LeetCode Core Problems: Python and Go Solutions

Table of Contents

LeetCode Core Problems: Python and Go Solutions

Assumptions:

1. Snippets are LeetCode-style function bodies.
2. Standard LeetCode type definitions are assumed for tree/list/graph nodes.

1. Arrays / Hashing

1) Two Sum

Problem Statement Given an array of integers and a target value return indices of two numbers whose sum equals the target and do not reuse an element.

Deep Dive

1. Identify the core pattern used by `1) Two Sum`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 1 Two Sum] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 1] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 1] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Group strings that are anagrams of each other and return the grouped lists in any order.

Deep Dive

1. Identify the core pattern used by `49) Group Anagrams`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 49 Group Anagrams] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 49] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

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

```python
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 Diagram

```
flowchart TD
    A[Start Go solution for problem 49] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```go
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 Return an array where each position is the product of all numbers in the input except the number at that position without using division.

Deep Dive

1. Identify the core pattern used by `238) Product of Array Except Self`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.

4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 238 Product of Array Except Self] --> B[Initialize data structures a
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 238] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```python
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 Diagram

```
flowchart TD
    A[Start Go solution for problem 238] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
```

```
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```go
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 Count how many contiguous subarrays in an integer array have sum exactly equal to k.

Deep Dive

1. Identify the core pattern used by `560) Subarray Sum Equals K`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 560 Subarray Sum Equals K] --> B[Initialize data structures and poi
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 560] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

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

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 Diagram

```
flowchart TD
    A[Start Go solution for problem 560] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Return the k elements that appear most frequently in an integer array in any order.

Deep Dive

1. Identify the core pattern used by `347) Top K Frequent Elements`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 347 Top K Frequent Elements] --> B[Initialize data structures and po
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 347] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 347] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```go
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]
}
```

2. Sliding Window / Strings

3) Longest Substring Without Repeating Characters

Problem Statement Find the length of the longest substring that contains no repeating characters.

Deep Dive

1. Identify the core pattern used by `3) Longest Substring Without Repeating Characters`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 3 Longest Substring Without Repeating Characters] --> B[Initialize c
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 3] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 3] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Find the minimum length substring of s that contains all characters from t including multiplicity and return an empty string if none exists.

Deep Dive

1. Identify the core pattern used by `76) Minimum Window Substring`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 76 Minimum Window Substring] --> B[Initialize data structures and po
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 76] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 76] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Return the length of the longest substring that can be made of one repeating character after replacing at most k characters.

Deep Dive

1. Identify the core pattern used by `424) Longest Repeating Character Replacement`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 424 Longest Repeating Character Replacement] --> B[Initialize data s
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
```

```
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 424] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 424] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Return true if s2 contains a permutation of s1 as a contiguous substring otherwise return false.

Deep Dive

1. Identify the core pattern used by `567) Permutation in String`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 567 Permutation in String] --> B[Initialize data structures and poi
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 567] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```
class Solution:
    def checkInclusion(self, s1: str, s2: str) -> bool:
```

```python
        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 Diagram

```
flowchart TD
    A[Start Go solution for problem 567] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```go
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 Determine whether a string is a palindrome after converting to lowercase and removing non alphanumeric characters.

Deep Dive

1. Identify the core pattern used by `125) Valid Palindrome`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 125 Valid Palindrome] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 125] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 125] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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
}
```

3. Binary Search

33) Search in Rotated Sorted Array

Problem Statement Search a target value in a rotated sorted array with distinct values and return its index or minus one.

Deep Dive

1. Identify the core pattern used by `33) Search in Rotated Sorted Array`.
2. Track the state variables at each iteration/recursion step.

3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 33 Search in Rotated Sorted Array] --> B[Initialize data structures
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 33] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 33] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Find the minimum element in a rotated sorted array with distinct values.

Deep Dive

1. Identify the core pattern used by `153) Find Minimum in Rotated Sorted Array`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 153 Find Minimum in Rotated Sorted Array] --> B[Initialize data stru
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 153] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 153] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Find the minimum integer eating speed so Koko can finish all banana piles within h hours.

Deep Dive

1. Identify the core pattern used by `875) Koko Eating Bananas`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 875 Koko Eating Bananas] --> B[Initialize data structures and pointe
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 875] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```
import math
from typing import List
```

```python
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 Diagram

```
flowchart TD
    A[Start Go solution for problem 875] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 based key value store supporting set key value timestamp and get key timestamp for the latest value at or before timestamp.

