
}

// Kth largest element
func kthLargest(nums []int, k int) int {
    // Simple approach without heap
    for i := 0; i < len(nums); i++ {
        for j := i + 1; j < len(nums); j++ {
            if nums[j] > nums[i] {
                nums[i], nums[j] = nums[j], nums[i]
            }
        }
    }
    return nums[k-1]
}

// Top k frequent elements
func topKFrequent(nums []int, k int) []int {
    freq := make(map[int]int)
    for _, num := range nums {
        freq[num]++
    }

    // Simple selection
    result := make([]int, 0, k)
    for i := 0; i < k; i++ {
        maxFreq := -1
        maxNum := 0
        for num, count := range freq {
            if count > maxFreq {
                maxFreq = count
                maxNum = num
            }
        }
        result = append(result, maxNum)
        delete(freq, maxNum)
    }
    return result
}

// 14. Linked List Utilities
type ListNode struct {
    Val  int
    Next *ListNode
}

// Reverse linked list
func reverseLinkedList(head *ListNode) *ListNode {
    var prev *ListNode
    current := head
    for current != nil {
        next := current.Next
        current.Next = prev
        prev = current
        current = next
    }
    return prev
}

// Detect cycle
func hasCycle(head *ListNode) bool {
    if head == nil {
        return false
    }
    slow, fast := head, head
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
        if slow == fast {
            return true
        }
    }
    return false
}

// Find middle node
func findMiddle(head *ListNode) *ListNode {
    if head == nil {
        return nil
    }
    slow, fast := head, head
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
    }
    return slow
}

// Merge two sorted lists
func mergeTwoLists(l1, l2 *ListNode) *ListNode {
    dummy := &ListNode{}
    current := dummy

    for l1 != nil && l2 != nil {
        if l1.Val <= l2.Val {
            current.Next = l1
            l1 = l1.Next
        } else {
            current.Next = l2
            l2 = l2.Next
        }
        current = current.Next
    }

    if l1 != nil {
        current.Next = l1
    } else {
        current.Next = l2
    }

    return dummy.Next
}
```

```go
// Remove nth node from end
func removeNthFromEnd(head *ListNode, n int)
*ListNode {
    dummy := &ListNode{Next: head}
    slow, fast := dummy, dummy

    // Move fast n+1 steps ahead
    for i := 0; i <= n; i++ {
        fast = fast.Next
    }

    // Move both until fast reaches end
    for fast != nil {
        slow = slow.Next
        fast = fast.Next
    }

    // Remove the node
    slow.Next = slow.Next.Next
    return dummy.Next
}

// 15. Tree Traversals
type TreeNode struct {
    Val   int
    Left  *TreeNode
    Right *TreeNode
}

// Inorder traversal
func inorderTraversal(root *TreeNode) []int {
    result := []int{}
    var inorder func(node *TreeNode)
    inorder = func(node *TreeNode) {
        if node == nil {
            return
        }
        inorder(node.Left)
        result = append(result, node.Val)
        inorder(node.Right)
    }
    inorder(root)
    return result
}

// Preorder traversal
func preorderTraversal(root *TreeNode) []int {
    result := []int{}
    var preorder func(node *TreeNode)
    preorder = func(node *TreeNode) {
        if node == nil {
            return
        }
        result = append(result, node.Val)
        preorder(node.Left)
        preorder(node.Right)
    }
    preorder(root)
    return result
}

// Postorder traversal
func postorderTraversal(root *TreeNode) []int {
    result := []int{}
    var postorder func(node *TreeNode)
    postorder = func(node *TreeNode) {
        if node == nil {
            return
        }
        postorder(node.Left)
        postorder(node.Right)
        result = append(result, node.Val)
    }
    postorder(root)
    return result
}

// Level order traversal
func levelOrderTraversal(root *TreeNode) [][]int {
    if root == nil {
        return [][]int{}
    }

    result := [][]int{}
    queue := []*TreeNode{root}

    for len(queue) > 0 {
        levelSize := len(queue)
        level := make([]int, levelSize)

        for i := 0; i < levelSize; i++ {
            node := queue[0]
            queue = queue[1:]
            level[i] = node.Val

            if node.Left != nil {
                queue = append(queue, node.Left)
            }
            if node.Right != nil {
                queue = append(queue, node.Right)
            }
        }
        result = append(result, level)
    }
    return result
}

// Height of tree
func treeHeight(root *TreeNode) int {
    if root == nil {
        return -1
    }
    leftHeight := treeHeight(root.Left)
    rightHeight := treeHeight(root.Right)
    if leftHeight > rightHeight {
        return leftHeight + 1
    }
    return rightHeight + 1
}
```

```go
// 16. Graph Algorithms
// Graph representation
type Graph struct {
    vertices int
    adjList  map[int][]int
}

func NewGraph(vertices int) *Graph {
    return &Graph{
        vertices: vertices,
        adjList:  make(map[int][]int),
    }
}

func (g *Graph) AddEdge(u, v int) {
    g.adjList[u] = append(g.adjList[u], v)
    g.adjList[v] = append(g.adjList[v], u)
}

