LeetCode Statements + Better Graphs + Solutions (Static PDF)

2026-02-22

LeetCode Statements + Better Graphs + Solutions (Static PDF)

Each problem includes: clear statement, variable-driven flowchart, Python solution, and Go solution.

1) Two Sum

Problem statement: Given an integer array nums and integer target, return indices of two different numbers whose sum equals target.

Core pattern: Hash map complement lookup

Variables to track: `i, x, y, seen`

```
flowchart TD
    A[Start with input for problem 1] --> B[Initialize variables: i, x, y, seen]
    B --> C[Apply pattern: Hash map complement lookup]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List

class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        seen = {}
        for i, x in enumerate(nums):
            y = target - x
            if y in seen:
                return [seen[y], i]
            seen[x] = i
        return []
```

Go Solution

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

49) Group Anagrams

Problem statement: Given an array of strings, group all anagrams together and return the groups in any order.

Core pattern: Anagram signature hashing

Variables to track: `s, key, groups`

```
flowchart TD
    A[Start with input for problem 49] --> B[Initialize variables: s, key, groups]
    B --> C[Apply pattern: Anagram signature hashing]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from collections import defaultdict
from typing import List

class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        groups = defaultdict(list)
        for s in strs:
            key = [0] * 26
            for c in s:
                key[ord(c) - ord("a")] += 1
            groups[tuple(key)].append(s)
        return list(groups.values())
```

Go Solution

```
func groupAnagrams(strs []string) [][]string {
    m := map[[26]int][]string{}
    for _, s := range strs {
        var key [26]int
        for _, ch := range s {
            key[ch-'a']++
        }
        m[key] = append(m[key], s)
    }
    out := make([][]string, 0, len(m))
    for _, v := range m {
        out = append(out, v)
    }
    return out
}
```

238) Product of Array Except Self

Problem statement: Given nums, return output where output[i] equals product of all nums[j] for j != i, without division.

Core pattern: Two-pass prefix and suffix

Variables to track: `i, out, prefix, suffix`

```
flowchart TD
    A[Start with input for problem 238] --> B[Initialize variables: i, out, prefix, suffix]
    B --> C[Apply pattern: Two-pass prefix and suffix]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```
from typing import List

class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        n = len(nums)
        out = [1] * n
        p = 1
        for i in range(n):
            out[i] = p
            p *= nums[i]
        s = 1
        for i in range(n - 1, -1, -1):
            out[i] *= s
            s *= nums[i]
        return out
```

Go Solution

```
func productExceptSelf(nums []int) []int {
    n := len(nums)
    out := make([]int, n)
    p := 1
    for i := 0; i < n; i++ {
        out[i] = p
        p *= nums[i]
    }
    s := 1
    for i := n - 1; i >= 0; i-- {
        out[i] *= s
        s *= nums[i]
    }
    return out
}
```

560) Subarray Sum Equals K

Problem statement: Given nums and integer k, return total count of contiguous subarrays whose sum is k.

Core pattern: Prefix sum frequency map

Variables to track: `cur, cnt, ans`

```
flowchart TD
    A[Start with input for problem 560] --> B[Initialize variables: cur, cnt, ans]
    B --> C[Apply pattern: Prefix sum frequency map]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```
from collections import defaultdict
from typing import List
```

```python
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        cnt = defaultdict(int)
        cnt[0] = 1
        cur = 0
        ans = 0
        for x in nums:
            cur += x
            ans += cnt[cur - k]
            cnt[cur] += 1
        return ans
```

Go Solution

```go
func subarraySum(nums []int, k int) int {
    cnt := map[int]int{0: 1}
    cur, ans := 0, 0
    for _, x := range nums {
        cur += x
        ans += cnt[cur-k]
        cnt[cur]++
    }
    return ans
}
```

347) Top K Frequent Elements

Problem statement: Given nums and integer k, return the k elements with highest frequency.

Core pattern: Frequency buckets

Variables to track: `freq, buckets, out`

```
flowchart TD
    A[Start with input for problem 347] --> B[Initialize variables: freq, buckets, out]
    B --> C[Apply pattern: Frequency buckets]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        freq = Counter(nums)
        buckets = [[] for _ in range(len(nums) + 1)]
        for num, c in freq.items():
            buckets[c].append(num)
        out = []
        for c in range(len(buckets) - 1, 0, -1):
            for num in buckets[c]:
                out.append(num)
                if len(out) == k:
                    return out
        return out
```

Go Solution

```
func topKFrequent(nums []int, k int) []int {
    freq := map[int]int{}
    for _, x := range nums {
        freq[x]++
    }
    buckets := make([][]int, len(nums)+1)
    for x, c := range freq {
        buckets[c] = append(buckets[c], x)
    }
    out := []int{}
    for c := len(buckets) - 1; c > 0 && len(out) < k; c-- {
        out = append(out, buckets[c]...)
    }
    return out[:k]
}
```

3) Longest Substring Without Repeating Characters

Problem statement: Given string s, return length of longest substring without repeating characters.

