Problem Tutorial: "Algorith For Robot"

We will keep the positions where the robot tries to collect the tokens and recalculate them using the data from previous cyclic shift. For the initial string we will find those positions with the simulation. Then we shall support the movement of one letter from the beginning of the string to the end:

- If the moved letter is L, then all positions where T is applied will move one step right.
- Similarly, if the moved letter is R, then all positions where T is applied will move one step left.
- If the moved letter is T. If we move it from the first position to the right, then we can't say for sure if the token at the position $n+1$ can be collected. Let the robot moves to the position $n+1+d$ at the end of program; then we can say for sure that the token at this position is collected. To check any real changes we need to keep for each position the number of attempts to collect the token from there. If for position $n+1$ number of attempts to collect was greater than 1, then the token is still collected, otherwise it will not be collected after the shift.

To keep for each position number of the attempts to collect the token from it, we can use the treap. The positions itself can be kept in the bitset, same as the cells where the tokens are placed. Then, to find for next shift number of the tokens collected, we can just apply bitwise AND operation and then count number of 1's in the result.

The working time can be estimated as $O(n \cdot \log_2(n) + n \cdot n/32)$.

Problem Tutorial: "Build The Tree"

Let T be a rooted tree. Let us introduce some notation.

- $size(T)$ — the number of vertices in T;
- $sumR(T)$ — the sum of distances from the root to each other vertex;
- $sum(T)$ — the sum of distances between each pair of vertices.

Now, we have to find a tree T such that $size(T) = n$ and $sum(T) = m$.

Let T_1 and T_2 be rooted trees, and T be a tree we get if we link the root of T_1 to the root of T_2 and set the root of T_2 as the root of T . Note that $size(T)$, $sumR(T)$ and $sum(T)$ can be calculated using only such values for T_1 and T_2.

Now we forget about trees and deal wuth tuples (n, r, m) such that there exist a tree T with $n = size(T)$, $r = sumR(T)$, $m = sum(T)$.

It turns out that there are not many such tuples: for example, only 391 for $n = 10$ and 32103 for $n = 20$. So, you can build such tuples in a straightforward manner, pairing whatever is possible.

Problem Tutorial: "Create the Polygon"

Note that s is either integer or semi-integer (that is, $x/2$ for an integer x).

- $If s > nm$, the answer does not exist. Further we show how to build the answer for any $s \le nm$.
- First, let m be as small as possible: that is, $nm > s$ and $n(m-1) < s$. It follows that $nm - s < n$.
- Now, if $m = 1$, the answer is obvious.
- For $m > 1$, take the points $(0, 0)$, $(0, m-2)$, $(nm-s, m)$, (n, m), $(n, 0)$.
- If s is not an integer, cut off the small triangle in the bottom-right corner.

Problem Tutorial: "Dodge The Bullets"

We can easily solve the problem in $O(nm + k)$ time if we consider the process backwards. Initially, mark all positions from 1 to n as safe, also move bullets m cells down (probably making some y's negative). Now process m rounds backwards.

In each round:

- Move all bullets up one cell, mark all covered positions of the spaceship as unsafe.
- For each currently safe position, make adjacent positions safe too (if they don't contain a bullet right now).

After m rounds, safe positions are exactly the starting positions where the player can win the game.

This is a bit too much operations to do within time limit (approx 10^9). However, bit operations can speed this up by a factor of bit word length w (32 or 64). Let us store the set of safe positions in a `std::bitset` s. Now we have to perform operations of several types:

- Make all positions that contain bullets unsafe. To do that, maintain the set of bullet-covered positions in a `std::bitset` b. Now we simply do `s = s & b`.
- Move all bullets up one cell. For each bullet remember when is enters the "danger strip" between first and H-th row, modify b accordingly.
- Propagate safe positions to both sides. `s = s | s << 1 | s >> 1`.

Problem Tutorial: "Encoded Permutations"

Let's calculate the array $dp[k][e]$ — the number of "source" permutations p with a property that exactly e of the values $p[1]$, $p[2]$, ..., $p[k]$ are greater than k (and at the same time exactly e of the values outside $p[1], \ldots, p[k]$ are not greater than k).

If $s[k] =$'-', then at the places $[1, k]$ are e values that are greater than k, and we know that $p[k] < k$, so it is not included in e, and all e "big" values are placed at $[1, k-1]$.

