Problem A. Palindromization

Let's start with some transformations. After applying them, it will be easier to think about what is required in the problem.

So, let's consider our array a.

1. Notice that there is no point in performing additions on segments that cross the middle of the array. Indeed, if m is the middle of the array, then adding x to the segment $[m-p, m+q]$, where $p < q$, is equivalent from the task's goal perspective to adding x to the segment $[m+p+1, m+q]$. Indeed, the difference between these two actions is only in adding x to $[m-p, m+p]$, and this action does not change the property of a being a palindrome.

2. Now let's reduce the problem to another one: consider the array b of the form

$$b = [a_1 - a_n, a_2 - a_{n-1}, \ldots, a_{\lfloor \frac{n}{2} \rfloor} - a_{\lceil \frac{n+1}{2} \rceil + 1}],$$

   that is, the array of differences of opposite elements of the array a. If a is a palindrome, then all elements of b are zeros. And adding x to a segment of the array a is equivalent to adding or subtracting x to a segment of the array b.

3. Now let's calculate c — the difference array of the array b, i.e.,

$$c = [b_1, b_2 - b_1, b_3 - b_2, \ldots, b_{\lfloor \frac{n}{2} \rfloor} - b_{\lfloor \frac{n}{2} \rfloor - 1}, 0 - b_{\lfloor \frac{n}{2} \rfloor}].$$

   In such an array, almost nothing changes in terms of our goal — when all elements of b are zero, all elements of c are also zero, and vice versa. However, now we have reduced adding x to a segment of b to adding x to one element of c and subtracting x from another.

Once we have calculated the array c, our task becomes to achieve a completely zero array in the minimum number of actions of the type "add x to one element and subtract from another". In case of $k = 1$, this is achieved quite straightforwardly: for each operation, you need to add 1 to a negative element and subtract 1 from a positive one.

Since the sum of the elements of the array c is zero, the necessary number of actions can simply be calculated as the sum of all its positive elements. The running time of the solution is $\mathcal{O}(n)$.

Similarly when we can add and subtract 1 or 2. The number c_i requires $\left\lceil \frac{|c_i|}{2} \right\rceil$ operations, it is impossible to do less. Let's calculate two sums of such quantities: for positive values and for negative ones. The answer will be the larger of them.

For this, first note that there is no point in subtracting from negative numbers and adding to positive ones — then it is always possible to either reduce the answer or get rid of such an operation without increasing it. And then note that it is not possible to achieve a smaller number of operations, and it is possible to get all zeros for such a number of operations. In the part where the maximum sum of $\left\lceil \frac{|c_i|}{2} \right\rceil$ was obtained, each c_i will be processed as $\left\lfloor \frac{|c_i|}{2} \right\rfloor$ operations of adding or subtracting two, and possibly one more operation of adding or subtracting one. In the other part, the sum turned out to be less, which means there were fewer odd numbers in it. Then each odd number will still have enough for one, and the "extra" ones will be grouped in pairs to add to those even numbers that lacked twos.

For the full solution it was necessary to make a few more constructive observations. First, let's reformulate the problem as "given a set of positive c_i and a set of negative c_i, it is required to find such a set of addends from 1 to 3 that they can be used to decompose all numbers of both the first and second sets". Indeed, each operation is an addition to a negative and a subtraction from a positive. In essence, we are decomposing the sets of positive and negative numbers among c_i into addends.

Now note that:

- Any decomposition into addends from 1 to 3 of a number $c_i \geq 8$ always contains several addends that give a sum of exactly 6 (proven by a careful case analysis). From this, it follows that instead of decomposing $c_i \geq 8$, you can independently decompose $c_i - 6$ and 6.
- In the decomposition of numbers $c_i = 5$ or $c_i = 7$, there will always be several addends with a sum of 3. This means you can decompose independently $c_i - 3$ and 3.

Then we will divide c into a group of negative and a group of positive numbers and transform them as described. In each group, there will only be numbers from the set $\{1, 2, 3, 4, 6\}$. Now we will present a solution that allows you to find the answer in time $\mathcal{O}(n^2)$ even though there is a quicker knapsack-based solution but it's not the intended solution.

