Problem A. New Functionality

Solution **by emulation**:

- emulate each press separately;
- recalculate the current task number according to the rules described in the statement;

The approximate pseudocode is as follows:

```
while (a != c || b != d) {
        ans = ans + 1

        if (b == n) {
                a = a + 1;
                b = 0;
        } else {
                b = b + 1
        }
}
```

The final asymptotic complexity is $O((c-a)\cdot n)$. c and n do not exceed 10^3, so the total will be no more than 10^6 iterations.

Solution **by formula**:

- the numbering of tasks is continuous and periodic (with a period of $n+1$);
- we will transition to a unified number $F(v,k) = v\cdot(n+1)+k$;
- in this case, the answer to the problem is $F(c,d) - F(a,b)$.

This is a classic technique for continuous numbering. For example, if $n = 59$, we get the standard problem of calculating the time between (h_1, m_1) and (h_2, m_2).

Problem B. Scientific Hypotheses

The problem presents a standard **minimax** problem.

One common way to solve such problems is as follows:

- We will try to fix a limit on the maximum $pmax$;
- We will check if there exists at least one answer that does not exceed this limit.
- We will decrease $pmax$ as long as an answer continues to exist.

In this case, "an answer exists" is equivalent to "there exists some array p where the maximum value does not exceed $pmax$, all resulting reactions are correct, and the values of f are non-negative".

We will prove that this approach works. We denote the function that checks the existence of an answer as $G(pmax)$.

We can notice two properties:

- If $G(pmax)$ is true, then $G(pmax+1)$ is also true — if we have constructed an array p with a limit above $pmax$, then the same array will also work with the limit $(pmax+1)$.

- If $G(pmax)$ is false, then $G(pmax - 1)$ is also false — if we could not construct an array p with the limit $pmax$, then further decreasing the limit will not help.

This means that we can apply **binary search on the answer** to find the optimal solution.

Let's understand how the function G should work internally:

- We note that due to the requirement of non-negativity of f, it is beneficial for us to keep the professor's imagination as high as possible.
- At the same time, we do not want to raise the plausibility of the hypotheses p too much, as we have an upper limit of $pmax$.

Let's consider several cases when processing the i-th reaction:

- If the professor does not believe, we simply assign $p_i = f_{i-1} + c + 1$.

  Thus, the professor does not believe in the hypothesis, and the plausibility of the story is kept as low as possible.
- If the professor believes, there are two possible scenarios:
  - The professor will believe in all remaining stories.
    In this case, we can assign $p_i = \min(pmax, f_{i-1} + c)$.
    Thus, we do not exceed the upper limit $pmax$, and we make the hypothesis sufficiently plausible for the professor.
  - There will be at least one reaction of disbelief.
    In this case, we will have to assign $p_i = \min(pmax - c - 1, f_{i-1} + c)$.
    Thus, in the case of a future reaction of disbelief, we will be able to add $(c + 1)$ and remain within our limit $pmax$.

In the end, $pmax$ will be suitable if during the emulation process:

- all values f_i are non-negative;
- all values p_i do not exceed the upper limit $pmax$.

As a right boundary for the binary search, we can use, for example, $R = f_0 + n \cdot (c + 1)$ — the case when the professor did not believe all n times.

The final asymptotic complexity is $O(n \cdot \log R)$.

Problem C. Tomorrow Will Be Better Than Yesterday

We will compute $(n - 1)$ vectors — the difference between neighboring points: $D_i = (x_{i+1} - x_i, y_{i+1} - y_i)$.

According to the problem conditions, we need to ensure that in the new coordinate system, the following conditions hold for all vectors:

- either $Dx_i > 0$.
- or $Dx_i = 0$ and $Dy_i > 0$.

Next, we will solve the problem concerning the vectors D.

We have n vectors D_i and we need to find vectors Ox and Oy such that:

- Ox and Oy are perpendicular;
- in the coordinate system (Ox, Oy), all $Dx_i \geq 0$.
- if $Dx_i = 0$, then $Dy_i > 0$.

We can formulate the second rule in terms of vectors: $(Ox \cdot D[i]) \geq 0$, where $\cdot$ denotes the **dot product** of the vectors.

Indeed, the dot product is related to the cosine of the angle, which is directly related to the x-coordinate of the vector in the coordinate system.

Next, we note that for a final answer to exist, all vectors D must lie in the same half-plane.

We will search for the vector Ox based on this assumption, and at the end, we will simply check that the coordinate system we found satisfies all the given conditions.

If all vectors lie in the same half-plane, then the maximum angle between any two vectors is strictly less than π. The angle cannot be equal to π, as in that case, the vectors could only lie strictly on the Oy axis, and at least one of them would have $Dy_i < 0$.

Let us assume that we have found one of the extreme vectors Dm.

