CP Questions

Competitive Programming – Interview/Exam Type Questions

Recursion and Basics

1. **Generate All Subsets of a Set**: Write a recursive function to generate all subsets of a given set.
   - *Explanation*: This involves exploring all combinations of elements using backtracking.
2. **Tower of Hanoi**: Solve for `n` disks.
   - *Explanation*: The problem demonstrates recursive problem-solving, breaking the problem into smaller sub-problems.
3. **Fibonacci Numbers**: Calculate the nth Fibonacci number using recursion.
   - *Explanation*: Highlights the importance of base cases and recursive calls.
4. **Count Number of Increasing Subsequences**: Count all strictly increasing subsequences in an array.
   - *Explanation*: Use dynamic programming to efficiently count subsequences in O(n^2).

Arrays and Array Lists

1. **Find the Missing Number in an Array**: An array contains integers from 1 to n with one missing number. Find it.
   - *Explanation*: Use summation formula or XOR operation for O(n) complexity.
2. **Two Sum Problem**: Find two numbers in an array that add up to a specific target.
   - *Explanation*: Use a hash table to achieve O(n) time complexity.
3. **Subarray with Given Sum**: Find a subarray with a given sum in an unsorted array.
   - *Explanation*: Use a sliding window or prefix sum technique for optimization.

4. **Print Minimum Element of the Array**: Find and print the smallest element in an array.
   - *Explanation*: Traverse the array once to determine the minimum element in O(n).
5. **Print Smallest K Elements in the Same Order**: Extract and print the smallest K elements while maintaining their order in the array.
   - *Explanation*: Use a min-heap for efficient extraction and a list for maintaining order.
6. **Find Duplicate and Missing Elements in O(n)**: Identify one missing and one duplicate element in an array.
   - *Explanation*: Use XOR or sum/difference techniques to achieve O(n) time complexity.
7. **Four Elements Such That a+b = c+d**: Find four numbers in an array that satisfy the equation a+b = c+d.
   - *Explanation*: Use a hash table to store and check pairs of sums efficiently.

Linked Lists

1. **Detect and Remove Loop**: Identify if a loop exists in a linked list and remove it.
   - *Explanation*: Use Floyd's cycle detection algorithm (slow and fast pointers).
2. **Merge Two Sorted Linked Lists**: Merge two sorted linked lists into a single sorted list.
   - *Explanation*: Use a two-pointer approach for efficient merging.
3. **Reverse a Linked List**: Reverse the nodes of a singly linked list.
   - *Explanation*: Use iterative or recursive techniques for in-place reversal.
4. **Last Nth Node of the Linked List**: Find the nth node from the end of a linked list.
   - *Explanation*: Use two pointers where the second pointer starts after advancing the first by n steps.
5. **Find the Middle Element of a Linked List**: Find the middle element of a singly linked list.
   - *Explanation*: Use slow and fast pointer traversal to locate the middle node in one pass.
6. **Remove a Loop in a Single Linked List**: Detect and remove a loop in a linked list.

- *Explanation*: Use Floyd’s algorithm to detect the loop and pointers to remove it.

Strings

1. **Longest Palindromic Substring**: Find the longest palindromic substring in a given string.
   - *Explanation*: Use dynamic programming or expand around the center for efficient computation.
2. **Anagram Check**: Determine if two strings are anagrams of each other.
   - *Explanation*: Use a frequency count approach with a hash table.
3. **String Compression**: Compress a string by replacing repeated characters with their count.
   - *Explanation*: Implement two-pointer traversal for in-place compression.
4. **First Unique Character**: Identify the first unique character in a string.
   - *Explanation*: Use a hash map to store character frequencies and iterate through the string.
5. **Reverse the Individual Words of the String**: Reverse each word in a given string while maintaining the word order.
   - *Explanation*: Split the string by spaces, reverse each word, and join them back.
6. **Custom Case of the Given String**: Implement logic to change a string into a custom case format.
   - *Explanation*: Modify the string as per the specified custom rules (e.g., alternating cases).

Stacks & Queues

1. **Evaluate Postfix Expression**: Given a postfix expression, evaluate its value using a stack.
   - *Explanation*: Push operands onto the stack and apply operators when encountered.
2. **Implement Min Stack**: Design a stack that supports push, pop, and retrieving the minimum element in O(1) time.

- *Explanation*: Use an auxiliary stack to keep track of minimum values.
3. **Check for Balanced Parentheses**: Determine if an expression has balanced brackets.
   - *Explanation*: Use a stack to match opening and closing brackets.
4. **Implement a Stack Using One Queue**: Design a stack using a single queue.
   - *Explanation*: Perform operations by rearranging elements in the queue.
5. **Implement Queue Using Stack**: Design a queue using two stacks.
   - *Explanation*: Use two stacks (one for enqueue, one for dequeue operations).

Trees

1. **Lowest Common Ancestor**: Find the lowest common ancestor of two nodes in a binary tree.
   - *Explanation*: Recursively traverse the tree to find the split point of the two nodes.
2. **Check if a Tree is a BST**: Verify if a binary tree satisfies the binary search tree property.
   - *Explanation*: Use in-order traversal or range checking for validation.
3. **Serialize and Deserialize a Binary Tree**: Convert a binary tree into a string and reconstruct it.
   - *Explanation*: Use pre-order traversal for serialization and de-serialization.
4. **Left View of a Binary Tree**: Print the nodes visible when the tree is viewed from the left.
   - *Explanation*: Use level order traversal to collect the first node of each level.
5. **Flatten a Binary Tree**: Convert a binary tree into a "flattened" linked list-like structure.
   - *Explanation*: Use recursive traversal to modify the tree in-place.
6. **Sum from Root to Leaf**: Calculate the sum of all root-to-leaf paths in a binary tree.
   - *Explanation*: Use DFS to accumulate path sums.

Graphs

1. **Detect Cycle in a Graph**: Detect a cycle in both directed and undirected graphs.
   - *Explanation*: Use DFS with visited and recursion stack arrays for directed graphs, and union-find for undirected graphs.
2. **Dijkstra's Algorithm**: Find the shortest path from a source to all vertices in a graph.
   - *Explanation*: Use a priority queue for efficient shortest path computation.
3. **Topological Sort**: Perform topological sorting of a directed acyclic graph (DAG).
   - *Explanation*: Use Kahn's algorithm or DFS-based approach.
4. **Check Whether the Graph is Bipartite**: Determine if a graph can be colored using two colors.
   - *Explanation*: Use BFS or DFS with alternating colors to validate bipartiteness.
5. **Shortest Distance Between Every Pair**: Find the shortest paths between all pairs of vertices.
   - *Explanation*: Use Floyd-Warshall algorithm for dense graphs.
6. **Detect Loop in a Directed Graph**: Identify if there's a cycle in a directed graph.
   - *Explanation*: Use DFS with a recursion stack to check for back edges.

Additional Problems

1. **Find Symmetric Pairs in a Relation**
   - *Explanation*: Use a hash table to store and check for symmetric pairs.
2. **Median of a Stream of Integers**
   - *Explanation*: Use two heaps (max-heap for lower half and min-heap for upper half).
3. **Last Non-Zero Digit of the Factorial**
   - *Explanation*: Remove trailing zeros by tracking factors of 2 and 5 while multiplying.
4. **Shortest Distance Between Every Pair**

- *Explanation*: Use Floyd–Warshall algorithm for all-pairs shortest path computation.