Problem Tutorial: "Anthill/Honeypot Simulator"

Idea: Nikolai Durov
Development: Nikolai Durov

The central limit theorem implies that the coordinates x and y of the ant after t units of time are independent normal random variables with zero mean and variance $t/2$. By scaling, we see that $f(t)$, the probability of the ant being in the honeypot at time t, is the Gaussian area of the disk with center at a distance $d' = d\sqrt{2/t}$ from the origin and of radius $R' = R\sqrt{2/t}$. It can be found in several different ways by numerical integration, for instance, by integrating the differential form $\omega = \frac{1}{\sqrt{2\pi}}\Phi(y)e^{-x^2/2}dx$ along the circle, where $\Phi(y)$ is any antiderivative of $\frac{1}{\sqrt{2\pi}}e^{-y^2/2}dy$, such as $\operatorname{erf}(y/\sqrt{2})$. Another option is to integrate $\omega' = \frac{1-e^{-(x^2+y^2)/2}}{2(x^2+y^2)}(x\,dy - y\,dx)$, which can be constructed by looking at the Gaussian density in polar coordinates. It is especially pleasant to integrate an analytic function along a circle: one simply has to compute the average of the values of the function on 2^k uniformly distributed points on the circle, and let k grow until two consecutive approximations are close enough. Once we know how to compute $f(t)$, the answer to the problem is $\left(f(T), \int_0^T f(t)\,dt\right)$.

Problem Tutorial: "Missing Number"

Idea: Vladislav Makarov
Development: Vladislav Makarov

On the first glance, this problem may seem impossible: having 4000 states can be seen as having less than 12 bits (because $2^{12} = 4096 > 4000$) and, intuitively, 3 of those 12 bits are needed to somehow store the index of the current pass over the array (because $8 = 2^3$), leaving barely any memory for us to work with. However, it turns out that 4000 states is more than enough.

The intended solution is based on the binary search. Before the first pass, we know that there is at least one integer from the range $[1, n+2)$ that is missing in the array. Partition $[1, n+1]$ into two ranges $[1, m)$ and $[m, n+2)$ of equal or almost equal length. On the first pass, compute the count of the elements of the array that fall in the former range. If there are less than $m-1$ such elements, then some integer from $[1, m)$ is missing from the array. On the other hand, if there are at least $m-1$ occurrences (because of repeats, the number of occurrences can be more than $m-1$) of integers from $[1, m)$, then there are at most $n - (m-1) = n - m + 1 < (n+2) - m$ occurrences of integers from $[m, n+2)$.

Hence, after the first run, we know a range of length approximately $n/2$ that contains at least one of the missing numbers. Similarly, if we currently know that an integer from the range $[\ell, r)$ is missing, we may choose $m = \lfloor \frac{l+r}{2} \rfloor$, count the integers from the range $[\ell, m)$ and, depending on the result, conclude that an integer from either $[\ell, m)$ or $[m, r)$ is missing. If $r - \ell = 2$, then both these ranges contain a single integer, meaning that we know the answer. If $r - \ell \geqslant 3$, we may need to make more passes over the array. It is clear to see that exactly 8 passes are needed with such a strategy.

Indeed, before the first pass we know a range of length less than 2^8 that contains a missing integer. After the first pass, we know a range of length at most 2^7. After the second pass, we know a range of length at most 2^6. And so on, meaning that after the eighth pass, we know a range of length at most 1, so no more passes are needed.

Why does this approach fit the requirement on the number of states? Well, a completely naive implementation does not, but it is easy enough to fix. Suppose that our current range is $[\ell, r)$ that splits into ranges $[\ell, m)$ and $[m, r)$ of almost equal length. Then, while counting elements of $[\ell, m)$ that appear in the array, it does not make sense to go above $m - \ell$ (if the count is at least $m - \ell$, we do not care about its exact value, because the count being so high ensures that at most $r - m - 1$ elements of $[m, r)$ appear in the array). So, a state of our automaton is a segment $[\ell, r)$ that may appear during the binary search process and an integer from 0 to $r - \ell$ inclusive. And, of course, we do not need to create any states when $r - \ell = 1$: whenever we would like to create a transition that leads to such a state, we could as well immediately halt and say that ℓ is a correct answer to the problem.

It turns out that the above two optimizations (do not count too high and do not create any states for ranges that contain exactly one element) make fitting into the limit on the number of states very easy: there is a single range that asks us to count from 0 to approximately $n/2$ ($n/2+1$ states for the original range), two ranges that ask us to count to approximately $n/4$, four ranges that ask us to count to approximately $n/8$, and so on. Hence, we need approximately $\sum_{k=0}^{7} 2^k \cdot (n/2^{k+1}+1) = \sum_{k=0}^{7} 2^k \cdot (n/2^{k+1}) + \sum_{k=0}^{7} 2^k = 4n + 2^8 - 1 = 1255$ states, which is much less than 4000 (depending on how the words "approximately" are handled, there may be a bit more or a bit less than 1255 states in the resulting automaton, but it is still clear that such an approach comfortably fits into the requirements).

