ChatGPT

Categorising Problems by Algorithmic Pattern

The uploaded **all_problems.md** file is a large collection of programming problems (≈1 900 problems) drawn from online judges like *LeetCode*. The first few entries include classic interview questions such as **Two Sum**, **Add Two Numbers**, **Longest Substring Without Repeating Characters**, **Median of Two Sorted Arrays**, **Longest Palindromic Substring** and **ZigZag Conversion**. Because the file contains hundreds of problems it is impractical to list and individually solve every one here; instead, a better approach is to recognise recurring **algorithmic patterns** and assign each problem to the appropriate pattern. Once a pattern is identified, a standard solution template can be applied.

Below is a summary of the major algorithmic patterns that appear repeatedly in the problem set, along with example problems and solution outlines. The examples come from the beginning of the uploaded file.

Algorithmic patterns and example problems

Problem (id & title)	Pattern / topic	Key idea & solution sketch
1. Two Sum	**Hash-table / Two-pointer**	We need to find two indices `i` and `j` such that `nums[i] + nums[j] = target`. The efficient solution uses a hash set or map to store complements: while scanning the array, compute `complement = target – nums[i]` and check if it already exists; if it does, return the indices. This approach runs in `O(n)` time and `O(n)` space. GeeksforGeeks explains that hashing stores each element and checks for its complement during iteration [1].
2. Add Two Numbers	**Linked-list addition**	Two non-empty linked lists represent numbers in reverse order (least significant digit first). The solution traverses both lists simultaneously, adds corresponding digits and a carry, and builds a new list. If one list is shorter, pad zeros or treat missing digits as zero. A recursive version pads the shorter list and propagates carries back to the head [2]. An iterative version maintains a running `carry`, sums digits and attaches nodes to the result in `O(max(m,n))` time.
3. Longest Substring Without Repeating Characters	**Sliding window / two pointers**	Maintain a window `[left, right]` over the string that contains no duplicate characters. Use a set or map to record characters in the window. Expand `right` forward; when a duplicate is seen, shrink `left` until the duplicate is removed. Track the maximum window size. This sliding-window method runs in `O(n)` time and is much faster than the brute-force `O(n²)` approach [3].

Problem (id & title)	Pattern / topic	Key idea & solution sketch
4. Median of Two Sorted Arrays	**Divide & conquer / binary search**	To find the median of two sorted arrays in `O(log(m+n))` time, avoid merging completely. The efficient solution compares the medians `m1` and `m2` of both arrays and recursively discards half of the elements. If `m1 < m2`, the combined median must lie in the right half of the first array and the left half of the second; if `m1 > m2`, discard the left half of the first and the right half of the second. Continue until a base case with ≤2 elements. This divide-and-conquer idea resembles binary search [4].
5. Longest Palindromic Substring	**Expand around center / dynamic programming**	A palindrome is symmetric around its center. For each character (and the space between characters), treat it as a potential center and expand outward while the characters match. Keep track of the longest palindrome found. There are `2n - 1` possible centers for a string of length `n`, and expanding around each center yields an `O(n²)` algorithm with constant extra space [5].
6. ZigZag Conversion	**String simulation**	Write the characters of the input string in a zig-zag pattern across `numRows` rows. Maintain an array of strings for each row, a `row` index and a `direction` flag. Append each character to the current row, and move `row` up or down; when the top or bottom row is reached, reverse the direction. At the end, concatenate all rows. GeeksforGeeks outlines this algorithm in four steps [6]. The simulation runs in `O(n)` time.

Common patterns in the full problem set

Although the uploaded file contains nearly two thousand questions, most of them fall into a small number of recurring categories. Recognising these patterns helps to design solutions quickly. The main categories include:

1. **Array & Hashing:** Problems such as *Two Sum*, *Subarray Sum Equals K*, and *Majority Element* require counting elements or tracking complements. Hash sets or maps are typical tools. The general tactic is to maintain a data structure that supports `O(1)` look-ups to detect previously seen values [1].

2. **Two-Pointers & Sliding-Window:** Many string/array problems ask for the longest, shortest or fixed-length subsequences (e.g., *Longest Substring Without Repeating Characters*, *Minimum Window Substring*, *Container With Most Water*). The sliding-window technique moves two pointers over the input, expanding and shrinking the window as needed [3]. This reduces time complexity from `O(n²)` to `O(n)`.