Core pattern: Sliding window unique chars

Variables to track: `l, r, last, best`

```
flowchart TD
    A[Start with input for problem 3] --> B[Initialize variables: l, r, last, best]
    B --> C[Apply pattern: Sliding window unique chars]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        last = {}
        l = 0
        best = 0
        for r, ch in enumerate(s):
            if ch in last and last[ch] >= l:
                l = last[ch] + 1
            last[ch] = r
            best = max(best, r - l + 1)
        return best
```

Go Solution

```go
func lengthOfLongestSubstring(s string) int {
    last := map[byte]int{}
    l, best := 0, 0
    for r := 0; r < len(s); r++ {
        ch := s[r]
        if i, ok := last[ch]; ok && i >= l {
            l = i + 1
        }
        last[ch] = r
        if r-l+1 > best {
            best = r - l + 1
        }
    }
    return best
}
```

76) Minimum Window Substring

Problem statement: Given strings s and t, return the minimum window in s that contains every character in t with multiplicity.

Core pattern: Valid window expand and contract

Variables to track: `l, r, need, have, formed, best`

```
flowchart TD
    A[Start with input for problem 76] --> B[Initialize variables: l, r, need, have, formed
    B --> C[Apply pattern: Valid window expand and contract]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from collections import Counter

class Solution:
    def minWindow(self, s: str, t: str) -> str:
        need = Counter(t)
        have = {}
        required = len(need)
        formed = 0
        l = 0
        ans = (float("inf"), 0, 0)
        for r, ch in enumerate(s):
            have[ch] = have.get(ch, 0) + 1
            if ch in need and have[ch] == need[ch]:
                formed += 1
            while formed == required:
                if r - l + 1 < ans[0]:
                    ans = (r - l + 1, l, r)
                c = s[l]
                have[c] -= 1
                if c in need and have[c] < need[c]:
                    formed -= 1
                l += 1
        return "" if ans[0] == float("inf") else s[ans[1] : ans[2] + 1]
```

Go Solution

```go
func minWindow(s string, t string) string {
    need := map[byte]int{}
    for i := 0; i < len(t); i++ {
        need[t[i]]++
    }
    have := map[byte]int{}
    required := len(need)
    formed := 0
    l := 0
    bestLen, bestL := 1<<30, 0

    for r := 0; r < len(s); r++ {
        ch := s[r]
        have[ch]++
        if v, ok := need[ch]; ok && have[ch] == v {
            formed++
        }
        for formed == required {
            if r-l+1 < bestLen {
                bestLen = r - l + 1
                bestL = l
            }
            c := s[l]
            have[c]--
            if v, ok := need[c]; ok && have[c] < v {
                formed--
            }
            l++
        }
    }
    if bestLen == 1<<30 {
        return ""
    }
    return s[bestL : bestL+bestLen]
}
```

424) Longest Repeating Character Replacement

Problem statement: Given s and k, return length of longest substring that can become all one character after at most k replacements.

Core pattern: Window with max frequency

Variables to track: `l, r, cnt, maxf, best`

```
flowchart TD
    A[Start with input for problem 424] --> B[Initialize variables: l, r, cnt, maxf, best]
    B --> C[Apply pattern: Window with max frequency]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        cnt = {}
        l = 0
        maxf = 0
        best = 0
        for r, ch in enumerate(s):
            cnt[ch] = cnt.get(ch, 0) + 1
            maxf = max(maxf, cnt[ch])
            while (r - l + 1) - maxf > k:
                cnt[s[l]] -= 1
                l += 1
            best = max(best, r - l + 1)
        return best
```

Go Solution

```go
func characterReplacement(s string, k int) int {
    cnt := [26]int{}
    l, maxf, best := 0, 0, 0
    for r := 0; r < len(s); r++ {
        idx := int(s[r] - 'A')
        cnt[idx]++
        if cnt[idx] > maxf {
            maxf = cnt[idx]
        }
        for (r-l+1)-maxf > k {
            cnt[int(s[l]-'A')]--
            l++
        }
        if r-l+1 > best {
            best = r - l + 1
        }
    }
    return best
}
```

567) Permutation in String

Problem statement: Given s1 and s2, return true if some permutation of s1 appears as a contiguous substring in s2.

Core pattern: Fixed-length frequency window

Variables to track: `m, need, win`

```
flowchart TD
    A[Start with input for problem 567] --> B[Initialize variables: m, need, win]
    B --> C[Apply pattern: Fixed-length frequency window]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
class Solution:
    def checkInclusion(self, s1: str, s2: str) -> bool:
        if len(s1) > len(s2):
            return False
        need = [0] * 26
        win = [0] * 26
        for c in s1:
            need[ord(c) - 97] += 1
        m = len(s1)
        for i, c in enumerate(s2):
            win[ord(c) - 97] += 1
            if i >= m:
                win[ord(s2[i - m]) - 97] -= 1
            if win == need:
                return True
        return False
```

Go Solution

```
func checkInclusion(s1 string, s2 string) bool {
    if len(s1) > len(s2) {
        return false
    }
    var need, win [26]int
    for i := 0; i < len(s1); i++ {
        need[s1[i]-'a']++
    }
    m := len(s1)
    for i := 0; i < len(s2); i++ {
        win[s2[i]-'a']++
        if i >= m {
            win[s2[i-m]-'a']--
        }
        if win == need {
            return true
        }
    }
    return false
}
```

125) Valid Palindrome

Problem statement: Given string s, return true if it is a palindrome after lowercasing and removing non alphanumeric characters.

