Problem A. Natural Division

Problem author: Vladimir Sukhov, developer: Vladimir Smagliy

We will look for coprime numbers a and b.

Let us denote the number formed by the sequence of digits after the decimal point as c. Then the following equality holds:

$$\frac{c}{10^n} = \frac{a}{b}$$

Thus, to find a and b, it is necessary to find all common divisors of 10^n and c, and then divide the denominator and the numerator by them. Note that the only common divisors can be the numbers 2 and 5.

This can be done with the following code:

```
while(c % 5 == 0 && ten_pow_n % 5 == 0) {
    c /= 5;
    ten_pow_n /= 5;
 }

while(c % 2 == 0 && ten_pow_n % 2 == 0) {
    c /= 2;
    ten_pow_n /= 2;
}
```

It remains to check that the resulting two numbers are less than 10^6.

Alternatively, one can find the greatest common divisor of the numbers c and 10^n and reduce the fraction by it to obtain the numbers a and b.

Problem B. Area of a Triangle

Author and problem developer: Nikolay Vedernikov

The area of a triangle is equal to $\frac{c \cdot h}{2}$. We need to determine when such a triangle does not exist.

Assume that such a triangle exists. We will construct the circumscribed circle around the triangle. Since the angle is right, the hypotenuse will be the diameter. Therefore, for such a triangle to exist, the height must not be greater than the radius.

Thus, if $2 \cdot h > c$, the answer is -1, otherwise it is $\frac{c \cdot h}{2}$.

Problem C. Cheburashka's Number

Author and problem developer: Ekaterina Vedernikova

Note that at step i, Cheburashka takes i oranges, and Gena takes one orange. In total, the number of oranges they take at step i is $i + 1$.

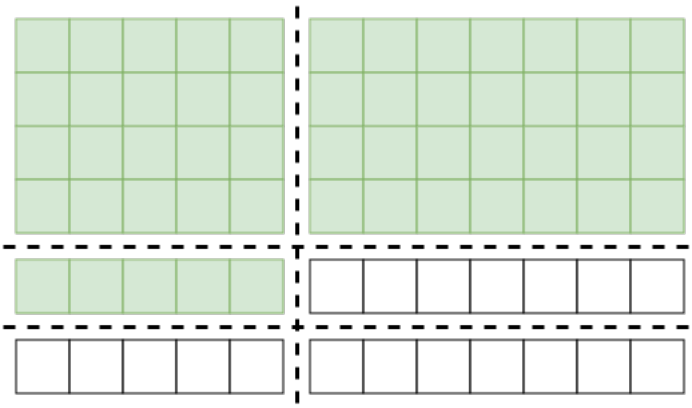
The total number of oranges taken over k steps will be:

$n = (1 + 1) + (1 + 2) + (1 + 3) + \ldots + (1 + k) = \sum_{i=1}^{k} i + k = \frac{k \cdot (k+1)}{2} + k = \frac{k \cdot (k+3)}{2}$

Problem D. Lazy Cuts

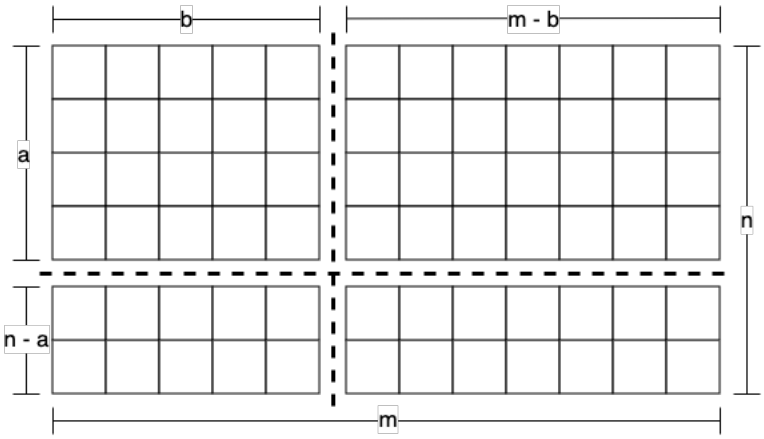
Problem Author: Nikita Golikov, Developer: Grigory Shovkoplyas

Note that to obtain an area s, no more than three cuts are needed. Using one cut, we can cut off the first k rows, and then, using two more cuts, obtain any number of cells from the next row.



