Arslan Ahmad
Posted on Nov 2, 2023

20 Essential Coding Patterns to Ace Your Next Coding Interview

#coding #interview #algorithms #programming

Navigating through coding interviews requires more than just a good grasp of algorithms and data structures; it demands a strategic approach and a keen eye for patterns. In todays competitive world of tech job interviews, understanding and mastering coding patterns can significantly enhance your problem-solving skills and boost your performance.

Coding patterns, or as we like to call them, are recurring techniques that provide a structured approach to solving complex problems. Think of them as the building blocks of algorithms, helping you to break down problems into more manageable parts.

In this blog, we will explore 20 essential coding patterns that are pivotal for acing coding interviews. We will delve into the pros and cons of each pattern, providing you with a balanced view to help you make informed decisions during your interviews. And

to top it off, we will equip you with real problem examples from the Grokking the Coding Interview course. I'm the author of this course, feel free to reach out to me if you have questions.

Let's go through each pattern one by one.

1. Two Pointers

Description

The Two Pointers technique is a clever strategy used in algorithm design, particularly when dealing with arrays or linked lists. Imagine you have two fingers, and you place each at different ends or positions of an array. These 'fingers' or pointers then traverse through the array, helping you to compare, search, or even manipulate the data efficiently.

Usage

- **Ordered Data Structures**: This pattern shines when applied to ordered arrays or lists, allowing for intelligent, position-based decisions that can significantly optimize the algorithm.
- **Efficiency**: By reducing the need for nested loops, the Two Pointers technique helps in achieving linear time complexity, making your algorithm faster and more efficient.

Pros and Cons

- **Pros**:
  - *Efficiency*: Achieves O(n) time complexity for problems that might otherwise require O(n^2).
  - *Simplicity*: Once mastered, it provides a straightforward and elegant solution.
- **Cons**:
  - *Applicability*: Mainly beneficial for problems involving sequences or intervals.
  - *Initial Complexity*: It might take some time to get the hang of this pattern and understand where and how to move the pointers.

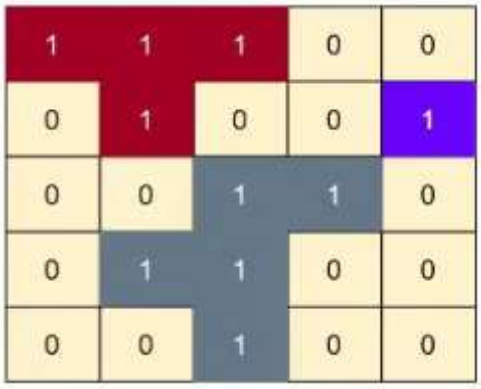
Example Problems from Grokking the Coding Interview

1. Pair with Target Sum: Find a pair in an array that adds up to a specific target sum.
2. Squaring a Sorted Array: Given a sorted array, create a new array containing squares of all the numbers of the input array in the sorted order.
3. Triplet Sum to Zero: Given an array of unsorted numbers, find all unique triplets in it that add up to zero.

2. Island (Matrix Traversal) Pattern

Description

The Island pattern, also known as Matrix Traversal, is a technique used to navigate through a 2D array or matrix. The primary goal is to identify and process contiguous groups of elements, often referred to as 'islands'. This pattern is particularly useful when you need to explore and manipulate grid-based data.



Usage

- **Grid-Based Problems**: Excelling in problems where you need to traverse a grid to find connected components or regions.
- **Contiguous Elements**: Ideal for situations where you need to group together adjacent elements that share a common property.

Pros and Cons

- **Pros**:
  - *Comprehensive*: Provides a thorough way to explore all the elements in a grid.
  - *Versatile*: Can be used to solve a variety of problems related to 2D arrays.
- **Cons**:
  - *Complexity*: Can be more complex to implement compared to linear data structure traversal.
  - *Space Overhead*: May require additional space for recursion or queue/stack for breadth-first/depth-first traversal.