To do this, note that the count of numbers not equal to 6 in each such set is linear from n (from each original c_i no more than two numbers from 1 to 4 were obtained). Moreover, it cannot be that in both sets there is a decomposition $6 = 2 + 2 + 2$, because then it can be replaced with $6 = 3 + 3$ with a smaller number of addends. The same is true for ones.

Then at least in one set, the number of ones does not exceed $\texttt{cnt}[1] + 2\texttt{cnt}[2] + 3\texttt{cnt}[3] + 4\texttt{cnt}[4] + 6$, and the number of twos does not exceed $\texttt{cnt}[2] + \texttt{cnt}[3] + 2\texttt{cnt}[4] + 3$. In fact, it will be seen that the necessary number of ones and twos is even less, but for now, we are satisfied with the fact that they are linear from n.

Then let's iterate through the number of used ones (m_1) and twos (m_2), find the number of threes as $\frac{\texttt{total} - m_1 - 2m_2}{3}$, and check whether both sets can be broken down into such addends. To check whether a set can be broken down into such addends, it is enough to check the following conditions:

1. If $m_1 < \texttt{cnt}[1]$, then it is not possible, because ones cannot be obtained in any other way. Otherwise, we will reduce m_1 by $\texttt{cnt}[1]$ and forget about them for now.
2. If $m_2 \leq \texttt{cnt}[2]$, then the remaining twos and all fours should have enough ones, everything else will decompose into addends equal to 3.
3. If $m_2 \leq \texttt{cnt}[2] + 2\texttt{cnt}[4]$, then the remaining fours should have enough ones. Moreover, if $m_2 - \texttt{cnt}[2]$ is odd, then the remaining two will clearly require an extra one. Indeed, either in the decomposition of some number, it will be necessary to put $2 + 1$, or from the current distribution of twos, one decomposition will have to be removed, and in it, use ones instead of twos. A small case analysis of how the addends 2 can be distributed among the "targets" shows that one extra one is enough for the remaining numbers to fall apart into addends of 3, and without it, it will not be possible to decompose.
4. If $m_2 \leq \texttt{cnt}[2] + 2\texttt{cnt}[4] + 3\texttt{cnt}[6]$, then we will break down everything as $2 = 2$, $4 = 2 + 2$ and $6 = 2 + 2 + 2$, and there will be either one or two extra twos left. Another small case analysis shows that for each of the remaining extra twos, an extra one will be needed regardless of whether to distribute the twos in the shown way or not.
5. If $m_2 > \texttt{cnt}[2] + 2\texttt{cnt}[4] + 3\texttt{cnt}[6]$, then similarly, for each extra two, one will be needed to combine them into $3 = 1 + 2$.
6. If $m_2 > \texttt{cnt}[2] + 2\texttt{cnt}[4] + 3\texttt{cnt}[6] + \texttt{cnt}[3]$, then there will not be enough "places" where twos can be distributed.

Thus, for each pair (m_1, m_2), it is possible to determine whether they give a correct decomposition, and choose the one that minimizes the total number of addends. As we already mentioned, there is also an alternative solution that reduces this problem to a knapsack problem, but we will not present it here, you can check it in on of the sources in the task's archive.

Problem B. Deck for Magic Tricks

First we will learn to find the number of the card with value x in the deck. Notice that the `base64` string is actually a representation of the number in base 64. We will convert this number into a familiar numerical

form, replacing each character with its six-bit binary representation of its position in the alphabet: '`$`' will be replaced with '000001_2', and '`a`' will be replaced with 38, which is '100110_2'. Now, if we concatenate the binary representations of all the characters of the card's value, we will get a bit string of length $6k \leqslant 60$, which can be converted into a `long long` or `Long` type number.

We must also remember that in the problem statement, the numbering of positions in the deck starts from 1, while the numbers we obtained range from 0 to $2^{6k} - 1$.

The next useful observation is that shuffling the deck has a period of $\log_2 n$, which could be observed by running simulations for small k.