Problem Tutorial: "String Workshop"

Idea: Mikhail Ivanov
Development: Mikhail Ivanov

We explain the intended C++ solution for "`String Workshop`" (the code shown in the statement). The whole approach relies on two observations: (1) an optimal sorting sequence can be chosen stable (swapping equal letters is useless and only increases the fee), hence every out-of-order pair must "cross" exactly once; (2) the swap fee depends on the current index k of the swapped adjacent positions, so we must compute a weighted sum over all mandatory crossings, not just the inversion count.

We sort substrings using local indices $1..m$. The stable sorted order is uniquely determined: all "`A`"s, then all "`B`"s, ..., then all "`Z`"s, preserving the original order inside equal letters. Every inversion (a pair $p < q$ with $s[p] > s[q]$) forces exactly one adjacent swap between those two concrete occurrences at the moment they become adjacent. However, the paid value is the index of that adjacent swap at that moment, which depends on how elements shift during the process. Therefore we need a data structure that can aggregate enough information to recover the minimal total fee.

The solution builds a segment tree over the whole string $s[1..n]$. Each node stores a monoid-like summary of its segment: summaries of left and right children can be merged to obtain the summary of the concatenation. This yields substring queries in $\mathcal{O}(\log n)$, and substring "mutation sorting" (type 3) via range assignments with lazy propagation.

For a segment of length ℓ, for each letter $c \in \{0, \ldots, 25\}$ it stores

$$cnt[c] = \#\{p : s[p] = c\}, \qquad sumPos[c] = \sum_{p:\ s[p]=c} p,$$

where positions p are local to the segment (from 1 to ℓ). These arrays let us shift all positions in the right child by the left length and quickly compute sums of positions over letter ranges.

It also stores the inversion count

$$inv = \#\{(p, q) : 1 \le p < q \le \ell,\ s[p] > s[q]\},$$

and the key quantity

$$cost = \text{the minimum possible sum of swap indices needed to stably sort this segment.}$$

To compute the cross contribution between two halves in $\mathcal{O}(26)$, the node additionally stores, for every letter threshold b,

$$pairCnt[b] = \#\{(p, q) : p < q,\ s[p] > b,\ s[q] \le b\},$$

and

$$preCnt[b] = \sum_{q:\ s[q]=b} \#\{p < q :\ s[p] \le b\}.$$

Intuitively, during the stable merge step the formula for the weighted fee involves terms like "how many elements $\le b$ are to the left" and "how many pairs of the form $(> b, \le b)$ exist", and these two arrays encode exactly those counts.

Now consider merging a left segment L of length a and a right segment R of length b.

1) Counts and position sums:

$$cnt = cnt_L + cnt_R, \qquad sumPos = sumPos_L + (sumPos_R + a \cdot cnt_R),$$

since every position in the right half is shifted by a.

2) Inversions:

$$inv = inv_L + inv_R + crossInv, \qquad crossInv = \sum_{x \in R} \#\{y \in L : \ y > x\},$$

computed as $\sum_{b=0}^{25} cnt_R[b] \cdot \#\{y \in L : \ y > b\}$.

3) The auxiliary arrays match their definitions:

$$pairCnt[b] = pairCnt_L[b] + pairCnt_R[b] + (\#\{y \in L : \ y > b\}) \cdot (\#\{x \in R : \ x \le b\}),$$

$$preCnt[b] = preCnt_L[b] + preCnt_R[b] + cnt_R[b] \cdot (\#\{p \in L : \ s[p] \le b\}).$$

4) The fee. One can think of an optimal stable process as: stably sort L, stably sort R, then stably merge the two sequences using adjacent swaps. In that merge, each element from R with value b must move left across all elements in L with values $> b$; each such move is one adjacent swap with a fee equal to the current swap index. Summing these mandatory fees produces the cross part of *cost*.

The code expresses the merged fee as

$$cost = cost_L + cost_R + a \cdot inv_R + cross1 + cross2.$$

Here $a \cdot inv_R$ accounts for the fact that the internal swaps of the right half happen at indices shifted by a in the full segment. The term

$$cross1 = \sum_{b=0}^{25} cnt_R[b] \cdot \Big(\sum_{y \in L: \ y > b} pos(y) \Big)$$

uses *sumPos* to include the position-dependent part contributed by left elements when right elements pass them. Finally,

$$cross2 = \sum_{b=0}^{25} \Big(cnt_R[b] \cdot pairCnt_L[b] + (\#\{y \in L : \ y > b\}) \cdot preCnt_R[b] \Big)$$

captures the remaining interaction terms caused by the fact that right elements move and affect each other's current indices; this is exactly why the node maintains *pairCnt* and *preCnt*.

Thus a substring query "`2 i j`" is answered by extracting the aggregate for $[i, j]$ in $\mathcal{O}(\log n)$ and outputting its *cost*; the collecting order is left-to-right so that the local indices inside the aggregate are $1..(j - i + 1)$ as required.

Updates and mutations.

Type 1 is a point assignment. The segment tree supports a lazy tag meaning "this whole segment is one letter". Applying such a tag sets $inv = 0$ and $cost = 0$, sets exactly one $cnt[c] = \ell$, and sets $sumPos[c] = 1 + 2 + \cdots + \ell$. Therefore a point change costs $\mathcal{O}(26 \log n)$.