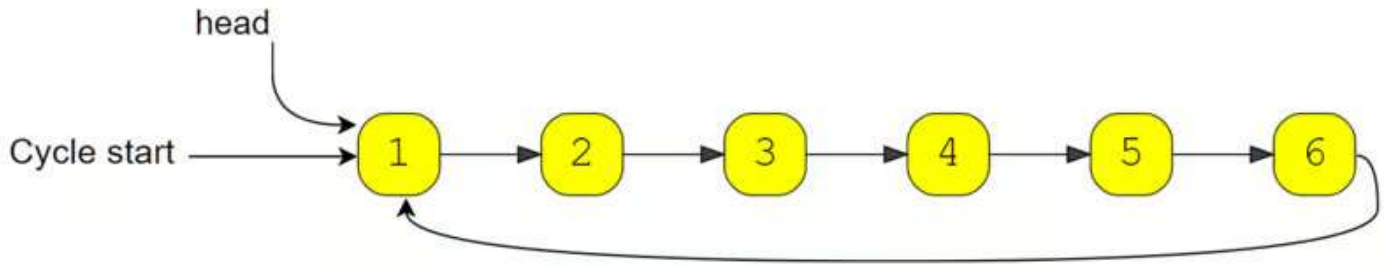
Example Problems from Grokking the Coding Interview

1. Number of Islands: Count the number of islands in a given 2D matrix.
2. Biggest Island: Find the largest island in terms of area or number of cells.
3. Flood Fill: Change the color of an image represented by a 2D array.

3. Fast & Slow Pointers

Description

The Fast & Slow Pointers technique involves two pointers traversing through a data structure at different speeds. This ingenious approach is particularly useful in identifying cycles, finding middle elements, and solving various other problems related to linked lists and arrays.



Usage

- **Cycle Detection**: Perfect for identifying cycles in a linked list or array, which is a common interview question.
- **Finding Middle Elements**: Efficiently find the middle element of a linked list without knowing the length beforehand.
- **Problem-Specific Solutions**: Solve specific problems like finding the start of a cycle in a linked list.

Pros and Cons

- **Pros**:
  - *Space Efficiency*: Achieves solutions without the need for extra space, adhering to O(1) space complexity.
  - *Versatility*: Applicable to a variety of problems, making it a versatile pattern to know.
- **Cons**:
  - *Initial Complexity*: Understanding how to move the pointers and at what speed can be tricky at first.

- *Specificity*: While versatile, it is mostly beneficial for problems related to linked lists and certain array problems.

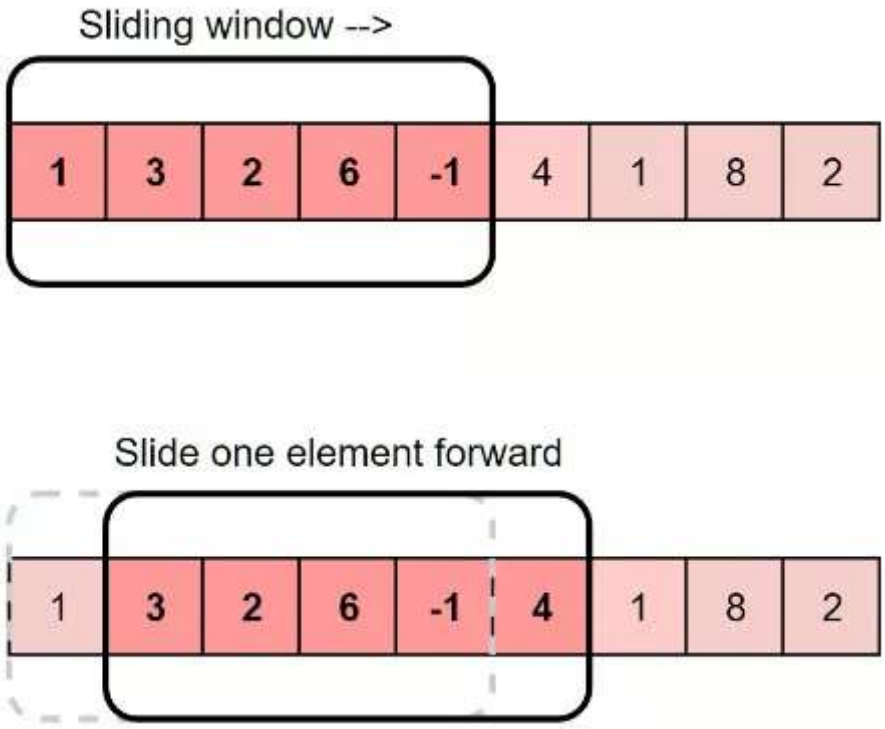
Example Problems from Grokking the Coding Interview

1. LinkedList Cycle: Determine if a linked list has a cycle.
2. Middle of the LinkedList: Find the middle node of a linked list.
3. Palindrome LinkedList: Check if a linked list is a palindrome.

4. Sliding Window

Description

The Sliding Window pattern involves creating a 'window' over a portion of data and sliding it across to solve problems efficiently. This technique is particularly useful for array or list-based problems where you need to find or calculate something among all the contiguous subarrays or sublists of a given size.



