Problem Set Analysis

Judging Panel

Name	Affiliation	Role
Prof. Dr. Zahidur Rahman	Jahangirnagar University	Chief Judge
Shahriar Manzoor	Southeast University	Judging Director, Problemsetter
Mohammad Ashraful Islam	Jahangirnagar University	Judging Coordinator, Problem Setter, Alternate Writer, Onsite judge
Md Mahbubul Hasan	Google	Reviewer, Problem Setter, Tester, Alternate Writer
Raihat Zaman Neloy	Google	Reviewer, Problem Setter, Tester, Alternate Writer
Nafis Sadique	Google	Reviewer, Problem Setter, Tester, Alternate Writer
Md. Imran Bin Azad	BRAC University	Alter Writer, Onsite judge, System Support
Md. Arifuzzaman Arif	Google	Problem Setter
Tanzir Pial	Stony Brook University	Problem Setter, Alternate Writer
Muhiminul Islam Osim	Kite Games Studio Ltd.	Onsite Judge, Problem Setter, Alternate Writer, Tester
Arghya Pal	Google	Problem setter, Tester, Alternate writer
Pritom Kundu	PriyoSys Ltd.	Reviewer
S. M. Shaheen Sha	Samsung Research	Problem setter, Alternate Writer, Tester, Onsite judge
Anik Sarker	Samsung Research	Problem Setter, Alternate Writer
Md Hasinur Rahman	Kite Games Studio Ltd.	Problem Setter, Alternate Writer
Abdullah Al Maruf	Affine DeFi	Tester, Alternate Writer
Mahdi Hasnat Siyam	Chaldal	Problem Setter, Tester, Alternate Writer
Chayan Kumar Ray	Google	Problem Setter, Tester

Problem A. A WeirdLand

Problem Setter: Tanzir Islam
Tester: Nafis Sadique, Arghya Pal
Category: Math, Probability

This can be thought of as an infinite sequence of coin tosses with a biased coin having probability = g for heads, and (1-g) for tails.

We can number the couples as $1, 2, 3, \ldots, \infty$. Then the first few child births (coin tosses) will belong to the first couple. The first couple stops having children after they have at least x boys and y girls. Then the 2nd couple starts having children and so on. By thinking of the couples having children sequentially instead of simultaneously, we convert the problem to a sequence of infinite coin tosses. For such an infinite sequence, it does not matter where we draw some boundaries where the i^{th} boundary will represent the i^{th} family. As a consequence, the values of x and y do not affect the final expected values at all[1].

There the ratio of expected number of heads vs expected number of tails count should be the ratio of their respective probabilities.

Formally, let x_n be the number of heads we get in n coin tosses. Since the $E(x_1) = g$, using linearity of expectation, $E(x_n) = n \times g$ Similarly let yn be the number of tails we get in n coin tosses.

$$E(y_n) = n \times (1 - g) \tag{1}$$

As n approaches infinity,

$$\frac{E(x_n)}{E(y_n)} = \frac{n \times g}{n \times (1 - g)} = \frac{g}{1 - g} \tag{2}$$

Problem B. Anagram Again

Problem Setter: Raihat Zaman Neloy
Tester: S.M Shaheen Sha, Anik Sarker
Category: Ad-hoc, Implementation

For each string in the dictionary, we can keep a counter of the characters, how many times they have occurred in that string. Now for each of the query, iterate over the dictionary words, and compare the counters, if all the characters are present in the query string more than the counter.

