Problem A. CosmoTile

General Observations

1. It is convenient to consider that the floor increases by a factor of x, rather than the tiles and their pieces decreasing by a factor of x;
2. If $lcm(a,b) \le a \cdot k$, then the answer is 0.

Since $lcm(a,b) = \dfrac{a \cdot b}{gcd(a,b)}$, the inequality $lcm(a,b) \le a \cdot k$ is equivalent to the inequality $\dfrac{b}{gcd(a,b)} \le k$.

In the inequalities above, $lcm(a,b)$ is the least common multiple of the numbers a and b, and $gcd(a,b)$ is the greatest common divisor of the numbers a and b.

Subgroups 1 and 2

In subgroup 1, the number b is odd. Therefore, the pieces after the cut will be of different sizes. Consequently, it will not be possible to use both pieces obtained after cutting one tile.

In subgroups 1 and 2, we can afford to model the tiling for each x and each row of tiles. The tiling continues until the total length of the tiles in the row reaches the length of the floor. The asymptotic complexity of such a solution is $\mathcal{O}(k^3 \cdot a)$. Pseudocode:

```
for x = 1 .. k
    sum = 0
    rem = 0
    for i = 1 .. x
        cur = 0
        while cur < a * x
            cur += b

        if (cur - a * x) * 2 != b
            sum += cur - a * x
        else if rem > 0
            sum -= cur - a * x
            rem -= 1
        else
            sum += cur - a * x
            rem += 1

    ans = min(ans, sum)
```

Subgroup 3

Notice that the rows of tiles are absolutely identical. Therefore, it is sufficient to model the tiling process in the first row and multiply the obtained result by the number of rows. The asymptotic complexity of such a solution is $\mathcal{O}(k^2 \cdot a)$. Pseudocode:

```
for x = 1 .. k
    cur = 0
    while cur < a * x
        cur += b

    if (cur - a * x) * 2 != b
        sum = (cur - a * x) * x
    else
        sum = (cur - a * x) * (x % 2)

    ans = min(ans, sum)
```

Subgroup 4

Notice that for a fixed x, the last tile in the first row has either dimensions a by a, if $a \cdot x \mod b = 0$, where $\mod$ is the modulo operation; or a by $a \cdot x \mod b$, if $a \cdot x \mod b \neq 0$. In the first case, there will be no unused pieces of tile left after laying the row, and in the second case, there will be a piece of size a by $b - a \cdot x \mod b$, which may be used in another row.

The asymptotic complexity of such a solution is $\mathcal{O}(k)$. Pseudocode:

```
for x = 1 .. k
    cur = a * x % b
    if cur != 0
        cur = b - cur

    if cur * 2 != b
        sum = cur * x
    else
        sum = cur * (x % 2)

    ans = min(ans, sum)
```

Subgroup 5 and Complete Solution

We will use the observation that the answer is 0 when $\frac{b}{gcd(a, b)} \leq k$, and obtain a solution with an asymptotic complexity of $\mathcal{O}(b)$. Pseudocode:

```
if b / gcd(a, b) <= k
    ans = 0
else
    for x = 1 .. k
        cur = a * x % b
        if cur != 0
            cur = b - cur

        if cur * 2 != b
            sum = cur * x
        else
            sum = cur * (x % 2)

        ans = min(ans, sum)
```

Solutions that encounter overflow of 64-bit data types, for example, when calculating $a \cdot k$, pass the tests of subgroup 5.

Problem B. Exponentiator-2025

Note that the result of the exponentiator's work is always a number of the form $a_i^{w_i}$ for some $1 \le i \le n$ and some natural w_i. Therefore, we want to determine for each a_i the maximum w_i such that the result of the exponentiator equals $a_i^{w_i}$, and check divisibility by x only for these n numbers of the form $a_i^{w_i}$.

Also, note that since x in the queries is up to 10^6, each prime factor appears in the factorization of x to no more than the 20th power. Therefore, if $w_i \ge 20$, we are not interested in the specific value, as the 20th power is sufficient to cover all possible prime factors from the factorization of x. Thus, we need to know the exact value of w_i only if it is less than 20.

Additionally, if $a_i = 1$, we are not interested in the value of w_i, as one raised to any power is still one. Therefore, we only consider w_i for interesting $a_i > 1$.