Core pattern: Two-pointer normalization

Variables to track: `l, r`

```
flowchart TD
    A[Start with input for problem 125] --> B[Initialize variables: l, r]
    B --> C[Apply pattern: Two-pointer normalization]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
class Solution:
    def isPalindrome(self, s: str) -> bool:
        l, r = 0, len(s) - 1
        while l < r:
            while l < r and not s[l].isalnum():
                l += 1
            while l < r and not s[r].isalnum():
                r -= 1
            if s[l].lower() != s[r].lower():
                return False
            l += 1
            r -= 1
        return True
```

Go Solution

```go
func isPalindrome(s string) bool {
    isAlnum := func(c byte) bool {
        return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9')
    }
    toLower := func(c byte) byte {
        if c >= 'A' && c <= 'Z' {
            return c - 'A' + 'a'
        }
        return c
    }
    l, r := 0, len(s)-1
    for l < r {
        for l < r && !isAlnum(s[l]) {
            l++
        }
        for l < r && !isAlnum(s[r]) {
            r--
        }
        if toLower(s[l]) != toLower(s[r]) {
            return false
        }
        l++
        r--
    }
    return true
}
```

33) Search in Rotated Sorted Array

Problem statement: Given rotated sorted array nums and target, return index of target or -1 if missing.

Core pattern: Rotated binary search

Variables to track: `l, r, mid`

```
flowchart TD
    A[Start with input for problem 33] --> B[Initialize variables: l, r, mid]
    B --> C[Apply pattern: Rotated binary search]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        l, r = 0, len(nums) - 1
        while l <= r:
            m = (l + r) // 2
            if nums[m] == target:
                return m
            if nums[l] <= nums[m]:
                if nums[l] <= target < nums[m]:
                    r = m - 1
                else:
                    l = m + 1
            else:
                if nums[m] < target <= nums[r]:
                    l = m + 1
                else:
                    r = m - 1
        return -1
```

Go Solution

```go
func search(nums []int, target int) int {
    l, r := 0, len(nums)-1
    for l <= r {
        m := (l + r) / 2
        if nums[m] == target {
            return m
        }
        if nums[l] <= nums[m] {
            if nums[l] <= target && target < nums[m] {
                r = m - 1
            } else {
                l = m + 1
            }
        } else {
            if nums[m] < target && target <= nums[r] {
                l = m + 1
            } else {
                r = m - 1
            }
        }
    }
    return -1
}
```

153) Find Minimum in Rotated Sorted Array

Problem statement: Given rotated sorted array with distinct values, return the minimum element.

Core pattern: Binary search minimum boundary

Variables to track: `l, r, mid`

```
flowchart TD
    A[Start with input for problem 153] --> B[Initialize variables: l, r, mid]
    B --> C[Apply pattern: Binary search minimum boundary]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List

class Solution:
    def findMin(self, nums: List[int]) -> int:
        l, r = 0, len(nums) - 1
        while l < r:
            m = (l + r) // 2
            if nums[m] > nums[r]:
                l = m + 1
            else:
                r = m
        return nums[l]
```

Go Solution

```go
func findMin(nums []int) int {
    l, r := 0, len(nums)-1
    for l < r {
        m := (l + r) / 2
        if nums[m] > nums[r] {
            l = m + 1
        } else {
            r = m
        }
    }
    return nums[l]
}
```

875) Koko Eating Bananas

Problem statement: Given piles and h, return minimum integer speed k so all bananas are eaten within h hours.

Core pattern: Binary search on answer

Variables to track: `l, r, mid, hours`

```
flowchart TD
    A[Start with input for problem 875] --> B[Initialize variables: l, r, mid, hours]
    B --> C[Apply pattern: Binary search on answer]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
import math
from typing import List

class Solution:
    def minEatingSpeed(self, piles: List[int], h: int) -> int:
        l, r = 1, max(piles)
        while l < r:
            m = (l + r) // 2
            hours = sum(math.ceil(p / m) for p in piles)
            if hours <= h:
                r = m
            else:
                l = m + 1
        return l
```

Go Solution

```go
func minEatingSpeed(piles []int, h int) int {
    l, r := 1, 0
    for _, p := range piles {
        if p > r {
            r = p
        }
    }
    can := func(k int) bool {
        hours := 0
        for _, p := range piles {
            hours += (p + k - 1) / k
        }
        return hours <= h
    }
    for l < r {
        m := (l + r) / 2
        if can(m) {
            r = m
        } else {
            l = m + 1
        }
    }
    return l
}
```

981) Time Based Key-Value Store

Problem statement: Design a time map supporting set(key,value,timestamp) and get(key,timestamp) returning latest value at or before timestamp.