Usage

- **Contiguous Subarrays**: Ideal for problems that require you to deal with contiguous subarrays or sublists.
- **Variable Sized Window**: Can be adapted for problems where the window size is not fixed and needs to be adjusted based on certain conditions.

Pros and Cons

- **Pros**:
  - *Efficiency*: Provides a way to reduce time complexity from O(n^2) to O(n) for specific problems.
  - *Versatility*: Can be used for a variety of problems, including maximum sum subarray, smallest subarray with a given sum, and longest substring with K distinct characters.
- **Cons**:
  - *Initial Complexity*: Understanding how to adjust the window size and when to slide the window can be challenging initially.
  - *Specificity*: Mainly beneficial for problems involving contiguous subarrays or sublists.

Example Problems from Grokking the Coding Interview

1. Maximum Sum Subarray of Size K: Given an array of positive numbers and a positive number ‘k’, find the maximum sum of any contiguous subarray of size ‘k’.
2. Fruits Into Baskets: Given an array of characters where each character represents a fruit tree, you are given two baskets, and your goal is to put maximum number of fruits in each basket.
3. Longest Substring with K Distinct Characters: Given a string, find the length of the longest substring in it with no more than K distinct characters.

5. Merge Intervals

Description

The Merge Intervals pattern is a powerful technique used to deal with overlapping intervals or ranges. It involves sorting and then merging intervals based on specific conditions. This pattern is incredibly useful for time-based problems, scheduling, and range manipulation.

Usage

- **Overlapping Intervals**: Perfect for problems where you need to merge overlapping intervals or find if an interval overlaps with any other.
- **Interval Scheduling**: Useful for problems that involve scheduling based on time intervals.

Pros and Cons

- **Pros**:

    - *Clarity*: Provides a clear and systematic way to deal with overlapping intervals.
    - *Efficiency*: Helps in reducing the problem complexity and achieving optimal solutions.
- **Cons**:
    - *Sorting Overhead*: Requires the intervals to be sorted beforehand, which could add to the time complexity.
    - *Specificity*: Mainly beneficial for problems involving intervals and ranges.

Example Problems from Grokking the Coding Interview

1. Merge Intervals: Given a list of intervals, merge all the overlapping intervals to produce a list that has only mutually exclusive intervals.
2. Insert Interval: Given a list of non-overlapping intervals sorted by their start time, insert a given interval at the correct position and merge all necessary intervals to produce a list that has only mutually exclusive intervals.
3. Intervals Intersection: Given two lists of intervals, find the intersection of these two lists. Each list consists of disjoint intervals sorted on their start time.

6. Cyclic Sort

Description

Cyclic Sort is a unique and intuitive sorting algorithm, particularly well-suited for problems where you are given a range of numbers and asked to sort them. The beauty of this pattern lies in its ability to sort the numbers in-place, utilizing the fact that the numbers are consecutive or have a specific range.

Usage

- **Consecutive Numbers**: Ideal for scenarios where you have an array of numbers in a specific range, and you need to sort them or find missing/duplicate numbers.
- **In-Place Sorting**: Provides a way to sort the numbers without using any extra space.

Pros and Cons

- **Pros**:
    - *Space Efficiency*: Achieves sorting without the need for additional space, adhering to O(1) space complexity.
    - *Time Efficiency*: Offers a linear time complexity solution for specific range-based sorting problems.
- **Cons**:

- *Limited Applicability*: Best suited for problems involving numbers in a specific range and may not be applicable for other types of sorting problems.
- *Initial Learning Curve*: Understanding the cyclic sort pattern and knowing when to apply it can take some time.

Example Problems from Grokking the Coding Interview

1. Find the Missing Number: Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.
2. Find all Duplicates: Find all the duplicate numbers (without using extra space and in O(n) runtime).
3. Duplicates In Array: Given an array of integers, 1 ≤ a[i] ≤ n (n = size of array), some elements appear twice and others appear once.

7. In-place Reversal of a Linked List

Description

The In-place Reversal of a Linked List pattern is a technique used to reverse the elements of a linked list without using additional memory. This is achieved by manipulating the pointers of the nodes in the linked list to reverse their direction.

Usage

- **Memory Efficiency**: Since no additional data structures are used, this pattern is memory efficient.
- **Reversing Sub-lists**: Can be extended to reverse sub-lists within a linked list, providing versatility in solving more complex problems.

