# Sylhet ICPC 2024 Collaborative Challenge: Episode 1

## Problemset Analysis

## Problem A: Free Food

### Type: Ah Hoc, Implementation

It was the easiest problem in the set and intended as the giveaway problem. The problem requires finding how many distinct days have free food across multiple events, where each event covers a specific range of days. To solve this, we can use a boolean array of size 365 (representing each day of the year). For each event, we mark the days in the array corresponding to the event's range `[si, ti]`. After marking all events for a test case, we count how many days were marked as having free food.

## Problem B: Dancing Digits

### Type: BFS

We can solve this problem using a breadth-first search (BFS), the algorithm explores all possible sequences, starting from the given arrangement, by encoding each configuration as a state and finding the shortest path to the target sorted state, where only valid prime-sum dances are allowed.

## Problem C: Undo History

### Type: Simulation

**Understanding the problem:** Some problems have heavy problem statements. It is best to make sure to understand them correctly before trying to solve them. A formal way to explain the problem:

`UndoHistory`: is a set of strings, initially contains `“”`.
`results`: list of strings, initially empty.
`buffer`: a string, initially `“”`.

There are three operations available:

**Enter**: Append buffer to results. (Costs 1).
**Type**: Append any letter to buffer. UndoHistory = UndoHistory U {buffer}. (Add the new contents of the buffer to UndoHistory. (Costs 1)
**Restore**: buffer = (Pick any element of UndoHistory). (Costs 2). This replaces the contents of the buffer. If it appended contents to the buffer, even some of the examples would be impossible (How to empty the buffer without replacing it with `“”` from the history?).

The objective is to make `results = lines` (as given by the input) using the minimum cost possible.

**The first line:** Typing the first line is straight-forward. We can just type all of the needed letters. There is no reason to use the UndoHistory. Then press enter. In total, we need `length(lines[0])+1` operations.

**The second line:** It may now be better to restore from the undo history.

The key is to notice that in order to type the first line. Each of the prefixes of `lines[0]` were at one point or another added to the undo history. We can restore any prefix of `lines[0]` with a cost equal to 2. If we find that any of these prefixes is also a prefix of `lines[1]`, then we may restore it, complete the remaining characters of `lines[1]` and press enter.

It may also be possible that `lines[0]` is a prefix of `lines[1]`, in that case it is possible to type the remaining characters of `lines[1]` and then press enter. For this case only, if this is possible, it is always less expensive to do this.

The solution is then, to check if `lines[0]` is a prefix of `lines[1]`, if it is, complete it. Else find any prefix of `lines[0]` that is in the history (Including `“”`). If it is a common prefix of `lines[1]`, we can restore it and complete it. Pick the prefix that gives the best result.

**Any other line:** We can actually generalize the logic. When writing `lines[i]`, we can assume that each of the previous lines was written. This means that each of their prefixes was added to the undo history one way or another. (Even if we restored from the history to type a line, it still holds true that each of its prefixes exists in the undo history). Thus we can restore any prefix of any of the past lines with a cost equal to 2.

If `lines[i-1]` is a prefix of `lines[i]`, we can continue typing without restoring from the undo history. However, it is important to notice that this is not necessarily better than the alternative. There may be a string in the `UndoHistory` that would yield a better result than using `lines[i-1]`. Because this time many of the previous lines could be longer than `lines[i-1]`. An extreme example would be the case where `lines[i]` is already in the `UndoHistory`, it would take 3 button presses to type it.

**Solution:** The solution is to, in each step, simulate all of the alternatives. Make no assumptions and just simulate them all and always pick the alternative that yields the smallest cost. The alternatives could be to restore something from the undo history (containing all of the prefixes of all the past lines) and the other alternative is to, if possible, continue typing. For the first alternative, we can always restore the largest common prefix between the current line and a past line. Here’s an implementation:

```
int minPresses(vector <string> lines) {
  int cost = lines[0].size() + 1;
  for (int i = 1; i < lines.size(); i++) {
    int icost = 1000000000; //A large number

    for (int j = 0; j < i; j++) {
      int k = 0;
      while (k < lines[i].size() && k < lines[j].size()) {
        if (lines[i][k] == lines[j][k]) {
          k++;
        } else {
          break;
        }
      }
      // k is the length of the largest common prefix between lines[i] and
lines[j]
      int ijcost = 2 + (lines[i].size() - k) + 1;
      if (j == i - 1 && k == lines[j].size()) {
        // lines[j] is a prefix of lines[i], then there is a cost to
continue
        // typing lines[i] equal to (lines[i].size() - k) + 1, same as
subtracting
        // 2 from ijcost:
        ijcost -= 2;
      }
      icost = std::min < int > (icost, ijcost);

    }
    cost += icost;
  }
  return cost;
}
```

## Problem D: Tic-Tac-Toe

### Type: Simulation, DP

This problem could be solved by simulating the game. The board has 3x3 = 9 cells, and there could be two different states for a single board cell. So, the total states could be $2^{3x3} = 2^9 = 512$. This could be solved by just brute force simulating the game.

For each step of the game, we will try every step and only take the winning move for that step. By playing the game in such a way, we will be able to simulate the game perfectly for both players.

## Problem E: Reciting to the Caterpillar

### Type: Counting, DP

The problem involves counting rhyme schemes for a verse with $N$ lines. Use dynamic programming to precompute the number of rhyme schemes for each possible $N$ up to 500. Build a DP table to count valid rhyme schemes efficiently, handling the constraints by performing calculations modulo `10`$^9$`+9`

The recurrence relation is:

$dp[i][j] = (dp[i-1][j] \times j + dp[i-1][j-1]) \% MOD$

## Problem F: Lighting Strategy

### Type: Graph (SCC)

To approach this problem, we can utilize Depth-First Search (DFS) or Breadth-First Search (BFS) to explore the graph and identify strongly connected components (SCCs). The key insight is that if a component is not reachable from any other component, at least one light in that component must be turned on manually.

## Problem G: Zeroing Out Integers

### Type: Math, Observation

Assuming `a >= b`, both integers can be simultaneously become `0` only if

- `a <= 2b` [otherwise, `a` can never become `0`]

- `(a+b)` is a multiple of 3 [since we are deducting `3x` from `(a+b)` in each operation]

If both conditions hold, it is possible to reduce both $a$ and $b$ to zero. Otherwise, it is impossible.

## Problem H: Special Number

### Type: Ad Hoc, Implementation

This was another ad hoc problem and was expected to be solved by all teams. Just check if there are repeated 1s in the binary representation of an integer (you can convert a decimal number into a binary number by repeatedly dividing the decimal number by 2). If not, the number can be treated as a *special number*. Find k-th such *special* number, and print it.

## Problem I: Salary Distribution

### Type: Greedy, Binary Search

Perform a *binary search* on the possible median salaries. For each candidate median, determine whether it is possible to get a salary distribution. Be careful about the corner cases.

**Courtesy to:**

- Araf Al Jami, *Software Engineer, Microsoft*.
- Tahseen Rasheed Chowdhury, *Software Engineer, Agrihand AI*.

**Contest Coordinator:**

- Dr. Arif Ahmad, *Associate Professor & Head, Dept. of CSE, NEUB*.