Let's analyze the cases when fewer than three cuts are sufficient, namely zero, one, or two. Zero cuts can occur if and only if: $s = n \cdot m$. One cut is enough if s is a multiple of n or s is a multiple of m.

With two cuts, it makes sense to make one cut vertically and the second horizontally. Let these cuts divide the grid after the first a rows and the first b columns ($0 < a < n$ and $0 < b < m$), as shown in the figure below.



There can be three ways to obtain s cells:

1. Take the upper left piece if $s = a \cdot b$.
2. Take everything except the upper left piece if $s = n \cdot m - a \cdot b$.
3. Take the upper left and lower right pieces if $s = a \cdot b + (n - a) \cdot (m - b)$.

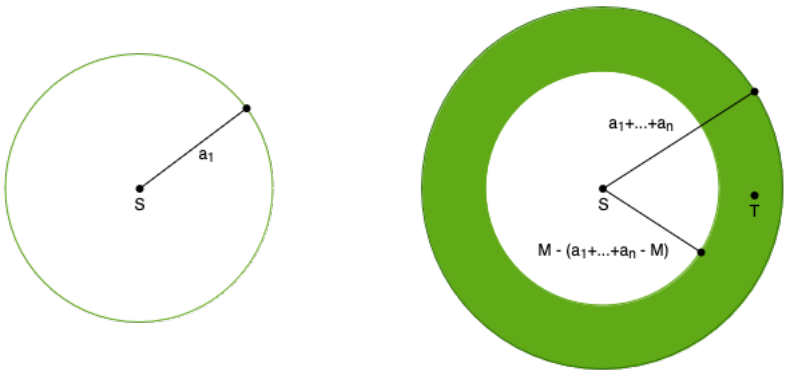
All methods of obtaining can be checked in $O(n)$ by iterating through the first cut and verifying in $O(1)$ the existence of a suitable second cut by solving an equation with one unknown.

Problem E. Decart the Grasshopper

Author of the problem: Demid Kucherenko, developer: Grigory Shovkoplyas

Let us denote the starting point as point S and the ending point as T.

Consider the geometric locus of points (GLP) of the grasshopper's positions. After the first jump, the GLP will be a circle centered at the starting point with a radius equal to the first jump. After the second jump and onwards, the GLP becomes a ring, possibly with a zero inner radius.



Thus, the grasshopper will not be able to reach point T if it is either outside the outer radius of the ring formed after all jumps or inside the inner radius. Let M denote the maximum jump length from the set, and D the distance between S and T. Point T is reachable from S if and only if $M - (a_1 + a_2 + \cdots + a_n - M) \le D \le a_1 + a_2 + \cdots + a_n$.

Next, we need to determine the trajectory of the jumps. We will make several assumptions for the convenience of constructing the solution.

- Let the trajectory start and end at S, and we will add a jump of length D at the end of the jump set.
- $M' = \max(M, D)$
- We will remove the jump of length M' from the set and assume that it can be performed at any moment.

In this case, the trajectory of the jumps will be a triangle with sides $a_1 + a_2 + \cdots + a_k$, $a_{k+1} + \cdots + a_n$ and M'. Let k be the minimum k such that $a_1 + a_2 + \cdots + a_k + M' \ge a_{k+1} + \cdots + a_n$.

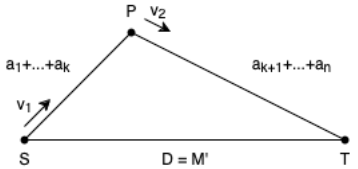
We will prove the existence of such a triangle (check the triangle inequality). Note that the previously verified condition guarantees that $M' \le a_1 + a_2 + \cdots + a_n$. We also specifically chose k so that $a_1 + a_2 + \cdots + a_k + M' \ge a_{k+1} + \cdots + a_n$, so it remains to prove that $a_1 + a_2 + \cdots + a_k \le a_{k+1} + \cdots + a_n + M'$.

