Problem A. Friendly Rooks

Author: Mikhail Anoprenko, developer: Grigory Shovkoplias

Note that the largest number of rooks that can be placed on the $n \times m$ chess board is $\min(n, m)$. Accordingly, if $\min(n, m) < k$, then there is no answer.

Otherwise, we can place all the rooks on the same diagonal. For example, along the main diagonal. Finally you need to accurately print the answer.

Problem B. Guess the Array

Author: Daniil Oreshnikov, developers: Daniil Oreshnikov, Grigoriy Khlytin

The first query we make is "1 n". The answer to that query will be the sum of the entire array $s_{1,n} = S$ with some blocked segment.

The goal of all the following queries will be to determine the sums of elements on each prefix of the array. To do this we will maintain the following invariant: at each moment of time for each yet unknown prefix $[1, i]$ at least one of the segments $[1, i]$ and $[i + 1, n]$ is not blocked.

To achieve this, when the interactor gives us a new blocked segment $[l, r]$ we consider three cases:

- If $l = 1$ then the next query will be "$r + 1$, n". We get the sum on that suffix $s_{r+1,n}$ and can calculate the sum on the blocked segment $[1, r]$ as $S - s_{r+1,n}$ where S is the sum on the whole array;
- If $r = n$ then the next query will be "1, $l - 1$" which will give us the sum on the prefix $[1, l - 1]$;
- If any other segment is blocked we will simply query the sum on the segment $[1, i]$ for any yet unused i. To choose such i we can maintain a queue of all numbers between 1 and $n - 1$ and a set of such i for which we already know this sum.

With this at any given time, for all i (possibly except one) for which the sum of $s_{1,i}$ is not yet known, both segments $[1, i]$ and $[i + 1, n]$ are not blocked. For the single i with one of the said segments blocked, the next query will be asking for a sum on the opposite segment, so the invariant described above holds.

Once we know the sums on all prefixes of the array in n actions, the i-th element can be calculated as $s_{1,i} - s_{1,i-1}$. All elements of the array can be calculated in such a way, and printed out after "**!**".

Problem C. Moving Cells

Author: Daniil Oreshnikov, developer: Grigoriy Khlytin

Let's use dynamic programming to solve this problem.

We will calculate a two-dimensional array $dp_{i,j}$ — the minimum number of actions to take with the first i columns to make *beautiful* part of the picture on these columns, and so that the consecutive segment of black cells in the i-th column starts in the j-th row.

The recalculation will be by the formula:

$$dp_{i,j} = |s_i - j| + \min_{k=j+s_{i-1}-t_{i-1}}^{j+t_i-s_i} dp_{i-1,k}$$

where s_i and t_i — numbers of the starting and ending positions of the consecutive segment of black cells in the i-th column of the picture.

In a naive implementation of this formula, we can go through all appropriate k for each j and compute the minimum among $dp_{i-1,k}$ with complexity $\mathcal{O}(n)$, then we get the asymptotic $\mathcal{O}(m \cdot n^2)$.

However, when j changes from j to $j+1$, among the elements of interest to us $dp_{i-1,k}$, no more than two elements can change, since the appropriate segment of k has shifted by one. So, we can keep the minimum in the set, for example with `std::set` in `C++`, then the asymptotics of such solution will be $\mathcal{O}(m \cdot n \cdot \log n)$, or with the data structure `queue with minimum`, then the asymptotic will be $\mathcal{O}(m \cdot n)$.

Problem D. Army of Clones

Author: Daniil Oreshnikov, developer: Saveliy Grigoriev

Let d_i — the maximum number of clones that can reach the room i. Define a weight of the edge $u \to v$ in the following way: if $d_u > a_v$, then the edge $u \to v$ has a weight 0, otherwise 1. Note that the original undirected edge $u - v$ trasforms into two directed edges $u \to v$ and $v \to u$.

The length of the shortest path from the room 1 to the room n in the given graph equals the minimum number of times when the army of clones loses the half of the group, that is when the army moves along this path, the number of survivor clones is the maximum.

Let's use the algorithm `0-1 bfs`. However we can determine the weight of an edge only if we calculate the shortest path to the outgoing end of the edge. `0-1 bfs` considers only such edges, so our algorithm will work correctly.

The total complexity — $\mathcal{O}(n)$.