If one of values $p[1], \ldots, p[k-1]$ is equal to k, then $e+1$ of the values $p[1], \ldots, p[k-1]$ are greater than $k-1$. Number of those values is $dp[k-1][e+1]$, and we have $e+1$ way to choose the i such as $p[i] = k$ and $e+1$ way to choose the value of $p[k]$ (since $k-1-(e+1)$ values are used inside the $[1, k-1]$), so we have $dp[k-1][e+1] \cdot (e+1)^2$ ways to do that.

So if $s[k] =$'-', $dp[k][e] = dp[k-1][e] \cdot e + dp[k-1][e+1] \cdot (e+1)^2$.

Similarly we can recalculate dp for $s[k] =$'+': $dp[k][e] = dp[k-1][e-1] + dp[k-1][e] \cdot e$.

The answer can be found in $dp[n][0]$, the time complexity is $O(n^2)$.

Problem Tutorial: "Font Processor"

To effectively solve the problem, we will build an additional data structure using the following method.

Let's add to the the set of nodes of the splines the points of their inflection in the direction of X. To find the inflection point of a cubic function, it is enought to equate its derivatives to zero. In other words, for this we need to find the roots of the quadratic equation:

$$3 \cdot AX_i \cdot t^2 + 2 \cdot BX_i \cdot t + CX_i = 0$$

.

We choose from them those that fall into the corresponding intervals $[0, 1]$. Thus, we define a set of parametric intervals on which the functions $X_i(t)$ retain their monotonicity. Let's subdivide the available

splines according to the corresponding intervals. Let's calculate the values of $X(t)$ on the boundaries of the specified intervals, and sort them along the x axis.

We will correspond the obtained values to the splines to whose nodes they refer. At the same time, we will arrange them in ascending order, marking them as *left* and *right*.

Drawing vertical lines through them, we obtain a partition of the plane by a set of strips.

Next, for each such strip, we define a set of splines intersecting it using the *sweeping line*.

Let's start the sequential traversal from the leftmost vertical line. In the process of traversal, we will add to the list the splines whose left boundaries lie on the next straight line, and exclude those with — right.

We select from the list all the splines contained in it and sort them along the y-axis so that they were located strictly one above the other.

To do this, it is enough to determine the Y coordinates of the points of intersection of each such spline with the line passing through the middle of the current strip.

By choosing, for example, the bottommost spline from the leftmost strip, we can determine the orientation of the path by simply looking at direction of change of the parameter from its left to its right point. If the parameter increases, then the direction is left-handed, if it decreases, then it is right-handed.

As a result, we get a search structure on the basis of which it will be possible to process the queries. For the each next entry point using a binary search, we determine the band in which it falls. If it does not fall into any of the bands, then it lies obviously outside the contour. Further, in a similar way, inside the current strip, we determine the spline closest to it.

To do this, we can compare the Y coordinate of the point and the point of intersection of the spline with the line passing through it. Then it is enough to compare the direction of the corresponding spline with the orientation of the contour.

Problem Tutorial: "Guess The Bit String"

Note that the initial integer is positive, i.e. the string S contains at least one 1.

We will consider two cases — when N is even and when N is odd.

Consider the binary strings of length 2. If we know that some string is 01 or 10, the comparsion with that string uniquely determines any other binary string of length 2:

```
    01  10
00  <<  <<
01  ==  <>
10  ><  ==
11  >>  >>
```

So as long as we know the placement for one of those pairs (we will consider it the *key*) and have the unknown continuous string of even length q, we can use $q/2$ queries to reach the goal.

In the first query we are asking about two characters at positions 0 and 1 (i.e `? 1 0 1`). If they are different, we have 01 for "<<" and 10 for ">>", i.e. the key is placed on position 0, and totally we have $1 + (N-2)/2 = N/2$ queries, that is OK with the query limit.

If they are equal ("=="), there are 00 or 11, so we continue comparing pairs of characters at positions $2i$ and $2i+2$ for $i = 0...(n-1)/2$. If all of those pairs are equal, we have the number consisting of 1's (because string contains at least one 1). Let the first i where they differ is k. Then all digits between 0'th and $2k+1$-th, inclusively, are equal (in first pair digits are equal, and each next pair is equal to previous one), and they differ with pair starting with $2k+2$. Note that if we compare 00 with any different string, we will get "<<", and for 11 it will be ">>", so at the last comparsion we know first $2k+2$ digits. Then we will compare the one-digit strings at positions $2k+2$ and $2k+3$. If they are not equal, we can determine digits precisely (as it was done for the first two characters). If they are equal, they differ from first $2k+2$ digits, so we know them precisely. In both cases we have a key (in digits $2k+2$ and $2k+3$ in the first case, and

in digits $2k+1$ and $2k+2$ in the second), so we used $k+3$ queries to know first $2k+4$ digits, and with the key we can use $(N-2k-4)/2$ queries to know the rest, so totally it caused $k+3+N/2-k-2 = N/2+1$ query, that is the same as $\lceil (N+1)/2 \rceil$ for the even N, so that is OK with the query limit.

