Week 5 editorial

Problem 5.1: Amogh Vs Creature

This problem asks for the minimum number of spell casts to completely drain a creature's power n. A spell targets one bit at a time, reducing its strength.

Intuition:
The problem statement and examples clearly indicate that each spell cast drains exactly one "set bit" (a bit with value 1) from the creature's power. Therefore, the minimum number of spell casts required is simply the count of set bits in the binary representation of n.

Steps:

1. For each given power n, convert n to its binary representation.
2. Count the number of '1's (set bits) in the binary string.
3. Print this count as the minimum number of spell casts.

Edge cases:

- If `n = 0`, the count of set bits is 0. (Though constraints state `n >= 1`).
- Powers that are powers of 2 (e.g., 1, 2, 4, 8, 16) will have only one set bit, resulting in 1 spell.

Time complexity:

- **Time:** O(log n) for each test case, as we iterate through the bits of n. Since n can be up to 10^9, `log n` is approximately 30 (for base 2).
- **Space:** O(1) or O(log n) depending on how the binary conversion is handled.

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

Problem 5.2: Dr. Pratham's Space Endeavours

This problem asks whether a drone, starting at point P and needing to reach point Q after exactly n hops with given lengths a_i, can devise a sequence of directions.

Intuition:
This is a classic problem that can be simplified by considering the minimum and maximum possible distances the drone can be from its starting point after n hops.
Let D be the Euclidean distance between P and Q.
The sum of all hop lengths is S = a_1 + a_2 + ... + a_n.
The drone can reach Q if and only if the distance D is achievable.
The minimum distance from P the drone can be after n hops is when all hops are aligned to minimize the final distance. This is |a_1 - a_2 - ... - a_n| (if you consider all hops in one line and some are forward and some are backward). More generally, it's max(0, S - 2 * max_hop_sum_subset) where max_hop_sum_subset is the largest sum of a subset of hops whose sum is less than or equal to S/2. A simpler way to think about it is that the minimum distance is max(0, 2 * max(a_i) - S) if there's a single dominant hop, or more generally, the sum of all hops minus twice the sum of the smallest hops that can "cancel out" the largest ones.

However, a more direct and correct approach for 3D space is to consider the triangle inequality. After n hops, the final position must be within a sphere of radius S centered at P. Also, the final position must be reachable by combining the vectors.

The problem essentially boils down to: can we form a polygon with side lengths a_1, ..., a_n such that the vector sum from P to Q is achievable?

The key insight for this type of problem in 3D (and 2D) is that if the sum of n-1 hop lengths is S', and the last hop length is a_n, then the drone can reach any point within the range [|S' - a_n|, S' + a_n] relative to the starting point of the n-th hop.

More generally, let S = sum(a_i). The minimum distance from P that can be reached is max(0, 2 * max(a_i) - S) (if one hop is significantly larger than the sum of all others, it dictates the minimum possible reach), and the maximum distance is S.

So, the drone can reach Q if and only if the distance D between P and Q satisfies:
max(0, 2 * max(a_i) - S) <= D <= S.

Steps:

1. Calculate the Euclidean distance `D` between `P=(px,py,pz)` and `Q=(qx,qy,qz)`: `D = sqrt((px-qx)^2 + (py-qy)^2 + (pz-qz)^2)`.
2. Calculate the sum of all hop lengths: `S = sum(a_i)`.
3. Find the maximum hop length: `max_a = max(a_i)`.
4. Check if `D <= S` and `D >= max(0, 2 * max_a - S)`.
   If both conditions are true, print "Yes". Otherwise, print "No".

Edge cases:

- If `n = 0`, then `P` must be equal to `Q`.
- If `P` and `Q` are the same point, `D = 0`. This is possible if `S` is even and `max_a <= S/2`, or more generally, if `0` falls within the reachable range.

Time complexity:

- **Time:** O(N) for calculating sum and max of hop lengths for each test case.
- **Space:** O(1) for storing variables.

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

Problem 5.3: Bandra Promenade

This problem asks to calculate the number of plots where new streetlights can be installed on a promenade of `n` rows and `m` columns, given `k` reserved stretches. Overlapping or touching reserved stretches in the same row should be treated as a single reserved block.

Intuition:
The total number of plots is `n * m`. We need to subtract the total number of reserved plots. Since reserved stretches can overlap within a row, we need an efficient way to merge these overlaps and count the unique reserved plots per row. A common approach for merging intervals is to sort them and then iterate, merging as needed.

Steps:

1. Initialize `total_reserved_plots = 0`.
2. Group reserved stretches by row. A map or dictionary where keys are row numbers and values are lists of `(c1, c2)` intervals would be suitable.
3. For each row:
   a. Get all reserved stretches for that row.
   b. Sort these stretches by their starting column `c1`.
   c. Merge overlapping intervals:
   i. Initialize `merged_intervals = []`.
   ii. Iterate through the sorted intervals. If the current interval overlaps with the last interval

in `merged_intervals`, merge them by updating the end column of the last interval. Otherwise, add the current interval to `merged_intervals`.
d. Calculate the number of reserved plots in this row from the `merged_intervals` (sum of `c2 - c1 + 1` for each merged interval).
e. Add this count to `total_reserved_plots`.