The complete solution is divided into several cases:

- For $n = 2$ and $n = 3$, in this case, it is necessary to explicitly enumerate all results of the exponentiator's work calculated modulo x for each query. The total number of ways the exponentiator can work will be $n! \cdot (n-1)! \cdots 1!$, which for $n = 3$ equals only 12, and easily fits within the time limit. For $n = 3$, it is important not to take both numbers modulo x after the first iteration of the exponentiator, as one of the numbers will later be the exponent. To avoid calculating a huge exponent completely, it is necessary to write a function `bounded_pow(a, b)` that returns $min(20, a^b)$, since again, passing numbers greater than 20 to the exponent does not make sense in the context of this problem.
- If all numbers in the array are equal to 1, the answer is `Yes` only for $x = 1$.
- If all numbers except one a_i are equal to 1, then the result of the exponentiator's work will either be 1 or a_i.
- For $n \ge 4$, there are at least two **non** ones in the array...

It turns out that if $n \ge 4$ and there are **at least** two numbers greater than 1 in the array, then for all i, w_i can always be greater than 20. We will demonstrate this with the example of $n = 4$ and the array $[x, y, 1, 1]$. The sequence of operations: $[x, y, 1, 1] \to [y^x, x, 1] \to [x^{(y^x)}, y^x] \to [y^{(x \cdot x^{(y^x)})}]$. Substituting $x = y = 2$ into the exponent, we get the number 32. It is clear that when $x, y \ge 2$, the exponent can only increase, similarly for increasing the number of elements both equal to 1 and greater than 1.

Thus, in this case, the problem reduces to "is there a number a_i in the array a such that $B \in A$, where A is the set of prime divisors of the number a_i, and B is the set of prime divisors of the number x."By writing the Sieve of Eratosthenes, we can compute the set of prime divisors for all numbers from 1 to 10^6. After that, each x can be transformed into $x' = "theproductoftheuniqueprimedivisorsof x"$, and for this number, we simply need to check if there is a number in the array a that is divisible by it. This can be done in $O(\frac{A}{x'})$ by simply enumerating all possible numbers $\le A$ that are divisible by x'. By remembering and reusing answers for identical x', the asymptotic complexity of such a solution will be $O(\frac{A}{1} + \frac{A}{2} + \ldots + \frac{A}{A}) = O(A \log A)$, where $A = 10^6$, which comfortably fits within the limit.

Problem C. Bunny 3.1

Working title "Fractal Bunny"

General Idea of the Solution

We will count $dp[i]$ — the number of ways to reach the step with number i.

The recalculation of such $dp[i]$ can be done by considering the last step before the bunny reaches step i. Then $dp[i] = \sum_{step \in STEP} dp[i - step]$, where $STEP$ — the set of possible last moves of the bunny. In other words, the number of ways to reach step i is equal to the sum of the number of ways to reach the steps one step before it.

Definition of Allowed Jumps

A step *step* is considered nice if the number $step - 1$ does not contain the digit 1 in its ternary representation. We will learn to check if a step *step* is nice. For this, we will write a function *check*, in which we will check the representation of the number $step - 1$ (how many steps the bunny jumped over) in the ternary numeral system.

```
bool check(int step) {
    step--;

    while (step) {
        if (step % 3 == 1) {
            return false;
        }
        step /= 3;
    }
    return true;
}
```

The function *check* works in $\mathcal{O}(\log n)$.

Subgroup 1. $n \leq 2000$

We will recalculate $dp[i]$ in a loop over i. We will go through all possible *step*. We recalculate only from the allowed ones. When recalculating, we update the value modulo: $dp[i] = dp[i] \% MOD$.

```
#define ll long long
const ll MOD = 998244353;
```

```
for (int i = 1; i <= n; i++) {
    for (int step = 1; step <= i; step++) {
        if (check(step)) {
            dp[i] += dp[i - step];
            dp[i] %= MOD;
        }
    }
}
```

We get a solution in $\mathcal{O}(n^2 \log n)$. This solution passes the first subgroup.

Subgroup 2. $n \leq 2 \cdot 10^4$

Notice that checking if each step is nice every time is slow. Instead, we can precompute for each step whether it is nice.

```
vector <bool> is_nice(n + 1, false);

for (int step = 1; step <= n; step++) {
    is_nice[step] = check(step);
}

for (int i = 1; i <= n; i++) {
    for (int step = 1; step <= i; step++) {
        if (is_nice[step]) {
            dp[i] += dp[i - step];
            dp[i] %= MOD;
        }
    }
}
```

This solution works in $\mathcal{O}(n \log n + n^2)$, passes the second subgroup of tests, and earns 45 points.

Subgroup 3. $n \leq 2 \cdot 10^5$

Notice that there are relatively few nice steps among all steps from 1 to n. Specifically, there are $2^{\log_3 n} = n^{log_2 3} \approx n^{0.63}$, which is about 2000 when $n = 2 \cdot 10^5$.