Problem C. Election

Problem Setter: Mohammad Ashraful Islam
Tester: Muhiminul Islam Osim, S.M Shaheen Sha
Category: Greedy

We have to split the votes into several group. Now think about the first group. It contains the indices $[1, i]$. Assume the optimal grouping of the indices $[i + 1, N]$ yields a margin of C ($X_{wins} - Y_{losses}$), then C must be the maximal margin. Now what about the margin with the group [1, i]? If we observe carefully we can see that:

- If $\sum_1^i vote_k > 0$ then this margin will increase by 1;
- Otherwise this margin will decrease by 1;

With this information we can now find the optimal margin of $[i, N]$ from $[j, N]$ where $i < j$. However, this would be too slow. We can also notice that, at any point of time we only need to know the largest margin possible and if $largest\ margin - 1$ is possible. If possible we also need to know the index which yields these margins. The reason for keeping the last two is that the optimal margin for [i, N] only moves by 1, so any margin more than 1 less of the largest available margin will never

[1] As a sanity test, solving the simplified problem for x = 1, y = 0 and a family stops having children as soon as they have 2 (or 3, or 4, or any integer) kids (regardless of their genders) can show that the values of x and y do not in fact matter.

give the best result. So all we have to do is to maintain these two margins. To break ties when multiple indices produce these margins we need to keep the ones that gives the largest $\sum_i^j vote_k$ for the margin.

To keep track of the sum of a range we can use the prefix sum technique ($prefix\_sum[1,i] = prefix\_sum[1,i-1] + vote_i$). With this approach we can get the optimal margin. This problem has a large input and tight time limit to avoid sub-optimal solutions.

Expected complexity: $O(N)$.

Problem D. Equal

Problem Setter: Shahriar Manzoor
Tester: Raihat Zaman Neloy, Nafis Sadique
Category: Math

Couple of ways to solve this problem.

- Use Big Integer Library in C++, Big Integer data type in Java or use Python.
- Use __int128 data type in C++.
- Check if $abs(\frac{a}{b} - \frac{c}{d}) < 10^{-20}$ using long double and $a * d = b * c$ using unsigned integer multiplication. Note that one way or the other won't work, but a combination of both does. Using double or float should fail as well.

Here is a python code that solves the problem.

```
import sys
input = sys.stdin.readline

n = int(input())

for i in range(n):
   a, b, c, d = map(int, input().split())

   if a * d == b * c:
      sys.stdout.write("Equal\n")
   else:
      sys.stdout.write("Not Equal\n")
```

Problem E. Function! Not Again!!!

Problem Setter: Raihat Zaman Neloy
Tester: Arghya Pal, Abdullah Al Maruf
Category: Data Structures, Tree

The first thing to figure out is - we can simplify the function as $\boldsymbol{F(x+y) = F(x) + F(y) - 9c}$ where c is the number of digits where carry over happens when we do $x + y$. Example: $x = 78$ and $y = 13$. Here, c will be 1 because for only one digit (least significant), we got a carry over. So, answer will be $F(78) + F(13) - 9 \cdot 1 = 10$ which is equal to $F(91)$.

Now from here, calculation is easy for the least significant digit. But for the other digits, we need to find if carry comes from the right side digits as well. To solve this, we will keep all the numbers using mod 10^i. i.e.: For calculating C for first digit, keep all numbers modulo 10, for calculating C for second digit, keep numbers modulo 100, for calculating C for third digit, keep numbers mod 1000, and so on. This way, if any carry comes from a less significant digit that will be kept in the consideration while calculation of C happens.

Now, when we are given a number for a query, for each digit of the number, we need to find how many digits will be there on the path U to V where carry will happen. Example, if one digit of the

query is 7, we will need to count how many digits there are who are greater than or equal to 3 for that position. This $\boldsymbol{counter} \times \mathbf{9}$ will need to be subtracted from the sum of F values.

To solve this we need some sort of data structure. Merge sort tree or persistent segment tree should work. But merge sort tree will require $O(10 \cdot N \cdot \lg^2(N))$ whereas persistent segment tree can bring this down to $O(10 \cdot N \cdot \lg(N))$. The time limit is strict, so merge sort tree should fail.

Expected complexity: $\boldsymbol{O(10 \cdot N \cdot \lg(N))}$.

Problem F. Good Path on a Tree

Problem Setter: Nafis Sadique
Tester: Raihat Zaman Neloy, Tanzir Islam
Category: Tree, Divide and Conquer

Lets run a depth-first search on the tree and calculate the depth of each vertex. Now, think about the good paths that is going through a particular vertex where that vertex is the lowest common ancestor (LCA) of those paths. How many situations do we have?

