Problem A. A + B

Let's consider $x = 0$. If you submit $x = 0$ you will receive a value

$$(a \,\&\, b \,\&\, 0) + (a \,|\, b \,|\, 0) + (a \oplus b \oplus 0) = (a \,|\, b) + (a \oplus b)$$

Then we can infer that $a + b$

- contains exactly one summand 2^i for every i that's present in the binary code of either a or b;
- contains two summands 2^i for every i that's present in both a and b, thus

$$a + b = (a \,|\, b) + (a \,\&\, b)$$

To get this value let's check $x = M = 2^{31} - 1$ that contains only ones in its binary code. Similarly with $x = 0$, here the interactor will return

$$(a \,\&\, b) + M + \neg(a \oplus b) = (a \,\&\, b) + 2M - (a \oplus b)$$

where $\neg$ represents a negation.

If we add these two results together, we'll get $(a \,|\, b) + (a \,\&\, b) + 2M$ which is also equal to $a + b + 2M$. Since M is a known constant, the value of $a + b$ can be easily calculated in one more action.

Problem B. Beautiful Dices

It is immediately noticeable that since the only number k is in the center of the sequence, all other die rolls must give values from 1 to $k - 1$, so there can be no more than $(k-1)^{\frac{n-1}{2}}$ different halves of the sequence. In fact, there are even fewer. Let's consider the right half. If we have already chosen its i-th element, then the element with index $2i$ is uniquely determined (as well as $4i$, $8i$, and so on). Therefore, only the arrangement of numbers at odd positions matters, and there are no more than $(k-1)^{\lceil \frac{n-1}{4} \rceil}$ different sequences.

A complete enumeration of all sequences works in $\mathcal{O}((k-1)^{n-1})$, which is too long and only passes a few tests. Enumeration that takes into account the condition of identical elements during generation works in $\mathcal{O}((k-1)^{\frac{n-1}{2}})$, which also does not fit into the constraints. If halves are enumerated independently, it results in $\mathcal{O}((k-1)^{\lceil \frac{n-1}{4} \rceil})$ but then we have to consider the restriction on favorite number pairs.

Let's generalize the idea described above: we perform a complete enumeration of all halves, prohibit favorite pairs from appearing more than once, and for each half, calculate a bit mask `inc` encoding the set of pairs of favorite numbers that it contains. Since favorite pairs are read from left to right, and for the left and right halves, the condition for identical values is different (for the left half, the distances to the center increase from right to left), we also calculate the mask $\texttt{inc}^R$ of the reversed favorite pairs included in the half of the sequence.

Now let's see which pairs of halves can be combined into a sequence. Take half a and half b. Write down the reversed a, the number k, and b one after the other. The favorite pairs that entered the resulting sequence are $\texttt{inc}^R(a) \mid \texttt{inc}(b)$. And such a combination can only be made if their intersection $\texttt{inc}^R(a) \ \&\ \texttt{inc}(b)$ is empty.

We iterate through all possible masks m_1, and for each of them, we pair them with all possible masks m_2, and if $m_1 \ \&\ m_2 = 0$, we add to the answer the product of the number of halves with $\texttt{inc}^R = m_1$ and the number of halves with $\texttt{inc} = m_2$. This solution works in $\mathcal{O}(\text{enumeration} + 4^m)$.

One possible optimization was to iterate not through all masks, but only through the "reachable" ones, i.e., those for which at least one corresponding half was generated. And then the only thing left is to optimize the iteration over the masks: for each m_1, we can only pair it with submasks of $\sim m_1$, and such pairs m_1 and submasks of $\sim m_1$ can be iterated through in $\mathcal{O}(3^m)$. This is enough to pass all tests.

Problem C. Historic Memories

There are two known solutions by the authors: one based on segment tree on the Euler tour of the tree, and the other on centroid decomposition. Here, we will provide detailed explanation of the first solution, and briefly mention the second.

