devsiddhartha@gmail.com

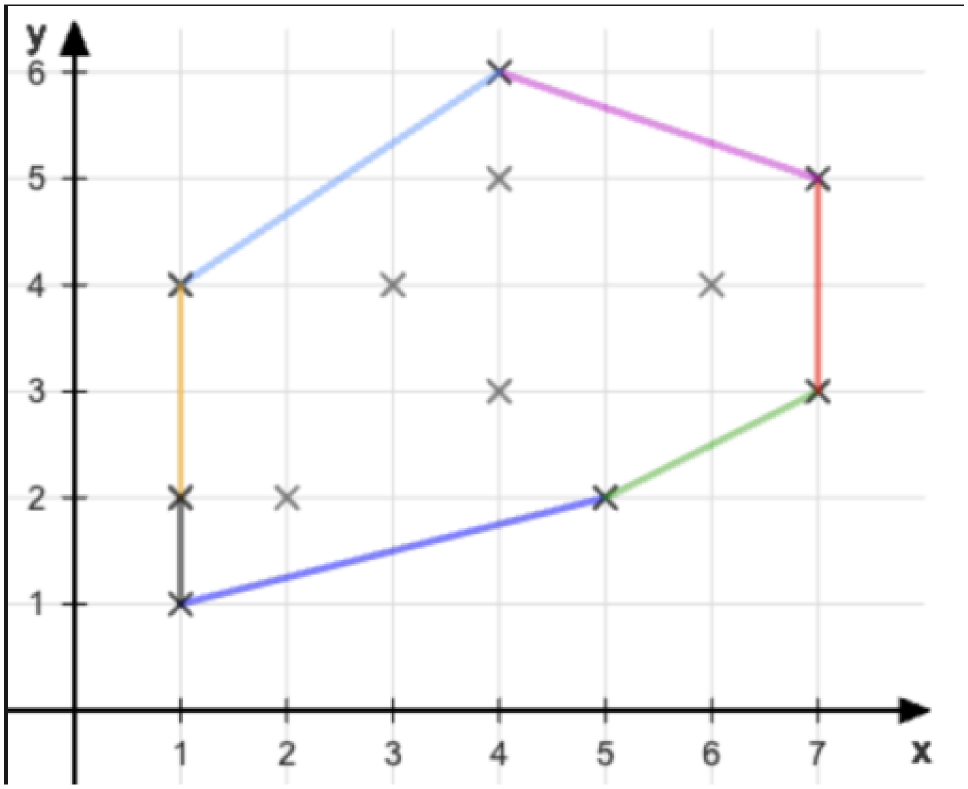
PenguHull - Tales of a (very) drunk Penguin

Global warming is affecting penguins. After extensive analysis, the excellently studied penguins of the PUM have determined that they have to start keeping farm animals in order to ensure sufficient food. Since penguins prefer to eat fish, they very quickly agreed that they wanted to mark off an area in the water in order to breed the fish there and then be able to catch them more easily. Fortunately, the researchers have been creating elaborate maps for generations to know where the schools of fish are usually located. In order to have fish in your breeding right from the start, you now want to set up a fence in the water around a group of such positions and start breeding. In a survey, the researchers have already determined which shoals of fish taste the best and one thing is certain: we can start breeding the tastiest fish tomorrow! … (A PinguUniParty at the Irish Pub later) … Oh jeh! The overzealous penguin Max seems to have drunk one Guinness too much. Max has haphazardly anchored all of the existing stakes into the seabed and there is no way to get them out again. There is only one option: we need to build the fence along the already set stakes. But how do we fence in the largest possible area with the existing posts? Surely you can write a program to find out. Or?

Explanation:

In this issue we will develop a program together to calculate approximately convex hulls. In order to reduce the mathematical complexity, the specification is kept quite informal. So if you want to learn more about the subject, or prefer mathematical definitions, you're welcomehere (Convex Hulls of Finite Sets of Points in Two and Three Dimensions - Maximilian Anziger)check. If there is enough interest, there will also be a dedicated (special) PGdP coffee party. You can imagine the problem formulated above as a board full of nails. The solution we're looking for is equivalent to what would happen if you put a rubber band around ALL the nails. Take a look at the following example:

Visual Representation (V1.0)