Assume this is not the case, then it must be true that $a_1 + a_2 + \cdots + a_{k-1} + a_k > a_{k+1} + \cdots + a_n + M'$. We chose k such that it holds that $a_1 + a_2 + \cdots + a_{k-1} + M' < a_k + a_{k+1} + \cdots + a_n$. Subtracting the first inequality from the second, canceling common terms, we get $a_k - M' > M' - a_k$, which simplifies to $a_k > M'$, which is a contradiction. Thus, the existence of a triangle with sides $a_1 + a_2 + \cdots + a_k$, $a_{k+1} + \cdots + a_n$ and M' is proven.

Now, for the final construction of the trajectory, we will consider two cases: $M' = D$ and $M' \ne D$.

$M' = D$

We will choose the segment from S to T as the base of the triangle. Next, we will find the third point of the triangle with sides $a_1 + a_2 + \cdots + a_k$, $a_{k+1} + \cdots + a_n$ and $M' = D$. This can be done using the formula for the coordinates of the intersection point of two circles. From the two options for the points, we will choose any. Let this be point P.



Then we will compute the unit vectors $\vec{v_1} = \vec{SP}/|SP|$ and $\vec{v_2} = \vec{PT}/|PT|$. Thus, to jump from point S to point P, we will compute each subsequent point as $P_{next} = P_{prev} + a_i \cdot \vec{v_1}$, and then, to jump from point P to point T — using the formula $P_{next} = P_{prev} + a_i \cdot \vec{v_2}$.

$M' \neq D$

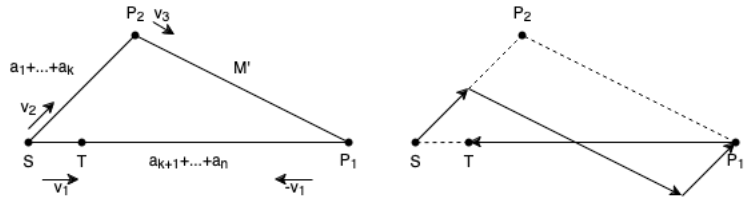
The second case is a bit more complicated. In this case, T is not a vertex of the triangle, but it must lie on one of its sides for everything to work out. Since we added the jump of length D at the end, point T must lie on the side of the triangle with length $a_{k+1} + \cdots + a_n$.

Also, recall the assumption that we removed the jump of length M' from the set and considered it could be performed at any moment. Since the order is strictly defined, when calculating $a_1 + a_2 + \cdots + a_k$, we will not consider M', if the first occurrence of the maximum is reached at index $i \leq k$, then we will not have confusion with indices in the further solution.

Let $\vec{v_1} = \vec{ST}/|ST|$. We will compute the vertex of the triangle P_1 as $P_1 = S + (a_{k+1} + \cdots + a_n) \cdot \vec{v_1}$. Then we will compute P_2, as the third point of the triangle, similarly to the previous case. Next, we need to compute two unit vectors $\vec{v_2}$ and $\vec{v_3}$, aligned with $\vec{SP_2}$ and $\vec{P_2P_1}$ respectively.

Then the solution looks similar to the previous case, we compute the next point from the previous one by adding the vector:

- $a_i \cdot \vec{v_3}$, if $a_i = M'$ and this is the first occurrence of the maximum.
- $a_i \cdot \vec{v_2}$, if $i \leq k$
- $a_i \cdot -\vec{v_1}$, if $i > k$



It is also necessary to carefully consider the case where $S = T$, in which case we can choose any unit vector as $\vec{v_1}$.

Problem F. Delivery Robot

Problem author: Artem Vasilyev, developer: Margarita Sablina

It is required to cover the points in a square of size $n \times n$ using $2n-2$ segments. The method of construction for $n = 3$ is provided in the example. We will construct the answer for the square of size $(n+1) \times (n+1)$ using the answer for the square of size $n \times n$ by adding two additional segments.

