Problem A. Coffee Cocktail

Problem author: Georgiy Charkovskiy, developer: Konstantin Bats

Let's simplify the problem right away: information about the total mass of each ingredient and the caffeine content in it is unnecessary, we can immediately move on to considering the mass of caffeine in each ingredient ($\frac{m_i \cdot k_i}{100}$), as the rest of the mass is not of interest to us. Let's denote this mass as c_i.

Subtask 1

In the first subproblem, it was guaranteed that there is no more than one type of snack. This means that if the answer exists, it is equal to 1. Therefore, it is sufficient to check that all the snacks taken together are enough to reach x caffeine.

Subtask 2

In the second subtask, on the contrary, all types of snacks are different. In this case, for all snacks, m_i and k_i are the same, and therefore c_i are also the same and equal to m_i. Therefore, to reach x caffeine, at least $\left\lceil \frac{x}{m_i} \right\rceil$ snacks will be needed, and this number will be the answer (or -1 if n snacks are not enough).

Subtask 3

The solution to the third subtask already brings us closer to the complete solution. If all k_i are equal, and t_i are different, then the answer will simply be the number of different snacks taken. At the same time, to reach x caffeine with the minimum number of snacks, it is sufficient to choose snacks with the maximum caffeine content, which in this case corresponds to the maximum m_i. So it is enough to sort all the snacks in descending order of m_i and choose in that order until x caffeine is reached or until the snacks run out. The time complexity of the solution is $\mathcal{O}(n \log n)$.

Subtask 4

Similar to the previous subproblem, but now it is necessary to explicitly calculate c_i and sort in descending order of c_i, after which to collect in that order. It was worth doing calculations in integers and using $100 \cdot c_i$ to reach $100 \cdot x$ caffeine, however, in the constraints of this problem, the precision of `double` should also have been sufficient.

Subtasks 5 and 6

The final idea necessary for the complete solution is grouping by types of snacks. If all k_i are equal, then it is most advantageous to first collect snacks of maximum mass. But if we have one snack with a mass of 100 of one type and two snacks with masses of 51 of another type, then it is advantageous to start with them. Indeed, as soon as we choose one snack of a certain type, there is no point in not taking all the other snacks of the same type, as we will approach the goal, and the answer will not increase.

Thus:

1. Calculate for each type of snack from 1 to q the total mass of caffeine in snacks of this type. In subproblem 5, it is sufficient to calculate the total mass and multiply by the same $\frac{k_i}{100}$.
2. Order the types in descending order of the sum of c_i.
3. Collect caffeine in this order until we reach x or until the types run out.

4. The number of types used will be the answer.

The time complexity of the solution is still $\mathcal{O}(n \log n)$.

Problem B. Fraction Conversion

Problem author and developer: Egor Yulin

Let's start by making some basic observations for all subtasks:

- a does not affect the answer — the integer part can be represented in any number system;
- $0.b(c)$ can be written as $\frac{b}{10^m} + \frac{c}{10^{m+k}} + \frac{c}{10^{m+2k}} + \ldots$;
- if a number is represented as a common fraction $\frac{x}{y}$, then the answer to the problem will be the minimum z such that z^k is divisible by y for some integer k.

More about the last point: we want to represent the number as a sum of terms of the form $\frac{t_i}{c^i}$ for integer i. Moreover, if the sum is finite, then all terms can be brought to a common denominator by the smallest entered i: for example, $\frac{3}{5^4} + \frac{1}{5^2} + \frac{2}{5} = \frac{3+5^2+2\cdot5^3}{5^4}$. Similarly, the reverse is also true: if a fraction is written in the form $\frac{x}{z^k}$, then it can be represented in the number system with base z by expanding this fraction into corresponding terms.