It is worth noting that if the tree is hanging from vertex 1, then starting from vertex 1, we will reach vertex v at time $\texttt{depth}(v)$, where $\texttt{depth}$ is the depth of the vertex. If the memory level is initially s_v and constantly decreasing, then at that time it will be $\max(0, s_v - \texttt{depth}(v))$. Similarly, if starting from vertex 1 at time t, the memory level in vertex v upon arrival will be $\max(0, s_v - \texttt{depth}(v) - t)$.

We can notice that when changing the starting vertex to an adjacent one (more precisely, to one of its children in the hanging tree), the distance to all vertices in the subtree decreases by 1 (correspondingly, the memory levels upon arrival in those vertices increase by 1), while the distance to all other vertices increases by 1.

We can do the following: construct the Euler tour of the tree, and for each vertex, write down $s_v - \texttt{depth}(v)$. Then build a segment tree on these values, and the answer for vertex 1 when starting at time 0 will simply be the maximum value in the entire segment tree. Then traverse the entire tree, subtracting 1 from the entire segment tree upon entering vertex u, and then adding 2 in the corresponding subtree. These two actions correspond to the memory level recalculation described above. Then the answer for vertex u is also found as the maximum value in the entire segment tree.

It is clear that when returning from lower vertices to upper ones, the described operations need to be reversed, but this does not affect the asymptotic complexity: in $\mathcal{O}(n \log n)$ time, we can calculate the answers for all vertices under the condition that the path starts at time 0. For a query with a start time at time t, we simply need to subtract the precomputed answer by t.

To store all those segment trees we can use persistency and make the described operations persistent, keeping a version of the segment tree for each of the vertices.

Now the remaining idea looks as follows:

1. Using the algorithm described earlier, find the answer assuming that the memory level has been decreasing for all vertices from the beginning (then we may obtain the maximum in a city where the memory did not actually decrease, but in this case, we will update the answer with a deliberately larger value on the second step, see below);

2. Account for the cities where the memory level stopped at a fixed value — for this, simply maintain the maximum of the memory levels in those cities where the decrease stopped, which is done with just one additional variable.

To do this efficiently we can use a square root decomposition based on the queries. For each city we'll maintain information about the last query that changed it and the memory level at that time. All change queries are divided into blocks ($Q \approx 6000$), and for each block three groups of cities are maintained:

- Cities where the memory decreased throughout the block;
- Cities where the memory did not change throughout the block;
- Cities for which change queries occurred during the block.

For cities of the first type, the previously described solution was applied. For cities of the second type we can simply find the maximum of the memory level values in those cities, and always update the answer with that maximum. There could be no more than Q cities of the third type: for each of them, simply find the distance between them and the current vertex, and update the answer. To find the distance, a sparse table for finding $\texttt{lca}$ in $\mathcal{O}(1)$ could be used.

The overall time complexity of this solution was $\mathcal{O}(\frac{q}{Q} \cdot n \log n + q \cdot Q)$.

An alternative solution using centroid decomposition has a better complexity. For this solution it was sufficient to build the centroid decomposition of the tree, then separately consider all cities where the memory level does not decrease, as described earlier, and store for each centroid c the maximum value $\texttt{opt}_c = s_v - \texttt{dist}(c, v)$ in its "component". Then the answer to a '?' query from vertex v is simply a traversal through all centroids in whose components the starting vertex lies, and updating the answer through $\texttt{opt}_c + \texttt{dist}(c, v)$.

Problem D. Doctor

Let's hang the tree by any vertex v, for example $v = 1$. It's quite easy to calculate the sizes of the subtrees in the resulting hanging tree, so we can do that using dynamic programming. Now the largest subtree of a root is an answer for this problem if the doctor choses to stay on square 1.

Using the calculated values of total number of people in subtrees we can also find the answer for the problem for every other square. Roads leading from square u lead to either its children or to the tree above. Sizes of the child subtrees are already calculated and size above can be calculated as $T - a_u - \sum\limits_{u \to w} \texttt{dp}[w]$ where T is the sum of all a_i.

Thus we can calculate max queue size for every square and choose the best answer in $\mathcal{O}(n)$ time.