Core pattern: Per-key timeline binary search

Variables to track: `store, timestamps, l, r, mid`

```
flowchart TD
    A[Start with input for problem 981] --> B[Initialize variables: store, timestamps, l, r
    B --> C[Apply pattern: Per-key timeline binary search]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from collections import defaultdict
from bisect import bisect_right
from typing import List, Tuple

class TimeMap:
    def __init__(self):
        self.m = defaultdict(list)

    def set(self, key: str, value: str, timestamp: int) -> None:
        self.m[key].append((timestamp, value))

    def get(self, key: str, timestamp: int) -> str:
        arr: List[Tuple[int, str]] = self.m.get(key, [])
        i = bisect_right(arr, (timestamp, chr(127))) - 1
        return arr[i][1] if i >= 0 else ""
```

Go Solution

```go
type pair struct {
    ts  int
    val string
}

type TimeMap struct {
    m map[string][]pair
}

func Constructor() TimeMap {
    return TimeMap{m: map[string][]pair{}}
}

func (tm *TimeMap) Set(key string, value string, timestamp int) {
    tm.m[key] = append(tm.m[key], pair{ts: timestamp, val: value})
}

func (tm *TimeMap) Get(key string, timestamp int) string {
    arr := tm.m[key]
    l, r := 0, len(arr)-1
    ans := -1
    for l <= r {
        mid := (l + r) / 2
        if arr[mid].ts <= timestamp {
            ans = mid
            l = mid + 1
        } else {
            r = mid - 1
        }
    }
    if ans == -1 {
        return ""
    }
    return arr[ans].val
}
```

102) Binary Tree Level Order Traversal

Problem statement: Given root of binary tree, return level order traversal values by depth.

Core pattern: BFS by levels

Variables to track: `queue, level`

```
flowchart TD
    A[Start with input for problem 102] --> B[Initialize variables: queue, level]
    B --> C[Apply pattern: BFS by levels]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from collections import deque
from typing import List, Optional

class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:
            return []
        q = deque([root])
        out = []
        while q:
            level = []
            for _ in range(len(q)):
                n = q.popleft()
                level.append(n.val)
                if n.left:
                    q.append(n.left)
                if n.right:
                    q.append(n.right)
            out.append(level)
        return out
```

Go Solution

```
func levelOrder(root *TreeNode) [][]int {
    if root == nil {
        return [][]int{}
    }
    q := []*TreeNode{root}
    out := [][]int{}
    for len(q) > 0 {
        sz := len(q)
        level := make([]int, 0, sz)
        for i := 0; i < sz; i++ {
            n := q[0]
            q = q[1:]
            level = append(level, n.Val)
            if n.Left != nil {
                q = append(q, n.Left)
            }
            if n.Right != nil {
                q = append(q, n.Right)
            }
        }
        out = append(out, level)
    }
    return out
}
```

199) Binary Tree Right Side View

Problem statement: Given root of binary tree, return values visible from right side.

Core pattern: BFS keep rightmost per level

Variables to track: `queue`, `level`, `last`

```
flowchart TD
    A[Start with input for problem 199] --> B[Initialize variables: queue, level, last]
    B --> C[Apply pattern: BFS keep rightmost per level]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from collections import deque
from typing import List, Optional

class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []
        q = deque([root])
        out = []
        while q:
            sz = len(q)
            for i in range(sz):
                n = q.popleft()
                if n.left:
                    q.append(n.left)
                if n.right:
                    q.append(n.right)
                if i == sz - 1:
                    out.append(n.val)
        return out
```

Go Solution

```go
func rightSideView(root *TreeNode) []int {
    if root == nil {
        return []int{}
    }
    q := []*TreeNode{root}
    out := []int{}
    for len(q) > 0 {
        sz := len(q)
        for i := 0; i < sz; i++ {
            n := q[0]
            q = q[1:]
            if n.Left != nil {
                q = append(q, n.Left)
            }
            if n.Right != nil {
                q = append(q, n.Right)
            }
            if i == sz-1 {
                out = append(out, n.Val)
            }
        }
    }
    return out
}
```

236) Lowest Common Ancestor of a Binary Tree

Problem statement: Given binary tree root and nodes p and q, return their lowest common ancestor.

Core pattern: Postorder DFS combine results

Variables to track: `node, left, right`

```
flowchart TD
    A[Start with input for problem 236] --> B[Initialize variables: node, left, right]
    B --> C[Apply pattern: Postorder DFS combine results]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeN
        if not root or root == p or root == q:
            return root
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)
        if left and right:
            return root
        return left or right
```

Go Solution

```go
func lowestCommonAncestor(root, p, q *TreeNode) *TreeNode {
    if root == nil || root == p || root == q {
        return root
    }
    left := lowestCommonAncestor(root.Left, p, q)
    right := lowestCommonAncestor(root.Right, p, q)
    if left != nil && right != nil {
        return root
    }
    if left != nil {
        return left
    }
    return right
}
```

200) Number of Islands

Problem statement: Given grid of 1 and 0, return number of islands connected by horizontal or vertical adjacency.

Core pattern: Flood fill each island

Variables to track: `r, c, grid`

```
flowchart TD
    A[Start with input for problem 200] --> B[Initialize variables: r, c, grid]
    B --> C[Apply pattern: Flood fill each island]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List

class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        m, n = len(grid), len(grid[0])
        def dfs(r: int, c: int) -> None:
            if r < 0 or c < 0 or r >= m or c >= n or grid[r][c] != "1":
                return
            grid[r][c] = "0"
            dfs(r + 1, c)
            dfs(r - 1, c)
            dfs(r, c + 1)
            dfs(r, c - 1)
        ans = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == "1":
                    ans += 1
                    dfs(i, j)
        return ans
```

Go Solution

```go
func numIslands(grid [][]byte) int {
    m, n := len(grid), len(grid[0])
    var dfs func(int, int)
    dfs = func(r, c int) {
        if r < 0 || c < 0 || r >= m || c >= n || grid[r][c] != '1' {
            return
        }
        grid[r][c] = '0'
        dfs(r+1, c)
        dfs(r-1, c)
        dfs(r, c+1)
        dfs(r, c-1)
    }
    ans := 0
    for i := 0; i < m; i++ {
        for j := 0; j < n; j++ {
            if grid[i][j] == '1' {
                ans++
                dfs(i, j)
            }
        }
    }
    return ans
}
```

133) Clone Graph

Problem statement: Given reference to node in connected undirected graph, return a deep clone.