Pros and Cons

- **Pros**:
  - *Space Efficiency*: Achieves in-place reversal, ensuring O(1) space complexity.
  - *Versatility*: Can be used to solve various problems related to linked lists, including reversing sub-lists and finding palindromes.
- **Cons**:
  - *Pointer Manipulation*: Requires careful manipulation of pointers, which can be error-prone.
  - *Initial Learning Curve*: Understanding how to reverse the pointers without losing the rest of the linked list can be challenging initially.

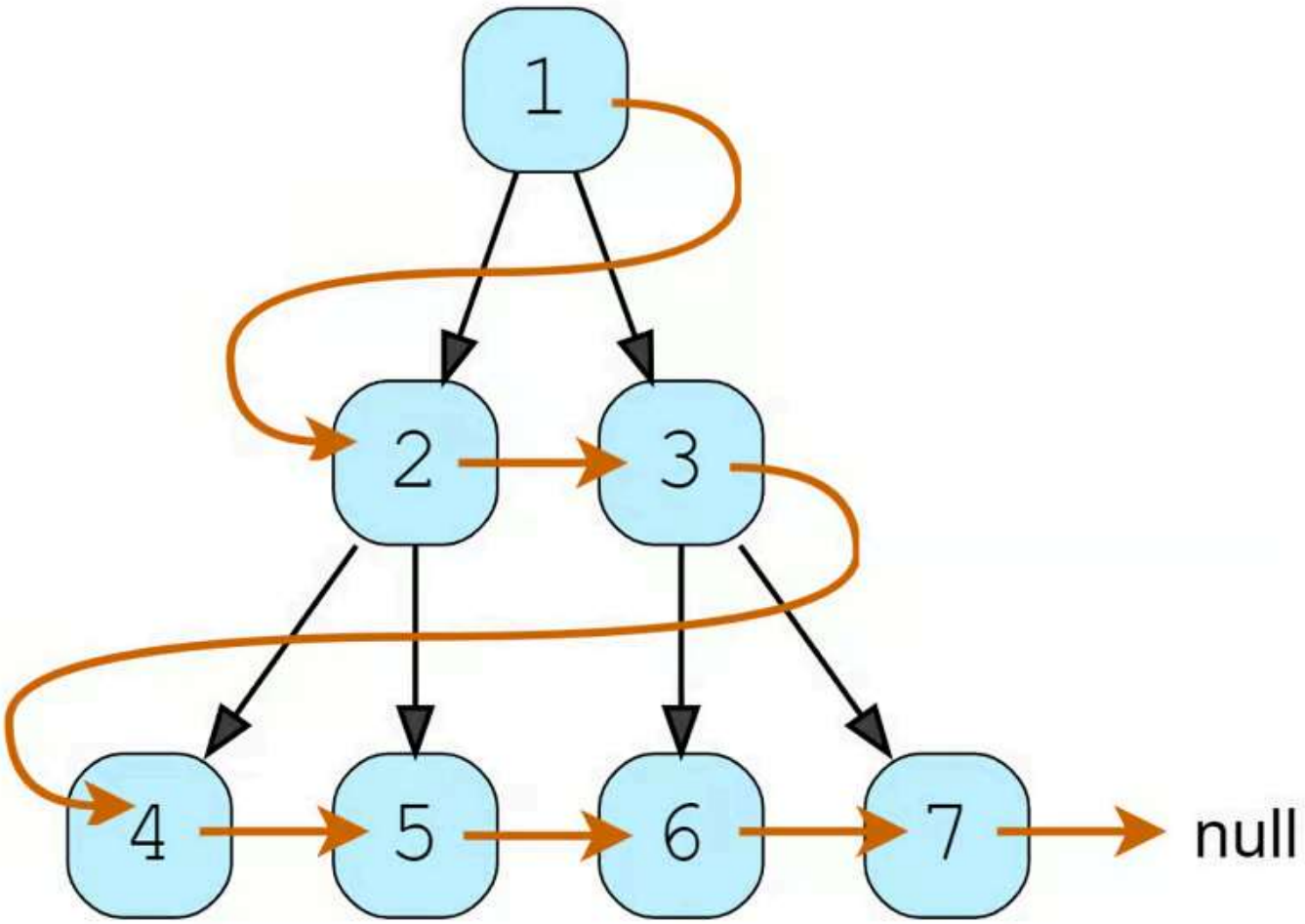
Example Problems from Grokking the Coding Interview

1. Reverse a LinkedList: Given the head of a Singly LinkedList, reverse the LinkedList.
2. Reverse a Sub-list: Given the head of a LinkedList and two positions ‘p’ and ‘q’, reverse the LinkedList from position ‘p’ to ‘q’.
3. Reverse Every K-element Sub-list: Given the head of a LinkedList and a number ‘k’, reverse every ‘k’ sized sub-list starting from the head.