Problem E. Haiku

Author: Artem Vasiliev, developer: Arsenii Kirillov

Let's check for each word whether there is a haiku starting with that word. To do that let's iterate over the words starting with this one, and count the number of phonetic units we see until we have seen at least 5 units. If we have seen strictly more that 5 units, there is no haiku here. Otherwise, the first haiku line is fine, and we use the same procedure to find the second line with exactly 7 units, and the last line with exactly 5 units. Since each word has at least one phonetic unit in it, for each starting position we will iterate over at most 17 words.

Problem F. Race

Author: Daniil Oreshnikov, developer: Mikhail Anoprenko

It is clear from the problem statement that at the moment of time t participant with number i will be located at the point $(s_i + t \cdot v_i, i)$.

Consider a moment of time when maximum number of sportsmen are on the same line. Let's fix one of participants in that line at that moment, denote its number by i. We will consider all possible values of i from 1 to n. From now on, we will only take into consideration lines passing through the point where i-th participant is located at particular moment of time.

For convenience when considering i-th participant we will locate coordinate center of the plane at the point where i-th participant is located (coordinate center will be moving together with this sportsman). Note that according to new coordinates at the moment of time t participant number j will be at the point $(s_j - s_i + t \cdot (v_j - v_i), j - i)$. Let's introduce a couple of new variables for convenience: $j' = j - i$, $s'_j = s_j - s_i$, $v'_j = v_j - v_i$. Thus at the moment of time t participant number j will be at the point $(s'_j + t \cdot v'_j, j')$. Now we are only interested in lines passing through the point $(0, 0)$.

Consider a moment of time t when participants with numbers j and k are located on the same line passing through $(0, 0)$. This condition is equivalent to the fact that cross product of vectors from $(0, 0)$ to positions of these participants equals to zero. Lets write down this condition using the formula for cross product: $(s'_j + tv'_j)k' - (s'_k + tv'_k)j' = 0$. Slight transformation of these equation: $s'_j k' - s'_k j' = t(v'_k j' - v'_j k')$. Not that if $v'_k j' - v'_j k' \neq 0$ then there is exactly one value of t satisfying the condition. In case when $v'_k j' - v'_j k' = 0$,

let's take a look at the value of $s'_j k' - s'_k j'$. If it is also equal to 0 than condition is satisfied for any value of t, otherwise it is never satisfied. Let's iterate through all possible pairs of $(j, k) \neq i$ and for each pair find a set of moments of time when the condition for this pair is satisfied.

For each value of j let's count the number of values of k when the condition is satisfied for any moment of time, and for each value of k if for pair (j, k) condition is satisfied for exactly one value of t, let's save this value. Now let's count the maximum number of same values of t among saved for particular value of j, add the number of participants who satisfy the condition always, and we will obtain the candidate for the answer. To get the real answer, we have to choose the maximum candidate for all values of i and j.

Time complexity of this solution if $O(n^3 \log n)$ when using a sort to find same values of t. When using a hash table time complexity may be reduced to $O(n^3)$. It's also worth noting that in order to avoid problems with real numbers precision solutions of this task should use rational fractions to store non-integer numbers.

Problem G. Maximaze XOR sum

Author: Daniil Oreshnikov, developer: Arsenii Kirillov

Let's note that when we swap the elements A_i and B_i, the values $X(A)$ and $X(B)$ are replaced by the new values $X(A) \oplus A_i \oplus B_i$ and $X(B) \oplus A_i \oplus B_i$. Let's denote the value $A_i \oplus B_i$ as C_i.

If a certain bit in some position in $X(A)$ and $X(B)$ is different, then it will stay different after any sequence of operations, an will be counted exactly once in the total sum. Otherwise — if this bit is equal in $X(A)$ and $X(B)$, it will always stay equal.

Therefore, we need to select some values C_i in such a way that in the positions where $X(A)$ and $X(B)$ both have zeroes, the xor-sum of selected C_i would have a one, and vice versa: where $X(A)$ and $X(B)$ both have ones, the xor-sum of selected C_i would have a zero. If it is impossible to satisfy all such desired conditions, then we should satisfy the ones that correspond to the higher (leftmost) bits first, because the corresponding power of two are higher that the sum of all the lower powers of two, thus the impact of the higher bit is more important.