Core pattern: DFS or BFS clone with map

Variables to track: `oldToNew`

```
flowchart TD
    A[Start with input for problem 133] --> B[Initialize variables: oldToNew]
    B --> C[Apply pattern: DFS or BFS clone with map]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
class Solution:
    def cloneGraph(self, node: 'Node') -> 'Node':
        if not node:
            return None
        mp = {}
        def dfs(n):
            if n in mp:
                return mp[n]
            copy = Node(n.val)
            mp[n] = copy
            for nei in n.neighbors:
                copy.neighbors.append(dfs(nei))
            return copy
        return dfs(node)
```

Go Solution

```
func cloneGraph(node *Node) *Node {
    if node == nil {
        return nil
    }
    mp := map[*Node]*Node{}
    var dfs func(*Node) *Node
    dfs = func(n *Node) *Node {
        if v, ok := mp[n]; ok {
            return v
        }
        copy := &Node{Val: n.Val}
        mp[n] = copy
        for _, nei := range n.Neighbors {
            copy.Neighbors = append(copy.Neighbors, dfs(nei))
        }
        return copy
    }
    return dfs(node)
}
```

206) Reverse Linked List

Problem statement: Given head of linked list, reverse the list and return new head.

Core pattern: Iterative pointer reversal

Variables to track: `prev, cur, nxt`

```
flowchart TD
    A[Start with input for problem 206] --> B[Initialize variables: prev, cur, nxt]
    B --> C[Apply pattern: Iterative pointer reversal]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        prev, cur = None, head
        while cur:
            nxt = cur.next
            cur.next = prev
            prev = cur
            cur = nxt
        return prev
```

Go Solution

```go
func reverseList(head *ListNode) *ListNode {
    var prev *ListNode
    cur := head
    for cur != nil {
        nxt := cur.Next
        cur.Next = prev
        prev = cur
        cur = nxt
    }
    return prev
}
```

21) Merge Two Sorted Lists

Problem statement: Given two sorted linked lists, merge and return one sorted list.

Core pattern: Two-list merge

Variables to track: `l1, l2, tail`

```
flowchart TD
    A[Start with input for problem 21] --> B[Initialize variables: l1, l2, tail]
    B --> C[Apply pattern: Two-list merge]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```
class Solution:
    def mergeTwoLists(self, list1: ListNode, list2: ListNode) -> ListNode:
        dummy = ListNode(0)
        cur = dummy
        while list1 and list2:
            if list1.val <= list2.val:
                cur.next = list1
                list1 = list1.next
            else:
                cur.next = list2
                list2 = list2.next
            cur = cur.next
        cur.next = list1 or list2
        return dummy.next
```

Go Solution

```
func mergeTwoLists(list1 *ListNode, list2 *ListNode) *ListNode {
    dummy := &ListNode{}
    cur := dummy
    for list1 != nil && list2 != nil {
        if list1.Val <= list2.Val {
            cur.Next = list1
            list1 = list1.Next
        } else {
            cur.Next = list2
            list2 = list2.Next
        }
        cur = cur.Next
    }
    if list1 != nil {
        cur.Next = list1
    } else {
        cur.Next = list2
    }
    return dummy.Next
}
```

143) Reorder List

Problem statement: Given linked list head, reorder nodes to sequence L0 Ln L1 Ln-1 and so on.

Core pattern: Split reverse weave

Variables to track: `slow, fast, second`

```
flowchart TD
    A[Start with input for problem 143] --> B[Initialize variables: slow, fast, second]
    B --> C[Apply pattern: Split reverse weave]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
class Solution:
    def reorderList(self, head: ListNode) -> None:
        slow, fast = head, head.next
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
        second = slow.next
        slow.next = None
        prev = None
        while second:
            nxt = second.next
            second.next = prev
            prev = second
            second = nxt
        first, second = head, prev
        while second:
            t1, t2 = first.next, second.next
            first.next = second
            second.next = t1
            first, second = t1, t2
```

Go Solution

```go
func reorderList(head *ListNode) {
    if head == nil || head.Next == nil {
        return
    }
    slow, fast := head, head.Next
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
    }
    second := slow.Next
    slow.Next = nil

    var prev *ListNode
    for second != nil {
        nxt := second.Next
        second.Next = prev
        prev = second
        second = nxt
    }
    first, second := head, prev
    for second != nil {
        t1, t2 := first.Next, second.Next
        first.Next = second
        second.Next = t1
        first = t1
        second = t2
    }
}
```

141) Linked List Cycle

Problem statement: Given linked list head, return true if list contains a cycle.

Core pattern: Floyd cycle detection

Variables to track: `slow, fast`

```
flowchart TD
    A[Start with input for problem 141] --> B[Initialize variables: slow, fast]
    B --> C[Apply pattern: Floyd cycle detection]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                return True
        return False
```

Go Solution

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

2) Add Two Numbers

Problem statement: Given two reversed linked-list numbers, return their sum as reversed linked list.