If N is odd, we will ask the query `? 2 0 1` first.

```
000 ==
001 <<
010 <>
011 <<
100 >>
101 ><
110 >>
111 ==
```

I.e. the "==" answer means that all digits in the triple are equal, and same signs define the first and the third digit, keeping the central one unknown. Two other answers let recognize the three first bits uniquely.

For $N = 3$, the "==" answer means that all digits are equal to one (because 000 is prohibited). So we have to distinguish `011` from `001` and `100` from `110`. It is posslble comparing the first 2 bits, so we use 2 queries for $N = 3$, that fits the restrictions.

For $N \geq 5$, the second query will be `? 2 2 3`, i.e similar question, but about triple on bits 2-4. If we have "==" in both queries, then all 5 first bits are equal, and we were used 2 queries. If $N = 5$, then answer is all ones (because 00000 is prohibited), if $N = 2k+3 > 5$, then consider the suffix of the string starting on bit 2. It have even length $2k$, we know that two first digits are the same, i.e. it is equivalent to task for $2k$ with one query asked (and the answer that first 2 digits are equal), that is solvable for $2k/2+1$ queries in total and for $2k/2+1-1 = k$ additional queries if we will take into account the first query that is already asked. So total number of operations for $N = 2k+3$ is $k+2$, that is equal to $\lceil (N+1)/2 \rceil = \lceil (2k+3+1)/2 \rceil$ and fits the restrictions.

If we have the different answers in both queries, then note that any non-equal answer uniquely determines last and first digit, i.e. in this case ambiguity of "==" is resolved. So only one ambiguity we can have is with the equal signs. If there is only one such case, we can use third query in a way similar to case with $N = 3$, if there are two cases (and it is possible in cases like `1?0?1` or `0?1?0`, we can use `? 2 1 2` for the third query. Note that with the known middle digit the values of other 2 are determined uniquely (look at the answer table before). So we used 3 queries to know the first 5 bits, and they are not all equal, i.e we know the key and can find other $N-5$ bits for $(N-5)/2$ queries, i.e. total number of queries is equal to $(N-5)/2+3 = (N+1)/2$, that fits the restrictions.

Problem Tutorial: "HR Codes"

This task can be solved using dynamic programming. Consider dp_i as the minimum recognition error for the first i columns of the given image. Then $dp_i = min(dp_{i-3} + error(i-3, 1), dp_{i-4} + error(i-4, 0))$, where the function $error(x, y)$ is the recognition error on the bounding box of the character y with the empty column before (i.e. 4×3 for $y = 0$ and 3×3 for $y = 1$), starting in column x.

Values for $i = 0..4$ shall be calculated manually. Obviously, $dp_0 = 0$; there is exactly one picture of length 2 (standalone 1) and exactly one picture of length 3 (standalone 0), so dp_2 and dp_3 can be calculated directly comparing the shape with the shapes for 1 and 0, respectively. For dp_1 and dp_4 there are no correct codes of the length i, so the set these values to "infinity" (the big positive integer).

Note that for any $n \geq 5$ the correct code of length n exists: for $n = 5$ it is '11', for $n = 6$ it is '10', for $n = 7$ it is '00', for $n = 8$ it is '111' etc, i.e to obtain the string with code of length $n+1$ we can replace 1 in representation of the string of length n with 0, or if there are no 1's, replace two last zeroes with three ones).

Problem Tutorial: "Integer Function"

Consider integer $f(x, a)$ for some x.

We have two cases. If $v = ab + ax$ is divisible by p, then $f(x, a) = 1$. If v is not divisible by p, then $f(x, a) = (ax + ab) \cdot inv_p(ab + ax) = 1$. Then when $k > a$ the equality $f(f(\dots f(f(k, k-1), k-2), \dots, 2), 1) = f(f(\dots f(f(1, a-1), a-2), \dots, 2), 1)$ is held. $f(x, y)$ can be calculated with time complexity $\log p$ using the known algorithms of modular inverse, so total time complexity is $O(min(a, k) \log p)$.