To transition from $n = 3$ to $n = 4$, we take the sequence of segments from the condition and extend the last segment to the point $(0; -1)$. After that, we will "wrap"the square of size $n \times n$ with two additional segments along the boundary of the previous square, covering all the missing points.

For example, for the case of $n = 4$, we will get the sequence of points

$$(2; 2), (0; 0), (3; 0), (0; 3), (0; -1), (4; -1), (4, 4).$$

At this point, we have already extended the last segment so that we can continue the construction for the square that is one size larger.

All that remains is to shift the origin so that the lower left corner of the resulting square of size $n \times n$ is at the point $(0; 0)$.

Problem G. Trip to Moscow

Author of the problem: Andrew Stankevich, developer: Mikhail Perveev

Note that in the optimal solution, Andrew will move only to the right along the number line. Besides the points 0 and m, he may stop at some gas stations, and on each segment between two consecutive stops, he will move at the maximum possible constant speed such that he has enough fuel.

Consider two points with coordinates x_1 and x_2 ($x_1 < x_2$). We will calculate the maximum speed at which Andrew can travel from point x_1 to point x_2, given that at point x_1 his car is fully fueled. Since at speed v the car consumes v liters of fuel per unit distance, and the tank size is c liters, the maximum possible speed is $v_{\max} = \frac{c}{x_2 - x_1}$. When moving at this speed, upon reaching point x_2, there will be no fuel left in the car. Thus, the minimum travel time from point x_1 to point x_2 will be $\frac{x_2 - x_1}{v_{\max}} = \frac{(x_2 - x_1)^2}{c}$.

Now we will use dynamic programming to solve the problem. We introduce additional points with coordinates $x_0 = 0$ and $x_{n+1} = m$. We say that $dp[i]$ is the minimum time sufficient to get from point x_0 to point x_i, and then refuel the car at point x_i.

Initially, we are at point x_0, so $dp[0] = 0$. To compute $dp[i]$, we need to iterate through the previous point where a stop will be made. The dynamic programming recurrence relation looks as follows:

$$dp[i] = \min_{j=0}^{i-1} \left(dp[j] + \frac{(x_i - x_j)^2}{c} \right) + t_i$$

The answer is found in $dp[n+1]$ (for convenience, let's say that $t_{n+1} = 0$, since refueling the car in Moscow is not mandatory). Thus, we have obtained a solution in $\mathcal{O}(n^2)$.

First, let's slightly modify the formulas to eliminate intermediate calculations in floating-point numbers, which may degrade accuracy. To do this, we multiply the obtained equation for $dp[i]$ by the number c. After this operation, we get the formula:

$$c \cdot dp[i] = \min_{j=0}^{i-1} \left(c \cdot dp[j] + (x_i - x_j)^2 \right) + c \cdot t_i$$

For convenience, we denote the quantity $c \cdot dp[i]$ as $dp'[i]$. Applying this notation, we obtain the formula:

$$dp'[i] = \min_{j=0}^{i-1} \left(dp'[j] + (x_i - x_j)^2 \right) + c \cdot t_i$$

Now we can compute the value of $dp'[n+1]$ using this formula, and at the end, divide the answer by the number c. This will help avoid unnecessary calculations in floating-point numbers and possible loss of precision.

Finally, let's optimize the time complexity of the solution. To do this, we expand the square in the dynamic programming recurrence relation:

$$dp'[i] = \min_{j=0}^{i-1} \left(dp'[j] + (x_i)^2 - 2 \cdot x_i \cdot x_j + (x_j)^2 \right) + c \cdot t_i$$

Now let's group the terms:

$$dp'[i] = \min_{j=0}^{i-1} \left((dp'[j] + (x_j)^2) + (-2x_j) \cdot x_i \right) + c \cdot t_i + (x_i)^2$$

Note that in this formula we are looking for the minimum among a certain set of linear functions of the form $y = k_j \cdot x + b_j$, where $k_j = -2x_j$, and $b_j = dp'[j] + (x_j)^2$, at the point x_i. After we compute the value of $dp'[i]$, another linear function is added to the set.

