Week 6 editorial

Problem 6.1: Jash's Academic Crisis

This problem asks whether a student, Jash, can complete all his courses without encountering a circular dependency in the prerequisite system. A circular dependency means that a set of courses forms a loop where each course requires another in the set, making it impossible to start any of them.

Intuition:
This is a classic problem that can be modeled as detecting cycles in a directed graph. Each course can be represented as a node, and a prerequisite relationship (course 'a' requires course 'b') can be represented as a directed edge from 'b' to 'a'. If a cycle exists in this graph, then a circular dependency is present, and Jash cannot graduate. If no cycles are found, it means a valid course order exists (a topological sort is possible).

Approach:
We can use Depth-First Search (DFS) to detect cycles in the directed graph.

1. **Represent the graph:** Create an adjacency list where `adj[b]` contains a if course a requires course b. This means an edge goes from b to a.
2. **Maintain states during DFS:** For each course, we need to track three states:
    - `unvisited`: The course has not been visited yet.
    - `visiting`: The course is currently in the recursion stack of the current DFS traversal (meaning we are exploring its prerequisites).
    - `visited`: The course and all its prerequisites have been processed, and no cycle was found through this course.
3. **Perform DFS:**
    - Iterate through all courses. If a course is `unvisited`, start a DFS from it.
    - In the DFS function:
        - Mark the current course as `visiting`.
        - For each prerequisite of the current course:
            - If the prerequisite is `visiting`, a cycle is detected. Return `false` (cannot graduate).
            - If the prerequisite is `unvisited`, recursively call DFS on it. If the recursive call returns `false`, propagate `false` upwards.
        - Mark the current course as `visited`.
        - Return `true` (no cycle found through this path).
4. **Final Result:** If all DFS traversals complete without detecting any cycles, then Jash can graduate. Print "YES". Otherwise, print "NO".

Time Complexity:

- **Time:** O(N + M), where N is the number of courses and M is the number of prerequisite relationships. This is because each node and each edge will be visited at most once during the DFS traversal.

Space Complexity:

- **Space:** O(N + M) for storing the adjacency list and the `visited` array (or equivalent data structures for tracking states).

Pseudo Code: https://pastebin.com/duumVRHB

Problem 6.2: Jash's Stock Trading Adventure

Jash wants to maximize profit from stock trading, but with a unique rule: after selling a stock, he cannot buy it again the very next day (a one-day cooldown). He can make multiple transactions.

Intuition:
This problem can be solved using dynamic programming. At any given day, Jash can be in one of two states: either he owns a stock, or he doesn't. The cooldown period adds a third consideration when he sells. We need to make decisions (buy, sell, or do nothing) that lead to the maximum profit.

Approach:
We can define a 2D DP array, `dp[i][k]`, where `i` represents the current day and k represents the state:

- `dp[i][0]` will store the maximum profit on day `i` if Jash does not own a stock.
- `dp[i][1]` will store the maximum profit on day `i` if Jash owns a stock.

We iterate backward from the last day to the first day to fill the DP table.

1. **If Jash owns a stock (`k = 1`):**
   - He can either sell the stock today: `prices[i] + dp[i + 2][0]` (profit from selling today plus profit from day `i + 2` because of the one-day cooldown). If `i + 2` is beyond the array bounds, it's just `prices[i]`.
   - Or he can choose to hold the stock: `dp[i + 1][1]` (profit from the next day if he continues to hold).
   - `dp[i][1]` will be the maximum of these two options.
2. **If Jash does not own a stock (`k = 0`):**
   - He can either buy a stock today: `dp[i + 1][1] - prices[i]` (profit from the next day if he buys today, minus the cost of buying).
   - Or he can choose to do nothing (cooldown or simply not buy): `dp[i + 1][0]` (profit from the next day if he remains without a stock).
   - `dp[i][0]` will be the maximum of these two options.

The base cases for the DP array are when `i` goes beyond the number of days. The final answer will be `dp[0][0]` (or `dp[0][1]` if we consider starting with the intent to buy on day 0, as in the provided code).

Time Complexity:

- **Time:** O(N), as we iterate through each day once, and for each day, we perform constant-time operations.

Space Complexity:

- **Space:** O(N) for the DP table. This can be optimized to O(1) by only storing the previous few days' states, as each day's calculation only depends on the next one or two days.\

Pseudo Code: https://pastebin.com/fUgP8Uuz

Problem 6.3: Jash and the Sorting Challenge

This problem asks for the minimum number of splitting operations to transform a sequence of positive integers into a non-decreasing order. A single card bearing number X can be fractured into precisely two new cards whose values sum to X. Every split inserts new cards, shifting subsequent elements.

Intuition:
To minimize operations, we should aim to make each number as large as possible while still satisfying the non-decreasing condition with the element to its right. This suggests a greedy approach working backward from the end of the array. For each element `curr` (at index `i`), we compare it with the `prev` element (which is the effective value of the element at index `i+1` after potential splits). If `curr` is already less than or equal to `prev`, no operations are needed, and `curr` becomes the new `prev`. If `curr` is greater than `prev`, we must split `curr`. To minimize splits, we want to divide `curr` into parts that are as large as possible, but none of which can exceed `prev`. The optimal strategy is to split `curr` into `parts` such that each part is approximately `prev` (or slightly less for the last part if `curr` is not a multiple of `prev`). The number of parts will be `ceil(curr / prev)`. This requires `ceil(curr / prev) - 1` splits. After splitting, the largest possible value for the leftmost part of `curr` (which will now be at index `i`) will be `curr / parts`. This new value becomes the `prev` for the next iteration (i.e., for `nums[i-1]`).