Problem Tutorial: "Jogging"

Note that the states of all runners change in a cycle with a period of $2l$. We need to determine whether there exists a moment in time when for any i a runner with number $p[i]$ is placed to the left of the runner $p[i+1]$. For a fixed i, let's find all times in the interval $[0, 2l]$ when the $p[i]$-th runner is placed to the left of the $p[i+1]$-th.

This is some *cyclic segment*, (i.e. either an ordinary segment or two segments such that the beginning of one coincides with 0, and the end of the second coincides with $2l$), since during the time $2l$ the runners are in the same point exactly twice. It remains only to determine whether there exists a point covered by all cyclic segments.

To do that, we will replace the cyclic segments of the second type by two ordinary segments. Now we need to check if there exists a point covered n times. Sort all the ends of the segments, and run through them in ascending order of the coordinate. We will increment the counter when we met the beginning of the segment, and decrement when we met the end. And all that we need is to detemine whether there is a moment when the counter is equal to n.

Time complexity: $O(n \log n)$.

Problem Tutorial: "King And Tree"

After the dissection of the initial tree there shall be several trees, such as each tree contains the same number of the apples. It is obvious that this number is the divisor of the total number of the apples, so we can for each divisor d count the number of ways to split the tree to the small trees such each small tree contains exactly d apples and then answer is the sum of all those numbers.

Let's solve the task for the fixed d. To do that, we will count the dp_h — number of ways to split the tree below, if we will start the new subtree at height h.

Denote the number of the subtrees starting at the height h as cnt, and sum of the apples in those subtrees as sum. Then at the height where the next subtree is starting, the sum of the apples in the subtrees will be equal to $nextSum = sum - cnt \cdot d$. Then dp_h is equal to the sum of dp_k, where at height k in the subtrees will be $nextSum$ apples, or zero, if the beginning of the next subtree cuts from the top the connected parts with the number of apples, not equal to d.

So all that we have to do is to check the correctness of the beginning of the new small tree. For that it is enough to test that each of cnt subtrees number of apples is divisible by d and that each of the connected parts contains at least one vertex. If all those conditions are held, then dp_h is the sum of dp_k, otherwise dp_h is 0. Then the situation when we have the connected part with number of the apples not equal to d will be processed automatically to $dp_h = 0$. To check those conditions efficiently, we can precalculate the sizes of the subtrees and position of the closest apple for the root of each subtree.

Then, for each divisor of k we can solve the task for the linear time, that is fast enough to solve this task.

Problem Tutorial: "List of Palindromes"

One of possible ways is to go through all possible combinations of a given set of characters in lexicographic order, checking each of them in turn for palindromicity. However, this will take us much more time than we can afford.

On the other hand, we can iterate over all possible halves of palindromes, completing them as we go, thereby avoiding options for iterating over strings that are obviously not palindromes.

However, the order of palindromes may differ from the order of their halves. For example, the `cba` and `cbaa` halves give us palindromes: `cbaabc` and `cbaaaabc`, respectively.

For the similar reason, it will be difficult for us to determine when to stop, and also understand which of the palindromes after sorting the resulting list will go beyond the given n.

The solution is based on a complete lexicographic enumeration of all possible combinations from a given set of characters, but it uses several heuristics to cut off deliberately false solutions, i.e it combines the ideas of both the above approaches.

We will build a recursive function that takes one argument i as the input, that is the current position in the output string. In this function we will alternately substitute each of the characters from the given alphabet and run recursively for $i + 1$. Thus, each time it is called, we will already have collected some string P of length i, at the end of which new characters will be added.

When choosing the characters, we should take into account our position relative to the string T. Thus, if the string P is a prefix of the string T, then the enumeration must also begin with the symbol $T[i]$. If this is not the case, or the length of the string T is less than the current i, then we can start the search from the smallest character available to us. To avoid unnecessary comparisons with the string T, we supplement our recursive function with one more Boolean argument, which will be equal to 1 if the string P collected earlier matches the corresponding string prefix T, and will be equal to 0 in all other cases.

Next, choosing the next character c and substituting it at the end of the string P, we can try to form a pair of new palindromes: *odd*: $P + c + P'$ and *even*: $P + c + c + P'$. At the same moment, it is also wise to check the number of characters available to us by comparing it with the number of already used.