From this point on, we will assume that the numbering of positions in the deck starts from zero. Notice the following fact: the operation of shuffling the deck is simply a cyclic shift of the binary representation of the card's position to the left by 1. Indeed, if we rewrite the formulas for changing the card's position from the previous subtask in 0-based numbering, we get

$$\texttt{next}(i) = \begin{cases} 2i & \text{if } i < \frac{n}{2} \\ 2i - n + 1 & \text{otherwise} \end{cases}$$

Let's see what these formulas actually do. If the number i has a zero in the most significant bit, it is simply multiplied by 2. If the most significant bit is one, the number is multiplied by 2, the most significant bit is erased, and the least significant bit is set. This precisely describes a cyclic bit shift to the left by 1. We obtain the key assertion of the problem: *if card x is at position y after t shuffles, y is a cyclic bit shift of x to the left by t.*

Finally, to obtain a solution in $\mathcal{O}(mk)$ time, it is sufficient to use the Knuth-Morris-Pratt algorithm, calculating the prefix function on the bit string obtained as the concatenation of x_i, a special symbol, and y_i repeated twice. Since the tandem repetition of y_i contains any cyclic shift of y_i as a substring, we will be able to find all suitable cyclic shift values in linear time relative to the length of the string and output the minimum.

Problem C. Unfair Game

Let's note several important criteria:

1. For the token to be able to be on the removed cell $(i_{\text{cur}}, j_{\text{cur}})$, it must be possible to reach $(i_{\text{cur}}, j_{\text{cur}})$ from (r, c) in $\sum\limits_{t \leq \text{cur}} k_t$ steps;
2. If we do not end up in a cell without unremoved neighbors along the way, each step changes the parity of the sum of the row and column indices, so the parity of the cell $(i_{\text{cur}}, j_{\text{cur}})$ must match;
3. If $\sum\limits_{t \leq \text{cur}} k_t$ is strictly greater than the length of some path from (r, c) to $(i_{\text{cur}}, j_{\text{cur}})$, and the final cell has at least one neighbor, it is possible to reach that cell via the found path and then make paired moves to a neighboring cell and back;
4. Alternatively — if the token could be in cell $(i_{\text{cur}}, j_{\text{cur}})$ exactly at the moment when its last neighbor was removed, then it can be in that cell at any moment afterward, as there will be no more moves from it.

To find distances between cells, a breadth-first search from the starting cell is required. Since at the moment we reach cells at a distance of $> \sum\limits_{t \leq \text{cur}} k_t$, we do not yet know if we will be able to pass through them after the computer's next move, we will

- maintain a global queue `bfs` with unprocessed cells;
- on each turn, continue `bfs` from the state where we stopped last time, skipping already removed cells;

- stop `bfs` upon reaching cells at a distance greater than the traveled distance;
- for all cells adjacent to the removed one, check how many of their neighbors are still unremoved — if all neighbors are removed, and the token can be in that cell at that moment (the distance is sufficient and the parity matches), we will remember that cell in a separate set.

Then the answer will be the first removed cell for which the criteria described at the beginning of this analysis are met.

The overall time complexity of the complete solution is $\mathcal{O}(nm)$ since it basically performs a `bfs` step-by-step with breakpoints as you would do in a debugging session.

Problem D. Hackathon

We will maintain the current position of each participant and his current "goal" — the position of the minimum on his prefix in the array a.

One possible approach to solving the problem was sorting events. Note that naive simulation is not efficient for several reasons:

1. we spend $\mathcal{O}(m)$ time processing each second, even those in which nothing happens;
2. we take a long time to find new goals for participants, even without separating those for whom they need to be recalculated and those for whom they do not.

These actions could have been implemented more efficiently. Let's create a heap that stores events like "participant came to his (possibly outdated) goal" and "participant approached the row of pizzas". We will order events in the heap by time, and if the time mark is the same, we will first process those who approached, and then those who reached the goal.

At the moment of approaching the row, we calculate the current goal of the participant — it will be the minimum on the prefix in the current state of the array a. We calculate the moment when he will approach this goal, and add the corresponding event to the heap. When processing the event "participant approached his goal", we recalculate events for all those who went to the same position. This can be done either immediately or deferred — recalculate the goal when the participant approaches his old goal and sees that the value known to him before is not relevant. These two approaches give the same result.

With an efficient implementation of finding the minimum on the prefix (for example, through a segment tree), such a solution passes approximately half of the tests.

The key idea for the full solution is to process "goals", that is, pieces of pizza, using a heap in ascending order, instead of processing people. Indeed, it will be difficult to find the final point for the participant by position and start time of movement. It is easier for each position to find the participant who will stop there.