Core pattern: Carry-propagating digit sum

Variables to track: `l1, l2, carry`

```
flowchart TD
    A[Start with input for problem 2] --> B[Initialize variables: l1, l2, carry]
    B --> C[Apply pattern: Carry-propagating digit sum]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        dummy = ListNode(0)
        cur = dummy
        carry = 0
        while l1 or l2 or carry:
            v1 = l1.val if l1 else 0
            v2 = l2.val if l2 else 0
            s = v1 + v2 + carry
            carry = s // 10
            cur.next = ListNode(s % 10)
            cur = cur.next
            l1 = l1.next if l1 else None
            l2 = l2.next if l2 else None
        return dummy.next
```

Go Solution

```go
func addTwoNumbers(l1 *ListNode, l2 *ListNode) *ListNode {
    dummy := &ListNode{}
    cur := dummy
    carry := 0
    for l1 != nil || l2 != nil || carry > 0 {
        v1, v2 := 0, 0
        if l1 != nil {
            v1 = l1.Val
            l1 = l1.Next
        }
        if l2 != nil {
            v2 = l2.Val
            l2 = l2.Next
        }
        s := v1 + v2 + carry
        carry = s / 10
        cur.Next = &ListNode{Val: s % 10}
        cur = cur.Next
    }
    return dummy.Next
}
```

56) Merge Intervals

Problem statement: Given intervals, merge overlaps and return merged interval list.

Core pattern: Sort then merge

Variables to track: `intervals, out`

```
flowchart TD
    A[Start with input for problem 56] --> B[Initialize variables: intervals, out]
    B --> C[Apply pattern: Sort then merge]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List

class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort(key=lambda x: x[0])
        out = []
        for s, e in intervals:
            if not out or s > out[-1][1]:
                out.append([s, e])
            else:
                out[-1][1] = max(out[-1][1], e)
        return out
```

Go Solution

```go
import "sort"

func merge(intervals [][]int) [][]int {
    sort.Slice(intervals, func(i, j int) bool { return intervals[i][0] < intervals[j][0] })
    out := [][]int{}
    for _, in := range intervals {
        if len(out) == 0 || in[0] > out[len(out)-1][1] {
            out = append(out, []int{in[0], in[1]})
        } else if in[1] > out[len(out)-1][1] {
            out[len(out)-1][1] = in[1]
        }
    }
    return out
}
```

57) Insert Interval

Problem statement: Given sorted non-overlapping intervals and newInterval, insert and merge as needed.

Core pattern: Append left merge overlap append right

Variables to track: `intervals, newInterval, out`

```
flowchart TD
    A[Start with input for problem 57] --> B[Initialize variables: intervals, newInterval, o
    B --> C[Apply pattern: Append left merge overlap append right]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List

class Solution:
    def insert(self, intervals: List[List[int]], newInterval: List[int]) -> List[List[int]]:
        out = []
        i, n = 0, len(intervals)
        while i < n and intervals[i][1] < newInterval[0]:
            out.append(intervals[i])
            i += 1
        while i < n and intervals[i][0] <= newInterval[1]:
            newInterval[0] = min(newInterval[0], intervals[i][0])
            newInterval[1] = max(newInterval[1], intervals[i][1])
            i += 1
        out.append(newInterval)
        out.extend(intervals[i:])
        return out
```

Go Solution

```go
func insert(intervals [][]int, newInterval []int) [][]int {
    out := [][]int{}
    i, n := 0, len(intervals)
    for i < n && intervals[i][1] < newInterval[0] {
        out = append(out, intervals[i])
        i++
    }
    for i < n && intervals[i][0] <= newInterval[1] {
        if intervals[i][0] < newInterval[0] {
            newInterval[0] = intervals[i][0]
        }
        if intervals[i][1] > newInterval[1] {
            newInterval[1] = intervals[i][1]
        }
        i++
    }
    out = append(out, newInterval)
    out = append(out, intervals[i:]...)
    return out
}
```

435) Non-overlapping Intervals

Problem statement: Given intervals, return minimum removals needed so remaining intervals do not overlap.

Core pattern: Greedy keep earliest end

Variables to track: `lastEnd, removed`

```
flowchart TD
    A[Start with input for problem 435] --> B[Initialize variables: lastEnd, removed]
    B --> C[Apply pattern: Greedy keep earliest end]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List

class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals.sort(key=lambda x: x[1])
        end = float("-inf")
        keep = 0
        for s, e in intervals:
            if s >= end:
                keep += 1
                end = e
        return len(intervals) - keep
```

Go Solution

```go
import "sort"

func eraseOverlapIntervals(intervals [][]int) int {
    sort.Slice(intervals, func(i, j int) bool { return intervals[i][1] < intervals[j][1] })
    end := -1 << 31
    keep := 0
    for _, in := range intervals {
        if in[0] >= end {
            keep++
            end = in[1]
        }
    }
    return len(intervals) - keep
}
```

452) Minimum Number of Arrows to Burst Balloons

Problem statement: Given balloon intervals, return minimum arrows required to burst all balloons.