// BFS traversal
func BFS(graph *Graph, start int) []int {
    visited := make(map[int]bool)
    result := []int{}
    queue := []int{start}
    visited[start] = true

    for len(queue) > 0 {
        vertex := queue[0]
        queue = queue[1:]
        result = append(result, vertex)

        for _, neighbor := range graph.adjList[vertex] {
            if !visited[neighbor] {
                visited[neighbor] = true
                queue = append(queue, neighbor)
            }
        }
    }
    return result
}

// DFS traversal
func DFS(graph *Graph, start int) []int {
    visited := make(map[int]bool)
    result := []int{}
    var dfs func(int)

    dfs = func(vertex int) {
        visited[vertex] = true
        result = append(result, vertex)

        for _, neighbor := range graph.adjList[vertex] {
            if !visited[neighbor] {
                dfs(neighbor)
            }
        }
    }

    dfs(start)
    return result
}

// Detect cycle in undirected graph
func hasCycleGraph(graph *Graph) bool {
    visited := make(map[int]bool)

    var dfs func(int, int) bool
    dfs = func(vertex, parent int) bool {
        visited[vertex] = true

        for _, neighbor := range graph.adjList[vertex] {
            if !visited[neighbor] {
                if dfs(neighbor, vertex) {
                    return true
                }
            } else if neighbor != parent {
                return true
            }
        }
        return false
    }

    for vertex := 0; vertex < graph.vertices; vertex++ {
        if !visited[vertex] {
            if dfs(vertex, -1) {
                return true
            }
        }
    }
    return false
}

// Count connected components
func countConnectedComponents(graph *Graph) int {
    visited := make(map[int]bool)
    count := 0

    var dfs func(int)
    dfs = func(vertex int) {
        visited[vertex] = true
        for _, neighbor := range graph.adjList[vertex] {
            if !visited[neighbor] {
                dfs(neighbor)
            }
        }
    }

    for vertex := 0; vertex < graph.vertices; vertex++ {
        if !visited[vertex] {
            count++
            dfs(vertex)
        }
    }
    return count
}

// Shortest path in unweighted graph
func shortestPathUnweighted(graph *Graph, start, end int)
[]int {
    if start == end {
```

```
    return []int{start}
  }

  visited := make(map[int]bool)
  parent := make(map[int]int)
  queue := []int{start}
  visited[start] = true

  for len(queue) > 0 {
    vertex := queue[0]
    queue = queue[1:]

    for _, neighbor := range graph.adjList[vertex] {
      if !visited[neighbor] {
        visited[neighbor] = true
        parent[neighbor] = vertex
        queue = append(queue, neighbor)

        if neighbor == end {
          // Reconstruct path
          path := []int{end}
          for path[len(path)-1] != start {
            path = append(path, parent[path[len(path)-
1]])
          }
          // Reverse path
          for i, j := 0, len(path)-1; i < j; i, j = i+1, j-1 {
            path[i], path[j] = path[j], path[i]
          }
          return path
        }
      }
    }
  }
  return nil
}

// 17. Dynamic Programming (1D)
// Fibonacci with memoization
func fibMemo(n int, memo map[int]int) int {
  if n <= 1 {
    return n
  }
  if val, ok := memo[n]; ok {
    return val
  }
  memo[n] = fibMemo(n-1, memo) + fibMemo(n-2,
memo)
  return memo[n]
}

// Climbing stairs
func climbStairs(n int) int {
  if n <= 2 {
    return n
  }
  dp := make([]int, n+1)
  dp[1] = 1
  dp[2] = 2
  for i := 3; i <= n; i++ {
    dp[i] = dp[i-1] + dp[i-2]
  }
  return dp[n]
}

// House robber
func rob(nums []int) int {
  if len(nums) == 0 {
    return 0
  }
  if len(nums) == 1 {
    return nums[0]
  }

  dp := make([]int, len(nums))
  dp[0] = nums[0]
  dp[1] = max(nums[0], nums[1])

  for i := 2; i < len(nums); i++ {
    dp[i] = max(dp[i-1], dp[i-2]+nums[i])
  }
  return dp[len(nums)-1]
}

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

// Maximum subarray sum
func maxSubArray(nums []int) int {
  if len(nums) == 0 {
    return 0
  }
  maxSum, currentSum := nums[0], nums[0]
  for i := 1; i < len(nums); i++ {
    currentSum = max(nums[i], currentSum+nums[i])
    maxSum = max(maxSum, currentSum)
  }
  return maxSum
}

// Coin change (minimum coins)
func coinChange(coins []int, amount int) int {
  dp := make([]int, amount+1)
  for i := range dp {
    dp[i] = amount + 1
  }
  dp[0] = 0

  for i := 1; i <= amount; i++ {
    for _, coin := range coins {
      if coin <= i {
        dp[i] = min(dp[i], dp[i-coin]+1)
      }
    }
  }
```