We will process pairs $(a_i, -i)$ in ascending order. For the next such pair, we will find the participant who will occupy it. For this participant j, three conditions must be met:

1. $s_j \geq i$, that is, the participant approached the row not to the left of this position;
2. by the time the participant approaches position i, that is, $t_j + v \cdot (s_j - i)$, a_i is the minimum on the corresponding prefix;
3. of all such participants, the one who approached this position earlier than everyone else is chosen, or, if there are several such, the one with the minimum number among them.

Let's transform the condition on time: the fact that at the time of approaching i it is the minimum on the prefix is equivalent to the fact that the time of approach is not less than any time of taking a piece of pizza located to the left (since we process positions in ascending order of a_i values). Let's call the moment

when a_i becomes the minimum on the prefix $\texttt{stable}(a_i)$. Then we want $t_j + v \cdot (s_j - i) \geq \texttt{stable}(a_i)$ or $t_j + v \cdot s_j \geq \texttt{stable}(a_i) + v \cdot i$. On the left is a function of j, on the right is a function of i and the current state of a_i.

Now let's reformulate the task as "find a participant with $s_j \geq i$ and the minimum $t_j + v \cdot s_j$, not less than $\texttt{stable}(a_i) + v \cdot i$". This can be done in several ways, one of which is to sort the participants by s_j, calculate the desired value $\texttt{target}_j = t_j + v \cdot s_j$ for each, and divide into blocks of size $\sqrt{m \log m}$, after which sort the participants in each block by `target`. Then, to find a participant whose `target` is minimal among all not less than $\texttt{stable}_i + v \cdot i$, it is enough to do a binary search in several blocks, and part of the block can possibly be processed separately, looking at each participant. Such a request works for $\mathcal{O}(\sqrt{m \log m})$.

It remains only to learn how to mark a participant who has received a piece of pizza as having left the row. To do this, you can either rebuild his block from scratch, or use the path compression technique — remember for each person the next `target` in his block, which has not yet been deleted, and when finding a participant, go through these links, compressing them into shorter paths in parallel.

Separately, it should be noted that if there are no participants taking a piece for themselves at position i, then this position will never become a goal for anyone, so it does not need to be added back to the heap. The full solution, thus, works for $\mathcal{O}((n+m)\sqrt{m \log m})$. There is also a more asymptotically efficient solution with a segment tree on sets to search for a participant with the same criteria, but it has a larger constant, and the physical running time is longer.

Problem E. Restore Permutation

For larger values of m 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.

With lesser values of m it becomes clear that encoding the permutation without the loss of information is impossible. Which means that some sort of hashing will be involved. 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.

For the permutations with 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.

By slightly changing the approach with hashes, we can improve our solution. To do this, we will divide the entire permutation into blocks of 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.

Another 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 we define c_0 as the sum over all positions with 1 in the zeroth bit, that is, over all odd positions, c_1 as the sum of all elements at positions with 1 in the first bit, and so on, $c_{\lceil \log_2 n \rceil}$ is defined as 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.

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.

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 archives.

Problem F. Self-Descriptive

In mathematics, this sequence is known as the *Golomb sequence*.

Firstly one can deduce that $a(n) = 1 + a(n - a(a(n)))$; it can be proven in several ways, however we won't focus on it here since it's a known mathematical construct with a lot of information about it online. Then there are only two things left to notice, first of which being that sum query problem can be reduced to prefix sum query.

For the last observation we need to use the specific structure of this sequence. Notice that it has a lot of repetitions that come in runs of adjacent elements. This means that we can use `Run-length encoding`. Instead of storing each number explicitly, we will work with pairs (`value`, `count`). It's a bit harder to calculate prefix sums on these pairs, but still not impossible: we calculate prefix sums for counts (to find the bound with binary search) and total sums (in other words, for value times count products). This solution works for $r_i \leq 10^{10}$.

The final solution requires one similar step. Let's look at segments in the previous solution. Using the same argument, the "counts" in these segments monotonically increase and have a lot of repeats. In fact, the sizes of segments from the previous solution are precisely the original sequence (because it is self-descriptive). So let's do the compression again, but this time, on segments. In the end, we'll store the sequence as blocks "numbers from a to b inclusive; each is repeated k times". Similarly, we can build prefix sums on these blocks and use binary search with prefix sums to answer the RSQ queries.