Problem E. Secure Prison

Let's see how much needs to be paid to build the i-th room inside and the j-th outside. If $a_j \geq a_i$ and $b_j \geq b_i$, then nothing needs to be paid. If $a_j < a_i$ and $b_j < b_i$, then the full price $a_i + b_i - a_j - b_j$ must be paid. Otherwise, one must pay either $a_i - a_j$ or $b_i - b_j$ depending on which dimension is lacking.

Consider the room (a_i, b_i). If we restrict the choice of the outer room to types with no smaller values of b_j, the answer will be $\max\left(0, \min\limits_{j\colon b_j \geq b_i} a_i - a_j\right)$. The same holds when considering rooms where the values of a_j are not less than a_i — the answer will be $\max\left(0, \min\limits_{j\colon a_j \geq a_i} b_i - b_j\right)$. Thus, to find the minimum amount of money that needs to be paid when choosing rooms where the i-th room can fit in at least one direction, we can sort all types of rooms independently by a and by b, and for each of the resulting sorted arrays, count the maximum suffix values of b and a, respectively. By remembering each room's position in these sorts, we can find the answer in $\mathcal{O}(1)$.

It remains to consider the case when it is beneficial to "expand" a room that is initially smaller than the i-th in both dimensions. For this, it is sufficient to iterate through all rooms in increasing order of b, and maintain a search tree on all previously considered rooms based on the explicit key a. For the room (a_i, b_i), it is necessary to split this tree by the key a_i and find the maximum value of $a_j + b_j$ in the left part, which can be specially maintained on the subtrees. The time complexity of such a solution in total is $\mathcal{O}(n \log n)$.

Problem F. Mixing Drinks

The main idea for solving this problem will be the observation that the strength of the coffee is continuous. That is, if we initially had coffee with strength P_0, and after some change the strength became P, then we could achieve any strength from P_0 to P by sipping from the same layers, but less.

Next, the following notations will be used: P_0 — the initial strength of the coffee in the cup, t_i — the required strength of the coffee in the query. Additionally, we will separately consider cases when $t_i < P_0$ and when $t_i > P_0$. Notice that by replacing all $p_i \leftarrow -p_i$ and all $t_i \leftarrow -t_i$, one case can be transformed into the other, so we will only describe the solution for $t_i < P_0$, while the reverse case will be completely similar.

Basic solution requires to iterate over how many of the top layers will be involved (essentially iterating over the answer to the query). Using continuity, we note that it is sufficient to check whether we can achieve a strength $\leq t_i$.

For such a check, we can observe that if we fix the available top x layers, it is sufficient to sip entirely those among them whose strength is greater than t_i. If that is not enough, then by sipping from layers with strength $\leq t_i$, we will not achieve the final strength of t_i. Accordingly, by iterating over x, we will sip all layers with strength $> t_i$, after which we will check that the final strength has become $\leq t_i$.

The total time complexity is $\mathcal{O}(q \cdot n^2)$ which is not enough.

Firstly, one could optimize the previous solution and instead of ordinary enumeration, search for the answer using binary search. This allows for monotonicity in the answer: the higher the straw we take, the wider the interval of achievable strength levels will be. This solution works in $\mathcal{O}(q \cdot n \cdot \log n)$.

To optimize even more we will iterate for each query over the affected layers of coffee from top to bottom. Then for each subsequent layer j, if $p_j \leq t_i$, we skip it, and if $> t_i$, we sip it entirely. As soon as the total strength of the drink reaches a value less than t_i, we have found our answer. To quickly compute the total strength of the drink, we need to maintain sums over the layers that have not been sipped, which appear in the numerator and denominator of the formula for P.

Using the reasoning described above, we can use a binary search on the answer. We understand that it is sufficient to sip all layers whose strength is greater than t_i, and if the layers are sorted by strength, the layers of interest will always form a certain segment.

Thus, one could precompute prefix sums of $p_i \cdot h_i$ and p_i over the layers, after which for answering a query:

- perform binary search on the answer;
- within the binary search, find the segment of interest among the layers;
- sip them and check that the final strength level is suitable for us.