Thus, if we convert the given fraction to canonical and irreducible form $\frac{x}{y}$ and factorize y into primes as $y = p_q^{\alpha_1} \cdot \ldots \cdot p_t^{\alpha_t}$, then the answer will be $c = p_1 \cdot p_2 \cdot \ldots \cdot p_t$, because by multiplying the numerator and denominator by the same number, the fraction can be brought to the form $\frac{x'}{c^{\max \alpha}}$.

Subtask 1

However, for solving the first subtasks, it was not necessary to conduct so many reasoning, although they will be useful for us. In the case of $m = 0$, $k = 1$, there are only 10 possible variations of how the fraction will look — from $0.(0)$ to $0.(9)$. For each of them, the answer can simply be calculated manually: $0.(0) = 0$, the answer is 1, $0.(1) = \frac{1}{9}$, which can be represented as 0.01 in the ternary system (the answer is 3), the rest will also give the answer 3, except for $0.(9) = 1$, for which the answer is 1.

Subtask 2

In the case when $m = 1$ and $k = 1$, the fraction is represented as $\frac{b}{10} + \frac{c}{90}$. In other words, $\frac{9b+c}{90}$. If the numerator and denominator are reduced, then for each possible denominator, the answer can be explicitly written, or in the program, the obtained denominator can be factorized into primes and multiplied: the answer will be the product of some subset of elements from $\{2, 3, 5\}$.

Subtask 3

In subtask 3, it was guaranteed that the fraction has no period. In this case, its conversion to a common fraction is simply finding $\texttt{gcd}(b, 10^m)$, and, moreover, we already know that the answer $c \leq 10$, because in the decimal system it can be represented without a period.

It is enough to check if b is divisible by 2^m and by 5^m, and if so, the corresponding prime will be canceled out from both the numerator and the denominator, and will not enter as a factor in the answer. For example, the second example — 0.25 represents $\frac{25}{100}$ or $\frac{1}{4}$, which can be represented as 0.01_2.

Subtask 4

Similar reasoning works here as well, but we just need to simplify the infinite sum. Let's simplify it immediately in the general form: $\frac{b}{10^m} + \frac{c}{10^{m+k}} + \frac{c}{10^{m+2k}} + \dots$ contains a geometric progression with a denominator of $\frac{1}{10^k}$, so this sum can be rewritten as

$$\frac{b}{10^m} + \frac{c \cdot 10^k}{10^m \cdot (10^k - 1)}.$$

In this subtask, $m = 0$, so b is absent, and the problem also reduces to finding $\texttt{gcd}(c \cdot 10^k, 10^k - 1)$ and factorizing $\frac{10^k - 1}{\texttt{gcd}(c \cdot 10^k, 10^k - 1)}$ into primes. Since 10^k and $10^k - 1$ are always coprime, it is sufficient to find $\texttt{gcd}(\texttt{c}, \texttt{10}^\texttt{k} - \texttt{1})$, after which the remaining denominator can be factorized into primes in $\mathcal{O}(\sqrt{10^k - 1})$.

Subtask 5

In the case when $m + k \le 12$, $\frac{b}{10^m} + \frac{c \cdot 10^k}{10^m \cdot (10^k - 1)}$, which can be rewritten as

$$\frac{(10^k - 1)b + 10^k c}{10^m \cdot (10^k - 1)}$$

has both the numerator and the denominator up to 10^{12}. Therefore, with the same approach, it remains to find their $\texttt{gcd}$, divide the numerator and denominator by it, and factorize the remaining denominator into primes, which fits within the time constraints.

Subtask 6

For a complete solution, it is necessary to use a small trick. Let's immediately factorize the denominator into prime numbers: $10^m \cdot (10^k - 1)$ contains 2^m, 5^m, and the factorization of $10^k - 1$, which can be found in $\mathcal{O}(\sqrt{10^k})$. It remains to determine which of these primes are included in the numerator and will be canceled out.