The time complexity consists of the precalculation (we need to calculate at most $q < 10^6$ blocks) + $O(\log q)$ per query.

Problem G. Space Battleship

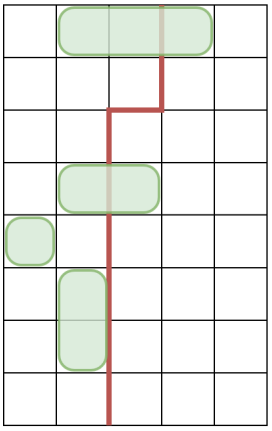
To solve this problem, we will apply dynamic programming on a skew profile.

We will gradually fill the board with ships, moving from left to right across the columns. To place a ship, we need to know how many ships of that type are left. Additionally, we need to keep track of previously placed ships — they may intersect or touch the one we want to place.

We will traverse the board from left to right and top to bottom. At any moment, we will have one cell for which we determine what will happen. There are four options:

1. Leave the cell empty. For this, there should not be an '`x`' symbol in it, and there should not be any already placed ships covering this cell.
2. Continue the ship placed previously to the right. For this, we need to remember whether any old ship is "sticking out" ahead. How to do this will be explained later.
3. Place a ship vertically in this cell. Then it will cover this cell and several cells below. To make this possible, we need to check if we are not overlapping with already placed ships and if we are not trying to cover cells marked with '`o`'.
4. Place a ship horizontally in this cell. Again, we need to check if any previous ship is obstructing us and if we are not violating the information about the board.

In the end, we need to remember the profile of the ships — how much they "stick out" beyond the already considered cells. It is proposed to store this using a bitmask, where each cell will be allocated two bits, encoding values from 0 to 3. A value of 0 means that there are no ship cells on the boundary, a value of 1 means that there is a cell on the boundary, 2 means that the ship protrudes one cell beyond the boundary, and 3 means that the ship protrudes two cells beyond the boundary.



In this example, the skew profile is indicated by a red line, and the green shapes represent the ships. The state of the board will correspond to a pair of numbers $(2, 0110202_4)$, where the first value of the pair indicates the position of the skew profile, and the second represents the aforementioned mask.

The final dynamic programming state will be $\texttt{dp}[c][\text{ship}_1][\text{ship}_2][\text{ship}_3][r][\text{mask}]$, where c is the column, and r is the skew point.

This problem required quite careful implementation. The total number of masks for each profile was no more than 2400, although the numerical values of the masks could be very large. To reduce memory consumption, it was necessary to compress the mask values before computing the dynamics. Additionally, it made sense to perform the dynamics forward and eliminate the first dimension, storing only the last two columns.

For further acceleration, it was possible to precompute whether it was possible to place each ship in each cell and for each of the five possible placements. This significantly sped up the solution. Also, for each mask and skew point, it was possible to precompute what the mask would be for each transition. The last optimization applied by the problem author involved changing the indices of the dynamics and the order of computations: the possibility of placing a ship of a certain type is unambiguously determined when fixing (c, r, mask), so it was worth iterating over them first and then unconditionally updating the dynamic values in loops over $\text{ship}_1, \text{ship}_2, \text{ship}_3$.

The final asymptotic complexity of the solution is $\mathcal{O}(wh \cdot s_1 s_2 s_3 \cdot \text{States})$, where States ≈ 2400.

Problem H. Ancestral Problem

We want to apologize for the fact that this problem seems to be already well-known for some of our participants. Since our direct sources of this problem are different (even though it's possible that the origin is the same) it was hard to predict that this problem was already used in some competitions you could have participated in.

We hope that this didn't cause any significant changes in today's contest difficulty or, more importantly, usefulness.

The task is to check whether the first tree has a subgraph isomorphic to the second tree. Let's try to solve the problem with any asymptotic complexity and then simplify the solution.

We will root the first tree and solve the problem using dynamic programming on its subtrees. Let $\texttt{dp}[u][x \to v]$, where u is a vertex of the first tree and $x \to v$ is a directed edge of the second tree, represent whether it is possible to match vertices u and v, while

- matching subtree of u with subtree of v;
- under the condition that in the second tree, the parent of v is x.

In other words, we are matching edges $p(u) \to u$ and $x \to v$, as well as subtrees of u and v.