Core pattern: Greedy arrow at overlap end

Variables to track: `end, arrows`

```
flowchart TD
    A[Start with input for problem 452] --> B[Initialize variables: end, arrows]
    B --> C[Apply pattern: Greedy arrow at overlap end]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List

class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        points.sort(key=lambda x: x[1])
        arrows = 0
        end = float("-inf")
        for s, e in points:
            if s > end:
                arrows += 1
                end = e
        return arrows
```

Go Solution

```go
import "sort"

func findMinArrowShots(points [][]int) int {
    sort.Slice(points, func(i, j int) bool { return points[i][1] < points[j][1] })
    arrows := 0
    end := -(1 << 62)
    for _, p := range points {
        if p[0] > end {
            arrows++
            end = p[1]
        }
    }
    return arrows
}
```

70) Climbing Stairs

Problem statement: Given n stairs, return number of distinct ways to reach top with moves of 1 or 2.

Core pattern: Fibonacci DP

Variables to track: `a, b`

```
flowchart TD
    A[Start with input for problem 70] --> B[Initialize variables: a, b]
    B --> C[Apply pattern: Fibonacci DP]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
class Solution:
    def climbStairs(self, n: int) -> int:
        a, b = 1, 1
        for _ in range(n):
            a, b = b, a + b
        return a
```

Go Solution

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

198) House Robber

Problem statement: Given house values in a line, return max amount robbable without robbing adjacent houses.

Core pattern: Two-state DP

Variables to track: `rob, skip`

```
flowchart TD
    A[Start with input for problem 198] --> B[Initialize variables: rob, skip]
    B --> C[Apply pattern: Two-state DP]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List
```

```
class Solution:
    def rob(self, nums: List[int]) -> int:
        prev2, prev1 = 0, 0
        for x in nums:
            prev2, prev1 = prev1, max(prev1, prev2 + x)
        return prev1
```

Go Solution

```
func rob(nums []int) int {
    prev2, prev1 := 0, 0
    for _, x := range nums {
        cur := prev1
        if prev2+x > cur {
            cur = prev2 + x
        }
        prev2, prev1 = prev1, cur
    }
    return prev1
}
```

322) Coin Change

Problem statement: Given coin denominations and amount, return minimum coins needed or -1 if impossible.

Core pattern: Unbounded knapsack DP

Variables to track: `dp, amount`

```
flowchart TD
    A[Start with input for problem 322] --> B[Initialize variables: dp, amount]
    B --> C[Apply pattern: Unbounded knapsack DP]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```
from typing import List

class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        INF = amount + 1
        dp = [INF] * (amount + 1)
        dp[0] = 0
        for a in range(1, amount + 1):
            for c in coins:
                if c <= a:
                    dp[a] = min(dp[a], dp[a - c] + 1)
        return -1 if dp[amount] == INF else dp[amount]
```

Go Solution

```
func coinChange(coins []int, amount int) int {
    inf := amount + 1
    dp := make([]int, amount+1)
    for i := 1; i <= amount; i++ {
        dp[i] = inf
        for _, c := range coins {
            if c <= i && dp[i-c]+1 < dp[i] {
                dp[i] = dp[i-c] + 1
            }
        }
    }
    if dp[amount] == inf {
        return -1
    }
    return dp[amount]
}
```

139) Word Break

Problem statement: Given string s and dictionary word list, return true if s can be segmented into dictionary words.

Core pattern: Prefix DP segmentation

Variables to track: `dp, j, i`

```
flowchart TD
    A[Start with input for problem 139] --> B[Initialize variables: dp, j, i]
    B --> C[Apply pattern: Prefix DP segmentation]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```
from typing import List

class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        words = set(wordDict)
        n = len(s)
        dp = [False] * (n + 1)
        dp[0] = True
        for i in range(1, n + 1):
            for j in range(i):
                if dp[j] and s[j:i] in words:
                    dp[i] = True
                    break
        return dp[n]
```

Go Solution

```
func wordBreak(s string, wordDict []string) bool {
    words := map[string]bool{}
    for _, w := range wordDict {
        words[w] = true
    }
    n := len(s)
    dp := make([]bool, n+1)
    dp[0] = true
    for i := 1; i <= n; i++ {
        for j := 0; j < i; j++ {
            if dp[j] && words[s[j:i]] {
                dp[i] = true
                break
            }
        }
    }
    return dp[n]
}
```

300) Longest Increasing Subsequence

Problem statement: Given nums, return length of longest strictly increasing subsequence.

Core pattern: Patience tails with binary search

Variables to track: `tails, x`

```
flowchart TD
    A[Start with input for problem 300] --> B[Initialize variables: tails, x]
    B --> C[Apply pattern: Patience tails with binary search]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from bisect import bisect_left
from typing import List

class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        tails = []
        for x in nums:
            i = bisect_left(tails, x)
            if i == len(tails):
                tails.append(x)
            else:
                tails[i] = x
        return len(tails)
```

Go Solution

```go
import "sort"

func lengthOfLIS(nums []int) int {
    tails := []int{}
    for _, x := range nums {
        i := sort.SearchInts(tails, x)
        if i == len(tails) {
            tails = append(tails, x)
        } else {
            tails[i] = x
        }
    }
    return len(tails)
}
```

39) Combination Sum

Problem statement: Given candidates and target, return unique combinations summing to target with unlimited candidate reuse.