8. Tree Breadth First Search

Description

The Tree Breadth First Search (BFS) pattern involves traversing a tree level by level, ensuring that you visit all the nodes at the current depth before moving on to the nodes at the next depth level. This is usually implemented using a queue.



Usage

- **Level Order Traversal**: Ideal for problems that require you to traverse a tree in level order or when you need to perform operations on nodes at the same depth.
- **Minimum Depth**: Useful for finding the minimum depth of a tree, as you can stop the traversal once you find the first leaf node.

Pros and Cons

- **Pros**:
  - *Complete Traversal*: Ensures that every node in the tree is visited.
  - *Level Order Information*: Provides information about the depth or level of each node.
- **Cons**:
  - *Space Overhead*: Requires additional space for the queue, which can be as large as the number of nodes at the largest level.
  - *Not as Efficient for Depth-Related Queries*: For problems that depend on depth information, a depth-first search might be more efficient.

Example Problems from Grokking the Coding Interview

1. Binary Tree Level Order Traversal: Traverse a tree in level order and return the values of the nodes at each level.
2. Reverse Level Order Traversal: Traverse a tree in reverse level order.
3. Zigzag Traversal: Traverse a tree in a zigzag order.

9. Tree Depth First Search

Description

The Tree Depth First Search (DFS) pattern involves traversing a tree in a depth-first manner, meaning you go as deep as possible down one branch before backing up and exploring other branches. This is typically implemented using recursion or a stack.

Usage

- **Path Finding**: Ideal for problems where you need to find a path or check the existence of a path with certain properties.
- **Complex Tree Traversals**: Useful for more complex tree traversal problems where you need to maintain state or perform operations as you traverse.

Pros and Cons

- **Pros**:
  - *Space Efficiency*: For a balanced tree, DFS uses less space than BFS.
  - *Simplicity*: Recursive implementations can be more straightforward and concise.
- **Cons**:
  - *Can Be Less Efficient for Wide Trees*: For very wide trees, DFS can use more space than BFS.

- *May Not Find the Shortest Path*: If you're looking for the shortest path in an unweighted tree, BFS is generally a better choice.

Example Problems from Grokking the Coding Interview

1. Binary Tree Path Sum: Given a binary tree and a number 'S', find if the tree has a path from root-to-leaf such that the sum of all the node values of that path equals 'S'.
2. All Paths for a Sum: Find all root-to-leaf paths in a binary tree that have a sum equal to a given number.
3. Count Paths for a Sum: Find the number of paths in a tree that sum up to a given value.

10. Two Heaps

Description

The Two Heaps pattern involves using two priority queues (heaps) to maintain a running balance or median of a set of numbers. One heap keeps track of the smaller half of the numbers, and the other keeps track of the larger half.

Usage

- **Running Median**: Perfect for problems where you need to find the median of a set of numbers as new numbers are added.
- **Balanced Partition**: Useful for problems where you need to maintain a balanced partition of numbers.

Pros and Cons

- **Pros**:
  - *Efficiency*: Provides a way to efficiently find the median or maintain balance in O(log N) time.
  - *Dynamic*: Can handle dynamic datasets where numbers are added over time.
- **Cons**:
  - *Complexity*: Implementation can be more complex due to the need to balance the two heaps.
  - *Space Overhead*: Requires additional space to store the numbers in the heaps.

Example Problems from Grokking the Coding Interview

1. Find the Median of a Number Stream: Design a class to calculate the median of a number stream.

2. Sliding Window Median: Find the median of all subarrays of size 'K' in the array.
3. Maximize Capital: Given a set of investment projects with their respective profits, we need to find the most profitable projects. We are given an initial capital and are allowed to invest only in a fixed number of projects. Our goal is to choose projects that give us the maximum profit.

11. Subsets

Description

The Subsets pattern involves dealing with problems that require generating all possible combinations or subsets of a set. This pattern is particularly useful when you need to explore all the different ways to combine elements, which is a common scenario in many coding problems.

Usage

- **Combinatorial Problems**: Ideal for problems where you need to generate all possible combinations of elements.
- **Exhaustive Search**: Useful when you need to perform an exhaustive search over all possible subsets of a set.

Pros and Cons

- **Pros**:
  - *Comprehensive*: Ensures that you consider all possible combinations of elements.
  - *Versatile*: Can be used to solve a variety of problems, including generating power sets, combinations, and permutations.
- **Cons**:
  - *Time Complexity*: Can lead to exponential time complexity, as the number of subsets of a set is 2^N.
  - *Space Complexity*: Requires additional space to store all the subsets.