To calculate such dynamics, we need the values of all $\texttt{dp}[u'][v \to v']$, where u' is a child of u, and v' is a child of v. The recalculation itself is a check for whether there is a matching in a bipartite graph that covers the left part: indeed, the subtrees of some children of u need to be "covered" by the subtrees of children of v, while all others need to be added separately.

Unfortunately, such dynamics work very slowly and do not provide an answer to the problem. Let's fix both of these issues at once. We will look at the recalculation of all dynamic states of the form $\texttt{dp}[u][x \to v]$ for different x. To recalculate them, we look for matches for graphs with the same left part (all children of u) and a right part that differs by one vertex (all neighbors of v, except x).

Instead, let's immediately run a matching search on the graph with all the vertices of interest, that is, only once for the pair of vertices (u, v). Then:

- if we found a matching that covers the right part, then the answer for all x will be `true`;
- if we could not cover at least two vertices, then the value of all $\texttt{dp}[u][x \to v]$ will be `false`;

- and if we managed to cover all vertices except one x', then for all other x we will only have to find an alternating path from x to x' using one `dfs`.

The solution ends here; for the answer, it remains only to consider some leaf as x and find if there is a corresponding `dp` equal to `true`.

Considering the time complexity: we run one matching search algorithm for each pair of vertices (u, v). Let's count the maximum possible number of edges in all the resulting graphs, that is, $\sum\limits_{u,v} \deg(u) \cdot \deg(v) = \sum\limits_{u} \deg(u) \cdot 2 \cdot m = 4nm$.

If we purely estimate the Kuhn algorithm, we will get the asymptotic complexity of the entire solution equal to $\mathcal{O}(nm^2)$; however, on specific graphs, the Kuhn algorithm actually works much faster, so such a solution already passes with a sufficiently careful implementation. If you really want to write an asymptotically faster solution, implement Dinic's algorithm to achieve a solution in $\mathcal{O}(nm\sqrt{m})$.

Problem I. Secret Folder

Note that we can immediately eliminate passwords that are substrings of other strings. To do this, we will check all possible pairs of strings and see if string s_i is contained in string s_j $(i \neq j)$ using any suitable algorithm (for example, the KMP algorithm or hashes). We also need to be careful with identical strings.

Now let's construct a directed graph with n vertices. The length of the edge (i, j) will be equal to the minimum number of characters that need to be added to the end of s_i so that the resulting string contains string s_j as a substring (less formally, we are interested in the maximum "overlap" of s_j on the right side of s_i). Let's learn how to calculate this value: since no two strings now contain each other, we will find the maximum suffix of s_i that matches the prefix of s_j of the same length. It is then easy to see that the sought value is the difference between the length of s_j and the length of the maximum matching suffix of s_i and the prefix of s_j. The maximum matching prefix and suffix can be found by taking the last value of the prefix function of the string "s_j#s_i", or by calculating a similar value using hashes.

Now note that the answer to the problem is the Hamiltonian path of minimal length in the graph described above, considering that starting from the i-th vertex "costs" the length of the string s_i. It is not difficult to show this: let's look at the occurrences of these strings in the answer. Obviously, there is no point in including one string in this order more than once, since the triangle inequality holds in the graph. And to minimize the answer for a given order, it is necessary to maximize the "overlap" of adjacent pairs of strings — this is precisely what we maximize in the sought graph.

As a result, we obtain a solution in $\mathcal{O}(2^n \cdot n^2 + S \cdot n^2)$ for finding the Hamiltonian path of minimal cost and removing strings that are contained in others.

Problem J. Tree Trisection

Let's introduce some notations first. Denote the minimal connected set containing all leaves from l to r with the root at v as $\texttt{cover}(v, l, r)$. Also, denote the minimal leaf in the subtree of vertex v as $\text{left}(v)$, and the maximal one as $\text{right}(v)$.

Now let's reformulate the problem statement a bit simpler and note several important facts. A complete binary tree is given, and there are queries specifying a segment of its leaves $[l, r]$. Initially, a set $V = \texttt{cover}(\text{lca}(l, r), l, r)$ is selected. Then it is required to choose two vertices x and y in this set so that if you "cut off" the set V itself and the subtrees of x and y, the maximum of the weights of the three resulting subtrees will be minimal.