Type 3 first queries $[i, j]$ and outputs *cost*, then replaces the substring by its sorted version. Since the aggregate already contains the counts $cnt[0..25]$, the sorted substring is just 26 consecutive uniform blocks. The code assigns

$$[i, \ i + cnt[0] - 1] \leftarrow \texttt{A}, \quad [i + cnt[0], \ i + cnt[0] + cnt[1] - 1] \leftarrow \texttt{B}, \ \ldots$$

using lazy range assignments. This is 26 assignments, each $\mathcal{O}(26 \log n)$, so type 3 is $\mathcal{O}(26^2 \log n)$ with a fixed constant.

Overall per test: build in $\mathcal{O}(26n)$, type 2 in $\mathcal{O}(26 \log n)$, type 1 in $\mathcal{O}(26 \log n)$, type 3 in $\mathcal{O}(26^2 \log n)$. The crucial point is that the answer for a substring is stored directly in the field *cost* of the aggregated monoid.

Problem Tutorial: "Distinguishable Distributions"

Idea: Mikhail Ivanov
Development: Mikhail Ivanov

One way to solve this problem is as follows: take $n = (t-1) + 2^{t-1}$. We will choose the distribution as follows: we will split each string s into two strings uv, where $|u| = t-1$, $|v| = 2^{t-1}$, and we will call s *allowed* if $v_u = \texttt{T}$, where u in the subscript is understood as a binary number. Our distribution will be uniform over the allowed strings (and with probability 0 it allows disallowed strings). With oracle access of length t, it is easy to check any string for allowance: the first $t-1$ queries will determine the string u, and at the end, we will find out the symbol v_u. The fact that this distribution is $(2^{-t}t, t)$-independent can be easily established by pairing all strings that differ only in the symbol v_u, considering any set of indices of size t and a subset T of possible binary strings at those indices, and examining how it can happen that one of the strings in the pair corresponds to T, while the other does not. The complete proof is left as an easy exercise for the reader in probability theory.

Problem Tutorial: "Four Sages Around an Oak Tree"

Idea: Mikhail Ivanov
Development: Mikhail Ivanov

There are a total of $3^4 = 81$ ways to color the hats of the four sages in three colors, so there are only 81 possible inputs in the first run. Based on the input of the first run, one can construct a *first run code* — an integer from 0 to 80 that uniquely identifies which colors were provided as input. The code can be constructed, for example, by converting the colors of the four hats into numbers from the set $\{0, 1, 2\}$, considering them as ternary digits of a number (for instance, the i-th digit from the right corresponds to the color of the hat of the i-th sage), and then calculating the resulting ternary number (possibly with leading zeros). For example, if we consider `red` $= 0$, `green` $= 1$, `blue` $= 2$, then the input example "`green red blue red`" corresponds to the digit sequence 1, 0, 2, 0, and the ternary number 0201_3 equals 13.

Let us call the *first run strategy* an array of 81 numbers from the set $\{1, 2, 3, 4\}$; then we can, having the first run strategy s_1, act as follows: calculate the code c_1 from the input data and respond with the number $s_1[c_1]$. Thus, the first run strategies correspond exactly to all 4^{81} deterministic ways to act during the first run.

Similarly, there are $4 \cdot 3 \cdot 3 = 36$ possible inputs in the second run, which can be encoded as numbers from 0 to 35, so the *second run strategy* can simply be considered as an array of 36 ternary digits.

Note that if we can somehow obtain the second run strategy of some correct solution, then constructing the first run strategy from it will not be difficult. Indeed, in the first run, we have four ways to respond: $1, 2, 3, 4$. We can try all of them, see what input we get in the second run, and what answer we would give according to our second run strategy. If, say, for number 3 we give the answer in the second run that matches the color of the hat of the third sage from the first run, then we can answer 3 in the first run. It may well be that there are several correct answers in the first run — we can choose any of them. The problem will only arise if none of the four answers fit. Thus, we will call a second run strategy *correct* if it can be complemented with a first run strategy to form a complete solution.

Thus, we have reduced the problem to finding a correct second run strategy. This is a string of 36 ternary digits (which can be encoded, for example, in nine bytes), so we can first find it locally (without sending it to the testing system, but checking against all 81 inputs of the first run on a working computer), and then use the resulting array of numbers in the actual submission, which will no longer rely on prolonged enumeration. The first thing that comes to mind is to randomly enumerate these 36 ternary digits, but

such enumeration may take hours. Let's act smarter. We will start with a string s_2, consisting of 36 threes. A three — is a special digit meaning "we will not respond during the second run for this input". Such a strategy is definitely incorrect, so we will start correcting it gradually. Let's imagine for a moment that not all inputs of the first run are equally important: there are inputs that we prioritize and on which we definitely want to succeed, while on other, less important inputs, our solution may not succeed as long as it does not fail on all more prioritized inputs. Formally, we express this relationship between inputs as a permutation of numbers [0..80], which we denote as $p \in \mathbb{S}_{[0..80]}$: the earlier the code of the first run appears in the permutation p, the more prioritized it is for our solution to succeed on this code.