Example Problems from Grokking the Coding Interview

1. Subsets: Given a set with distinct elements, find all of its distinct subsets.
2. Subsets With Duplicates: Given a set of numbers that might contain duplicates, find all of its distinct subsets.
3. Permutations: Given a set of distinct numbers, find all of its permutations.

12. Modified Binary Search

Description

The Modified Binary Search pattern involves adapting the classic binary search algorithm to solve various problems, often related to searching in a sorted array or finding the boundary of a condition.

Usage

- **Sorted Arrays**: Perfect for problems involving searching or making decisions based on sorted arrays.
- **Finding Boundaries**: Useful for finding the start or end of a condition in a sorted array.

Pros and Cons

- **Pros**:
  - *Efficiency*: Provides a logarithmic time complexity solution for searching problems, making it highly efficient.
  - *Versatility*: Can be adapted to solve a wide range of problems beyond simple searching.
- **Cons**:
  - *Applicability*: Mainly beneficial for problems involving sorted arrays or conditions with clear boundaries.
  - *Implementation Nuances*: Requires careful implementation to handle edge cases and avoid infinite loops.

Example Problems from Grokking the Coding Interview

1. Order-agnostic Binary Search: Given a sorted array of numbers, find the index of a given number. The array could be sorted in ascending or descending order.
2. Ceiling of a Number: Given an array of numbers sorted in ascending order, find the ceiling of a given number. The ceiling of a number is the smallest number in the given array greater than or equal to the given number.
3. Next Letter: Given an array of lowercase letters sorted in ascending order, find the smallest letter in the given array greater than a given ‘key’.

13. Bitwise XOR

Description

The Bitwise XOR pattern involves using the XOR bitwise operator to solve problems, often related to finding missing numbers or duplicate numbers in an array. XOR is a binary operator that returns 1 when the two bits are different and 0 when they are the same.

Usage

- **Finding Missing or Duplicate Numbers**: Ideal for problems where you need to find a missing number or duplicate numbers in an array.
- **Bit Manipulation**: Useful for problems that require manipulation of bits to achieve the desired result.

Pros and Cons

- **Pros**:
  - *Efficiency*: Provides a constant space solution for certain problems, making it highly efficient.
  - *Simplicity*: Once understood, the XOR operator can be used to create elegant and simple solutions.
- **Cons**:
  - *Specificity*: Mainly beneficial for problems involving finding missing or duplicate numbers.
  - *Learning Curve*: Understanding how the XOR operator works and when to use it can take some time.

Example Problems from Grokking the Coding Interview

1. Single Number: In a non-empty array of integers, every number appears twice except for one, find that single number.
2. Two Single Numbers: In a non-empty array of numbers, every number appears exactly twice except two numbers that appear only once. Find the two numbers that appear only once.
3. Complement of Base 10 Number: For a given positive number N in base-10, return the complement of its binary representation as a base-10 integer.

14. Top 'K' Elements

Description

The Top 'K' Elements pattern involves finding the 'K' largest or smallest elements in an array or stream of data. This pattern is particularly useful when dealing with large

datasets and you need to maintain a subset of the data based on certain criteria.

Usage

- **Priority Queue**: Utilizes a min-heap or max-heap to efficiently keep track of the 'K' largest or smallest elements.
- **Streaming Data**: Ideal for scenarios where the data is streaming in, and you need to maintain the 'K' largest or smallest elements at any given time.

Pros and Cons

- **Pros**:
  - *Efficiency*: Provides a way to find the 'K' largest or smallest elements in O(N log K) time.
  - *Space Efficiency*: Only requires O(K) space, regardless of the size of the dataset.
- **Cons**:
  - *Limited to 'K' Elements*: Only maintains information about the top 'K' elements, not the entire dataset.
  - *Heap Maintenance*: Requires careful maintenance of the heap to ensure efficiency.

Example Problems from Grokking the Coding Interview

1. Top 'K' Numbers: Given an unsorted array of numbers, find the 'K' largest numbers in it.
2. Kth Smallest Number: Given an unsorted array of numbers, find the Kth smallest number in it.
3. 'K' Closest Points to the Origin: Given an array of points in the a 2D plane, find 'K' closest points to the origin.

15. K-way Merge

Description

The K-way Merge pattern involves merging multiple sorted arrays or lists into a single sorted list. This pattern is highly useful in scenarios where you have multiple sorted datasets that you need to combine and maintain the sorted order.