Deep Dive

1. Identify the core pattern used by `981) Time Based Key-Value Store`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 981 Time Based Key-Value Store] --> B[Initialize data structures and
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 981] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 981] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```
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
}
```

4. Trees / Graphs

102) Binary Tree Level Order Traversal

Problem Statement Return the level order traversal of a binary tree as values grouped by depth from left to right.

Deep Dive

1. Identify the core pattern used by `102) Binary Tree Level Order Traversal`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 102 Binary Tree Level Order Traversal] --> B[Initialize data structu
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 102] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

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

```python
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 Diagram

```
flowchart TD
    A[Start Go solution for problem 102] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```go
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 Return the values visible from the right side of a binary tree from top to bottom.

Deep Dive

1. Identify the core pattern used by `199) Binary Tree Right Side View`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 199 Binary Tree Right Side View] --> B[Initialize data structures ar
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 199] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

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

class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
```

```python
        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 Diagram

```
flowchart TD
    A[Start Go solution for problem 199] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 a binary tree and two nodes return their lowest common ancestor.

Deep Dive

1. Identify the core pattern used by `236) Lowest Common Ancestor of a Binary Tree`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 236 Lowest Common Ancestor of a Binary Tree] --> B[Initialize data s
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 236] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 236] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```
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 Count the number of islands in a binary grid where islands are connected groups of land cells.

Deep Dive

1. Identify the core pattern used by `200) Number of Islands`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 200 Number of Islands] --> B[Initialize data structures and pointers
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
```

```
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 200] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 200] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Return a deep copy clone of a connected undirected graph.

Deep Dive

1. Identify the core pattern used by `133) Clone Graph`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 133 Clone Graph] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
```

```
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 133] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 133] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```go
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)
}
```

5. Linked List

206) Reverse Linked List

Problem Statement Reverse a singly linked list and return the new head.

Deep Dive

1. Identify the core pattern used by `206) Reverse Linked List`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 206 Reverse Linked List] --> B[Initialize data structures and pointe
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 206] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 206] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Merge two sorted linked lists and return the merged sorted list.

Deep Dive

1. Identify the core pattern used by `21) Merge Two Sorted Lists`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 21 Merge Two Sorted Lists] --> B[Initialize data structures and poi
```

```
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 21] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```python
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 Diagram

```
flowchart TD
    A[Start Go solution for problem 21] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```go
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 Reorder a linked list from L0 L1 and so on into L0 Ln L1 Ln minus one and so on.

Deep Dive

1. Identify the core pattern used by `143) Reorder List`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 143 Reorder List] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 143] --> B[Initialize state]
```

```
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 143] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Return true if a linked list contains a cycle otherwise false.

Deep Dive

1. Identify the core pattern used by `141) Linked List Cycle`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 141 Linked List Cycle] --> B[Initialize data structures and pointers
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 141] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 141] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Add two non negative integers represented by reversed linked lists and return the sum as a linked list in the same reversed digit order.

Deep Dive

1. Identify the core pattern used by `2) Add Two Numbers`.

2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 2 Add Two Numbers] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 2] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 2] --> B[Initialize state]
```

```
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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
}
```

6. Intervals / Greedy

56) Merge Intervals

Problem Statement Merge all overlapping intervals and return the non overlapping intervals covering the same ranges.

Deep Dive

1. Identify the core pattern used by `56) Merge Intervals`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 56 Merge Intervals] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
```

```
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 56] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 56] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Insert a new interval into sorted non overlapping intervals and merge if needed.

Deep Dive

1. Identify the core pattern used by `57) Insert Interval`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 57 Insert Interval] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 57] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```
from typing import List

class Solution:
```

```python
    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 Diagram

```mermaid
flowchart TD
    A[Start Go solution for problem 57] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Return the minimum number of intervals to remove so the rest are non overlapping.