Accordingly, two useful facts:

1. The vertex $\texttt{root(V)}$ can be found as $\text{lca}(l, r)$, where in the numbering from 1 the left child of vertex i has number $2i$, and the right one has $2i + 1$, so $\text{lca}(l, r)$ has a number whose binary representation is the longest common prefix of the binary representations of l and r.

2. To minimize the maximum of the weights of the three trees, it is enough to try to bring their weights as close as possible to one third of the weight of V. Indeed, the sum of their weights is fixed and equal to the weight of V, and therefore the maximum of them will not be less than $\frac{\texttt{weight}(V)}{3}$.

Let's note that it is possible to calculate the total weight of $\texttt{cover}(p, a, \text{right}(2p))$ or $\texttt{cover}(p, \text{left}(2p+1), a)$ for each leaf a and its ancestor p, i.e., the subtree containing either all the leaves between a and the "middle" of the subtree p. Then for $v = \text{lca}(l, r)$, the weight of $\texttt{cover}(v, l, r)$ decomposes into the weight of $\texttt{cover}(v, l, \text{right}(2v))$ and the weight of $\texttt{cover}(v, \text{left}(2v+1), r)$, which can be calculated in $\mathcal{O}(1)$ using precomputed values.

For a complete solution, it was required to calculate some auxiliary information or the answers themselves faster. Since it is forbidden to choose x and y that are an ancestor and a descendant, we want to independently cut out two subtrees from V so that their weights and the weight of what remains are as close to each other as possible. As we have already noted, for this it is enough to choose the subtrees x and y with weights closest to one third of the weight of V, although this statement requires clarification (see below).

But for now, let's consider two cases — either x and y lie on the same side of $v = \texttt{root}(V)$, or on different sides. If they are on different sides, then without loss of generality, x is in the left subtree, and y is in the right. We will then immediately respond to all queries for which $a = \texttt{lca}(l, r)$ is equal to the current vertex. To do this, we independently find x closest to one third of the weight of V in the left subtree of v, and y in the right.

For this, we remember for each query its l, and we will move along the leaves from the "central left" ($c_1 = \text{right}(2v)$) to the left, and for each leaf i we will recalculate $\texttt{cover}(v, i, c_1)$. Among all the vertices of this set, we want to be able to select a tree with a weight closest to a certain value at any moment. Note that the entire set $\texttt{cover}(v, i, c_1)$ consists of

- "complete" vertices, for which each vertex of the subtree also lies in this set,
- and "incomplete" ones, for which some of the vertices of the subtree have not yet entered this set.

There are no more than $\log_2 n$ "incomplete" vertices since they are all ancestors of the last added leaf; and the rest can be stored together with the weights of their subtrees in some kind of search tree that allows finding the nearest value element in logarithmic time (for example, in a Cartesian tree).

Then for each leaf i: we go through its ancestors, update the weights of their subtrees, those that have become "complete" are inserted into the search tree. After that, if for some query $l = i$, we look for a subtree of the required weight in the search tree, and check $\mathcal{O}(\log n)$ "incomplete" subtrees manually. A similar process is performed for the right borders, only starting from the leaf $c_2 = \text{left}(2v+1)$ to the right up to $\text{right}(v)$.

The running time of such a solution is $\mathcal{O}(n \log^2 n + m \log n)$, the first term comes from the fact that we will put each vertex once in the search tree during the processing of each of its ancestors, and the second from the response to queries.

It remains only to find answers to queries for which x and y lie on the same side of $v = \text{lca}(l, r)$. To do this, note that this can be beneficial only if the weight of some part of V on one side of v, including the weight of v, is less than the current answer found. This can only happen for one of the two sides (left or right), so we will "postpone" this query to the corresponding child of v.

For this, we will represent our queries not as (l, r), but as (l, r, Δ), where for queries from the input data $\Delta = 0$. This will mean that we are not minimizing $\max(w_v, w_x, w_y)$, but $\max(w_v+\Delta, w_x, w_y)$. "Postponing" the query down the tree, we will increase its Δ by the weight of the part that we left "above" — later this part will be attached to the upper of the three parts obtained from the postponed query. Such queries can be answered in the same way described above, but now we need to look for weights closest to $\frac{\texttt{weight}(V)+\Delta}{3}$. The running time of the responses to queries then becomes $\mathcal{O}(m \log^2 n)$, because each query generates no more than $\log_2 n$ new queries, postponed down the tree.