Instead of the internal binary search, one could sort the queries by t_i and use the two-pointer method: for smaller t_i, we will always need no less height of the straw, and on the corresponding suffix of layers, we will be interested in a segment that is at least as long. The time complexity: $\mathcal{O}(q \cdot \log^2 n)$ or $\mathcal{O}(q \cdot \log n)$, with the two-pointer solution already being quite close to complete.

Now let us have two strengths t_i, t_j, where $t_i < t_j$, then the answer for strength t_i will be no less than the answer for strength t_j.

We will precompute answers for all $1 \leq t \leq P_0$ in descending order and for all $P_0 < t \leq 10^5$ in ascending order symmetrically. Next, as usual, we consider $t < P_0$. Let the current strength be t, then

- we will iterate over the sipped layer from top to bottom;
- we will also maintain a set of all unsipped layers that remain above, ordered by p_i (for example, using `std::set`);
- as noted above, it is sufficient to sip all layers with strength $> t$, so upon encountering a new layer, we will add it to the set, and then remove it from the set and sip all layers with $p_i > t$;
- if the final strength turns out to be $< t$, then we have found the answer for strength t.

By precomputing the answer for all strength values from 1 to 10^5, we can then respond to "small" queries in $\mathcal{O}(1)$, achieving a time complexity of $\mathcal{O}(\max(p) \cdot \log n + q)$.

For the complete solution, we will use the same idea, simply implementing the two-pointer method not for all possible strength values, but only for those appearing in the queries. That is, we will sort all queries by t_i and find the answer using the described algorithm, iterating over $t_i \leq P_0$ in descending order and $t_i > P_0$ in ascending order symmetrically.

Problem G. Crazy Arrangements

We will prove that the answer does not depend on the tree. We introduce a variable h_v, which is equal to the XOR of the values along the path from vertex v to the root (any vertex can be taken as the root). Then the XOR along the path between vertices u and v is equal to $h_u \oplus h_v$.

Based on the values of h_v, for any tree and root r, the weights of the edges can be uniquely restored if $h_r = 0$. Thus, we have reduced the problem to counting valid arrays h.

We will fix a boundary that separates the paths where the XOR is equal to 0 from the paths where the XOR is equal to 1.

Then we have the conditions $h_{u_i} \oplus h_{v_i} = q_i$ (where $q_i = 0$ for paths to the left of the boundary, and $q_i = 1$ for paths to the right of the boundary), and we need to count the number of arrays that satisfy these constraints.

This can be done in $O(n + m)$ using an algorithm similar to graph coloring with two colors. The solution will then run in $O(m \cdot (n + m))$. To further optimize this solution, we propose using the "divide and conquer" technique. We will proceed similarly to the Dynamic Connectivity algorithm. When we go to the left half of the segment, we add the right half to the DSU with weights 1. When we go to the right half, we add the left half to the DSU with weights 0.

Thus, the time complexity will be $O(m \log^2 n)$, and it can also be optimized to $O(m \log n)$ using depth-first search and explicit graph compression instead of DSU with rollbacks.

Problem H. Mood Balance

Consider the quantity $y_i = x_i + 2i$, where x_i is your mood at the i-th minute, and i is the minute of the walk. Let's track how this quantity changes as we move from minute i to minute $i + 1$.

- If you have a negative thought, x_i becomes $x_{i+1} = x_i - 1$, so $y_{i+1} = (x_i - 1) + 2(i + 1) = y_i + 1$;
- If you have an ambiguous thought, x_i becomes $x_{i+1} = 2(x_i + (i + 1) - 2) = 2(x_i + i - 1)$, which means $y_{i+1} = 2(x_i + i - 1) + 2(i + 1) = 2y_i$.

Thus, with each subsequent minute of the walk, the quantity y either increases by one or doubles. Note also that initially, before the first minute of the walk, $y_0 = 0$, and at the end of the walk, we want to obtain $y_n = 0 + 2n$. This means we have reduced the problem to obtaining the number $2n$ from the number 0 using addition and multiplication operations. Moreover, we need to use the operation that gives $y_{i+1} = y_i + 1$ as few times as possible, since we want to minimize the number of negative thoughts.