Fixing p, let's iterate through the codes from it from left to right and act greedily. Suppose we have taken the next code c_1; let's see what happens if we respond to it with 1. This will yield some input for the second run $c_2^{(1)}$. If on $c_2^{(1)}$ we already output the color of the hat of the first sage from c_1 (in other words, $s_2\left[c_2^{(1)}\right] = c_1[1]$), then we have already passed the test c_1, and we can move on in p. If $s_2\left[c_2^{(1)}\right] = 3$, then we have not yet decided what to do during the second run for such an input. Let's then assign $s_2\left[c_2^{(1)}\right] := c_1[1]$ (now the solution passes the test c_1, we have just guaranteed that!) and also move on along p. If $s_2\left[c_2^{(1)}\right]$ has already been assigned and contains something other than $c_1[1]$, then we will try to respond to c_1 with the number 2, construct the input for the second run $c_2^{(2)}$ — and consider the same three cases: $s_2\left[c_2^{(2)}\right] = 3$, $s_2\left[c_2^{(2)}\right] = c_1[2]$, $s_2\left[c_2^{(3)}\right] = c_1[3]$; in the last of these cases, we will also have to try responding with the number 3 and finally with the number 4. If none of the four options work, then the strategy s_2 has already reached a state where it cannot be complemented to be correct, so let's reject the permutation p as bad.

Now we can organize the enumeration as follows: we will randomly choose $p \in \mathbb{S}_{[0..80]}$ in a loop and see if a correct strategy can be obtained by acting as described in the previous paragraph. This method already works relatively quickly, and it is possible to find a working strategy locally in just five minutes.

Problem Tutorial: "Pluses and Minuses"

Idea: Ivan Bochkov
Development: Ivan Bochkov

We are asked to calculate the sum $\sum_a d_a \binom{2n-1-a}{n-1}$, where $d = 0, 1$ depending on whether it is written in Misha's notebook.

Let us choose such coefficients e_a that $\sum_a d_a \binom{2n-1-a}{n-1} = \sum_a e_a \binom{2n}{n+a}$.

Note that $\binom{2n}{n+a} = \sum_k \binom{2n-k-1}{n-1}\binom{k}{a}$. From here

$$\sum_a e_a \binom{2n}{n+a} = \sum_a e_a \sum_k \binom{2n-k-1}{n-1}\binom{k}{a} = \sum_k \binom{2n-k-1}{n-1} \sum_a \binom{k}{a} e_a$$

.

That is, if $d_k = \sum_{a=0}^{k} \binom{k}{a} e_a$ for any k, then this set of e's is suitable.

But in this case, $e_k = \sum_{a=0}^{k} \binom{k}{a} (-1)^{a-k} d_a$ are suitable.

Now we need to calculate the sum $\sum_a e_a \binom{2n}{n+a}$ for any n. Which is the same as calculating $[x^0](x+1/x)^{2n} Q(x)$ for $Q(x) = \sum e_a x^{-a}$ and all n. We do this using a divide-and-conquer algorithm. Let's solve this problem on a segment of length L. Then, in Q, we only care about the powers in the segment $[-L, L]$. Divide the segment into 2 halves and solve the problem for $Q(x)$ and $Q(x)(x+1/x)^{2(L/2)}$ in the halves, respectively. The bottleneck is at multiplying polynomials on a segment of length L in $O(L \log L)$, which results in a total running time of $O(n \log^2 n)$.

Problem Tutorial: "Traffic Lights"

Idea: Mikhail Ivanov
Development: Mikhail Ivanov

Problem (algorithm view). We have a tree with vertex colors in $\{R, G, Y\}$ and two query types:

- Type 1: recolor a vertex v.
- Type 2: for a given vertex v, find the minimum length of a path between some red r and green g *such that* v *lies on the path* $r \leftrightarrow g$. If impossible, output -1.

Key reformulation. For fixed v we need

$$\min_{\substack{r:\, col(r)=R \\ g:\, col(g)=G \\ v \in path(r,g)}} d(r, g).$$

If $v \in path(r, g)$ then $d(r, g) = d(r, v) + d(v, g)$, so we want two endpoints "meeting" at v.

LCA + Euler tour. Root the tree at 1. Compute $\text{tin}[u], \text{tout}[u]$ by an iterative DFS. Then subtree(u) is exactly the Euler interval $[\text{tin}[u], \text{tout}[u]]$. Binary lifting gives LCA, distances, and `kth_ancestor`. The helper `next_neighbor(v,x)` returns the first step from v toward x: child on the path if v is ancestor of x, otherwise $parent[v]$ (and 0 if $x = v$). This lets us describe "the branch of v containing x" via a single forbidden subtree interval.

Centroid decomposition (CD). Build centroid decomposition. For every vertex u store its centroid-ancestor chain $cdAnc[u] = [c_0, \ldots, c_t]$ (iterated from closest to farthest in the code). For each centroid c consider all vertices whose centroid chain contains c (the centroid component of c). Store their Euler entry times in a sorted array $cenTin[c]$.

Why store $cenTin[c]$? Any rooted subtree constraint is an Euler interval $[L, R]$. Intersecting that subtree with the centroid component becomes a contiguous index range in $cenTin[c]$ (found by `lower_bound`/`upper_bound`).

Per-centroid data structures (for each color $X \in \{R, G\}$).

- Segment tree $segX[c]$ over indices of $cenTin[c]$: leaf for vertex x stores $d(x, c)$ if $col(x) = X$, else ∞. So range minimum gives $\min d(x, c)$ among allowed vertices.
- A min-heap $heapX[c]$ storing pairs $(d(x, c), x)$ for fast "unrestricted nearest". It is lazy: on query we pop while the stored vertex is no longer color X.

