Cutting the Cake

Input file:	`standard input`
Output file:	`standard output`
Time limit:	2 seconds
Memory limit:	512 megabytes

This problem can only be solved in the C++ programming language.

The organizers of the Closed Olympiad for school students in programming decided to order a large cake for the evening banquet. They ordered a cake with n candles numbered from 0 to $n-1$. It is known that the candles are located at different points on the plane, no three candles lie on the same line, and no four candles form a trapezoid (a quadrilateral with two parallel sides).

To ensure that everyone gets a piece of cake, the organizers want to pre-construct a way to cut the cake into the maximum number of pieces. Unfortunately, you do not know the positions of the candles on the cake, and the only way to find out what the cake looks like is by corresponding with the pastry shop via email.

In one message, you can request to weigh no more than four triangular pieces, with vertices at the candles:

- You provide two lists of triangles `left` and `right`, containing no more than four triangles in total. The triangles may intersect, share common candles, and even coincide.
- The pastry shop will cut the triangular pieces according to the requested lists from several copies of the cake and place the pieces from `left` on the left scale and the pieces from `right` on the right scale.
- You will receive the result of the weighing. We will consider that the weight of a piece equals its area. As a result, you will find out which of the triangle lists has a greater total area, or if the areas are equal.

After several weighings, you must find a way to cut the cake. Each piece must be a convex polygon with vertices at the candles. The vertices of the piece must be listed in the order of traversal either clockwise or counterclockwise. Any two pieces must not intersect at internal points. You must return a list of pieces whose total area is maximally possible. Also, in some groups, it is required to maximize the number of pieces.

Solution Format

This is an unusual problem. It has a testing format with a grader, where you need to implement only the function `solve` with the solution. This function will be called by the testing program of the jury (the grader), and the return value of the function will be accepted as the solution to the problem.

In particular, this means that the code you submit **must not contain input or output**. Your code **must not** contain a `main` function. If necessary, you can implement any number of helper functions, structures, classes, and global variables, but all the code of your solution must be in one file.

You must implement the following function:

```
std::vector<std::vector<int>> solve(int n);
```

The function `solve` takes a single integer n — the number of candles.

In the implementation of the function `solve`, you can use the function `compare`, which is implemented by the grader:

```
int compare(const std::vector<Triangle>& left, const std::vector<Triangle>& right);
```

This function takes two non-empty lists of triangles with a total size of no more than 4. It returns -1 if the total area of the triangles in `left` is less than the total area of the triangles in `right`, 0 if the

areas are equal, and 1 otherwise. If an incorrect request is made or the limit on the number of requests is exhausted, the program will terminate automatically.

To access the function `compare` in your solution, the first line of your code must include the header file with the following line:

```
#include "triangles.h"
```

In this file, the structure `Triangle` used in the function `compare` is defined as follows:

```
struct Triangle {
    int i, j, k;

    Triangle() = default;
    Triangle(int i_, int j_, int k_) : i(i_), j(j_), k(k_) {}
};
```

This structure describes a single triangular piece, where the parameters `i`, `j`, and `k` describe the indices of the candles that form the vertices of the triangular piece. The indices of the candles must be distinct for a single triangular piece, but they can repeat in multiple triangles.

You do not need to include the definition of the structure `Triangle` in the code you submit; it will be automatically taken from the header file.

All parameters (indices of candles in requests, as well as in the result you return) are specified in **0-indexing**.

Your function `solve` should use calls to the function `compare` to find any partitioning of the cake into convex pieces, where any two pieces do not intersect at internal points, the total area of the pieces is maximized, and possibly the number of pieces is maximized.

The function `solve` should return a list of pieces into which the cake is divided. Each piece is described by a structure `std::vector<int>`, consisting of the candles that form the convex polygon. The candles must be listed in **the order of traversal along the boundary of the polygon** (either clockwise or counterclockwise). Any two pieces must not intersect at internal points. You must return a list of pieces whose total area is maximally possible, and in some groups, it is additionally required to maximize the number of pieces while maintaining the condition of maximizing the total area.

During one run of the grader, the **jury will make exactly one call to the function** `solve`. The **grader is not adaptive**, meaning that the positions of the candles are fixed in advance and do not depend on the implementation of the function `solve`.

Testing

You are provided with a solution template `triangles.cpp`, as well as a header file `triangles.h`, containing the definitions of the functions `solve` and `compare`. For convenience in testing, you are provided with a grader — the file `grader.cpp`. This file implements reading input data from the standard input stream, calling the function `solve`, and outputting the result of the function `solve` to the standard output stream. In the testing system, the grader may differ.

To compile your code `triangles.cpp` in C++, use the command

```
g++ -std=c++20 grader.cpp triangles.cpp -o grader
```

After executing this command, an executable file of the grader `grader` or `grader.exe` will be created, depending on your operating system, which you can run to input tests in the specified format.

If compilation via commands causes you difficulties, for local testing, you can copy the implementation of the function `solve` into the file `grader.cpp` (it must be inserted before the function `main`) and run the file `grader.cpp`. In this case, before submitting the solution to the testing system, you will need to leave only the implementation of the function `solve`, **remembering to include the header file at the beginning of the code** (this is done by adding the line `#include "triangles.h"` at the beginning

of the code you submit). If you are using any C++ libraries, their inclusion must also be added at the beginning of the submitted code.

In case of receiving a compilation error verdict, ensure that **the code you submit does not contain a `main` function, nor does it contain definitions of the function `compare` and the structure `Triangle`**. You can only use the function `compare` and the structure `Triangle` in the code you submit.