Let's look at the first time the operation of increasing by 1 will be applied. Notice that after this, y_i will only grow, and at the same time:

- we cannot perform more than $\lfloor \log_2(2n) \rfloor$ multiplications by two, because then the final y_n will exceed $2n$;
- we must perform at least $\texttt{bitcount}(2n)$ addition operations, because the multiplication operation does not increase the number of one bits in the number.

From these two observations, we derive an algorithm: we will "collect" the number $2n$ bit by bit from the most significant to the least significant: each time we multiply the current result by two (an ambiguous thought), and if the corresponding bit in the number $2n$ is one, we add 1 to the result (a negative thought). The time complexity of the algorithm is $\mathcal{O}(\log n)$.

Problem I. Delivery Robot

Note that:

1. After executing the first operation, the point (x, y) will move to point $(-y, x)$.
2. After executing the second operation, the point (x, y) will move to point $(y, -x)$.
3. After executing the third operation, the point (x, y) will move to point $(1 + y, -(x - 1))$.

4. After executing the fourth operation, the point (x, y) will move to point $(1-y, x-1)$.

After each operation, the parity of the sum of the coordinates does not change. Therefore, we cannot reach the destination point from the starting point if they have different parities of the sum of coordinates. We will prove that in other cases we can construct the required path. To do this, we will present command sequences that move from point (x, y) to points $(x+1, y+1)$, $(x+1, y-1)$, $(x-1, y+1)$, and $(x-1, y-1)$. With such transitions, we can construct a path to any point with the same parity of the sum of coordinates.

1. After executing the program "23", the point (x, y) will first move to point $(-y, x)$, and then to point $(x+1, y+1)$.
2. After executing the program "14", the point (x, y) will first move to point $(y, -x)$, and then to point $(x+1, y-1)$.
3. After executing the program "32", the point (x, y) will first move to point $(1+y, -(x-1))$, and then to point $(x-1, y+1)$.
4. After executing the program "41", the point (x, y) will first move to point $(1-y, x-1)$, and then to point $(x-1, y-1)$.

Problem J. Pizza

This task is solved using inclusion-exclusion formula.

Let's fix a subset of unsatisfied lists. Then we know whether we should include all the ingredients on unsatisfied lists. If there are no contradictions, change the answer by $+2^k$ or -2^k (depending on the parity of the number of lists in the subset), where k is the number of ingredients missing from the lists in the chosen subset (each of these ingredients can be either included or not).

If one goes over subsets with a for-loop, the solution works in $O(2^m \cdot m \cdot \max(a_i))$, which is too slow.

If one uses a recursive search instead, friend i's wish list is traversed only 2^i times instead of 2^m. Thus, the solution works in $O(2^m \cdot \max(a_i))$. This is the intended solution.

An alternative solution uses bit compression. During the recursive search two bit vectors are maintained — for good (T) and bad (F) ingredients. Before the search, similar vectors are calculated for the input wish lists. For a given subset of lists, one has to check for contradictions and count missing ingredients. The former is checked as $T \mathbin{\&} F = 0$, the latter is done by simply counting 1-bits in T and F. This solution works in $O(2^m \cdot \frac{n}{32})$.

Problem K. String Mutation

It is immediately noted that the third query is typically solved using polynomial hashing. If we can quickly determine the hash for each substring of the string, we can quickly compare substrings for equality. The remaining task is to efficiently process the first two types of queries.

We will learn to quickly update hashes for substrings. To do this, we will handle each possible character from '`a`' to '`z`' separately. We will create a Cartesian tree for each character with explicit keys, which will store the positions of occurrences of that character. Additionally, we will maintain the sum prime^i for all positions i in the subtree at each node of the Cartesian tree.