Clarification about weights: in fact, the words about "weight closest to one third" are not entirely correct. It is sufficient to select at least one of x and y with a subtree weight that is one of the two closest to $\frac{\texttt{weight}(V)}{3}$. But if one of the vertices has already been chosen, for example, x, then y must be chosen with a subtree weight closest to $\frac{\texttt{weight}(V)-\texttt{subtree}(x)}{2}$. Therefore:

1. we will make a pass through the leaves in the left subtree of v, find candidates for x with a weight closest to one third of the weight of V (the nearest not smaller and the nearest not larger);
2. we will make a pass through the leaves in the right subtree of v and for each corresponding candidate x find the most suitable y; at the same time, we will find candidates for y with a weight closest to one third of the weight of V;
3. we will make another pass through the left subtree, and find for each candidate y the most suitable x;
4. from all the found pairs of candidates (x, y) we will choose the optimal pair.

Problem K. Public Transportation

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. If we iterate through all possible pairs of angles in the first row in $\mathcal{O}(m^2)$, we can check if the corresponding triangle can be good.

Firstly, 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.

Also 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 $d = x - y - z$.

Then we can 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 find the answer, 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 L. Crossbreeding

We will start from the end: let the answer to the problem be the number x. We will look at its bit representation and find the first (most significant) bit $\texttt{lead}(x)$. Let it be at position y from the end, which corresponds to 2^y in the decomposition of x into a sum of powers of two. Then we notice that any a_i for which $\texttt{lead}(a_i) > y$, meaning the most significant one is earlier, must be paired with a_j whose bits above y match a_i, so that their $\texttt{xor}$ gives 0 in the higher bits.

Thus, to have $\texttt{lead}$ equal to y in the answer, all a_i with $\texttt{lead}(a_i) > y$ must be paired with the same $\texttt{head}_{y+1}(a_i) = a_i >> (y + 1)$. Here, $>>$ denotes the right bit shift operation, which discards the last bits of the number. In this case, $a_i >> (y + 1)$ discards all bits from the 0-th to the y-th inclusive.

The algorithm is as follows: we will iterate y from 29 to 0 (instead of iterating, we could perform a binary search to slightly increase efficiency), group all a_i by their $\texttt{head}_y$, that is, by the $30 - y$ most significant bits, and check that all groups with a non-zero $\texttt{head}_y$ contain an even number of elements. If this is the case, it is possible to ensure that all $30 - y$ most significant bits of each number in the answer are zeros by simply pairing the original numbers within the groups.

As soon as we find such a y for which at least one group has an odd size, it is clear that the answer will have a 1 in the corresponding bit. At the same time, for all $\texttt{head}_{y-1}$ groups, the sizes are still even, so the numbers need to be paired within those groups. We will then group a_i by their $\texttt{head}_{y-1}$ and within each group, separate those whose next bit is 0 from those whose next bit is 1.

After that, we will solve the problem independently for each group.

- If sub-group with 1 in the next bit has even number of elements, they can be paired within that group so that all resulting numbers have a 0 in the next bit, and in this case, they will all be less than the answer.
- Otherwise, we can pair all but one. The remaining number can either be left as is, in which case the answer for this group will simply be the minimum of the numbers with a 1 in the next bit, or it can be paired with a number from the same group but with a 0 in the next bit.

  In this case, it is sufficient to find $\min\limits_{\substack{a_i \in \texttt{group}_0 \\ a_j \in \texttt{group}_1}} a_i \oplus a_j$, where $\texttt{group}_0$ and $\texttt{group}_1$ are the elements of the current group with 0 or 1 in the next bit, respectively.

  Finding such a minimum is a classic problem that can be solved using a bit trie: we will treat each number as a string of 30 zeros and ones, add all elements of one subgroup to such a trie, and for each element of the second subgroup, we will search for a pair that minimizes their `xor`, trying to go through the matching bit in the trie each time.
- The size of the sub-group with 0 in the next bit doesn't matter since all of those numbers are already less than the answer.

Now that we have obtained the minimally possible maximum number in each group, the answer will be the maximum of the answers for all groups. The overall runtime of such a solution is $\mathcal{O}(n \log A)$, where A is the upper limit on the values of a_i.