Input

The grader reads the test in the following format:

The first line contains an integer n ($4 \leq n \leq 10\,000$) — the number of candles.

In the next n lines, there are two integers x_i, y_i ($-10^9 \leq x_i, y_i \leq 10^9$) — the coordinates of the i-th candle.

Output

The grader outputs the results of the function `solve` — the list of found pieces.

In the first line, the number of found polygons (the size of the vector returned by the function `solve`) is printed.

In the following lines for each polygon (an element in the vector returned by the function `solve`), the size of the polygon is printed first, followed by the line of indices of the candles that are the vertices of the polygon (elements of each `std::vector<int>` in the return value of the function `solve`). Each polygon must be convex, and its vertices must be listed in the order of traversal (either clockwise or counterclockwise). No two polygons should intersect at internal points.

In the file `grader.cpp`, there is a variable `verbose`, which is initially set to 0. By increasing its value, the grader will write more detailed information about your solution and its requests.

Examples

standard input	standard output
4 1 2 1 4 0 0 3 -1	3 3 0 1 2 3 0 2 3 3 0 1 3
5 -1 -1 4 4 4 -2 1 2 -2 2	1 4 0 4 1 2
6 2 2 0 -2 -1 3 -2 0 7 0 2 -3	4 3 2 4 3 3 3 4 5 3 1 3 5 3 0 2 4

Note

In the first example, a possible sequence of function calls could look like this:

Initially, the function `solve(4)` is called.

From the function `solve`, the function

```
compare({Triangle(0, 1, 2)}, {Triangle(1, 2, 3)})
```

is called.

During the execution of the function, the sizes of the triangular pieces formed by the triangle with vertices at candles numbered 0, 1, and 2 and the triangle with vertices at candles 1, 2, and 3 are compared.

Since the first triangle is smaller than the second, the function returns -1.

Next, from the function `solve`, the function

```
compare({Triangle(1, 2, 3)}, {Triangle(0, 1, 2), Triangle(0, 1, 3)})
```

is called.

The response will return 1, as the area of the triangle with vertices at candles 1, 2, and 3 is greater than the total area of the triangle with vertices at candles 0, 1, and 2 and the triangle with vertices at candles 0, 1, and 3.

Next, from the function `solve`, the function

```
compare({Triangle(1, 2, 3)}, {Triangle(0, 1, 2), Triangle(2, 3, 0), Triangle(0, 1, 3)})
```

is called.

The response will return 0.

In the end, the function `solve` returns `{{0, 1, 2}, {0, 2, 3}, {0, 1, 3}}` — a description of the partitioning of the cake into convex pieces that have the maximum possible total area and do not intersect at internal points.

In the first and third examples, the found method of cutting the cake has the maximum possible number of pieces, so such answers would be considered correct in groups with maximization. In the second test, the number of pieces is not maximally possible, so such an answer would be considered correct in groups without maximization, but would receive 0 points in groups with maximization.

Note that the first test fits the additional constraints of groups 1 and 2, while the third test fits the additional constraints of group 5.

Scoring

The tests for this problem consist of 11 groups. The rules by which points are awarded for groups are described below. Note that passing the tests from the statement is not required for some groups. The final score for each group is equal to the maximum score obtained for that group of tests across all submissions.

The score for a solution for a group of tests is equal to the minimum score obtained across all tests in the group.

- If your solution makes an incorrect call to `compare` or returns an answer that does not satisfy the mandatory conditions, it will receive 0 points for the test.
- In each test, your solution is allowed to make no more than $2 \cdot 10^6$ calls to the function `compare`. If the limit is exceeded, your solution will receive 0 points for the test.
- In some groups, it is required to maximize the number of pieces in the answer. In such groups, if the number of pieces is not maximally possible, your solution will receive 0 points for the test.

If none of the described conditions for the test are violated, the answer for the test is considered correct. Let the full score for the group of tests be p. Some groups are evaluated using a formula, while others are evaluated without using a formula:

- If the group is evaluated without using a formula, the score for the test is equal to p.
- Otherwise, if the group is evaluated using a formula, let q be the number of calls to `compare` made by your solution. Then the score for the test is equal to $\left\lfloor p \cdot \frac{20n}{\max(q, 20n)} \right\rfloor$.

Group	Full score	Score		Additional constraints	Required groups	Comments
		Max.	Formula	n		
0	0	no	no	–	–	Tests from the statement.
1	13	no	no	$n = 4$	–	
2	4	yes	no	$n = 4$	1	
3	13	no	no	–	–	$(10^9, -10^9)$, $(-10^9, 10^9)$, $(10^9, 10^9)$ are in the set, the rest are random points inside this triangle
4	8	yes	no	–	3	$(10^9, -10^9)$, $(-10^9, 10^9)$, $(10^9, 10^9)$ are in the set, the rest are random points inside this triangle
5	11	yes	yes	–	–	points are the vertices of a convex polygon
6	9	no	no	$n \leq 100$	–	random points
7	8	yes	no	$n \leq 100$	6	random points
8	9	no	yes	–	–	random points
9	8	yes	yes	–	–	random points
10	9	no	yes	–	–	
11	8	yes	yes	–	–	

In groups 6–9, all points were chosen randomly and independently from the square $[-10^9, 10^9] \times [-10^9, 10^9]$.

In groups 3–4, points other than $(10^9, -10^9)$, $(-10^9, 10^9)$, $(10^9, 10^9)$ were chosen randomly and independently from the triangle with vertices $(10^9, -10^9)$, $(-10^9, 10^9)$, $(10^9, 10^9)$.

In these groups, among such random tests, only those where all points are distinct, no three lie on the same line, and no four form a trapezoid are retained.