- The current vertex is the tallest sequence in the path, so we need to check which vertices under the subtree have strictly increasing paths until the current vertex. Then we count the pairs that are at the same depth. We need to make sure that the current vertex is the LCA of them though.
- The current vertex has a strictly increasing path to another vertex in the subtree. So, that sequence may continue to its parent vertex. In this case we need to propagate this sequence to the parent.
- The current vertex is in the decreasing part of the sequence. If we know the vertex or at least the depth of the vertex where the sequence started then we can find out how many numbers are left of the sequence. So, we have to pass it to the parent. We also need to count under the subtree how many increasing sequences are there which can be used to fill the remaining numbers.
  - The current vertex could be the last item of the decreasing part of the sequence. In that case we just count them.
  - Whether the current vertex is the first item in the decreasing part of the sequence can be found out by checking if the child is bigger than it and has an increaing sequence. In that case we need to convert it to a decreasing sequence.

We can use subtree-merging technique to solve these situations. Basically we solve the problem for each subtree from a vertex using the subtrees from its children. Then lets say from each vertex we return the two following information.

- Which subtree vertices are still on the increasing sequence path. We track the depth from which the subtree started. We can keep it in a map to count the number of occurences of depth.
- Which subtree vertices had a increasing sequence, but now is in a decreasing sequence path. For these sequences we store at which depth of the ancestor path this sequence will complete.

When we process a vertex, we cheack the subtree under each child. We first check if the child is smaller than the vertex. If that is true then all the increasing sequences can be carried upward to the parent of the current vertex. Otherwise the increasing sequences become new decreasing sequences. For example if the current vertex is at depth d_i and a vertex in one of the subtrees below which was increasing until the current vertex is at depth d_j then the path

If we know these information then we can merge them create a good sequence. For each vertex, we find the child with the largest subtree. This one becomes our base. The for every other child subtrees, we first query against the base and then we merge it with the base. This is a pretty well known concept. But if this feels new to you there are some other problems that may help.

- APIO'12 Dispatching. Check the discussion in codeforces.

- Xenia and Tree.
- This blog post in codeforces.

Overall expected complexity $O(n \cdot \lg^2(n))$.

Problem G. Hate Me if You Can

Problem Setter: Mohammad Ashraful Islam
Tester: S.M Shaheen Sha, Anik Sarker
Category: DP

The straightforward solution for this problem is DP. We will keep four states, x y z and which batsman is batting right now. Then in dp, for the captain try all possible scores while the other batsman will only play dot balls. Given the number of test cases is huge, we must pre-calculate the DP for all possible states.

Expected complexity: $O(100 \times 200 \times 200 \times 2)$

Problem H. Palindromic Subsequence

Problem Setter: Muhiminul Islam Osim
Tester: Abdullah Al Maruf, Mahdi Hasnat Siyam
Category: DP

Let's consider that there is one single query where $l = 1$ and $r = n$. So, we have to find the longest P in the entire S. We can solve this using dynamic programming. So, we introduce $\boldsymbol{dp(i, j, rem, mode)}$ which returns the length of the longest P from $\boldsymbol{S_i S_{i+1} \ldots S_{j-1} S_j}$ where **rem** is the number of characters remaining to be matched with S_i or S_j depending on the mode. If mode is 1, we try to match some characters of the remaining suffix of S_i of length rem with S_j. If mode is 2, we try to match some characters of the remaining prefix of S_j of length rem with S_i. If mode is 0, then we haven't any remaining characters to match; we have the entire S_i and S_j. In this way, we try to increase i and decrease j to match the mirror characters of our desired palindrome.

Now, as we use memoization, we can get the desired answer from the method for every query. For each query, we just need the value of $\boldsymbol{dp(l, r, 0, 0)}$.