Deep Dive

1. Identify the core pattern used by `435) Non-overlapping Intervals`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 435 Non-overlapping Intervals] --> B[Initialize data structures and
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 435] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 435] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Return the minimum number of arrows needed to burst all balloons represented as intervals.

Deep Dive

1. Identify the core pattern used by `452) Minimum Number of Arrows to Burst Balloons`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 452 Minimum Number of Arrows to Burst Balloons] --> B[Initialize dat
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
```

```
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 452] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 452] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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
}
```

7. Dynamic Programming

70) Climbing Stairs

Problem Statement Count distinct ways to climb n stairs when each move can be one or two steps.

Deep Dive

1. Identify the core pattern used by `70) Climbing Stairs`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 70 Climbing Stairs] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 70] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 70] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Return the maximum money that can be robbed from houses in a line without robbing adjacent houses.

Deep Dive

1. Identify the core pattern used by `198) House Robber`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 198 House Robber] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 198] --> B[Initialize state]
    B --> C[Process input with pattern logic]
```

```
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 198] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```go
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 a target amount return the fewest coins needed to make that amount or minus one if impossible.

Deep Dive

1. Identify the core pattern used by `322) Coin Change`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 322 Coin Change] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 322] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```python
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 Diagram

```
flowchart TD
    A[Start Go solution for problem 322] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```go
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 Return true if a string can be segmented into one or more dictionary words.

Deep Dive

1. Identify the core pattern used by `139) Word Break`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 139 Word Break] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 139] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
```

```
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```python
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 Diagram

```
flowchart TD
    A[Start Go solution for problem 139] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```go
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 Return the length of the longest strictly increasing subsequence in an array.

Deep Dive

1. Identify the core pattern used by `300) Longest Increasing Subsequence`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 300 Longest Increasing Subsequence] --> B[Initialize data structures
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 300] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 300] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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)
}
```

8. Backtracking

39) Combination Sum

Problem Statement Return all unique combinations of candidate numbers that sum to target where each candidate may be chosen unlimited times.

Deep Dive

1. Identify the core pattern used by `39) Combination Sum`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 39 Combination Sum] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
```

```
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 39] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 39] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Return all possible permutations of a list of distinct integers.

Deep Dive

1. Identify the core pattern used by `46) Permutations`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 46 Permutations] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 46] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 46] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Return all subsets of a list of unique integers.

Deep Dive

1. Identify the core pattern used by `78) Subsets`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 78 Subsets] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 78] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```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 Diagram

```
flowchart TD
    A[Start Go solution for problem 78] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```
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 Return true if a word exists in a grid by sequentially adjacent cells without reusing the same cell in one path.

Deep Dive

1. Identify the core pattern used by `79) Word Search`.
2. Track the state variables at each iteration/recursion step.
3. Enforce the key invariant before moving to the next state.
4. Validate edge cases (empty input, boundaries, duplicates).

Diagram

```
flowchart TD
    A[Input for problem 79 Word Search] --> B[Initialize data structures and pointers]
    B --> C[Iterate / recurse with pattern rules]
    C --> D{Invariant holds?}
    D -- No --> E[Adjust state and continue]
    D -- Yes --> F[Record candidate/best answer]
    E --> C
    F --> G{More work?}
    G -- Yes --> C
    G -- No --> H[Return final answer]
```

Python Solution Diagram

```
flowchart TD
    A[Start Python solution for problem 79] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```python
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 Diagram

```
flowchart TD
    A[Start Go solution for problem 79] --> B[Initialize state]
    B --> C[Process input with pattern logic]
    C --> D[Update running result]
    D --> E{More work remaining}
    E -- Yes --> C
    E -- No --> F[Return final answer]
```

```go
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
}
```