Using this fact, we will modify the solution. First, we will find a list of all allowed steps, after which we will recalculate dp only through them.

```
vector <int> nice_steps;

for (int step = 1; step <= n; step++) {
    if (check(step)) {
        nice_steps.push_back(step);
    }
}

for (int i = 1; i <= n; i++) {
    for (int step: nice_steps) {
        if (i - step < 0) {
            break;
        }
        dp[i] += dp[i - step];
        dp[i] %= MOD;
    }
}
```

This solution already works faster, but may not pass subgroup 3 — in this case, we can, for example, notice that we often perform the heavy operation of division by modulo, while we can perform it just once after summation.

```
for (int i = 1; i <= n; i++) {
    for (int step: nice_steps) {
        if (i - step < 0) {
            break;
        }
        dp[i] += dp[i - step];
    }
    dp[i] %= MOD;
}
```

This code works in $\mathcal{O}(n \cdot n^{\log_3 2})$, passes subgroup 3, and earns 60 points.

Subgroup 4. $n \le 10^6$

Let's consider the set of steps from which we could reach the i-th step.

```
last_j = "#*#"
digit0 = "012"

last_j = "#*#***#*#"
digit0 = "012012012"
digit1 = "000111222"

last_j = "#*#***#*#*********#*#***#*#"
digit0 = "012012012012012012012012012"
digit1 = "000111222000111222000111222"
digit2 = "000000000111111111222222222"
```

Notice that this set of points when $n = 3^k$ does not contain allowed steps in the middle third of the steps. We want to calculate for each i the sum in such a structure ending at point $i - 1$.

It is clear that the sum over all # on a segment of length 3^k, ending at point i, is recalculated as follows: $sum_k[i] = sum_{k-1}[i - 2 \cdot 3^{k-1}] + sum_{k-1}[i]$.

Then we can store $sum_k[i]$ at each point for all k such that $0 \le k \le \log_3 n$. We will recalculate $sum_k[i]$ in increasing order of k. Then the next sum will be recalculated through the previous one.

This solution works in $\mathcal{O}(n \log n)$, passes subgroup 4, and earns 85 points.

Subgroup 5. $n \le 2 \cdot 10^6$

Notice that the solution in the previous subgroup results in Memory Limit Exceeded. Therefore, we need to reduce memory consumption.
We will store our $sum_k[i]$ as $int32_t$ instead of $int64_t$ ($long\ long$). It is also worth noting that $vector < vector < int >>$ can take more memory than we expect. We can use a two-dimensional array, $vector < array < int, MAXK >>$, or store everything in a one-dimensional vector, obtaining values as follows: $sum_k[i] = dp_sum[MAXK \cdot i + k]$.

This solution receives full points.

Problem D. Metro Repair

We will solve the problem immediately for the case $\alpha = 1.5$.

Let $Rsum$ be the sum of r_i for the completed works. Let $Lsum$ be the sum of l_i for the completed works.

Then, if both conditions are met, we can write the following chain of inequalities:

$m \geq Rsum \geq \alpha \cdot Lsum \rightarrow \frac{m}{\alpha} \geq Lsum \rightarrow \frac{2m}{3} \geq Lsum$

This means that $m - Lsum \geq m - \frac{2m}{3} = \frac{m}{3}$.

In other words, to satisfy the second condition, it is necessary for all works with $r_j \leq \frac{m}{3}$ to be included in the plan, since $m - Lsum \geq \frac{m}{3}$ is guaranteed.

At the same time, it will be possible to complete at most two works with $r_j > \frac{m}{3}$, since $Rsum$ cannot exceed m.

After this observation, it is not difficult to count all work plans that include 0 or 1 work with $r_j > \frac{m}{3}$ by simply enumerating all such plans.

The counting of plans with 2 works with $r_j > \frac{m}{3}$, after some simple transformations, reduces to a similar problem:

«Given two arrays $x_1 \ldots x_n$ and $y_1 \ldots y_n$, as well as two constants X and Y. Count the number of $1 \leq i < j \leq n$ such that $x_i + x_j \geq X; y_i + y_j \leq Y$ simultaneously»

After sorting the array x in non-decreasing order and iterating over i, such a problem reduces to queries of the form «how many numbers in the prefix of the array $\leq k$», which can be answered, for example, using a MergeSortTree data structure, with a total asymptotic complexity of $O(n \log^2 n)$. However, it can also be solved with a sweep line and a Fenwick tree, with an asymptotic complexity of $O(n \log n)$. Both solutions should have scored full points, but the solution with MergeSortTree might have needed some adjustments, depending on the efficiency of your implementation.

Problem E. MAX MEX MEX

Subgroup 2. $c_i = 0$

In this subgroup, we cannot change the sets of numbers in the baskets. Therefore, we can calculate MEX_i for all baskets, after which we can compute the MEX of the resulting set.

Such a solution passes the second subgroup and earns 10 points.