To calculate the hash of a substring, we will iterate over all characters and for each one perform a `split` of the corresponding Cartesian tree to obtain the total hash of the positions of that character in the segment. The hash of the string will thus be equal to the sum $c \cdot \texttt{hash}(c, l, r)$ for all possible characters c. The only difference is that the hash will be multiplied by prime^l, so when comparing hashes, the hash of the leftmost substring must be multiplied by $\text{prime}^{|l_2 - l_1|}$.

To respond to the first two types of queries,

1. for the first type of query, we simply move the corresponding index i from the character tree of c_1 to the character tree of c_2,
2. for the second type of query, we move all elements of the smaller tree to the larger one.

Since we always pour the smaller tree into the larger one, each element will be moved no more than $\log n$ times, except for those elements that are moved by first-type queries. Thus, we obtain a time complexity of $\mathcal{O}(26 \cdot (n+q) \cdot \log^2 n)$.

Problem L. Magical Clock

Let's represent each pair of divisions (minute and hour hands) on the clocks as a vertex of a graph. Then for each vertex, we can calculate the next state to which the clocks transition after one tick. Then for each query, one could simulate the process to determine if the desired state is reached, but such a solution would take too long.

Notice that if the minute hand catches up with the hour hand at least once while transitioning between two states, there was a moment in time between the two states when the minute hand was at zero. Denote the starting state as (h_1, m_1) and the final state in the query as (h_2, m_2). Then we can separately check if the second state is reachable from the first without the minute hand passing through zero, and then rely on the fact that at least once the minute hand jumped to zero. To check the first case, it is enough to note that the hour hand moves at a constant speed and completes strictly less than a full circle, so through $h_2 - h_1$, we can calculate exactly how much time has passed and check if m_1 transitioned to m_2 in that time.

The case of passing through 0 can be considered as follows: from (h_2, m_2), we can uniquely restore from which state $(h_2^*, 0)$ and how long ago the minute hand started moving (for this, it must hold that $m_2 \bmod 12 = 0$), and similarly, how much time will pass before the state (h_1, m_1) transitions to $(h_1^*, 0)$, and what that h_1 will be. This will happen in $\left\lceil \frac{(h_1 - m_1) \bmod (60 \cdot 12 \cdot t)}{11} \right\rceil$ ticks.

After that, it is enough to check if the state $(h_2^*, 0)$ is reachable from $(h_1^*, 0)$, and if so, in how many ticks. Each such state $(h, 0)$ corresponds to a vertex of the graph, and from each vertex, there will be one edge forward — what the next state of this kind will be obtained from this one (the weight of the edge will be the number of ticks). In general, the minute hand will catch up with the hour hand in $\left\lceil \frac{h}{11} \right\rceil$ ticks. However, one must not forget the special case $h = 0$, in which case $60t$ ticks will pass before the minute hand reaches zero.

A graph in which each vertex has exactly one outgoing edge can be represented as a set of cycles and paths that start from some other vertices and end at vertices of these cycles. Using DFS, we can identify all paths and cycles. For each cycle, we precompute prefix sums of the edge weights (starting from any of its vertices) and the position of the vertex in the cycle, and for each path the sum of the edge weights on the path from each vertex to the first vertex of the cycle to which this path leads, as well as the number of edges to the first vertex of the cycle, and which specific vertex of the cycle this path leads to.

Now, to answer the query, it is enough to check that the vertex h_2^* is reachable from h_1^*. For this, either both must belong to the same cycle, in which case the distance can be found using prefix sums, or one h_2^* must lie on the cycle, while h_1^* lies on the path leading to it, in which case the answer will be the sum of the distance to the cycle and the distance within the cycle. The remaining case is when both vertices lie on the same path; however, since paths can connect and form a tree, it is also worth compressing all cycles into vertices in advance, and using DFS on the resulting trees to calculate the entry and exit times for each vertex, which can then be used to determine the reachability relationship on paths in the original graph in $\mathcal{O}(1)$. If h_2^* is reachable from h_1^*, the distance will be the difference of the distances from them to the cycle.

In the end, do not forget to add the number of ticks to reach h_1^* from (h_1, m_1) and to reach (h_2, m_2) from h_2^*. The overall time complexity of the solution is $\mathcal{O}(720t + q)$.