In order to do that, consider C_i as binary rows, and run Gaussian elimination algorithm. It will show which bit positions are possible to satisfy, and how to do that from higher ones to lower ones. Since the input numbers are at most 10^{18}, the binary representations are no more than 60 bits, so there will be no more than 60 iterations of Gaussian elimination.

Problem H. Octopus Game

Easy to see that it is optimal to alternate the types of operations, because two consecutive operations of the same type can be easily combined.

Let's use Euclidean algorithm to find GCD of a and b. After each move we would like to receive a new number that correspond to the next step of Euclidean algorithm. Since in the end one of the numbers becomes zero in Euclidean algorithm, we would find the desired sequence k_i.

However, the problem is the length of such sequence. It it well known that the worst cast for Euclidean algorithm is the pair of two consecutive Fibonacci numbers: the algorithm will always move to the previous consecutive pair. So in the given constrains our approach will lead to the solution in 85 steps.

Let's use the "Least Absolute Remainder" technique: let us allow the existence of negative remainders in the formulation of Euclidean algorithm, so now on each step there are two possible remainders to choose from ($a\%b$ and $a\%b - b$), and the technique proposes to always select the one with the smaller absolute value. Note that such action is correct in terms of the game with cards: we simply can take $k + 1$ instead of k that we would use in the usual formulation of Euclidean algorithm.

It can be shown that the number of steps doesn't exceed $\log_{1+\sqrt{2}} \max(a, b)$.

Problem I. Third Group Exam

Author and developer: Daniil Oreshnikov

Let's see what happens if Ilya chooses to be graded in all blocks by answering theoretical questions. If there were no limit on the number of blocks with a practical problem solved then his score would be $\sum_{i=1}^{n} x_i$.

Note that in some blocks there are $x_i > y_i$ and in some blocks x_i is less than y_i. When theory is replaced by practice, in blocks of the first type the total grade decreases, and in blocks of the second type increases. For blocks in which $x_i = y_i$, the total score is not affected by whether the theory or practice was chosen.

Let's return to the situation where Ilya chooses to pass all blocks by answering theory questions. He needs to choose practice instead of theory in some (at least b, but no more than $n - a$) blocks in a way that maximizes his score. When the theory is replaced with practice in the i-th block, the sum changes by $y_i - x_i$. If we denote by t the number of blocks with $y_i > x_i$, then the correct solution is to choose practice in $\max(b, \min(n - a, t))$ blocks with **maximal** values of $y_i - x_i$.

Indeed, if $t > n - a$, then the maximal sum can be obtained when $n - a$ blocks with maximal $y_i - x_i$ are graded as y_i, and the remaining a blocks as x_i, since there must be at least a blocks of theory. If $b \le t \le n - a$ the in the end Ilya can pass all the blocks in which x_i is greater by theory and all the blocks in which y_i is greater by practice, achieving the maximal possible $C = \sum_{i=1}^{n} \max(x_i, y_i)$. And if $t < b$ there is no point in solving more than b practical problems instead of theory, since this will reduce the score.

So one of the optimal answers is achieved as follows: first, Ilya sorts all blocks by value $y_i - x_i$, then he chooses to be graded by practice in $\max(b, \min(n - a, t))$ last blocks and by theory in the rest of them. Time complexity is $\mathcal{O}(n \log n)$ because of sorting (although it can be done in linear time with ordinal statistics calculation).

Problem J. Computational ethnography

Author: Nikolay Vedernikov, developer: Mikhail Anoprenko

Let s be an interesting number. It is a square of some integer number r. In other words, $r \le \sqrt{B}$ is an integer.

Let's iterate over all possible values of r in the interval from 1 to $\lfloor \sqrt{B} \rfloor$. For each value $s = r^2$ let's check whether this number s is interesting. In order to do this we should check that the reversed decimal notation of s is a square number. To do this, we get all the digits of s, concatenate them in the reversed order, and check whether the square root of the received number is an integer.

The answer to the problem is the number of candidates that passed our check and that fit into the interval from A to B. The working time of the solution is $O(\sqrt{B} \log B)$, because for each of $\sqrt{B}$ candidate value s we iterate over all the digits of s.

Problem K. Work or Sleep!