Thus, we need to perform operations of two types:

- Add to the considered set the function $y = k \cdot x + b$ with some known parameters k and b;
- Find the minimum among all considered functions at a certain point x.

These operations can be performed using the Convex Hull Trick technique. Adding a new line can be done in amortized $\mathcal{O}(1)$ time, as the slope of the added line is not greater than the slopes of the lines that were added earlier. Finding the minimum at a point can be done in $\mathcal{O}(\log n)$ using binary search. Thus, we obtain a solution in $\mathcal{O}(n \log n)$.

It can also be noted that the points at which the minimum needs to be computed do not decrease, so we can eliminate binary search by using the two-pointer method. It is sufficient to maintain a pointer to the line that is optimal at the current point. Since the queries are monotonic, the pointer will only move to the right. This will allow us to achieve a time complexity of $\mathcal{O}(n)$. However, this was not required to solve the problem.

Problem H. Colorful Graph

Author and developer of the problem: Egor Ulin

This problem can be solved in several ways. Let's consider a solution that uses the data structure *map*.

For each color c, we will store the number of *bad* vertices. We define bad vertices as those from which more than one edge of color c originates. The number of such vertices can be counted using a *map*, where for each vertex we store the number of edges of color c connected to it.

A color is considered excellent if the number of bad vertices of that color is equal to 0. Thus, after each query, we can determine in $O(1)$ time whether color c is excellent.

The processing of queries works as follows:

- If the color was excellent and after adding an edge it is no longer excellent, we subtract the number of edges of that color from the answer;
- If the color was not excellent and after removing an edge it became excellent, we add the number of edges of that color to the final answer;
- If the color was not excellent and after the query it did not become excellent, the answer remains unchanged;
- If the color was excellent and remains excellent, we add 1 to the answer.

The number of edges of each color can be stored using an array.

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

Problem I. "Galactic Timeline"

Problem author: Margarita Sablina, developer: Daniil Golov

First, let's understand that cards with the same numbers written on both sides can be disregarded separately, because in the correct timeline they go one after the other. Therefore, in the solution, we will assume that there are no cards with matching numbers on both sides.

Now, let's look at the cards that have the minimum number of all. We will examine their reverse sides and choose the card with the minimum number on the reverse side, let's call it X. Notice that the chosen card can only be in the first position in the answer.

Indeed, if we assume that X is not in the first position, then consider the card that is in the timeline before it, let's call it Y. It is known that the number on one side of X is minimal. Accordingly, the card Y on that side can only have the same number as on X, otherwise the order is not maintained. But in that case, the number on the other side of X will be less than the number on the other side of Y, and the order will also not be maintained. Thus, the card X can only be in the first position.

Then we will choose card X, place it in the first position, exclude it from consideration, and repeat the process for the remaining cards. The card that we choose not first will be flipped if necessary. Notice that in this way we will select n cards with the minimum numbers, which means we will simply sort the cards by their minimum side.

Then we will sort the cards first by the minimum number on them, and in case of ties in the minimums, by the maximum. After such sorting, we will check that the side with the larger numbers on the cards is also in the correct order, and if not, we will say that it is impossible to find the correct order.

Problem J. Chess Bishop

Author and problem developer: Zakhar Iakovlev

There are many different solutions to this problem. We will provide one constructive solution.

First, we will perform the following sequence of operations (essentially a binary search) and remember which operations were successful:

1. $(4, 4)$
2. $(2, 2)$
3. $(1, 1)$

After performing these operations, the bishop is guaranteed to be on the last 8th rank or on the last file `h`. This way, we learn the position on its diagonal (the distance to the right or top edge of the board). Now we will perform the following sequence of actions:

1. $(-4, -4)$
2. $(-2, -2)$
3. $(-1, -1)$

After this, the bishop is guaranteed to be on the 1st rank or on the first file `a`. Now the length of the diagonal and the position on it are known.

Finally, we will make the move $(1, -1)$. This will determine whether the original diagonal is above the main diagonal or not. Ultimately, the diagonal on which the bishop was located and the position on it are known.

A total of exactly 7 operations were performed.