Core pattern: Backtracking with reuse

Variables to track: `path, remain, i`

```
flowchart TD
    A[Start with input for problem 39] --> B[Initialize variables: path, remain, i]
    B --> C[Apply pattern: Backtracking with reuse]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List

class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        out = []
        path = []
        candidates.sort()
        def dfs(i: int, rem: int) -> None:
            if rem == 0:
                out.append(path[:])
                return
            if i == len(candidates) or candidates[i] > rem:
                return
            path.append(candidates[i])
            dfs(i, rem - candidates[i])
            path.pop()
            dfs(i + 1, rem)
        dfs(0, target)
        return out
```

Go Solution

```go
func combinationSum(candidates []int, target int) [][]int {
    sort.Ints(candidates)
    out := [][]int{}
    path := []int{}
    var dfs func(int, int)
    dfs = func(i, rem int) {
        if rem == 0 {
            cp := append([]int{}, path...)
            out = append(out, cp)
            return
        }
        if i == len(candidates) || candidates[i] > rem {
            return
        }
        path = append(path, candidates[i])
        dfs(i, rem-candidates[i])
        path = path[:len(path)-1]
        dfs(i+1, rem)
    }
    dfs(0, target)
    return out
}
```

46) Permutations

Problem statement: Given distinct integers nums, return all permutations.

Core pattern: Permutation backtracking

Variables to track: `path, used`

```
flowchart TD
    A[Start with input for problem 46] --> B[Initialize variables: path, used]
    B --> C[Apply pattern: Permutation backtracking]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List

class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        out = []
        def backtrack(i: int) -> None:
            if i == len(nums):
                out.append(nums[:])
                return
            for j in range(i, len(nums)):
                nums[i], nums[j] = nums[j], nums[i]
                backtrack(i + 1)
                nums[i], nums[j] = nums[j], nums[i]
        backtrack(0)
        return out
```

Go Solution

```go
func permute(nums []int) [][]int {
    out := [][]int{}
    var dfs func(int)
    dfs = func(i int) {
        if i == len(nums) {
            cp := append([]int{}, nums...)
            out = append(out, cp)
            return
        }
        for j := i; j < len(nums); j++ {
            nums[i], nums[j] = nums[j], nums[i]
            dfs(i + 1)
            nums[i], nums[j] = nums[j], nums[i]
        }
    }
    dfs(0)
    return out
}
```

78) Subsets

Problem statement: Given unique integers nums, return all subsets.

Core pattern: Include exclude recursion

Variables to track: `i, path`

```
flowchart TD
    A[Start with input for problem 78] --> B[Initialize variables: i, path]
    B --> C[Apply pattern: Include exclude recursion]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```python
from typing import List

class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        out = []
        path = []
        def dfs(i: int) -> None:
            if i == len(nums):
                out.append(path[:])
                return
            path.append(nums[i])
            dfs(i + 1)
            path.pop()
            dfs(i + 1)
        dfs(0)
        return out
```

Go Solution

```
func subsets(nums []int) [][]int {
    out := [][]int{}
    path := []int{}
    var dfs func(int)
    dfs = func(i int) {
        if i == len(nums) {
            cp := append([]int{}, path...)
            out = append(out, cp)
            return
        }
        path = append(path, nums[i])
        dfs(i + 1)
        path = path[:len(path)-1]
        dfs(i + 1)
    }
    dfs(0)
    return out
}
```

79) Word Search

Problem statement: Given board and word, return true if word exists via adjacent cells without reusing a cell in one path.

Core pattern: Backtracking DFS on grid

Variables to track: `idx, r, c`

```
flowchart TD
    A[Start with input for problem 79] --> B[Initialize variables: idx, r, c]
    B --> C[Apply pattern: Backtracking DFS on grid]
    C --> D{Need more iterations or recursion}
    D -- Yes --> E[Update variables and continue]
    E --> C
    D -- No --> F[Return computed answer]
```

Python Solution

```
from typing import List

class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        m, n = len(board), len(board[0])
        def dfs(r: int, c: int, i: int) -> bool:
            if i == len(word):
                return True
            if r < 0 or c < 0 or r >= m or c >= n or board[r][c] != word[i]:
                return False
            tmp = board[r][c]
            board[r][c] = "#"
            ok = dfs(r + 1, c, i + 1) or dfs(r - 1, c, i + 1) or dfs(r, c + 1, i + 1) or dfs
            board[r][c] = tmp
            return ok
        for i in range(m):
            for j in range(n):
                if dfs(i, j, 0):
                    return True
        return False
```

Go Solution

```
func exist(board [][]byte, word string) bool {
    m, n := len(board), len(board[0])
    var dfs func(int, int, int) bool
    dfs = func(r, c, i int) bool {
        if i == len(word) {
            return true
        }
        if r < 0 || c < 0 || r >= m || c >= n || board[r][c] != word[i] {
            return false
        }
        ch := board[r][c]
        board[r][c] = '#'
        ok := dfs(r+1, c, i+1) || dfs(r-1, c, i+1) || dfs(r, c+1, i+1) || dfs(r, c-1, i+1)
        board[r][c] = ch
        return ok
    }
    for i := 0; i < m; i++ {
        for j := 0; j < n; j++ {
            if dfs(i, j, 0) {
                return true
            }
        }
    }
    return false
}
```