Approach:

1. Initialize `ans` (total operations) to 0.
2. Set `prev` to the last element of the array (`nums[n-1]`).
3. Iterate backward from the second-to-last element (`i = n-2`) down to the first element (`i = 0`).
4. In each iteration, let `curr` be `nums[i]`.
5. If `curr > prev`:
    - Calculate `parts = ceil((double) curr / prev)`. This is the minimum number of pieces `curr` must be split into.

    - Add `parts - 1` to ans (each split creates one new part, so `parts` parts require `parts - 1` splits).
    - Update `prev` to `curr / parts`. This represents the largest possible value for the leftmost piece of `curr` after splitting, ensuring the non-decreasing order is maintained for the next comparison.
6. Else (`curr <= prev`):
    - No splits are needed for `curr`.
    - Update `prev` to `curr` for the next comparison.
7. Return `ans`.

Time Complexity:

- **Time:** O(N), as we iterate through the array once. The `ceil` and division operations are constant time.

Space Complexity:

- **Space:** O(1), as we only use a few variables to store the answer and previous element.

Pseudo Code: https://pastebin.com/iGursnfQ

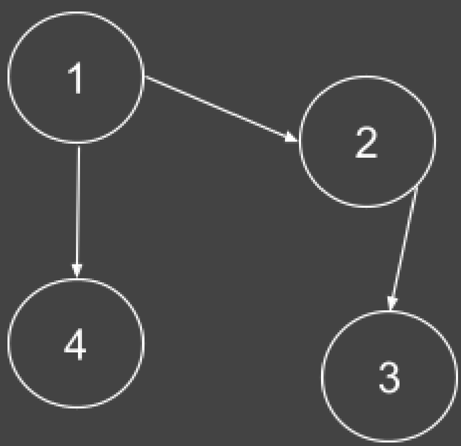
Problem 6.4: Ryn Blackwood's Adventure: Labyrinth of Lost Voices

The naive way to solve this problem is to take each node (apart from s) as t and then check whether there is a path that will not coincide with each other. But this operation is very costly since N is 2e5.

To optimise it, we use dfs with a from array, parent array and visited array which marks 3 states:

- `0`: node not visited
- 1: node visited but not fully explored
- 2: node fully explored (no more children left)

For example, in the graph 1→2→3 and 1→4 (starting from `s = 1`):



`vis[1]=1` → `vis[2]=1` → `vis[3]=1`, **and since 3 has no children,** `vis[3]=2`**. Then backtrack:** `vis[2]=2`**, explore** `4`**, set** `vis[4]=2`**, and finally** `vis[1]=2`**.**

During DFS, if we encounter a node that's already explored (`vis[u] = 2`)**, it may indicate a valid target node. To verify this, we use the from array, where** `from[x]` **denotes the source node from which** `x` **was reached.**

To ensure paths don't overlap, we check that `from[u] ≠ from[v]` **for nodes** `u` **and** `v` **with** `vis[u] = 2`**. The parent array then helps reconstruct the final path.**

Time Complexity:

O(n+m) as each node is visited just once and each edge is explored utmost once. All other operations are O(1)

Pseudo code: https://pastebin.com/SPS3wby5

Problem 6.5: Cash & Clash

This problem involves finding the minimum total cost to have the n-th fighter stand in an arena, given a set of fighters with various trait values and costs. The core challenge is to determine the shortest path from a starting fighter to a target fighter, considering that increasing an attribute of one fighter can help defeat another.

Intuition:
The problem can be modeled as finding the shortest path in a graph. Each fighter can be considered a node, and the ability of one fighter to defeat another (possibly after increasing an attribute) represents a directed edge. The cost of increasing an attribute would be the weight of this edge. A brute-force graph construction would be too slow. The key is to optimize graph construction by considering each attribute separately and using virtual nodes to represent attribute increases.

Approach:
The problem can be solved by constructing a specialized graph and finding the shortest path.

1. **Graph Representation:**
    - Consider each fighter as a node.
    - An edge u -> v exists if fighter u can defeat fighter v .
2. **Optimized Graph Building (Attribute-wise):**
    - Instead of brute-forcing all possible defeat scenarios, process each attribute independently.
    - For each attribute (e.g., the x-th attribute), create 2n virtual nodes: X1, ..., Xn and

Y1, ..., Yn.

- Sort all a_i, x values (trait values for the x -th attribute for all fighters). Let these sorted values be val_1, ..., val_n.
- Add edges between virtual nodes:
  X_i -> X_{i+1} with a weight of (val_{i+1} - val_i) for 1 <= i < n. This represents the cost to increase an attribute to the next sorted value.
- Connect real fighter nodes to these virtual nodes based on their attribute values. For example, if fighter j has attribute x value val_k, you might have edges like j -> X_k and Y_k -> j.
- The goal is to find the shortest path from the n-th fighter to the 1st fighter in this constructed graph. This can be achieved using algorithms like Dijkstra's.

Time Complexity:

The time complexity will depend on the number of fighters (nodes), attributes, and the efficiency of the shortest path algorithm. Sorting attributes takes O(m * n log n). The shortest path algorithm (e.g., Dijkstra's) on the constructed graph (which has O(n * m) nodes and edges) would be roughly O(E log V), where V is the number of nodes and E is the number of edges.

Space Complexity:

The space complexity will be dominated by storing the graph, which would be O(n * m) for the adjacency list representation.

Pseudo Code: https://pastebin.com/zpQ60DUw