Expected complexity: $O(n^2 \cdot |S_i|^2)$

Problem I. Picturesque Skyline II

Problem Setter: Nafis Sadique
Tester: Md. Mahbubul Hasan
Category: Greedy, DP

First lets think about how to fix one segment of size $2 \cdot k + 1$ without thinking about creating any gaps. We just need to know what is the minimum number of swaps required to turn that segment into a pyramid-like pattern group. Here is one approach to calculate the minimum number of swaps needed to make a segment [l, r] a pyramid-like. Lets find the smallest number in the segment. We must either push it to the beginning or the end of the segment, so how many move required to move this number? If we try to move it to the front then we would need to swap with all the bigger numbers before it in the segment. Similarly if we try to move it to the end then we would have to swap with all the bigger numbers after it in the segment. Now, lets think about the second smallest number, similarly we either move it to the front or move it to the end, but we can't move it past wherever we placed the smallest number. So, we continue placing all the numbers in sorted order either to the front or the end. But we can only place k numbers in the front and k numbers in the end and we don't do anything with the largest number as it will automatically be fixed in the middle.

Lets calculate for each building, the number of taller buildings that appears before the building in the segment and how many taller buildings appear after the building in the segment. Lets call

these values a_i and b_i. To make a pyramid like shape we need to have k sorted numbers at the beginning of the segment and k reverse-sorted numbers at the end with the largest number in the middle. Which means we need to move some of the buildings to the front and some of them to the back. The buildings that needs to be moved to the front incures a_i cost while the others incure b_i cost.

So basically, we assign each buildings to either type **a** or type **b** and make sure they both have the same size. We are not counting the tallest building in anything since it doesn't contribute anything to the minimum cost. To do the assignment, we can take $\sum b_i$ and convert a_i to $a_i - b_i$. The smallest k values of $a_i - b_i$ will be added to $\sum b_i$ and that would be the minimum number of moves requires to make the segment pyramid-like. We have to do it for all odd length segments and for each segment we can easily do it in $O(n \cdot \lg(n))$ complexity. But with the total number of segments this can grow very large.

We can improve it even more. For a segment of length 3, we never need more than a single swap. Which means, the upper bound of the result is $\frac{n}{3} + \epsilon$. Here $\epsilon = 6$ would work. We can get substantial speedup if we stop processing the segments that we know will need more than $\frac{n}{3} + \epsilon$ swaps. One way to maintain that is to build [l,r] from [l+1, r-1]. If we know the number of swaps needed for [l+1, r-1] is more than $\frac{n}{3} + \epsilon$ then we simply ignore these segments. However we are not sure how much speedup does this actually provide, what we only know is that it is very good.

After that we can just do a dynamic-programming approach to split the groups. That part can be done in $O(n^2)$. This performs somwhere close to $O(n^2 \cdot \sqrt{n})$, but we don't have an exact proof.

Problem J. Rik and Vik

Problem Setter: Md. Arifuzzaman Arif
Tester: Raihat Zaman Neloy, Nafis Sadique
Category: Math, Combinatorics

Assume $K = \frac{S}{N}$. Now, lets fix the first K columns with S distinct characters. We can do that in S! ways. Then we fill out the remaining $M - K$ columns one by one. We can notice that the i^{th} column must have the same set of characters as the $(i-k)^{\text{th}}$ column. We can arrange the N characters in the i^{th} column in N! ways. So, we simply take their products. As a result we have the following formula.

$$S! \cdot N!^{M-K} \tag{3}$$

Problem K. Sort and &

Problem Setter: Arghya Pal
Tester: Nafis Sadique, Raihat Zaman Neloy
Category: Observation

If we look closely, the minimum cost of sorting the array will be either 0 or 1. If the array size is $2^k - 1$, then the answer will be 1, otherwise 0. Because for all other indices i we can find at least one index 2^j such that $(i \wedge 2^j) = 0$. We can use those indies to swap the numbers to their correct position. So, the optimal moves should be like following:

- If the array size is $2^k - 1$, first take $2^k - 1$ to its correct position.
- Move all the indices to their position other than 2^i indices.
- Now move the 2^i indices to their correct position.

Expected complexity: $O(n)$.