Problem A. (A + B) mod P

Please read a paper "Do Machine Learning Models Memorize or Generalize?" with very nice visualizations: `https://pair.withgoogle.com/explorables/grokking/`.

It doesn't contain the solution immediately, but it shouldn't be very hard to construct a solution for this problem when you understand the ideas in the paper.

Problem B. The Best Wife

If there are two segments, where one segment lies entirely inside another, we can discard the bigger one because if a solution exists with a bigger segment, we can replace it with a smaller one and get the same result. We can understand when we must discard some segment using a segment tree in $O(n \log n)$ time.

After discarding such segments, we can use greedy to construct the answer. Let's sort all segments in the increasing order of the left end and try to add them in this order. We only add a segment if it does not intersect with already used segments.

We can reformulate the greedy solution as dynamic programming:

$$d(s) = \begin{cases} 1 + d(t+1), & \text{if segment}[s..t] \text{ exists} \\ d(s+1), & \text{otherwise} \end{cases}$$

In this case, $d(0)$ is the answer.

We need to recalculate d_i after each segment addition or deletion. To do so, we can represent each element d_s as a node of a forest, which has an outgoing edge to either d_{s+1} or d_{t+1}. This forest could be stored in a data structure like a Link-cut tree. The answer is a sum on the path from d_0 to the root. The overall complexity is $O(n \log n)$.

It is also possible to solve this problem with a persistent segment tree if you don't like a Link-cut tree.

Problem C. Cardinality

A lot of different solutions are possible, we will describe one of them.

Let's split all the sets into small and big based on the size.

If the set is small, let's store all the elements directly.

Let's also pick 4000 random indexes in the range 1 to n, and for each big set, store a bitset, which shows which elements out of 4000 are present. We can estimate the full size of the set based on it.

Another possible option is to use `https://en.wikipedia.org/wiki/MinHash`.

Problem D. 3D

We can use the hill climbing algorithm to solve the problem. Let's put points in random locations and calculate the score function by definition (the biggest difference from $d_{i,j}$ to real $dist_{i,j}$).

Let's try to move a random point in a random direction. If the score improves, we apply the change. Otherwise, do nothing. We can iterate this process until it finds the solution.

We can also try to move points by a larger distance at the beginning and decrease the changes closer to the end of the process.

We are guaranteed that tests are random, so we can choose parameters for the algorithm locally before submitting.

Problem E. Equal Strings

Let's start by calculating the distance from the first string to all other strings. Then, we can split all strings into groups by that distance. If two strings are equal, they will be in the same group. Then, we can recursively run the same algorithm for each group.

The inputs are guaranteed to be random, so we can test locally that the total number of queries is not too big. In fact, it is much less than 25 000.

Problem F. Fast Tree Queries

Computers are fast nowadays, so we can solve this problem in $O(nq)$.

Let's first discuss how to solve this problem with the array instead of the tree.

It is important to make sure your program uses SIMD instructions. Usually, the compiler could vectorize loops, so you don't need to do anything special. Sometimes, it helps to extract the hot path into a separate function like:

```
void add(int *a, int len, int x) {
  for (int i = 0; i < len; i++) {
    a[i] += x;
  }
}
```

You can use websites like Compiler Explorer to understand if SIMD instructions are used: `https://godbolt.org/z/vfEr8a17G`. You can see the main loop uses "ymm" registers (this is a good sign!):

```
.L4:
        vpaddd  ymm0, ymm1, YMMWORD PTR [rax]
        add     rax, 32
        vmovdqu YMMWORD PTR [rax-32], ymm0
        cmp     rdx, rax
        jne     .L4
```

It is important to use correct pragmas to optimize your code. In this case, it is important to enable avx2 instructions and use O3:

```
#pragma GCC target("avx2")
#pragma GCC optimize("O3")
```

If we have a fast enough solution for the array, we can solve the problem on the tree using heavy-light decomposition, which stores arrays for each continuous path. We need to run $O(\log n)$ times some SIMD-optimized function to answer each query. The total length of arrays for each query doesn't exceed $O(n)$, so it should be as fast as on the array.

Problem G. Geo Sharding

There are several possible constructions.

For example, we can build a hexagonal grid.

Another way is to use rectangles of size 10×15 with some shifts on each layer.

Problem H. Have You Seen This Subarray?