For this, let's immediately notice that the sum of two numbers is always divisible by the smaller of the exponents of the prime in which it enters as terms (for example, $4 + 32$ is divisible by 2^2, but not by 2^3). And then:

- 2 and 5 enter $10^k c$ with an exponent of $k + \texttt{deg}(c)$, where $\texttt{deg}(c)$ — the exponent of their entry in c, which can be found in $\mathcal{O}(\log c)$;
- in $(10^k - 1)b$, they enter exactly with an exponent of $\texttt{deg}(b)$, because $10^k - 1$ is not divisible by either 2 or 5;
- the primes from the factorization of $10^k - 1$, on the contrary, do not enter 10^k, so in the numerator $(10^k - 1)b + 10^k c$, their exponent of entry is equal to $\texttt{deg}(c)$ for the reasons described above.

For each prime from the factorization of $10^k - 1$, we can independently find its exponent of entry in c, or we can immediately calculate $\texttt{gcd}(10^k - 1, c)$. After that, it remains only to cancel for each prime its entry in the numerator and denominator, and multiply the primes that remain in the denominator in a non-zero exponent. The obtained product will be the answer.

Problem C. Public Transportation

Problem authors: Daniil Oreshnikov and Georgiy Charkovskiy, developer: Daniil Oreshnikov

Subtask 1

Let's look at a matrix where $1 \le t_{i,j} \le 2$. Notice that for a triangle to be good, there must be at least the number 2 in its right angle (since $a > 0$ and $b > 0$). At the same time, the sum of the other two angles must be equal to the right angle. If we add $d < 0$ to the matrix, there will be no suitable candidate to be a right angle. If we add $d > 0$, then the sum of the two acute angles will be at least $2(1 + d)$, which is guaranteed to be greater than the maximum possible value in the right angle $2 + d$.

Thus, it is sufficient to simply count the number of good triangles in the given matrix. Such triangles have the form $t_{i,j} = 2$, $t_{i+1,j} = t_{i,j+1} = 1$. The number of twos, next to which there are ones to the right and below, can be counted by traversing the matrix in $\mathcal{O}(nm)$ time.

Subtask 2

For each triangle when $n = 2$, it must have a right angle in the first row, and one of the acute angles in the same column in the row below. We will iterate through all possible pairs of angles in the first row in $\mathcal{O}(m^2)$ and check if the corresponding triangle can be good.

Let's say we consider $t_{1,j} = x$, $t_{2,j} = y$ and $t_{1,k} = z$. Then we need to add such a d so that $t_{1,k}$ becomes equal to the opposite side, i.e., 1. In other words, the only suitable d is $1 - t_{1,k}$. We will add it to $t_{1,j}$ and $t_{2,j}$, and check that the remaining conditions are met.

Subtask 3

Similar to the first subtask, we notice that it makes sense to consider only d from $-\max(T)+2$ to $\max(T)$. If we add a smaller value, there will be no numbers ≥ 2 left. If we add a larger value, the sum of any two cells will be greater than any other.

We will iterate through d from -50 to 50, and then in $\mathcal{O}(n^2m^2)$ we will iterate through all possible triangles and check them for "goodness". Such a solution, with efficient implementation, passes the tests of the third subtask.

To guarantee passing this subtask, it should be noted that each selected triangle can be good for no more than one specific d. Indeed, if one d fits, then any other will violate the equality between the value in the right angle and the sum of the values in the acute angles. At the same time, a potentially suitable d can be calculated using the formula. If there is an x in the right angle, and y and z in the acute angles, then we need to ensure that $x + d = (y + d) + (z + d)$, from which $d = x - y - z$. It remains to iterate through all triangles in $\mathcal{O}(n^2m^2)$.

Subtask 5

If we further optimize the iteration, we can pass subtask 5. Let $t_{i,j} = x$, and $t_{i+a,j} = y$. Notice that by the condition, the value y must be exactly a less than the value x. If this is not the case, we will stop the iteration and move on to the next values.