Let $N[c]$ indicate the initial number of characters c, $M[c]$ – the number of characters c encountered in the string P. Thus, to obtain an odd palindrome, it is enough to check the condition $(2 \cdot M[c] \leq N[c] + 1)$, and for an even palindrome — the condition $(2 \cdot M[c] \leq N[c])$.

However, this does not take into account all other characters previously encountered in the string P. But here we can use the following fact.

If some prefix P cannot be extended to an even palindrome, then the process of forming new palindromes on it will be completed, i.e.no subsequent prefix having P as a sub-prefix, cannot be extended to a palindrome.

To prove this fact we can note that if there really was a palindrome, whose half has the form $P + Q$, then among its suffixes P' would be sure to be found. And then we could get $P + P'$, and we get the contradiction.

Thus, in order to skip the stage of creating new palindromes in subsequent runs of the recursive function, we will pass one more boolean argument to it, which tells whether prefix P is half an even palindrome or not.

The palindromes created in this way are not printed immediately, but are stored in a special data structure.

As such a structure, we will use a two-dimensional array $D[i][c]$ – containing lists of palindromes, whose prefixes match the previously received string P, and the i-th position is the symbol c. In this case, the palindromes themselves are written as indices of their positions opposite to i.

Before the very first run of the recursive function, $D[0][c].push(0)$ is initialized for all valid $c \geq T[0]$. When creating an odd palindrome $P + c + P'$, a new entry is created in the array $D[i + 1][P[i - 1]].push(i - 1)$, and when creating an even $P + c + c + P'$: $D[i + 1][c].push(i)$.

Now we can say that when substituting the next character c at the end of the string P, we should traverse all l contained in the list $D[i][c]$. For each such $l > 0$, we extend the corresponding palindrome by creating a new entry $D[i + 1][P[l - 1]].push(l - 1)$.

If there is $l == 0$, it will mean that the previously created palindrome ends in this position, then print the resulting string $\{P + c\}$.

If at some point we find that there is no palindromes with the prefix P or the number of all given palindromes exceeded n, the procedure ends.

Problem Tutorial: "Matrix"

Let's first say that $A[i]$ is built as the array of the maximums in the i'th row, and $B[i]$ as array of the maximums in the ith column, and show that if there exists at least one pair of the parent arrays, then arrays A and B should be the parent ones.

Look at the i-th row of the matrix $X[][]$. Its element $X[i][j]$ is equal to the minimum of $A[i]$ and $B[j]$. Thus, if the next $B[j]$ is less than $A[i]$, then $X[i][j] = B[j]$, otherwise it is equal to $A[i]$. This means that either the largest of the elements of the row is equal to $A[i]$, or all elements of the row are equal to $B[j]$. But then we can say that $A[i]$ is equal to the largest of them, also without violating anything. Absolutely similar reasoning can be done for each column to justify the construction of the array $B[j]$.

Having constructed the arrays $A[]$ and $B[]$ in this way, we need to check that the matrix $X[][]$ is correct and corresponds to them. If this is not the case, you should immediately print 0 as the answer to the problem.

Now, in order to count the number of valid pairs of parent arrays, let's see which elements in them can be changed. If for some i and j $A[i] \neq B[j]$, then the lesser of $A[i]$ and $B[j]$ cannot change, because then the value of $X[i][j]$ will change, which should not be. This means that we can only change elements of $A[]$ that are greater than all elements of $B[]$, and elements of $B[]$ that are greater than all elements of $A[]$. But by the construction of arrays $A[]$ and $B[]$, these elements are the same in both arrays and equal to the largest value in the matrix $X[][]$.

Let us now find the value of the maximum in the matrix $X[][]$ and the number of such maxima in the arrays $A[]$ and $B[]$. We will denote these values as $maxval$, $k1$, $k2$, respectively. Note that we cannot simultaneously change the values in the $A[]$ and $B[]$ arrays, because if we change the values of $A[i]$ and $B[j]$, then the value of $X[i][j]$ will become invalid.

Summarizing all of the above, we get that we can only change the largest values, and only in one of the arrays at a time. Moreover, this variable maximum value can be make it arbitrarily large, but not more than the number C from the problem statement. This means that the final answer can be calculated by the formula: $(C - maxval + 1)^{k1} + (C - maxval + 1)^{k2} - 1$: we choose one of the arrays, in which each of the $k1$ (or $k2$ in the case of the $B[]$ array) maximum elements can take any value, greater than or equal to its current value. 1 must be subtracted, since the option in which we do not change anything will be taken into account in both terms, that is, twice.