Problem K. Magic Spell

Author and developer of the problem: Nikita Golikov

We will create a directed graph with $n + 1$ vertices, numbered from 1 to $n + 1$, where each edge will have a color — a number from 1 to k. Consider a subarray $[l_q : r_q]$ of the array a. Note that we are interested in several cases for using it in the answer:

1. p is a subarray of $a[l_q; r_q]$ — the answer is 1.
2. $a[l_q : r_q] = p[i : j]$ — edge $i \to j + 1$ has color k.
3. $a[l_q : l_q + x] = p[n - x : n]$ — edge $n - x \to n + 1$ has color k.
4. $a[r_q - x : r_q] = p[1 : x + 1]$ — edge $1 \to x + 2$ has color k.

To determine which of the cases hold, we will precompute the arrays left[i] and right[i], which store the length of the maximum subarray from i to the left and right respectively, that is a subarray of p. To do this efficiently, we will precompute the inverse permutation for p, after which we will compute the arrays left and right in a linear pass. Using these arrays and the inverse permutation, we can determine cases 2-4.

To efficiently determine the first case, we will find all occurrences of p in the array a in advance; this can be done naively by looping through each occurrence of p_1 in the array a, which will work in linear time overall since p is a permutation. After that, for each index i, we will precompute the nearest occurrence to the right, and when considering the segment $[l_q; r_q]$, we will find the nearest occurrence to the right of l_q and check that it fits within the segment.

Now, if the answer is not equal to one, the problem can be reformulated as follows: we need to find the shortest path in terms of the number of edges from vertex 1 to vertex $n + 1$, where all edge colors are different. Note that all edges in our graph go from a smaller vertex to a larger one, so it is acyclic. We also note that the number of edges of each color is no more than two, and if there are two, one edge goes from vertex 1, and the other edge goes to vertex $n + 1$.

Without loss of generality regarding colors, the problem of finding the shortest path in an acyclic graph can be solved using dynamic programming (dp[v] — the length of the shortest path from vertex v to vertex $n + 1$) in linear time. To account for the possibility of two edges of the same color, we will count two shortest paths from each vertex v, which have different colors for the last used edge (in other words, edges to vertex $n + 1$). To do this, we will maintain the arrays dp[v] and $\text{dp}_2[v]$, as well as the array edgeInd[v], which store the length of the minimum shortest path, the second shortest path differing by the index of the last edge, and the index of the last edge used in the shortest path, respectively. This can be computed in a linear pass from the end, similar to finding the second minimum in an array.

Now we will find the answer as follows: we will iterate over the first edge on the path, let this edge be $1 \to v$ with color c. If edgeInd[v] $\neq c$, we can update the answer via $1 + \text{dp}[v]$, otherwise via $1 + \text{dp}_2[v]$. We will find the minimum over all first edges, after which we will reconstruct the answer in the standard way, saving the ancestor arrays for dp[v] and $\text{dp}_2[v]$. We have obtained a solution in linear time.

Problem L. Multimino

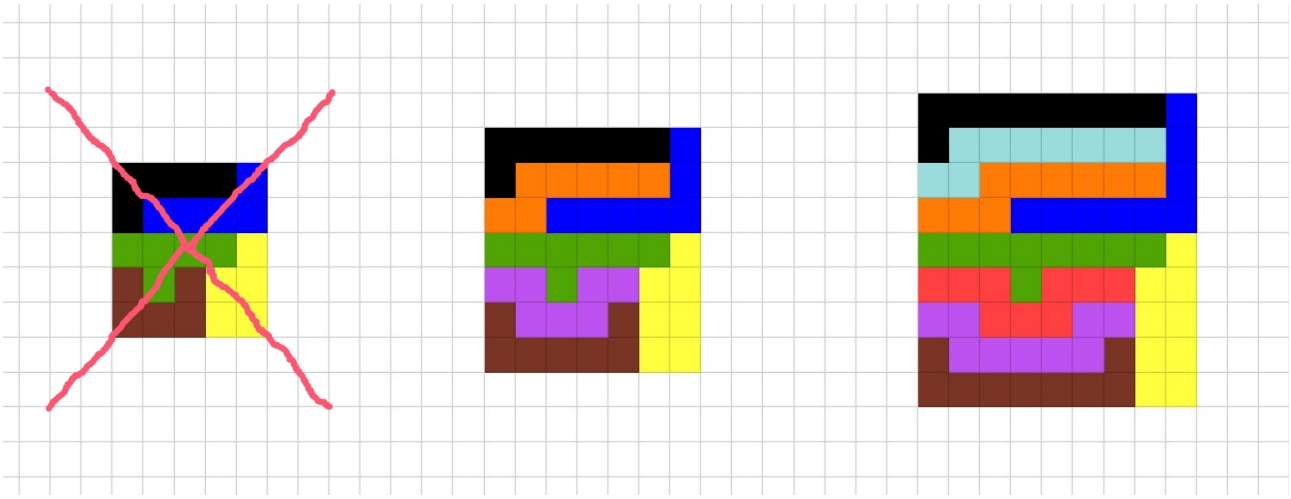
Author and problem developer: Mikhail Savvateev