In a randomly generated table, there will be few pairs of values with this property, so the iteration in $\mathcal{O}(n^2m^2)$ will actually perform about $\mathcal{O}(n^2m)$ actions, since for a large number of cell pairs, the third angle of the triangle will not even be iterated.

Subtasks 4 and 6

Let's further improve the idea described above. Notice that for all three cells of a good triangle, the condition $\texttt{row} + \texttt{col} + \texttt{value} = \texttt{const}$ is satisfied. In other words, the sum of the value, row number, and column number is the same and equal to $i + j + a + b$. Thus, it is sufficient to only consider groups of cells for which this value takes the same value.

It remains only to notice that for any triangle, where all three vertex cells have the same value of this quantity, there will be a unique suitable d to make the triangle good. Then, to solve the fourth subtask, it is sufficient to independently iterate through cells with the same value in the row and cells with the same value in the column, and then add the product of their quantities to the answer.

For a complete solution, it is only necessary to make a pre-calculation using a hash table and find for each possible value $t_{i,j} + i + j$ the number of cells with this value in each row and each column. Then, it is sufficient to iterate through all cells and for each cell add to the answer the number of cells with the same value below it, multiplied by the number of cells with the same value to the right of it. The time complexity of the complete solution is $\mathcal{O}(nm)$.

Problem D. Restore Permutation

Problem author and developer: Daniil Oreshnikov

Subtask 2

We will skip a separate solution for subtask 1, as almost any of the solutions described below can be generalized to it. To solve subtask 2, it is sufficient to directly encode the original permutation, and then reconstruct it from the encoded string. To do this, note that to represent numbers from 1 to n, $\lceil \log_2 n \rceil$ bits are sufficient, that is, in general, no more than 20 bits per number. We will write out the bit representations of each element of the permutation in turn, using the same number of bits, and then on the second run, we will reconstruct the entire permutation.

Subtask 3

This subtask stands slightly apart from the others, as the expected solution in it does not directly generalize to the following subtasks. We will calculate the polynomial hash of the permutation, that is, $\left(\sum_{i=1}^{n} x^{i-1} \cdot p_i\right) \bmod M$ for some prime x and M. We will write the binary representation of this hash as the answer.

Note that when swapping elements p_i and p_j, the hash of the permutation changes by $(p_i - p_j) \cdot (x^i - x^j)$. After reading the permutation q and calculating the initial hash, we will calculate the new hash from q, and then iterate over i and j, and check if the old hash differs from the new one by this value.

Since the permutation can have a length of up to 2000, there are around $4 \cdot 10^6$ possible outcomes, so if the hash is calculated modulo $M \approx 10^9$, with a sufficiently high probability, there will be a unique suitable pair (i, j). This solution works in $\mathcal{O}(n^2)$, which does not allow it to be applied to permutations of greater length.

Subtask 4

Subtask 4 did not have a specific author's solution and was designed for participants to come up with their own approach, fundamentally different from the ideas described in this tutorial.

Subtask 5

By slightly changing the approach with hashes, we can solve the following subtask. To do this, we will divide the entire permutation into blocks of approximately size 5000 and independently calculate the hash for each of them. On the second run, we will divide the permutation q into the same blocks, calculate the hashes, and see which blocks have changed.

If only one block has changed, we can use the previous solution to find the pair of changed elements in it. If two blocks have changed, we will similarly iterate over the position in the first and second blocks, and check that their exchange returns all hashes to their original values.

Subtask 6

The second method that could be applied in this problem is similar to the *Hamming code*. We will calculate several sums of the elements of the permutation, so that we can reconstruct the changed elements with some accuracy. Specifically, for each b, we will calculate the sum of all elements at positions whose bit number b is equal to 1. For this, we will first pad the permutation with zeros to a length equal to a power of two.

Then c_0 — the sum over all positions with 1 in the zeroth bit, that is, over all odd positions, c_1 — the sum of all elements at positions with 1 in the first bit, and so on, $c_{\lceil \log_2 n \rceil}$ — the sum in the second half of the permutation. We will output all $\lceil \log_2 n \rceil$ such sums in turn as a bit string.