Unrestricted nearest to v of color X. For each centroid $c \in cdAnc[v]$:

$$cand = d(v, c) + \min_{x:\, col(x)=X} d(x, c),$$

where the minimum is read from the cleaned heap top. Take the best over all c, also returning the node id (`nearest_any_with_node`).

Nearest to v of color X with subtree constraints. Given an Euler interval $[L, R]$: for each $c \in cdAnc[v]$ compute the corresponding index range $[l_c, r_c]$ in $cenTin[c]$. If we want "inside" $[L, R]$, query $segX[c]$ on $[l_c, r_c]$. If we want "outside", query the complement: $[0, l_c - 1]$ and $[r_c + 1, m - 1]$ and take min. Add $d(v, c)$ and take global min. This is `nearest_interval_dist`.

Updates (Type 1). When recoloring x: for every $c \in cdAnc[x]$ find the position of $\text{tin}[x]$ in $cenTin[c]$, set old-color leaf to ∞, new-color leaf to $d(x, c)$ in segment trees, and push $(d(x, c), x)$ to the new-color heap (if R or G).

Type 2 query logic (function `min_traffic_light`). Let (u, d_R) be the unrestricted nearest red to v, and (w, d_G) nearest green. Also handle the trivial endpoint case: if v is itself R (or G), we can set one endpoint to v.

If v lies on the path $u \leftrightarrow w$, then (u, w) is feasible and gives candidate $d_R + d_G$.

Otherwise, u and w are in the same branch from v. Let $compR = \texttt{next\_neighbor}(v, u)$ and similarly $compG$. Any feasible pair must have at least one endpoint *outside* that branch, otherwise the path never reaches v. So we try two options:

$$d_R + \min_{g \notin branch(compR)} d(v, g), \qquad d_G + \min_{r \notin branch(compG)} d(v, r),$$

where "outside the branch" is implemented as either "inside subtree(v)" (if $comp = parent[v]$) or "not in subtree($comp$)" otherwise (using Euler intervals + segment tree filtering). Take the best among all candidates.

Complexity. CD depth is $O(\log n)$. For each centroid we do $O(\log n)$ binary searches on $cenTin[c]$ and $O(\log n)$ segment tree operations:

$$\text{Type 2: } O(\log^2 n), \quad \text{Type 1: } O(\log^2 n), \quad \text{memory: } O(n \log n).$$

Problem Tutorial: "Sweet Remainders!"

Idea: Mikhail Ivanov
Development: Mikhail Ivanov

We are given multiple test cases. Each test contains n, m, and an array $b_1, \ldots, b_n$. If $b_i = -1$, vertex i is free. Otherwise, we must have $\deg(i) \bmod m = b_i$. We need to output either "`NO`" or "`YES`" and any tree that satisfies the constraints.

The solution splits into two independent parts. First, we decide whether a valid degree sequence exists, and construct one if it does. Second, we build a tree that realizes the chosen degrees using a Prüfer-code construction.

A tree on $n > 1$ vertices is completely characterized by a degree sequence $d_1, \ldots, d_n$ with $d_i \geq 1$ and $\sum_i d_i = 2(n-1)$. Therefore, our goal is to pick integers d_i that satisfy:

- $1 \leq d_i \leq n-1$ for all i (for $n = 1$, the only possible degree is 0);
- for every constrained vertex, $d_i \bmod m = b_i$;
- $\sum_i d_i = 2(n-1)$.

We start with the smallest feasible degrees. For $n = 1$, the answer is immediate: the only degree is 0, so we output "`YES`" iff $b_1 = -1$ or the residue condition matches zero.

Assume now $n > 1$. Define a base degree $d_i^{(0)}$:

- if $b_i = -1$, set $d_i^{(0)} = 1$;
- if $m = 1$, all residues must be zero, so also set $d_i^{(0)} = 1$;
- if $m > 1$ and $b_i \neq -1$, pick the smallest positive integer with the required residue: $d_i^{(0)} = b_i$ for $b_i > 0$, and $d_i^{(0)} = m$ for $b_i = 0$.

If any $d_i^{(0)} > n - 1$, we must answer "`NO`".

Let $S_0 = \sum_i d_i^{(0)}$ and let $R = 2(n-1) - S_0$ be the remaining sum we must add. If $R < 0$, the constraints are already too tight, so the answer is "`NO`".

Now we distribute R without breaking residues and without exceeding $n - 1$:

- for free vertices, we can add 1 repeatedly up to the cap $n - 1 - d_i^{(0)}$;
- for constrained vertices (when $m > 1$), we can only add multiples of m, again up to the cap.

This is easiest as a two-stage check. First, we spend as much of R as possible on free vertices using unit steps. If free capacity is insufficient, the leftover must be covered by constrained vertices in steps of size m. In particular, if there are no free vertices at all and $m > 1$, then R must be divisible by m, otherwise no distribution can preserve all residues.