4. The total available plots for streetlights will be `(n * m) - total_reserved_plots`. Print this value.

Edge cases:

- No reserved stretches (`k = 0`): `total_reserved_plots` will be 0, and the answer is `n * m`.
- All plots in a row are reserved.
- Large values of n and `m` (up to 10^9) mean we cannot use a 2D array. The approach of grouping by row and merging intervals is efficient enough.

Time complexity:

- **Time:** O(K log K) due to sorting intervals within each row (in the worst case, all K intervals are in one row). Grouping by row using a hash map takes O(K). Merging intervals for each row takes O(number of intervals in that row).
- **Space:** O(K) for storing the grouped intervals.

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

Problem 5.4: Bandra Neighborhood

This problem asks to count the number of separate building clusters in an `m` rows by `n` columns map, where 'B' is a building and 'E' is an empty lot. Clusters are connected horizontally or vertically, and city edges are considered empty.

Intuition:
This is a classic graph traversal problem, specifically counting connected components in a grid. We can use either Breadth-First Search (BFS) or Depth-First Search (DFS) to find and mark visited buildings within a cluster. Each time we start a new traversal from an unvisited building, it signifies a new cluster.

Steps:

1. Initialize `cluster_count = 0`.
2. Create a 2D `visited` array (or set of coordinates) of the same dimensions as the map, initialized to `false`.
3. Iterate through each cell `(r, c)` of the map from `(0, 0)` to `(m-1, n-1)`.
4. If the cell `(r, c)` contains a 'B' (building) and has not been visited:
   a. Increment `cluster_count`.
   b. Start a traversal (BFS or DFS) from `(r, c)`:
   i. Add `(r, c)` to a queue (for BFS) or call a recursive function (for DFS).
   ii. Mark `(r, c)` as visited.
   iii. While the queue is not empty (BFS) or the recursive calls continue (DFS):
   * Dequeue/process the current cell.
   * For each of its four neighbors (up, down, left, right):
   * If the neighbor is within grid bounds, contains a 'B', and has not been visited:
   * Mark it as visited and add it to the queue/make a recursive call.
5. After iterating through all cells, print `cluster_count`.

Edge cases:

- An empty map or a map with no buildings ('B's) will result in 0 clusters.
- A map entirely filled with buildings will result in 1 cluster.
- The "city edges are considered empty lots" means we don't need to worry about clusters extending beyond the map boundaries.

Time complexity:

- **Time:** O(M * N) because each cell is visited at most a constant number of times (once for checking, once for traversal if it's a building).
- **Space:** O(M * N) for the `visited` array and the queue/recursion stack.

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

Problem 5.5: Ryn Blackwood's Adventure: Escape

This problem asks for the minimum number of tiles Ryn must visit to escape from the top-left `(0,0)` to the bottom-right `(n-1, m-1)` of an `n x m` grid, with safe (0) and unsafe (1) tiles. Ryn can move up, down, left, right, and can make at most k magical hops (jumping over one unsafe tile).

Intuition:
This is a shortest path problem on a grid with a special ability, which suggests a Breadth-First Search (BFS). Since Ryn has a limited number of hops, we need to incorporate the remaining hops into the state of our BFS.

A state in our BFS should be `(row, col, hops_remaining)`. The distance will be the number of safe tiles visited.

Steps:

1. Initialize a 3D `distance` array (or map) `dist[n][m][k+1]` with infinity, representing the minimum tiles to reach `(row, col)` with `hops_remaining`.
2. Create a queue for BFS and add the starting state: `(0, 0, k)`.
3. Set `dist[0][0][k] = 1` (starting tile counts as 1).
4. While the queue is not empty:
   a. Dequeue `(r, c, hops)`.
   b. If `(r, c)` is the target `(n-1, m-1)`, we have found a path. The current `dist[r][c][hops]` is a candidate for the minimum.
   c. Explore four direct neighbors (up, down, left, right):
   i. For a neighbor `(nr, nc)`:
   * If `(nr, nc)` is safe (grid value 0) and `dist[r][c][hops] + 1 <`

`dist[nr][nc][hops]`:
* Update `dist[nr][nc][hops] = dist[r][c][hops] + 1` and enqueue `(nr, nc, hops)`.
d. Explore four hop neighbors (jumping over one tile):
i. For a hop direction, consider `(nr1, nc1)` (the intermediate tile) and `(nr2, nc2)` (the landing tile).
* If `hops > 0`, `(nr1, nc1)` is unsafe (grid value 1), `(nr2, nc2)` is safe (grid value 0), and `dist[r][c][hops] + 1 < dist[nr2][nc2][hops-1]`:
* Update `dist[nr2][nc2][hops-1] = dist[r][c][hops] + 1` and enqueue `(nr2, nc2, hops-1)`.

5. After the BFS completes, find the minimum value among `dist[n-1][m-1][hops]` for all `hops` from 0 to `k`. If no path is found (all values are infinity), print -1.

Edge cases:

- Start and end are the same: 1 tile.
- No path exists, even with hops: print -1.
- `k = 0`: Standard BFS without hops.

Time complexity:

- **Time:** O(N * M * K) because each state `(r, c, hops)` is visited at most once, and from each state, we explore a constant number of neighbors.
- **Space:** O(N * M * K) for the `distance` array and the queue.

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