In this case, the angles between Dm and all vectors D_i will have the same sign (either always ≤ 0 or always ≥ 0).

For definiteness, let us choose the non-positive sign (≤ 0).

In terms of vectors, this can be written as $(Dm \times D_i) \leq 0$, where $\times$ denotes the **cross product** of the vectors.

Indeed, the cross product is related to the sine of the angle, which is non-negative for all angles from 0 to π.

Thus, the vector Dm can be found in a "maximum search" format:

- initialize $Dm = D_1$;
- if $(Dm \times D_i) > 0$ — replace $Dm = D_i$.

As a result, Dm will contain the "extreme maximum" vector from the given set. Now we need to understand how to obtain the vector Ox from Dm.

We know that Dm is, in the worst case, located somewhere "on the edge" of the half-plane — close to the positive direction of the future Oy axis.

In this case, nothing prevents us from taking the vector Dm itself as Oy — its x coordinate will be equal to 0, and $y > 0$.

In this case, we need to choose a vector for Ox that is perpendicular to Dm.

Out of the two possible vectors, we will choose the one for which $(Dm \times Ox) < 0$. With this choice, the angle between Ox and all other vectors will not exceed $\frac{\pi}{2}$ in absolute value.

Calculating a suitable Ox is easy — it is equal to $(Dmy, -Dmx)$:

- $(Dm \cdot Ox) = x \cdot y - y \cdot x = 0$.
- $(Dm \times Ox) = x \cdot (-x) - y \cdot y < 0$.

Now, for each vector D_i, it is necessary to check that all conditions are met (otherwise the answer is `No solution`):

- $(D_i \cdot Ox) \geq 0$.

- If $(D_i \cdot Ox) = 0$, then $(D_i \cdot Oy) > 0$.

Problem D. Conspiracy Theory

To begin with, let's note that:

- The order of elements only affects the output format.
- Any conspiracy contains only unique a_i.

First, we will eliminate duplicate values in the array a. From now on, we will assume that all values a_i are unique.

Next, we will sort the array of pairs $B_i = (a[i], i)$ in lexicographical order in ascending order. In the future, we will refer to the first element of the pair B_i as a_i.

Let's formalize the condition:

- You can transition from a_j to a_i if
  - $j < i$;
  - $\gcd(a_j, a_i) > 1$.
- Find the longest sequence of transitions.

Note that the transition conditions prohibit the presence of cycles by definition. Thus, we are solving the problem of the **longest path in a directed acyclic graph**.

First, let's recall how such a problem is solved in general:

- The vertices are processed in topological order of the graph;
- Dynamic programming is used: dp_i — the maximum length of the path ending at vertex i.
- $dp_i = \max(dp_j) + 1$ — calculated in $O(deg_i)$ by iterating over the edges (j, i).
- The answer is computed as $\max(dp_i)$ over all i.

The final asymptotic complexity is $O(\sum deg_i) = O(E)$ for processing edges and $O(V)$ for processing vertices.

In this problem, the graph is implicit: the vertices are positions from 1 to n, and the edges are defined implicitly through the transition condition.

It turns out that a naive solution will work in $O(n^2 \cdot \log A)$:

- The vertices are implicitly given in topological order due to $j < i$.
- $deg_i = i - 1$, hence $E = O(n^2)$;
- Checking for the presence of an edge (j, i) is done in $O(\log A)$ — we need to compute the GCD (gcd) using Euclid's algorithm.
- No additional processing of the vertex (position) is required.

How can we optimize the solution described above?

Notice that if $\gcd(X, Y) > 1$, then X and Y have at least one common prime factor. In other words, we do not care about the numbers a_i themselves — we only care about which prime factors they contain.

Let's denote the **set** of prime factors of the number a_i as P_i.

It is worth noting that the size of P_i in this problem does not exceed 7: $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 < 10^6 < 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19$.

Let's denote this limitation as $Pmax$.

So far, this only allows us to modify the edge presence check: instead of calculating $\gcd(a_i, a_j)$, we will check for the presence of a common element in the sets P_i and P_j.

This will not significantly change the asymptotic complexity — $O(n^2 \cdot Pmax)$ — but will allow us to perform subsequent optimizations.

We will introduce an additional array $dpF_{i,q}$ — the maximum length of the path ending at vertex i, where the **next** transition can be made using the prime factor q.

In this case, the recalculation of the main array will be as follows:

- Calculate $dp_i = \max(dpF_{j,q}) + 1$ for $q \in P_i$ and for $j < i$.
- Record $dpF_{i,q} = dp_i$.

Note that in the first step, we are interested in the maximum path ending at any of the vertices $1 \ldots (i-1)$ (considering the prime number q) regardless of the vertex j.

Therefore, let's introduce an array $dpP_{i,q} = \max(dpF_{j,q})$ for $j \le i$ — the optimum on the prefix.

