BIJELE

For each of the six input numbers, we output the expected number of pieces minus the input number.

CRNE

If Mirko makes H horizontal and V vertical cuts, Slavko's board will fall apart into $(H+1)\cdot(V+1)$ pieces.

The first solution uses two nested for-loops to try all possible values of H and V (such that their sum is not greater than N) and outputs the largest product.

The second solution is to notice that it never makes sense to make less than N cuts, and loops over all values of H, and uses the expression $N-H$ for V.

The third solution doesn't use any loops, instead observing that the product will be the largest if H and V are as close as possible. The actual expressions are H = N div 2, V = (N+1) div 2.

PRVA

We need to find all words in the crossword and choose the lexicographically smallest one. A vertical word starts in a square if that square is on the first row or if the square above it is blocked, and a horizontal word starts if a square is in the first column or if the square to its left is blocked.

To check if a string A is lexicographically less than string B in the programming language C we can use the expression strcmp(A, B) < 0, and in Pascal and C++ (assuming A and B are variables of type `string`) simply A < B.

TURBO

When the number of elements N is small, it's possible to simulate the algorithm swap by swap. The complexity of this algorithm is $O(N^2)$, and it will surely work for N up to 5000, and the problem statement guarantees that in most of the test cases it will be at most 100.

For longer arrays we need an algorithm more efficient than simulation. Suppose in some phase the number X is to be moved to its final position. The number of swaps in that phase is the difference between the current and final positions of X. The final position is known, it is exactly X, but the current position isn't.

The current position of X can be calculated from its starting position, provided sufficient information about numbers moved in previous phases. More precisely, each number moved in an odd phase (when numbers are moved to the left) that was initially to the right of X "jumped over" X during its phase and because of this X moved one place to the right. Similarly, each number moved in an even phase that was initially to the left of X will move X one place to the left.

These two numbers (how much X has moved left and right) can be efficiently calculated (in logarithmic complexity) using two Fenwick trees; one for odd phases and one for even phases. There is also a similar solution which uses only one Fenwick tree. Such a solution is implemented in the official source code.

Whenever a task includes smaller and larger constraints with algorithms of vastly different implementation complexity to solve them, it is a good idea to include both algorithms in the source code, and determine which one to use at runtime, depending on N. That way, if there is an error in the more complex algorithm, we don't lose points on the smaller test cases.

KEMIJA

For smaller inputs (all numbers up to 100) we can try all possible pairs of values for the first two numbers in the ring (which uniquely determines all remaining numbers), calculate all remaining numbers and check if the ring generated by adding neighbours to a number is equal to the input ring.

For larger rings, the solution is much more complex. Label the numbers in the first ring A_1 to A_N, and the second ring B_1 to B_N. We need to find a ring A such that it generates ring B, and that all numbers in it are positive.

First, notice that the sum of ring A must be exactly one third of the sum of ring B.

Also, from the input ring we can determine for each position k the difference $A_{k+3} - A_k = B_{k+2} - B_{k+1}$. This, with the sum of ring B, suffices to solve the task.

When the length of ring N is not divisible by 3, the solution is unique. Suppose the first element of the ring (A_1) equals 1. From the calculated differences we determine A_4, A_7, ..., A_{N-2}. Because N is relatively prime to 3, we will have determined all numbers A_2 to A_N before returning to A_1. We just need to ensure that the sum of ring A is appropriate (one third of the sum of ring B). For this it suffices to add to each element of A the value of the expression [sum(B)/3 − currentsum(A)] / N.

When N is divisible by 3, the solution is not unique and it is less obvious how to ensure that the numbers are positive. This time the differences generate three separate "chains":

- A_1, A_4, ..., A_{N-2};
- A_2, A_5, ..., A_{N-1};
- A_3, A_6, ..., A_N.

We need to determine the numbers A_1, A_2 and A_3 (and increase the numbers in their respective chains so that the differences are right) so that all numbers are positive and that the sum of ring A is correct. It is not hard to show that a ring which satisfies this generates the input ring B.

For each of the chains it is possible to determine the smallest possible value of the first number so that all numbers in the chain are positive. For example, if the differences generate the chain 1, −4, 5, then we need to add at least 5 to all numbers so that all numbers are positive.

We can set A_1 to the smallest value so that the first chain is positive (in the previous example A_1 would be 6) and similarly for A_2 and the second chain. A_3 is uniquely determined by the expression $B_2 - A_1 - A_2$.

PRAVOKUTNI

We need to find an algorithm of complexity better than $O(N^3)$. Here we will describe three such algorithms.

The basic idea of the first one is: choose the point in which the angle will be right, fix one other point in the triangle and quickly calculate how many of the remaining points form a right triangle with the first two points. The algorithm is based on the so-called canonical representation of a line. Each line can be, regardless of which two points it is generated from, transformed into a unique form. For a pair of points we use the usual formulas to calculate three integers A, B and C such that $Ax + By + C = 0$. The equation can be multiplied by a constant, because this doesn't change the line it represents. Now divide A, B and C by their greatest common divisor so that they become relatively prime, and also negate the entire equation if the first non-zero number is negative. With this we can build a function/map f(line), which tells us how many points are on a line. Now it is easy to implement the idea from the start of this paragraph, and using appropriate date structures, the complexity will be good.

The second algorithm, after choosing the first point (where the right angle will be), sorts the remaining points around it by angle. Now we can use two variables (the so-called "sweep" method) to count all right triangles. We move the first variable point by point, and have the second one 90 degrees ahead of it, moving forward, trying to form right triangles with the first variable.

The third algorithm is different from the first two, but also easier to implement. Choose a point P and translate the coordinate plane so that the point P is the origin (more precisely, subtract the coordinates of point P from every point). Now, for each point, first determine which quadrant it is in, and then rotate it by 90 degrees until it is in the first quadrant. After that, sort all points by angle (y divided by x). Two points form a right triangle with point P if they have the same angle and if they were in neighbouring quadrants before rotating. After sorting, for each set of points with the same angle, count how many of them were in each of the four quadrants and multiply the numbers for neighbouring quadrants.

The complexity of all three algorithms is $O(N^2 \cdot \log N)$. The official source code features the last algorithm.