3. **Linked-List Manipulation:** Questions like *Add Two Numbers*, *Reverse Linked List* or *Merge Two Sorted Lists* involve traversing lists, adjusting pointers and handling edge cases. Common patterns include iterative traversal with dummy nodes and recursion with carry/borrow propagation [2].

4. **Divide & Conquer / Binary Search:** Some problems (e.g., *Median of Two Sorted Arrays*, *Kth Smallest Element in a Sorted Matrix*) require searching sorted data. Using divide-and-conquer or binary search on the answer range can reduce complexity to `O(log n)` [4].

5. **Dynamic Programming:** Problems such as *Longest Palindromic Substring*, *Longest Common Subsequence*, and *0/1 Knapsack* exploit overlapping subproblems. Techniques include memoisation, bottom-up tables and optimisation of space. For palindromic substrings, expanding around centers is an optimised form of dynamic programming [5].

6. **Graph & Tree Traversals:** Breadth-first search (BFS) and depth-first search (DFS) are ubiquitous in questions about graphs and trees (e.g., *Binary Tree Level Order Traversal*, *Number of Islands*). Patterns include using queues for BFS, recursion or stacks for DFS, and colouring/visited sets to avoid re-visiting nodes.

7. **Backtracking & Recursion:** Problems like *Permutations*, *Combination Sum*, and *Sudoku Solver* use backtracking to explore candidate solutions and backtrack when constraints are violated. The pattern is to build partial solutions recursively and prune branches that cannot lead to a valid answer.

8. **Greedy Algorithms:** Certain tasks (e.g., *Jump Game*, *Minimum Number of Arrows to Burst Balloons*) are solved optimally by making locally optimal choices. Proving correctness often relies on exchange arguments or the structure of the problem.

9. **Monotonic Stack & Heap:** Problems that ask for next greater/smaller elements or efficient retrieval of top `k` values (e.g., *Largest Rectangle in Histogram*, *Sliding Window Maximum*) use stacks or heaps to maintain candidate values and achieve `O(n)` or `O(n log n)` complexity.

10. **Bit Manipulation:** Some problems, like *Single Number* or *Bitwise AND of Numbers Range*, require bitwise operations, counting set bits or using XOR's properties.

11. **Matrix / 2-D Arrays:** Tasks such as *Spiral Matrix*, *Rotate Image*, or *Island Perimeter* involve scanning or transforming matrices. Techniques include direction vectors for traversal and in-place transposition/rotation.

12. **Trie & Advanced Data Structures:** Problems like *Implement Trie*, *Word Search II* and *Design Search Autocomplete System* use prefix trees, segment trees or Fenwick trees to support efficient queries and updates.

Conclusion

The uploaded problem set contains hundreds of challenges, but they largely revolve around a handful of algorithmic paradigms. By categorising problems into patterns such as **hashing**, **sliding windows**, **divide**

and conquer, **dynamic programming**, **linked-list manipulation**, **tree/graph traversal**, **backtracking**, **greedy methods**, **monotonic stacks/heaps**, **bit manipulation**, **matrix operations** and **tries**, one can select the correct solution template quickly. The example solutions above illustrate how to apply these templates to representative problems. Armed with these patterns, one can tackle the remaining problems in the set efficiently.

---

1 Two Sum - Pair with given Sum - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/check-if-pair-with-given-sum-exists-in-array/

2 Add Two Numbers Represented by Linked Lists - InterviewBit
https://www.interviewbit.com/blog/add-two-numbers-represented-by-linked-lists/

3 Understanding the Sliding Window Pattern: Efficient Utilization Through Examples | HackerNoon
https://hackernoon.com/how-to-find-the-longest-substring-without-repeating-characters

4 Median of two sorted arrays of same size
https://www.enjoyalgorithms.com/blog/median-of-two-sorted-arrays

5 Longest Palindromic Substring (With Visualization)
https://www.finalroundai.com/articles/longest-palindromic-substring

6 Print Concatenation of Zig-Zag String in 'n' Rows - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/print-concatenation-of-zig-zag-string-form-in-n-rows/