Let's update the recalculation formulas:

- $dp_i = dpP_{i-1,q} + 1$ for $q \in P_i$.
- $dpP_{i,q} = \max(dpP_{i-1,q}, dp_i)$ for all q.

This allowed us to optimize the first step, but the second step (updating dpP) remains inefficient.

Notice that the calculation of dp_i depends only on dpP_{i-1}, while dpP_i depends on dpP_{i-1} and dp_i.

In this case, we can apply the **scanning line** technique, which allows us to eliminate the index i in the array dpP:

- First, calculate $dp_i = dpP_q + 1$ for $q \in P_i$.
- Then update $dpP_q = \max(dpP_q, dp_i)$ for $q \in P_i$.

As a result, we obtained the dynamic calculation in $O(n \cdot Pmax)$.

One question remains: how to quickly compute the set P_i for a given a_i?

We will build an array $minP_x$ — the smallest prime divisor of the number x.

In this case, to construct P_i, we only need to start with the number $x = a_i$ and divide by $minP_x$ until we reach $x = 1$.

All unique primes q encountered in the process will form our set P_i.

We can efficiently compute $minP$ using the Sieve of Eratosthenes:

- For prime numbers, set $minP_q = q$;
- When iterating over the factors of the prime number $x = q \cdot j$, update $minP_x = q$ if $q \le j$.

The final asymptotic complexity is $O(A \cdot \log\log A + n \cdot Pmax)$, where the first term is the computation of $minP$, and the second term is the computation of dp and dpP.

Problem E. Training Camps

Let's build a formal model of the problem:

- Given a directed acyclic graph (DAG).
- In the graph, there are two vertices Tm and Tk such that no edge enters them.
- We need to construct two paths (starting from Tm and Tk) such that the union of the paths contains all n vertices of the graph.

First, we will perform a topological sort of the graph by running the corresponding traversal from the vertices Tm and Tk.

If any vertex of the graph **was not visited** during these traversals, then the answer definitely does not exist.

Otherwise, we can at least reach all vertices (but it is not guaranteed that two paths will suffice).

For each vertex v, we introduce a "level" indicator L_v:

- $L_{Tm} = L_{Tk} = 0$;
- for a vertex t, the level $L_t = \max(L_f) + 1$ over all edges (f, t).

Since the graph is acyclic, the levels of the vertices L_v on any path will strictly increase.

This allows us to construct the answer using a greedy emulation, iterating through levels L from 1 to n in increasing order:

- denote the current "ends" of the paths as m and k.
- if there are no vertices at level L — then the traversal is complete.
- if there is more than one vertex at level L — then there is no answer (since there are only two paths).
- if there is exactly one vertex v at level L — we try to transition there from both vertex m and vertex k.

  If neither transition is possible, then the answer does not exist.
- if there are two vertices v_1 and v_2 at level L, we try to transition:
  - from m to v_1 and from k to v_2 (both transitions must succeed);
  - from k to v_1 and from m to v_2 (both transitions must succeed);

  If neither option succeeds, then the answer does not exist.

Problem F. Volunteering

Given n small segments $[b_i; e_i]$ and one large segment $[S; F]$.

It is required to compute for each $h = 1 \dots n$ the value a_h — the total length of the large segment covered by fewer than h small segments.

First, let's compute the value c_h — the total length of the large segment covered by **exactly** h small segments.

In this case, $a_h = c_0 + c_1 + \cdots + c_{h-1} = a_{h-1} + c_{h-1}$.

To compute c_h, we can use the classic **event sorting** technique:

- For each small segment, create two events $(b_i, OPEN)$ and $(e_i, CLOSE)$.
  - If $e_i < S$, do not add the events for the segment;
  - If $b_i > F$, similarly ignore the segment;
  - The $OPEN$ event has the coordinate $\max(b_i, S)$;
  - Similarly, the $CLOSE$ event has the coordinate $\min(e_i, F)$.
- Process the events in lexicographic order of increasing time;
  in case of equal times, place the closing events before the opening events ($CLOSE < OPEN$).

Processing the i-th event (T_i, D_i) works as follows:

- Let OpS be the number of "open" segments, and LT be the last processed time.
- Increase c_{OpS} by the amount $(T_i - LT)$.
- If $D_i = OPEN$, then OpS increases by 1, otherwise, it decreases by 1.
- Update $LT = T_i$.

Problem G. Exhausting Training

First, let's calculate N_p and N_m — the minimum number of days required to "get in shape" for the first and second skills.

Let's consider the calculation of N_p (the reasoning for N_m is similar):

- If $E \leq S_p$, then $N_p = 0$.
- Otherwise, $N_p = \lceil \frac{E-S_p}{D_p} \rceil$ — the ceiling of the number of "increases" in skill.