Once we obtain final degrees d_i, we construct a tree with exactly these degrees using the Prüfer method. Create a multiset that contains each vertex v exactly $d_v - 1$ times, which forms a Prüfer sequence. Maintain a min-heap of leaves (vertices with current degree 1). Process the Prüfer sequence from left to right: pop the smallest leaf u, connect it to the next sequence value x, decrement degrees, and if x becomes a leaf, push it into the heap. After processing all $n-2$ symbols, two vertices remain; connect them with the last edge. The resulting tree has degrees d_i by construction; hence it satisfies all residue constraints.

The complexity per test case is $\mathcal{O}(n)$ for building the degrees and $\mathcal{O}(n \log n)$ for the heap-based Prüfer construction. Memory usage is $\mathcal{O}(n)$.

Problem Tutorial: "Wooden Checker"

Idea: Vladislav Makarov
Development: Nikita Gaevoy

Our task is to check two conditions: the correctness of the forest and the correctness of the descendant numbers. We will check them sequentially.

The correctness of the forest condition is checked quite straightforwardly: it is enough to verify that the in-degree of each vertex does not exceed one and that the graph does not contain (undirected) cycles. The latter condition can be checked using a disjoint set system, while the first condition can be verified with a simple array.

To check the correctness of the descendant numbers condition, we will need to make a few simple observations. First, we note that when adding the edge $v \to u$, the sets of descendants change only for the vertices on the path from v to the root of the tree containing v. This observation alone is sufficient to solve the problem using data structures like a link-cut tree, but we will make another observation to obtain a simpler solution.

Consider the sets of vertices in the connected components containing v and u. These sets must form segments (since they are the sets of descendants of the roots of the trees), which we denote as $[l_u, r_u]$ and $[l_v, r_v]$. If the two segments do not form a segment in their union, we can immediately state that the root of the new merged connected component has a set of neighbors that violates the problem's condition. In this case, we can find this vertex using the disjoint set system and immediately output an error message.

Otherwise, without loss of generality, we can assume that $r_u + 1 = l_v$, and therefore, the criterion for the condition to be satisfied at vertex v is the reachability of vertex l_v from it. It is easy to see that if l_v is reachable from v, then the condition is satisfied not only for v but also for all its ancestors.

Thus, we need to learn how to solve the following problem: for a given vertex v, determine whether the vertices l_v and r_v lie in the subtree of vertex v. To check this condition, we can explicitly store the paths from the root to the vertices l_v and r_v for each connected component. When merging two components, we can perform the check by removing elements from the end until we encounter v. This operation works in total for all queries in linear time since no removed vertex can reappear in the path.

Recomputing the paths for the merged component is expressed using a single concatenation operation. To perform this operation quickly, we can use a doubly linked list or a deque, merged in a smaller-to-larger manner (this will increase the time complexity to $O(n \log n)$).

The final time complexity is estimated as the time complexity of the disjoint set system, and the required memory is linear.

Problem Tutorial: "Euler Line"

Idea: Mikhail Ivanov
Development: Mikhail Ivanov

Mathematical Facts about Euler's Line

For a non-degenerate triangle ABC, we denote: O — the circumcenter (intersection of the perpendicular bisectors), H — the orthocenter (intersection of the altitudes), G — the centroid (intersection of the medians), O_9 — the nine-point center.

From classical geometry, it is known that the points O, G, H, O_9 lie on the same line (Euler's line), and G divides the segment OH in the ratio $OG : GH = 1 : 2$, while O_9 is the midpoint of OH.

A convenient vector formulation (starting the count from O): there exists the equality

$$\overrightarrow{OH} = \overrightarrow{OA} + \overrightarrow{OB} + \overrightarrow{OC}.$$

Then automatically

$$H = A + B + C, \qquad G = \frac{A + B + C}{3}.$$

In particular, the coordinates of G always lie in the lattice $\frac{1}{3}\mathbb{Z} \times \frac{1}{3}\mathbb{Z}$.

Necessary Condition for Existence

The input provides a line

$$kx + \ell y + m = 0, \qquad k^2 + \ell^2 > 0.$$

Since the centroid G must lie on Euler's line, and thus on the given line, we have

$$kG_x + \ell G_y + m = 0.$$

Multiplying by 3 and substituting $3G_x = A_x + B_x + C_x$, $3G_y = A_y + B_y + C_y$, we obtain

$$k(A_x + B_x + C_x) + \ell(A_y + B_y + C_y) + 3m = 0.$$

The left-hand side is divisible by $g = \gcd(k, \ell)$, therefore it is necessary that

$$g \mid 3m.$$

A construction will be described that always produces a valid triangle under this condition, meaning this condition will also be sufficient.

Why You Can't "Simply Transform the Line to Vertical" Using an Affine Transformation

The definitions of O and H critically rely on perpendicularity (perpendicular bisectors and altitudes). An arbitrary integer linear/affine transformation does not preserve perpendicularity, so it does not generally map O to the circumcenter of the image and H to the orthocenter of the image. Therefore, to transfer the solution between coordinate systems, one must use transformations that preserve angles, i.e., similarities (rotation + scaling), reflections, and translations.

Key Idea: Construct a Vertical Euler Line and Then Apply Similarity

Consider an integer matrix of the form

$$M = \begin{pmatrix} p & -q \\ q & p \end{pmatrix}.$$

Such a transformation is a composition of rotation and homothety (similarity): it preserves angles, and thus preserves perpendicularity. Therefore, if we apply the transformation

$$(x, y) \mapsto (X, Y) = M\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix},$$

then the images of the points O, H, G, O_9 will correspond to the centers of the image triangle, and the Euler line will transform into the Euler line of the new triangle.