Let's try to answer one query. For each pair of neighboring integers in the query, let's determine intervals of time when they were neighbors in the array. We should pick the smallest time which works for all pairs. Because swaps are guaranteed to be random, if $n = 10^5$, there are $O(1)$ time intervals for each pair. Which means we can intersect intervals naively.

If n is very small (e.g., smaller than 10), we can't use the same algorithm because the number of intervals could be big for each pair of integers. But if n is small, there are not too many different permutations, so for each permutation, we can remember the first time the array is equal to it.

Potentially it is possible to squeeze a solution that combines those two ideas into a Time Limit. But if we don't want to fight with a Time Limit, we can create a separate solution for n between 10 and 100. Additionally to storing intervals for all neighboring pairs, we can store intervals for all triples of integers.

Problem I. Interactive Casino

The expected value of each round is positive, so we should generally play in rounds. But we don't need to optimize the expected value; we should instead maximize the probability of getting 10 000$ after all rounds.

The idea is not to participate in risky rounds, which could significantly decrease our balance. More formally, let's pick some constant r and only accept rounds for which $b \leq r \cdot m$, where m is our current balance and b is the suggested bet.

How to choose good r? Let's try different ones and locally check if we win with a good probability. Empirically, r around 0.3 works well.

Problem J. Jigsaw Puzzle

Because splits are guaranteed to be random, we can assume that all lengths of all edges of polygons are different, except for cases when polygons are touching via this edge.

We can start from any polygon and iteratively find another polygon with an edge of the same length and uniquely determine the relative position of that polygon. After all polygons are added, we need to rotate them so that the square is parallel to coordinate axes.

Problem K. Knapsack

We can use the Karmarkar–Karp algorithm to solve this problem. Let's convert each integer a_i into triple $(0, 0, a_i)$. We can join two triples (x_1, y_1, z_1) and (x_2, y_2, z_2) into a new triple $(x_1 + x_2, y_1 + y_2, z_1 + z_2)$. We can also shuffle integers inside the triple. For example, we can transform triple (x, y, z) into (y, z, x). We aim to get a triple of equal integers (x, x, x).

For simplicity, we can always assume $x \leq y \leq z$ in the triple (x, y, z).

Karmarkar–Karp algorithm takes two triples (x_1, y_1, z_1) and (x_2, y_2, z_2) such that $z_i - x_i$ is the biggest. Later, it combines them into $(x_1 + z_2, y_1 + y_2, z_1 + x_2)$ and adds it to the set. The algorithm continues until only one triple is left in the set or some triple (x, x, x) is generated.

With high probability, this algorithm finds a solution for given constraints. But if it doesn't succeed, we can add some randomness to the process and start it again. For example, we can randomly discard some integers at the beginning.

Problem L. London Underground

Let's analyze the following algorithm. We store the list of possible solutions. It initially contains one empty set. Then, we iterate over all stations in some order. For each new station, we try to add it to each existing solution, and if successful, add a new solution to the list. This solution works in a time proportional to the number of solutions, which is too big.

We can do the following optimization. After analyzing a prefix of stations, we don't need to store full solutions. If, for a station, we already analyzed all other stations connected to it, we don't need to remember the state of that station. So, if two solutions exist that only differ in this station, we can merge them and save the number of such solutions.

How fast is the new solution? If for a prefix of stations, there are k stations, which are directly connected to some not analyzed station, we need to store 2^k integers. Unfortunately, k could be pretty large for some prefixes, so this solution is still too slow.

But it is possible to find such permutation of stations that the sum of 2^k for all prefixes is not too big. To find this permutation, we can use the fact that the graph used in the problem is always the same. We can run simulated annealing locally to find a good permutation in a few minutes and hardcode it to the final solution.

Problem M. Meta

The problem is pretty easy — we just need to calculate the minimum time for each problem, sort problems by that time, and solve problems until the total time doesn't exceed 300.

The more interesting fact is that a sample of this problem could give you some ideas about how hard are other problems in this contest.

For example, you can compare the smallest time for problems and understand which of the problems are easier and which are harder.

You can also notice that for some of the problems, the difference between the biggest time and smallest time is quite big, which means this problem is quite tricky, and it is possible to come up with a wrong solution and spend a lot of time implementing it. So it is better to discuss solutions to such problems with your teammates.