Assume that $N_p \leq N_m$.

In this case, we will train the skill of creativity (m) intensively all the time, while the skill of typing (p) can sometimes be trained in the "regular" way.

Let U denote the number of days with regular training for skill p.

In this case, the inequality $N_p \leq (N_m - U) + \frac{U}{2}$ must hold, or $2 \cdot N_p \leq 2 \cdot (N_m - U) + U = 2 \cdot N_m - U$.

This gives us the constraint $U \leq 2 \cdot (N_m - N_p)$. We must also remember that $U \leq N_m$ from our assumption.

Since we want to train "regularly" for as many days as possible, it is optimal to take the upper bound $U = \min(N_m, 2 \cdot (N_m - N_p))$.

Knowing U, we can easily compute the total amount of the answer:

- $N_m \cdot 4 \cdot T_m$ for intensive training of creativity.
- $U \cdot T_p$ for regular typing training.
- $(N_m - U) \cdot 4 \cdot T_p$ for intensive typing training.

The case $N_p > N_m$ is considered similarly.

Problem H. Tiring Wait

This problem involves a simple (but efficient) **emulation** of the waiting process:

- If Anatoly is at stop B at time T, we will find the nearest bus heading towards stop A.
  - Let this bus be b_i, in which case $b_{i-1} < T \leq b_i$.
  - Anatoly will take this bus if $b_i + d < a_n - d$.
  - We will increment the bus counter and move Anatoly to stop A at time $b_i + d + 1$.
- Similarly, if Anatoly is at stop A at time T, we will find the nearest bus heading towards stop B.
  - Let this bus be a_j, in which case $a_{j-1} - d < T \leq a_j - d$.
  - Anatoly will take this bus if $a_j < a_n - d$.
  - We will increment the bus counter and move Anatoly to stop B at time $a_j + 1$.

An efficient search for the "nearest" buses can be performed in one of two ways:

- **Binary search** on the arrays a and b.

  The languages `C++` / `Java` / `Python` contain built-in implementations of such a search.

  The final asymptotic complexity is $O(n \cdot \log n + m \cdot \log m)$.
- The technique of **two pointers**.
  - Maintain pointers i and j on the last considered buses.
  - Over time, the pointers can only increase.

  The final asymptotic complexity will be $O(n + m)$.

Problem I. Fair Diversity

We will calculate for each establishment the value c_i — the number of visits to establishment i over the first D days.

In this case, the answer cannot be less than $M = \max(c_i)$.

From c_i, we move to $h_i = M - c_i$ — how many visits establishment i lacks to reach the maximum.

We will emulate the process as follows:

- The process stops when all $h_i = 0$.
- If there is **exactly one** $h_v > 0$, we increase **all** h_i by 1 (after which we increase M).
- We choose two establishments with the largest values of h_i, after which we visit them (decreasing h_i by 1).

Why will this process yield the correct result?

Suppose that during the emulation process, only one $h_v > 0$ remains.

- Let $h_v > 1$.

  In this case, this establishment has been the "least visited" from the very beginning, and every day we must have visited it and somewhere else.

  Since even such "active" visits were not enough for the answer, we must visit each establishment at least one more time.

- Now let $h_v = 1$.

  In this case, we know that the total required number for the current value of M is odd, and such an answer cannot exist according to the problem statement.

  Since all previous steps were performed optimally (based on the case $h_v > 1$), we only need to visit all establishments one more time to make the total number of visits even.

For efficient emulation of the process, we will need a data structure that can efficiently perform the following operations:

- add a value;
- return an extremum (minimum or maximum);
- extract the corresponding extremum.

It is known that data structures such as "binary heap" and "search tree" can perform all these operations in $O(\log N)$ or more efficiently:

- C++: `std::priority_queue` and `std::set`;
- Java: `PriorityQueue` and `TreeSet`;
- Python: the `heapq` package.

Now we need to understand the maximum number of emulation steps:

- The value of M can be upper-bounded as $2 \cdot D$;

  for example, this bound is achieved when $n = 3$ and we visit establishments $(1, 2)$ for D days.
- In this case, the maximum number of iterations will be $2 \cdot D \cdot n = O(D \cdot n)$.

The final asymptotic complexity of the solution through emulation is $O(D \cdot n \cdot \log n)$.

There also exists a solution to this problem without using data structures.

In fact, this is the same emulation solution, but performing some checks and calculations in $O(1)$.

Let us introduce the notations $maxH = \max(h)$ and $S = \sum h$.

The current set of establishments can be visited without increasing M if

- $S \mod 2 = 0$ — the total number of remaining visits is even;
- $maxH \leq S - maxH$ — we must have enough paired establishments to fully visit the "least visited" establishment.

As long as these conditions are not met, we will increase all H by 1 (and also increase M).