Thus, it is sufficient to:

1. Have several small integer triangles in advance, for which the Euler line is vertical and has the form $3x + r = 0$, where $r \in \{0, 1, 2\}$;
2. For the given line, choose the appropriate r modulo 3;
3. Adjust the shift (t_x, t_y) so that after applying M, the required line $kx + \ell y + m = 0$ is obtained.

Transition from (k, ℓ, m) to Similarity Parameters

Let $g = \gcd(k, \ell)$ and the primitive normal:

$$p = \frac{k}{g}, \qquad q = \frac{\ell}{g}, \qquad \gcd(p, q) = 1.$$

Suppose we start with a vertical line

$$3x + r = 0 \quad \Longleftrightarrow \quad x = -\frac{r}{3}.$$

We apply the transformation with the matrix $M = \begin{pmatrix} p & -q \\ q & p \end{pmatrix}$ and the shift (t_x, t_y). Then

$$X = px - qy + t_x, \qquad Y = qx + py + t_y.$$

We restore x in terms of (X, Y):

$$x = \frac{p(X - t_x) + q(Y - t_y)}{D}, \qquad D = p^2 + q^2.$$

Substituting into $3x + r = 0$, we obtain the equation of the image of the vertical line:

$$3pX + 3qY + \big(Dr - 3(pt_x + qt_y)\big) = 0.$$

This is a line with normal $(3p, 3q)$, thus having the same direction of the normal as the original line (k, ℓ).

To match the constant, it is sufficient to equate the free term (in the scale of $3/g$):

$$Dr - 3(pt_x + qt_y) = M_0, \qquad M_0 = \frac{3m}{g}.$$

Here, M_0 is an integer if and only if the condition $g \mid 3m$ is satisfied.

Choosing the Remainder r and Solving the Shift Equation

We rewrite:

$$pt_x + qt_y = S, \qquad S = \frac{Dr - M_0}{3}.$$

For S to be an integer, we need to achieve the comparison

$$Dr \equiv M_0 \pmod 3.$$

With $\gcd(p, q) = 1$, it is impossible for $p \equiv q \equiv 0 \pmod 3$, thus $D = p^2 + q^2 \not\equiv 0 \pmod 3$ and is therefore invertible modulo 3. Consequently, there exists a unique $r \in \{0, 1, 2\}$ satisfying $Dr \equiv M_0 \pmod 3$; for this S is automatically an integer.

After this, the equation $pt_x + qt_y = S$ is solvable in integers, since $\gcd(p, q) = 1$. One particular shift can be found using the extended Euclidean algorithm. The general solution has the form

$$t_x = t_x^{(0)} + q \cdot t, \qquad t_y = t_y^{(0)} - p \cdot t, \qquad t \in \mathbb{Z}.$$

This allows us to choose t to minimize the values $|t_x|, |t_y|$ (for example, making t_x close to 0 modulo $|q|$), thereby ensuring that the final coordinates do not exceed the limits of $\pm 10^6$.

Base Triangles

Three small integer non-degenerate scaled triangles are fixed in advance: for $r = 0$, the Euler line is $3x + 0 = 0$, for $r = 1$ — $3x + 1 = 0$, for $r = 2$ — $3x + 2 = 0$. They can be obtained by enumerating small coordinates with exact verification (by calculating O and H and checking the equation of the line OH). Since the transformation used is a similarity, it preserves non-degeneracy and (non)isosceles properties: the image of a scaled triangle will remain scaled.

Final Algorithm

For each test (k, ℓ, m):

1. Compute $g = \gcd(k, \ell)$. If $g \nmid 3m$, output six zeros.
2. Set $p = k/g$, $q = \ell/g$, $D = p^2 + q^2$, $M_0 = 3m/g$.
3. Find $r \in \{0, 1, 2\}$ from the comparison $Dr \equiv M_0 \pmod 3$.
4. Compute $S = (Dr - M_0)/3$ and find an integer solution (t_x, t_y) to the equation $pt_x + qt_y = S$.
5. Choose the parameter t in the general solution to minimize the shift and stay within coordinate limits.
6. Take the base triangle for the chosen r and apply the transformation to its vertices

$$(x, y) \mapsto (X, Y) = (px - qy + t_x,\ qx + py + t_y).$$

7. Output the resulting integer coordinates.

Why This Works

- The condition $g \mid 3m$ is necessary because the centroid always has coordinates with a denominator of 3.
- When $g \mid 3m$, the choice of r allows for the exact alignment of the free term modulo 3; then the shift solves the linear equation exactly.
- The matrix M defines a similarity, so right angles and perpendicularities are preserved, meaning O, H, G, O_9 and Euler's line correctly transition to the corresponding objects in the image triangle.