Problem author: Sergey Kopeliovich, developer: Grigory Shovkoplias

To solve this problem, it is necessary to derive a formula for the dependence of perfomance on sleep time. For example, you can apply the formula to the equation of a straight line at two points to get that

$$f(t_{sleep}) = \begin{cases} \frac{X \cdot t_{sleep}}{\frac{T}{6}} & 0 \le t_{sleep} < \frac{T}{6} \\ 100 - \frac{(100-X) \cdot (\frac{T}{3} - t_{sleep})}{\frac{T}{6}} & \frac{T}{6} \le t_{sleep} \le \frac{T}{3} \\ 100 & t_{sleep} > \frac{T}{3} \end{cases}$$

From this we obtain the formula for the dependence of the amount of work on the sleep time:

$$Work(t_{sleep}) = \begin{cases} (T - t_{sleep}) \cdot \frac{X \cdot t_{sleep}}{\frac{T}{6}} & 0 \le t_{sleep} < \frac{T}{6} \\ (T - t_{sleep}) \cdot (100 - \frac{(100-X) \cdot (\frac{T}{3} - t_{sleep})}{\frac{T}{6}}) & \frac{T}{6} \le t_{sleep} \le \frac{T}{3} \\ (T - t_{sleep}) \cdot 100 & t_{sleep} > \frac{T}{3} \end{cases}$$

Then the problem is reduced to finding the maximum of this function.

1. `Method 1: for lovers of mathematics`. It is possible to find the points at which the derivatives are equal to zero (the vertex of the parabola) in each part of this function, and obtain that the maximum is attained at the point $t_{sleep} = \frac{T}{6}$ in the first part of this function, at the point $t_{sleep} = \frac{T}{3}$ in the third part of this function, as expected. And the maximum is reached at the point $t_{sleep} = \frac{T \cdot (2(100-X)-25)}{3(100-X)}$ in the second part of this function. Then it remains to substitute the values and takes the maximum, or notice that the maximum will always be in the second part (possibly at the border).

2. `Method 2: for lovers of algorithms`. For those participants who do not really want to deduce many formulas, there is a wonderful algorithm ternary search that can be run on each part of the function and take the maximum of the results.

3. `Method 3: for universal lovers`. You can combine ideas from the above methods. For example, run ternary search only in the second part of the function, because you can guess the fact that the maximum is in the second part of the function. So there is no use in deriving complex formulas.

Problem L. Permutation Transformation

Author: Nikolay Vedernikov, developer: Sergey Khargelia

If $n = k$, then p can be transformed to the q only in the case of $p = q$.

If $n = k + 1$, then p can be transformed to the q only if q is a cyclic shift of p. In this case, q can be obtained by sequential application of k-transfers with parameters $a = 1$, $b = 2$. Obviously, at most $n - 1$ k-transfer will be required.

If $n \ge k+2$ and k is odd, then p always can be transformed to the q. Let's learn how to swap two adjacent elements of the permutation. If $k = 1$, it can be done by applying one k-transfer. So let's assume that $k \ge 2$. Note that it's enough to learn how to swap the first two elements of the permutation and how to move any two adjacent elements to the beginning of the permutation, because then we can move the elements to the beginning, swap them, and then return them back (it can be done by swapping parameters a and b for the operations of transfer to the beginning and applying this operations in reverse order).

The first two elements of the permutation can be swapped by applying $\lfloor \frac{k+1}{2} \rfloor$ k-transfers with parameters $a = 1$, $b = 3$ and then a k-transfer with parameters $a = 2$, $b = 3$. For example, if $n = 6$, $k = 3$ and the permutation is equals to $1, 2, 3, 4, 5, 6$, it will change as follows: $1, 2, 3, 4, 5, 6 \to 4, 5, 1, 2, 3, 6 \to 2, 3, 4, 5, 1, 6 \to 2, 1, 3, 4, 5, 6$.

Any two adjacent elements can be moved to the beginning as follows: let's denote the index of the first of them by *pos*. if $pos + k - 1 \le n$, then after a k-transfer with parameters $a = pos$, $b = 1$, this two elements will be moved to the beginning. Otherwise, let's apply a k-transfer with parameters $a = n - k + 1$, $b = 1$. Then *pos* will decrease by at least $n - k \ge 2$, and the considered elements will remain adjacent. Note that this algorithm requires at most $\lfloor \frac{n-1}{2} \rfloor + 1$ operations.