On the second run, we will calculate the same sums for the permutation q. Each sum either changed or did not. If the sum did not change, then both numbers that changed places either were included in the sum or were not, that is, the corresponding bit of the changed positions is the same. If the sum changed, then exactly one of the numbers was included in the sum, and the corresponding bit of the changed positions is different. In other words, we precisely reconstruct $i \oplus j$, where i and j are the sought positions, and $\oplus$ is the bitwise exclusive OR. Moreover, the sum that changed changed exactly by $\pm(p_i - p_j)$.

This code stores about $\log_2 n$ sums, each of which is about n^2, that is, it takes up $2 \log_2 n$ bits. Unfortunately, this is not always enough to uniquely reconstruct the exchanged numbers. However, in a similar way, we can calculate the sums not of p_i, but of p_i^2, then we will know $|p_i - p_j|$ and $|p_i^2 - p_j^2|$. Knowing these two values, we can reconstruct $p_i + p_j$, and with its help — the numbers p_i and p_j themselves. In total, this approach requires about $5 \log_2^2 n \approx 2000$ bits.

Subtask 7

To further reduce the number of stored bits, we will do two things:

1. we will calculate XOR instead of sums, reducing the number of bits from $2 \log_2^2 n$ to $\log_2^2 n \approx 400$;
2. we will write down the polynomial hash of the permutation next to it (in fact, two hashes, so that the probability of collision is negligibly small) — this is an additional ≈ 60 bits.

Since the Hamming sums allow us to find $i \oplus j$, we will simply iterate over i, find the corresponding j with a single bitwise operation, and check that when they are exchanged, the hashes return to the original remembered values.

Full solution [credit to: Sergey Zolotarev]

The full solution requires moving away from the idea of Hamming codes and simply coming up with a more suitable hash function. Let's introduce the following function:

$$f(k) = \left(\sum_{i=1}^{n} i^k \cdot p_i \right) \bmod (4n^{k+1}).$$

Now we will calculate and write $f(1)$, $f(2)$, and $f(3)$ as the answer. This will take $42 + 62 + 82$ bits, which, however, requires calculations using `__int128_t` or long arithmetic. Now, if we know these values for both p and q, it is enough to look at how they change when p_i and p_j are swapped.

With such an exchange, $f(k)$ changed by $-i^k p_i - j^k p_j + i^k p_j + j^k p_i$, that is, by $(p_i - p_j) \cdot (j^k - i^k)$. Thus, we will know $(p_i - p_j) \cdot (i - j)$, $(p_i - p_j) \cdot (i^2 - j^2)$, and $(p_i - p_j) \cdot (i^3 - j^3)$ according to the corresponding

moduli. By iterating over i, we can with a high probability uniquely reconstruct the only suitable j. For a detailed description of the required transformations, we suggest you refer to the source code of the full solution in the Olympiad archives.

Problem E. DequeQL

Problem author: Daniil Oreshnikov, developers: Daniil Oreshnikov and Konstantin Bats

Subtasks 1 and 4

To solve subtask 1, it was possible to write any inefficient solution that correctly performed the operations described in the statement. In general, it should be noted that if deque d_2 has exactly i children, and deque d_1 is at position j in it, numbered from 1, then to extract d_1 from d_2, if d_2 is already the root, it will take $\min(j, i - j + 1)$ `pop` operations.

Therefore, for subtask 4, it was sufficient to write a careful simulation of all operations and maintain for each deck the sequence of deques directly nested in it $\mathrm{ch}(d)$, its "parent" $\mathrm{p}(d)$ — the deque in which it is contained, and its position in the parent $\mathrm{idx}(d)$. Then the answer to the query `pop_complexity(`d`)` is

$$\min(\mathrm{idx}(d), |\mathrm{ch}(d) - \mathrm{idx}(d) + 1|) + \texttt{pop\_complexity}(\mathrm{p}(d)),$$

because to extract the deque, you need to first extract its parent to the top level, and then extract this element from the parent. Such a formula can be computed in $\mathcal{O}(n)$.