Usage

- **Multiple Sorted Arrays**: Ideal for merging multiple sorted arrays or lists.

- **External Sorting**: Useful in external sorting, where the data to be sorted does not fit into memory and is stored in sorted chunks.

Pros and Cons

- **Pros**:
  - *Efficiency*: Provides a way to merge multiple sorted arrays in O(N log K) time, where 'N' is the total number of elements across all arrays, and 'K' is the number of arrays.
  - *Space Efficiency*: Only requires O(K) space for the priority queue.
- **Cons**:
  - *Dependent on Sorting*: The efficiency of this pattern depends on the arrays being sorted.
  - *Priority Queue Overhead*: Requires maintenance of a priority queue, which adds to the complexity.

Example Problems from Grokking the Coding Interview

1. Merge K Sorted Lists: Given an array of 'K' sorted LinkedLists, merge them into one sorted list.
2. Kth Smallest Number in M Sorted Lists: Given 'M' sorted arrays, find the K'th smallest number among all the arrays.
3. Find the Smallest Range Covering Elements from K Lists: Given 'M' sorted arrays, find the smallest range that includes at least one number from each of the 'M' lists.

16. Topological Sort

Description

Topological Sort is a pattern used for linearly ordering the vertices of a directed graph in such a way that for every directed edge (U, V), vertex U comes before V. This pattern is particularly useful in scenarios where you have a set of tasks and some tasks depend on others.

Usage

- **Task Scheduling**: Ideal for problems where tasks need to be scheduled in a specific order, respecting their dependencies.
- **Course Scheduling**: Useful in scenarios like course scheduling where some courses have prerequisites.

Pros and Cons

- **Pros**:
  - *Clarity*: Provides a clear and systematic way to order tasks or vertices.
  - *Detecting Cycles*: Helps in detecting cycles in a directed graph, which is important for understanding if a valid ordering is possible.
- **Cons**:
  - *Applicability*: Mainly beneficial for problems involving directed graphs and ordering of vertices.
  - *Complexity*: Implementation can be complex, especially for beginners.

Example Problems from Grokking the Coding Interview

1. Topological Sort: Given a directed graph, find the topological ordering of its vertices.
2. Tasks Scheduling: Find if it is possible to schedule all the tasks.
3. Tasks Scheduling Order: Find the order of tasks we should pick to finish all tasks.

17. Trie

Description

A Trie, also known as a prefix tree, is a tree-like data structure used to store a dynamic set of strings, where the keys are usually strings. It is particularly useful for retrieval of a key in a dataset of strings, which makes it highly efficient for solving word-based problems.

Usage

- **Autocomplete**: Ideal for implementing autocomplete functionality in search engines or text editors.
- **Spell Checker**: Useful for building spell checkers in word processors.
- **IP Routing**: Used in IP routing to store and search routes.

Pros and Cons

- **Pros**:
  - *Efficiency*: Provides fast retrieval of strings and is more efficient than hash tables or sets when it comes to string keys.
  - *Prefix Searching*: Excellent for problems that require prefix searching or matching.
- **Cons**:
  - *Space Overhead*: Can use more space compared to other data structures when the dataset is sparse.

- *Complexity*: Implementation can be complex, especially when handling deletion of words from the Trie.

Example Problems from Grokking the Coding Interview

1. Insert into and Search in a Trie: Implement insertion and search in a Trie.
2. Longest Common Prefix: Find the longest common prefix of a set of strings.
3. Word Search: Given a 2D board and a word, find if the word exists in the grid.

18. Backtracking

Description

Backtracking is a general algorithmic technique that considers searching through all the possible configurations of a search space in order to solve computational problems. It is particularly useful for optimization problems and when a complete search of the solution space is required. The main idea is to explore each possibility until the solution is found or all possibilities have been exhausted.

Usage

- **Combinatorial Problems**: Ideal for solving problems that require generating all possible configurations like permutations, combinations, and subsets.
- **Puzzle Solving**: Useful for solving puzzles such as Sudoku, crossword puzzles, and the N-Queens problem.

Pros and Cons

- **Pros**:
  - *Completeness*: Ensures that the entire solution space is explored, guaranteeing that the optimal solution will be found if it exists.
  - *Space Efficiency*: Uses less memory as it only needs to store the current state and the decision stack.
- **Cons**:
  - *Time Complexity*: Can lead to exponential time complexity, as it explores all possible configurations.
  - *Optimization Required*: May require additional optimizations like pruning to be practical for larger instances.