We learned how to swap two adjacent elements, so now we can easily transform p to the q: let's find the first element of q in the p and move it to the first position, then find the second element of q in the p and move it to the second position, and so on. In total, the solution uses at most $(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$

swaps of adjacent elements, so at most $\frac{n(n-1)}{2} \cdot \left(\left\lfloor \frac{k+1}{2} \right\rfloor + 1 + 2\left(\left\lfloor \frac{n-1}{2} \right\rfloor + 1\right)\right) \le \frac{n(n-1)}{2} \cdot \left(\left\lfloor \frac{n}{2} \right\rfloor + n + 2\right) \le n^3$ k-transfers.

Now let's consider the case when $n \ge k+2$ and k is even. Then p can be transformed to the q only if the permutations have the same parity.

Let's prove that if k is even, then k-transfer does not change the parity of the permutation. Note that for a fixed pair of elements k-transfer changes their relative order only if exactly one of these elements is in the subsegment, to which we apply k-transfer. Also, the relative order has changed either with all elements of subsegment, or with none. Therefore, the parity of the number of inversions has changed $k \cdot c$ times, for some $0 \le c \le n-k$. Since k is even, $k \cdot c$ is also even, so k-transfer didn't change the parity of the number of inversions.

If p and q have the same parity, then p always can be transformed to the q. Let's learn how to take three adjacent elements of the permutation and shift them cyclically, leaving the rest elements on their positions. If $k = 2$, it can be done by applying one k-transfer. So let's assume that $k \ge 3$. Note that it's enough to learn how to cyclically shift the first three elements of the permutation and how to move any three adjacent elements to the beginning of the permutation. Moreover, it's easy to make sure that the algorithm of transferring two adjacent elements to the beginning of the permutation considered above is also suitable for the current case.

The first three elements of the permutation can be cyclically shifted by applying a k-transfer with parameters $a = 3$, $b = 1$, and then two k-transfers with parameters $a = 2$, $b = 3$. For example, if $n = 7$, $k = 4$ and and the permutation is equals to $1, 2, 3, 4, 5, 6, 7$, it will change as follows: $1, 2, 3, 4, 5, 6, 7 \to 3, 4, 5, 6, 1, 2, 7 \to 3, 2, 4, 5, 6, 1, 7 \to 3, 1, 2, 4, 5, 6, 7$.

Note that cyclically shifting three elements to the left is the same as cyclically shifting them to the right twice, so we learned cyclically shift three elements in any direction.

Now we should learn how to transform p to the q using these cyclic shifts: let's find the first element of q in the p and denote its index by j. If $j > 2$, we will cyclically shift to the right the elements with indices $j-2, j-1, j$. By applying such operations, we can move the considered element to one of the first two positions. If it turns out that element is in the second position at the end, then we will cyclically shift to the left the elements with indices $1, 2, 3$. Then we will do a similar sequence of operations for the second element of q and so on up to the $(n-2)$-th element. In the end we will get either $q_1, \ldots, q_{n-2}, q_{n-1}, q_n$, or $q_1, \ldots, q_{n-2}, q_n, q_{n-1}$. Note that we couldn't get the second permutation, because its parity differs from the parity of q, and hence from the parity of p. That's why we will get exactly q.

In total, the solution uses at most $\sum\limits_{i=1}^{n} \left(\left\lfloor \frac{n-i}{2} \right\rfloor + 2\right) \le 2n + \frac{1}{2}\sum\limits_{i=1}^{n}(n-i) = \frac{n(n-1)+8n}{4} = \frac{n(n+7)}{4}$ cyclic shifts of three adjacent elements to the right. So, if $k = 2$ then $n \ge 4$ and cyclic shift of three elements is a single k-transfer, therefore the solution uses at most $\frac{n(n+7)}{4} \le n^3$ k-transfers. Otherwise, $k \ge 4$, so $n \ge 6$ and the solution uses at most $\frac{n(n+7)}{4} \cdot \left(3 + 2\left(\left\lfloor \frac{n-1}{2} \right\rfloor + 1\right)\right) \le \frac{n(n+7)}{4} \cdot (n+4) \le n^3$ k-transfers.