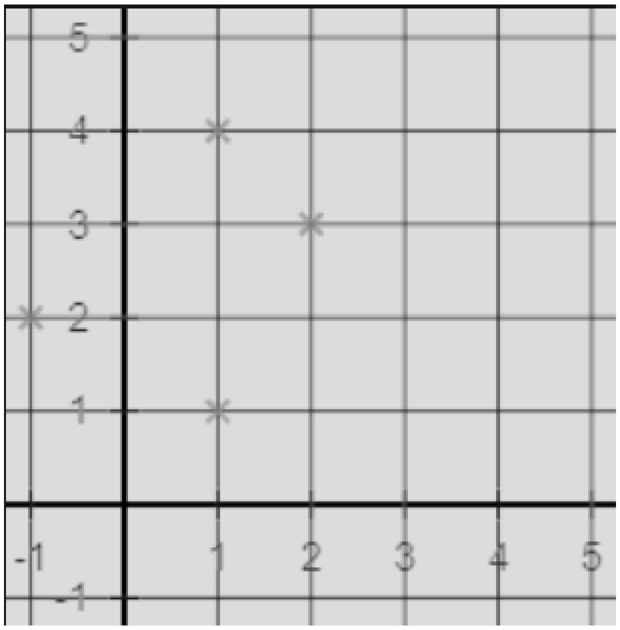
The points (X) correspond to the stakes/nails that are firmly anchored. The colored edges show how the fence/rubber band should run along the posts/nails. So, we calculate the smallest possible polygon that encloses all given points. Strictly speaking, a "real" convex hull should cover the two segments of (1, 1) to (1, 2) and from (1, 2) to (1, 4) by a single stretch of (1, 1) after (1, 4) substitute. To simplify the problem, we ignore this condition in this exercise.

Presentation of points/point listings

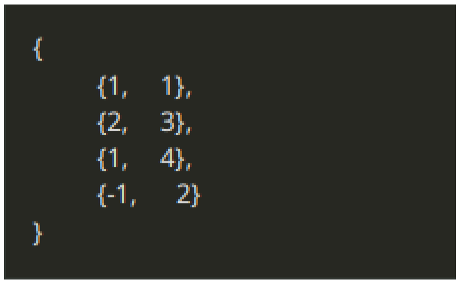
Points and listings of points are presented in our program as follows:

- One point (x, y) is represented by an integer array with two entries: {x, y}. The entry 0 corresponds to the x-coordinate and the entry 1 to the y-coordinate.
- A collection (a set, so to speak - except that they occur in a fixed order and there may be duplicates) of multiple points is represented as a 2D array (often int[][] points) shown. The inner arrays all have two entries and represent the individual points in the manner just described.

The point cloud



can be represented by the following array, for example:

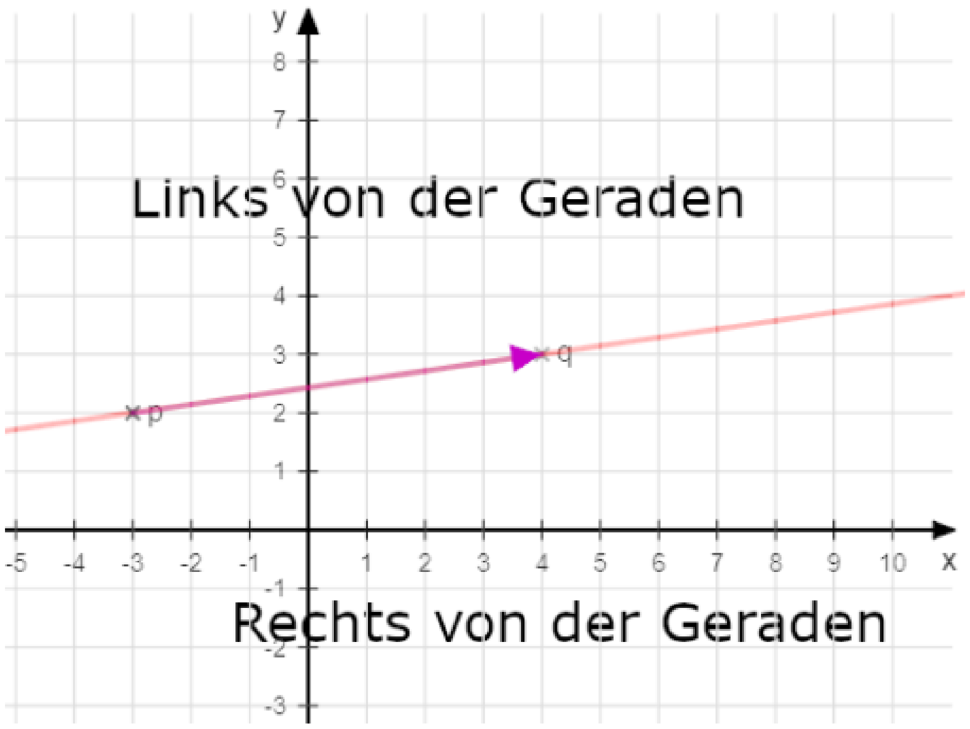


```
{
    {1,   1},
    {2,   3},
    {1,   4},
    {-1,    2}
}
```