Obtaining the set of valid MEX_i

Let's learn how to obtain the set of valid values MEX_i for each basket, which can be achieved by performing several (possibly zero) actions in the i-th basket.

To achieve $MEX_i = m$ for some m, we need all numbers $[0, 1, 2, \ldots, m-1]$ to be present in the basket. Additionally, the number m itself must not be present in the basket.

Consequently, if a certain number $0 \le x \le m-1$ is not present in the basket, we need to obtain it from a smaller number through several increment actions.

We will go through all numbers from 0 to n_i and check if we can achieve them as MEX_i. We will maintain a set of "free"smaller numbers and, if necessary, place the largest of them in an empty spot (it can be proven that this is optimal). We will also keep track of how many actions we have left and check for each value whether it is possible to move all numbers equal to m at least one position to the right (for MEX_i to equal m, it is necessary that the number m does not appear in the set).

Subgroup 1. $n_i \le 15$, $k \le 1000$

Let's define the set of valid MEX_i. The answer depends on which specific values of MEX_i we choose for each basket. Essentially, we are interested in what sets of values we can obtain before the final action of counting $MEX(MEX_1, MEX_2, \ldots, MEX_k)$.

So let's simply enumerate all possible subsets of values. In this subgroup, $n_i \le 15$, thus $MEX_i \le 15$. Therefore, there are a total of 2^{16} possible subsets.

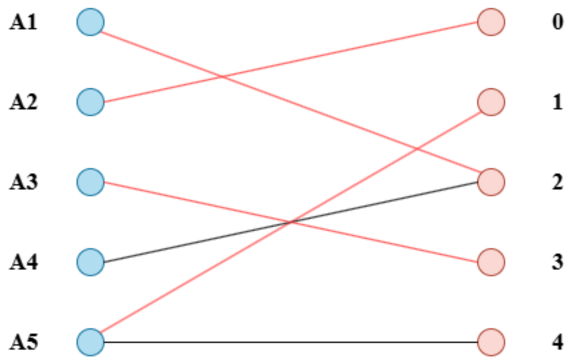
Let's obtain all achievable subsets. This can be done by maintaining $dp[i][mask]$ equal to 1 if we can obtain the set $mask$, and 0 otherwise. We recalculate the valid masks through the i-th set MEX_i as follows:

```
for (int i = 0; i < k; i++) {
    for (int mask = 0; mask < (1 << 16); mask++) {
        for (int m: possible[i]) {
            dp[i + 1][mask | (1 << m)] |= dp[i][mask];
        }
    }
}
```

Subgroups 3 and 4

Let's assume we have obtained the set of valid values MEX_i for each basket. We then get a bipartite graph between the baskets and the MEX values. The left part contains the set of baskets, and the right part contains the set of MEX_i values.

We want the first m elements on the right to be "covered"by baskets. Essentially, we are looking for a matching that covers the first m vertices in the right part.



Blue baskets and red MEX values

Such a matching can be found in the following ways:

- The Kuhn algorithm can be run sequentially from the vertices of the right part.
- A binary search on the answer along with a maximum flow search or the Kuhn algorithm also solves the problem and passes all tests.

Why does such a solution work quickly:

First, from a basket of size n_i, we can achieve a maximum of $MEX_i = n_i$, but not more. Therefore, the total number of edges in the graph does not exceed s, which is equal to 10^6.

Second, the answer to the problem cannot be too large.
Let the answer to the problem be M. Then the baskets have achieved MEX values equal to $0, 1, 2, \dots, M-1$. To achieve $MEX_i = j$, it is necessary that there are at least j numbers in the i-th basket. Therefore, to obtain an answer equal to M, it is necessary that the total number of numbers in the baskets is at least $\sum_{m=0}^{M-1} m = \frac{(M-1)\cdot M}{2}$. That is, $s \geq \frac{(M-1)\cdot M}{2}$, hence $M \leq \sqrt{2 \cdot s} + 1$. Therefore, the answer does not exceed 1000, which explains the speed of the algorithm.

Subgroup 5. b_i — any, $k \leq 1000, s \leq 3 \cdot 10^4$

In the last subgroup, there is an added complication—now each number a_j appears with a frequency of b_j. This changes the approach to finding valid values MEX_i—now, when traversing from left to right, we will maintain a set of "free"numbers along with their frequencies.

Due to b_i, the restriction on MEX_i no longer works as in subgroup 4.

Note that the answer does not exceed k, so it is sufficient to consider values $MEX_i \leq k$. Thus, there will be no more than 10^6 edges in the graph, while the answer does not exceed k. Therefore, the Kuhn algorithm or the flow search algorithm with binary search will also solve the problem in this subgroup.