We want to find some pattern that, when repeated with different parameters, will yield many different n-minos that are structurally similar and fit well together. There are surely many implementations of this idea, and below is one of the quite natural ones.

Let's devise a construction for even sufficiently large n. We will build it sequentially from the example for $n = 6$ shown on the left, adding additional n-minos—"zigzags— on top of the cut square and n-minos—"trays— on the bottom each time we transition from n to $n + 2$:

We just need to slightly modify the construction so that it also works for odd n. For example, this can be done as shown on the right:

This results in a sequence of cuts that is almost identical to the sequence for even n, with the only important note being that they are valid only starting from $n = 7$, since for $n = 5$, the black and blue n-minos turn out to be equal:



For small n, we will find the answer manually. Fortunately, the most complex cases $n = 4, 5$ have already been addressed in the problem statement; for $n = 2, 3$, it can be trivially proven that the required partition does not exist; for $n = 1$, one 1-mino will suffice, and for $n \geqslant 6$, our construction above is already fully valid, so we will apply it.

Problem M. Three Points

Author of the problem: Artem Vasilyev, developer: Ekaterina Vedernikova

We will consider the arrangements of a line (trench) and three points on a plane (three houses).

Let the line not pass through any of the points. Then there are two cases:

1. If the line lies outside the triangle formed by these points. We will use a parallel translation and move the line to the nearest vertex. One of the distances will be reduced to 0, while the others will decrease (since this is a parallel translation).
2. If the line passes inside the triangle, then it is clear that on one side of it there are two points, and on the other side, there is one. Then we will parallelly move the line to the nearest vertex (from the two that lie on one side). Let's say we moved the line a distance of h, then the distance to the two points decreased by h each, while the distance to one point increased by h, resulting in a total distance decrease of h. Thus, when the line passes through at least one of the points, the total distance to it will be less than if it does not pass through any of them.

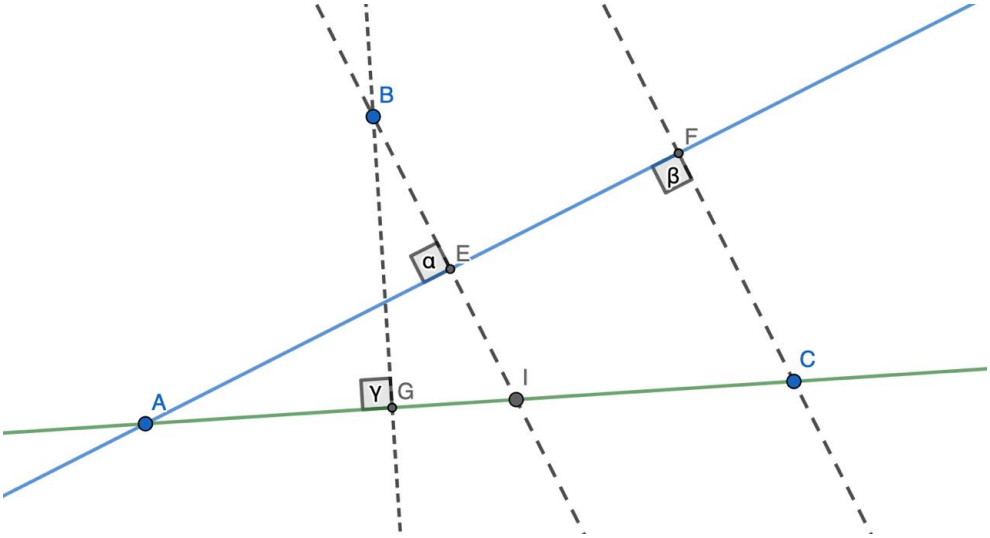
Therefore, the line passes through at least one point.