Subtasks 2 and 3

In these two subtasks, it was guaranteed that at any given time, each deque contained no more than 2 or 3 other deques directly. In the first case, it is clear that any element can be extracted from its parent in one operation: one `pop_front` if it is the first element, or `pop_back` if it is the last.

In this case, the `pop_complexity` of any deque is simply its depth in the tree of nesting. It is not difficult to maintain the depths of vertices during all operations: any operation changes the depths of the entire subtree by the same amount, so they could be maintained using implicit treaps built on Euler tours. Alternatively, it could have been done by performing a root decomposition on the queries, rebuilding all active nesting trees in each block and calculating the depths, after which each individual query from the block would be considered separately.

In the third subtask, deques appear, the extraction of which from the parent requires two operations. Such a deque among the "children" of a specific deque can only be one — the central one out of three. In this case, the answer for its entire subtree increases by 1 compared to the above-described answer for the case $|\mathrm{ch}(d)| \le 2$. Such additions and subtractions could also be carefully maintained using Euler tours on the nesting trees. Euler tours in this case should be stored, listing each vertex only once - when entering it. The time complexity in both cases is $\mathcal{O}((n + m) \log n)$.

Subtask 5

This subtask differs from the previous ones in that it is sufficient to simply simulate operations on the deques, building a forest of nesting trees, and then calculate for each vertex its `pop_complexity` using the formulas mentioned above in a single `dfs` pass. Then the precomputed values can be used to answer queries.

Subtask 6

If there are no `pop` operations, the nesting trees only grow. It can be noticed that in this case, the `pop_complexity` of each deque does not decrease, and moreover, as soon as a deque ceases to be the root,

it will never become the root again.

If we expand the sum considered above, it can be noticed that during the life of any deque, only the last term in the sum changes, and sometimes (when the root deque is added to another), a new term equal to 1 is added to the end. Again, as noted in the solutions for the third and fourth subtasks, the operation "add 1" on the subtree of a vertex can be implemented using Euler tours on the nesting trees. And the operation of increasing the last term in the sum can be handled more implicitly:

1. for each deque, divide all time into periods with the same ancestor-root;
2. for several deque with the same ancestor-root, all subsequent periods will be the same (the period ends when the ancestor-root is added to another deque, and it becomes the new ancestor-root);
3. at the beginning of the period, maintain the current `pop_complexity` of all deques;
4. during the period, to answer a query, it is enough to take the known sum at the beginning of the period, and then take into account how many children were added to the ancestor-root on each side by that moment;
5. at the end of the period, it is necessary to fix and recalculate the `pop_complexity` of all deques — for this, it can be noted that for each immediate child of the current ancestor-root, the "distance" to the edge of the parent deque has changed by a known amount, and it is possible to simply update the Euler tour treap for its entire subtree.

Subtask 7

And finally, for the complete solution, it was necessary to notice that with each `push` or `pop` operation, the value of `pop_complexity` changes for all children of deque d_2 located between the "middle" (the central child by position) and the changing edge, exactly by 1.

Sequential children of the same deque together with their subtrees, speaking in terms of nesting trees, form a continuous segment in the Euler tour. Therefore, if the size of its subtree and all its children is stored for each deque, then for each operation it is sufficient to add $+1$ or -1 to the segment in the Euler tour corresponding to the left or right half of the children of the corresponding deque (which can be identified if `split` is implemented by a pointer to the vertex, and not by the number of vertices). Accordingly, it only requires a small carefully implemented modification to the solution described above for subtasks 3 and 4. The time complexity remains $\mathcal{O}((n+m)\log n)$.