Example Problems from Grokking the Coding Interview

1. Subsets: Given a set of numbers, find all of its subsets.

2. Permutations: Given a set of distinct numbers, find all of its permutations.
3. N-Queens: Place N queens on an N×N chessboard so that no two queens threaten each other.

19. Monotonic Stack

Description

A Monotonic Stack is a specialized data structure that maintains elements in a sorted order while supporting stack operations. It is particularly useful for problems where you need to find the next greater or smaller element in an array or when you need to maintain a running maximum or minimum value efficiently.

Usage

- **Next Greater Element**: Ideal for finding the next greater element for each element in an array.
- **Maximum Area Histogram**: Useful for problems like finding the largest rectangular area under a histogram.

Pros and Cons

- **Pros**:
  - *Efficiency*: Provides a way to solve certain problems in linear time, making it highly efficient.
  - *Simplicity*: Once understood, the monotonic stack can lead to concise and elegant solutions.
- **Cons**:
  - *Specificity*: Mainly beneficial for problems involving finding the next greater or smaller element and related problems.
  - *Learning Curve*: Understanding how and when to use a monotonic stack can take some time.

Example Problems

1. **Next Greater Element**: Given an array, find the next greater element for each element in the array.
2. **Maximum Area Histogram**: Given a histogram, find the largest rectangular area under the histogram.
3. **Largest Rectangle in Histogram**: Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest

rectangle in the histogram.

20. 0/1 Knapsack (Dynamic Programming)

Description

The 0/1 Knapsack problem is a classic optimization problem that falls under the category of Dynamic Programming. In this problem, you are given a set of items, each with a weight and a value, and a knapsack with a maximum capacity. The goal is to determine the maximum value that can be accommodated in the knapsack without exceeding its capacity. The "0/1" part of the name reflects the fact that you can't break an item, you either take it or leave it.

Usage

- **Resource Allocation**: Ideal for problems where you need to optimally allocate limited resources to maximize profit or minimize cost.
- **Budgeting**: Useful for budgeting scenarios where you need to choose a subset of projects or investments to maximize return.

Pros and Cons

- **Pros**:
  - *Optimality*: Ensures that the optimal solution is found, provided that the problem satisfies the principle of optimality.
  - *Generality*: Can be adapted to solve a wide variety of optimization problems.
- **Cons**:
  - *Time and Space Complexity*: The naive implementation has a time and space complexity of O(nW), where n is the number of items and W is the capacity of the knapsack. This can be prohibitive for large inputs.
  - *Requires Integer Weights and Values*: The classic 0/1 Knapsack problem requires weights and values to be integers.

Example Problems from Grokking the Coding Interview

1. 0/1 Knapsack: Given the weights and profits of 'N' items, put these items in a knapsack which has a capacity 'C'. The goal is to get the maximum profit out of the items in the knapsack.
2. Equal Subset Sum Partition: Given a set of positive numbers, find if we can partition it into two subsets such that the sum of elements in both subsets is equal.

3. Subset Sum: Given a set of positive numbers, determine if there exists a subset in the set whose sum is equal to a given number ‘S’.

Conclusion

Mastering these patterns is definitely not about memorizing solutions; it’s about understanding the underlying principles and learning how to apply them to a wide array of problems. The versatility of these patterns ensures that you are well-equipped to handle different challenges, making you a formidable candidate in any coding interview.

Remember, practice is key. The more problems you solve using these patterns, the more proficient you will become in recognizing problem types and applying the appropriate patterns. So, keep practicing, stay persistent, and you will find yourself excelling in coding interviews, ready to tackle any problem that comes your way.

Want to read more about coding patterns:

1. Don’t Just LeetCode; Follow the Coding Patterns Instead
2. Top LeetCode Patterns for FAANG Coding Interviews
3. Grokking Dynamic Programming Patterns for Coding Interviews

---

Top comments (3)

Saikiran R Hegde • May 28 • Edited

Please add the correct link "In-place Reversal of a Linked List" onwards. FYI @arslan_ah

Lakin Mohapatra • Jan 10

very nice

Leo • Nov 7 '23

after 6, most of the link are wrong

Arslan Ahmad

Founder www.designgurus.org | Ex-Facebook, Microsoft, Hulu, Formulatrix, Techlogix | Entrepreneur, Software Engineer, Writer.

LOCATION
Seattle, WA

JOINED
Sep 24, 2017

More from Arslan Ahmad

Key Steps to Prepare for a Software Engineer Interview
#interview #systemdesign #leetcode #programming

Top LeetCode Patterns for FAANG Coding Interviews
#programming #career #leetcode #python

System Design Interviews: A Step-By-Step Guide
#programming #distributedsystems #career #architecture