```
    if dp[amount] > amount {
        return -1
    }
    return dp[amount]
}

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

// 18. Dynamic Programming (2D)
// Longest common subsequence
func longestCommonSubsequence(text1, text2 string) int {
    m, n := len(text1), len(text2)
    dp := make([][]int, m+1)
    for i := range dp {
        dp[i] = make([]int, n+1)
    }

    for i := 1; i <= m; i++ {
        for j := 1; j <= n; j++ {
            if text1[i-1] == text2[j-1] {
                dp[i][j] = dp[i-1][j-1] + 1
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
            }
        }
    }
    return dp[m][n]
}

// Longest common substring
func longestCommonSubstring(text1, text2 string) int {
    m, n := len(text1), len(text2)
    dp := make([][]int, m+1)
    for i := range dp {
        dp[i] = make([]int, n+1)
    }
    maxLength := 0

    for i := 1; i <= m; i++ {
        for j := 1; j <= n; j++ {
            if text1[i-1] == text2[j-1] {
                dp[i][j] = dp[i-1][j-1] + 1
                maxLength = max(maxLength, dp[i][j])
            }
        }
    }
    return maxLength
}

// Edit distance
func editDistance(word1, word2 string) int {
    m, n := len(word1), len(word2)
    dp := make([][]int, m+1)
    for i := range dp {
        dp[i] = make([]int, n+1)
    }

    for i := 0; i <= m; i++ {
        dp[i][0] = i
    }
    for j := 0; j <= n; j++ {
        dp[0][j] = j
    }

    for i := 1; i <= m; i++ {
        for j := 1; j <= n; j++ {
            if word1[i-1] == word2[j-1] {
                dp[i][j] = dp[i-1][j-1]
            } else {
                dp[i][j] = 1 + min(dp[i-1][j-1], min(dp[i-1][j],
dp[i][j-1]))
            }
        }
    }
    return dp[m][n]
}

// Unique paths
func uniquePaths(m, n int) int {
    dp := make([][]int, m)
    for i := range dp {
        dp[i] = make([]int, n)
        dp[i][0] = 1
    }
    for j := 0; j < n; j++ {
        dp[0][j] = 1
    }

    for i := 1; i < m; i++ {
        for j := 1; j < n; j++ {
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
        }
    }
    return dp[m-1][n-1]
}

// 0/1 Knapsack
func knapsack01(weights, values []int, capacity int) int {
    n := len(weights)
    dp := make([][]int, n+1)
    for i := range dp {
        dp[i] = make([]int, capacity+1)
    }

    for i := 1; i <= n; i++ {
        for w := 0; w <= capacity; w++ {
            if weights[i-1] <= w {
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-
1]]+values[i-1])
            } else {
                dp[i][w] = dp[i-1][w]
            }
        }
    }
    return dp[n][capacity]
```

```
}

// 19. Bit Manipulation
// Check if number is power of two
func isPowerOfTwo(n int) bool {
    return n > 0 && (n&(n-1)) == 0
}

// Count set bits
func countSetBits(n int) int {
    count := 0
    for n > 0 {
        count += n & 1
        n >>= 1
    }
    return count
}

// Find single non-repeating number
func singleNumber(nums []int) int {
    result := 0
    for _, num := range nums {
        result ^= num
    }
    return result
}

// Toggle ith bit
func toggleBit(n int, i uint) int {
    return n ^ (1 << i)
}

// Generate subsets using bitmask
func subsetsBitmask(nums []int) [][]int {
    n := len(nums)
    total := 1 << n
    result := make([][]int, 0, total)

    for mask := 0; mask < total; mask++ {
        subset := []int{}
        for i := 0; i < n; i++ {
            if mask&(1<<i) != 0 {
                subset = append(subset, nums[i])
            }
        }
        result = append(result, subset)
    }
    return result
}

// 20. Math & Number Theory
// GCD and LCM
func gcd(a, b int) int {
    for b != 0 {
        a, b = b, a%b
    }
    return a
}

func lcm(a, b int) int {
    return a / gcd(a, b) * b
}

// Prime check
func isPrime(n int) bool {
    if n <= 1 {
        return false
    }
    if n <= 3 {
        return true
    }
    if n%2 == 0 || n%3 == 0 {
        return false
    }
    for i := 5; i*i <= n; i += 6 {
        if n%i == 0 || n%(i+2) == 0 {
            return false
        }
    }
    return true
}

// Sieve of Eratosthenes
func sieveOfEratosthenes(n int) []int {
    isPrime := make([]bool, n+1)
    for i := 2; i <= n; i++ {
        isPrime[i] = true
    }

    for p := 2; p*p <= n; p++ {
        if isPrime[p] {
            for i := p * p; i <= n; i += p {
                isPrime[i] = false
            }
        }
    }

    primes := []int{}
    for i := 2; i <= n; i++ {
        if isPrime[i] {
            primes = append(primes, i)
        }
    }
    return primes
}

// Fast exponentiation
func fastExponentiation(x float64, n int) float64 {
    if n == 0 {
        return 1
    }
    if n < 0 {
        x = 1 / x
        n = -n
    }

    result := 1.0
    for n > 0 {
        if n&1 == 1 {
            result *= x
```

```
        }
        x *= x
        n >>= 1
    }
    return result
}

func main() {
    // Test functions as needed
    fmt.Println("Go Data Structures and Algorithms
Utilities")
}
```