Problem Tutorial: "Traversal of a Triangular Grid"

Idea: Vladislav Makarov
Development: Vladislav Makarov

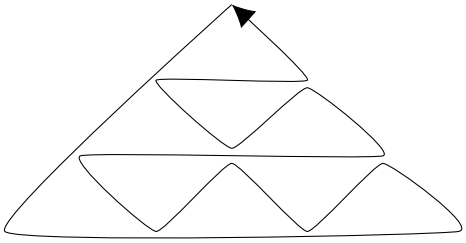
This is a simple problem that can be approached in two different ways. Firstly, one can apply the standard algorithm that finds an Eulerian circuit (see, for example, `https://cp-algorithms.com/graph/euler_path.html` for some exposition on the algorithm) in the given graph as long as it exists. This approach is completely straightforward but requires quite a bit of time to implement.

Secondly, the problem can be solved by many explicit constructions. Jury used the following one:

1. Go down-left for n steps (that is, until you hit the left corner of the large triangle).
2. Go right for n steps (until you hit the right corner of the triangle).

3. Move in a zigzag manner to the left for $2n-1$ steps. Specifically, move left-up for a single step, then move left-down for one step, then move left-up again, et cetera. The last step in this section will be a left-up step that will move you to the second lowest grid vertex on the left side of the triangle).
4. Go right for $n-1$ steps (until you hit the right side of the triangle), then move in a zigzag manner to the left for $2n-3$ steps (until you hit the left side of the triangle again).
5. Repeat the same process until you end up in the second highest vertex on the right side of the triangle after making a step to the right. After that, make a single left-up step, completing an Eulerian circuit.

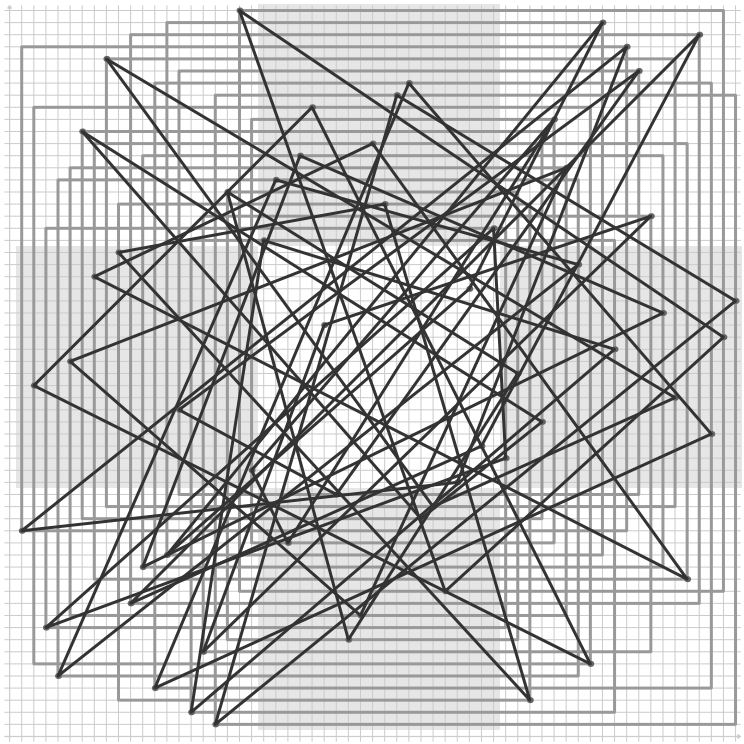
The following figure depicts how the Eulerian circuit described above looks like when $n = 3$:



Problem Tutorial: "Triangles"

Idea: Ivan Kazmenko
Development: Ivan Kazmenko

Draw a board over the points that will consist of 3×3 cells of size $(n/3) \times (n/3)$. Each of our points is located in one of the cells. To maximize the answer, each triangle has to contain a point from the left column of the board, a point from the right column, a point from the top row, and a point from the bottom row.



For each triangle, look at the coordinates of the cells that the points belong to. We want them to be $(1, p_1)$, $(2, p_2)$, and $(3, p_3)$, where p is a permutation of $(1, 2, 3)$. We will choose any such permutation and proceed to solve a subproblem with cells of size $(n/3-1) \times (n/3-1)$.

It can be proven, either by case analysis or through flows, that for at least one permutation, each of the selected three cells contains a point.

Problem Tutorial: "Construction Company"

Idea: Ivan Bochkov
Development: Ivan Bochkov

Consider a graph with vertices $0, 1, \ldots, 2m$ (corresponding to the ends of the time slots). We will discuss the bandwidth later.

Draw an edge between vertices i and $i+1$ for each i, and if there is a task $[i, j]$, draw an edge $(i-1) \to j$. We want the paths from 0 to $2m$ to correspond to sober workers (the taken jobs correspond to their edges, and between jobs the path goes along the edges of the form $i \to i+1$). What do we need from these a paths? They must cover all the complex jobs. We also have conditions such as "the i-th time slot is covered by no more than b simple jobs after the distribution of sober workers."

But this condition means that among the simple tasks covering the i-th time slot, at least c_i of them are completed. This means that no more than $a - c_i$ of the flow passes through the $i \to i+1$ edge.

In other words, the flow conditions are as follows:

1. A simple task has a capacity of 1.
2. A complex task has exactly 1 unit of flow passing through the edge.
3. The capacity of edge $i \to i+1$ is equal to $a - c_i$.
4. The total flow is a.

This is an l-r flow, and the graph and the flow are of size $O(m)$. Time complexity is $O(m^2)$.