Let the line pass through two points; then the total distance to it will be smaller when it passes through the two most distant points from each other. If we recall the formula for the area of a triangle using height, we find that the larger the side to which the height is drawn, the smaller the height itself. The total distance from all points will equal the length of the height (since the distance from the other points will be zero). It remains to show that this is the optimal solution.

Let's compare the total distance to the line in the case when the line passes through exactly one point and when the line passes through the two most distant points from each other.

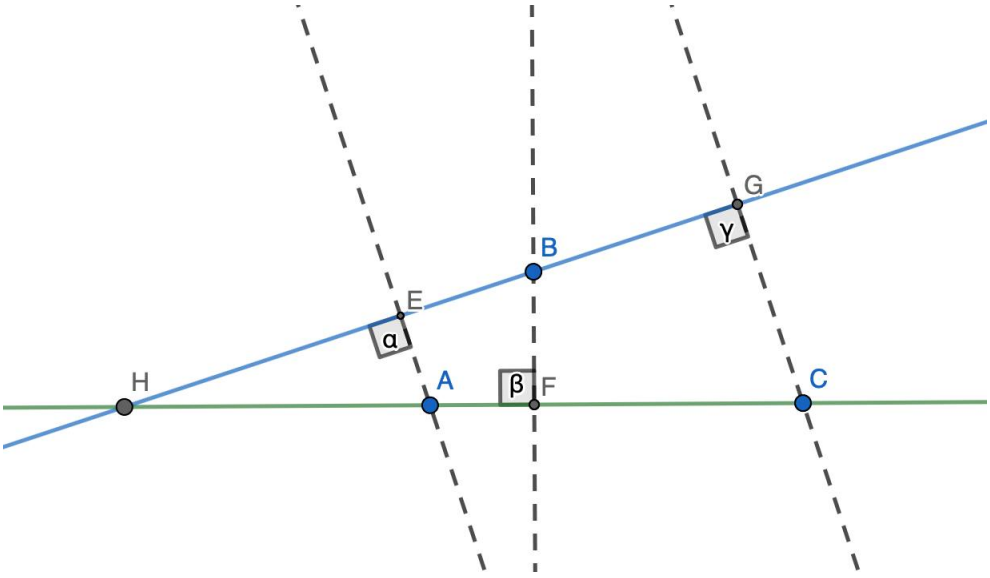
We will consider two cases.

First case:



When the line passes through one of the points that lie on the line (which passes through the two most distant points). We need to compare the height BG and the sum of the heights BE and FC. Extend BG to intersect with AC at point I. Triangle BGI is a right triangle, with hypotenuse BI (BI = BE + EI) > BG (the leg). We also note that EI < FC (due to the parallelism of these lines, it is easy to see the similarity of the triangles, as well as that this side will be smaller since the other sides are also smaller than the corresponding sides). Thus, we conclude that the distance to the line passing through the two most distant points is less than the sum of the distances to the line passing through one of these points but not through the other.

Second case:



When the line passes through the third point (which does not lie on the line passing through the two most distant points from each other). We constructed point H – the intersection of these lines (in the case where they are parallel, the distances will be equal, and it is clear that $h < h + h$). The angle at this vertex is common, which means triangles EHA, FHB, and GHC are similar to each other. Therefore, height BF < EA + GC (remember that HB < HC). In this case, it turns out that if we draw a line through the two most distant points from each other, the distance will be less than if we draw a line through the third point.