Straights

We will often here in after refer to two points p and q reference spanned line. By this we mean the straight line that runs through the two points (implicit here, of course !Arrays.equals(p, q)- the points are not the same). When we talk about "to the left of the straight" or "to the right of the straight," we imagine we're standing up p and would towards q look. What is then to our left is "to the left of the straight line", what is then to our right is "to the right of the straight line":

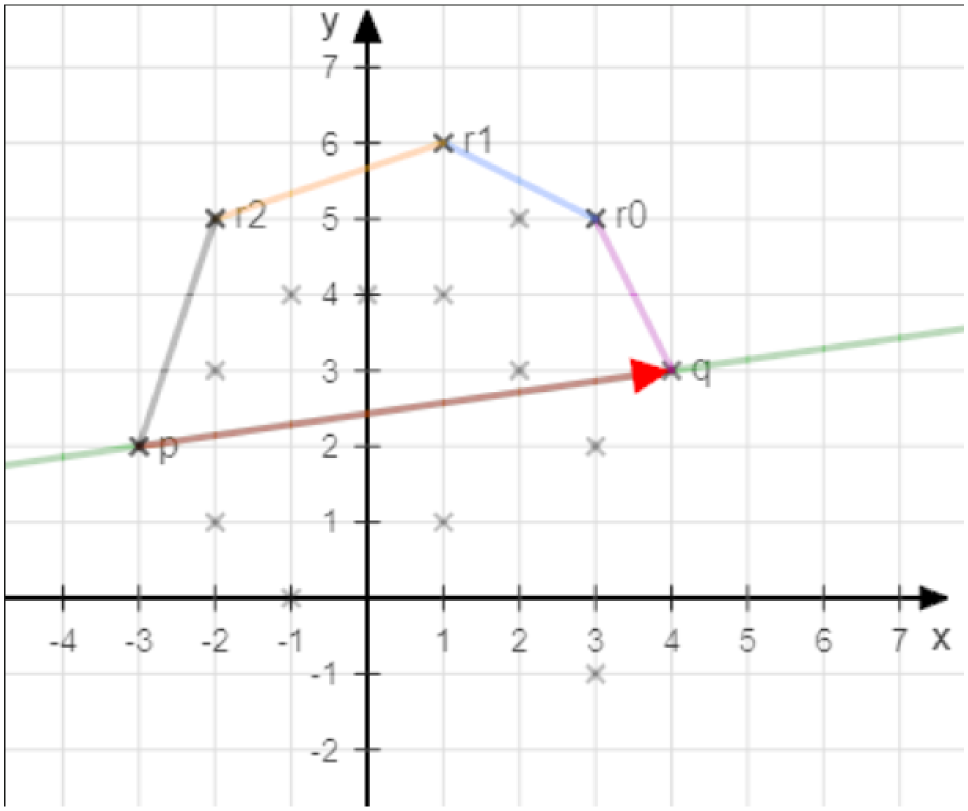
Visual representation



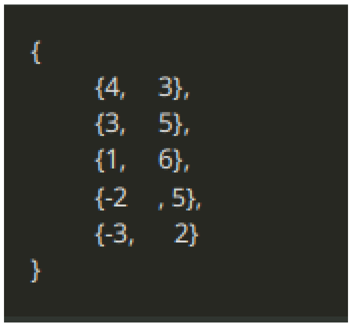
Links von der Geraden: To the left of the straight
Rechts von der Geraden: To the right of the straight

Representation of convex hulls

The convex hull of a point cloud is represented by a point listing (i.e., a 2D array) of all the points that lie on that hull. The points should be entered in the list of points counterclockwise from a starting point. Two variants can be distinguished: Open Shells: Still open (unfinished, so to speak) convex hulls, as in the methodquickHullLeftOf() handled and returned always refer to a pair of points P, Q. They describe the hull around all points to the left of the through P and Q spanned straight lines. The envelope is then delimited at the right-hand edge by the straight line itself. Such a shell is indicated by the list{Q, ..., P} of all points on the envelope of Q from counterclockwise up to P shown. For example, the one in the following image would result in the straight-line p and q related open shell through the array{q, r0, r1, r2, p} being represented:



So explicitly through

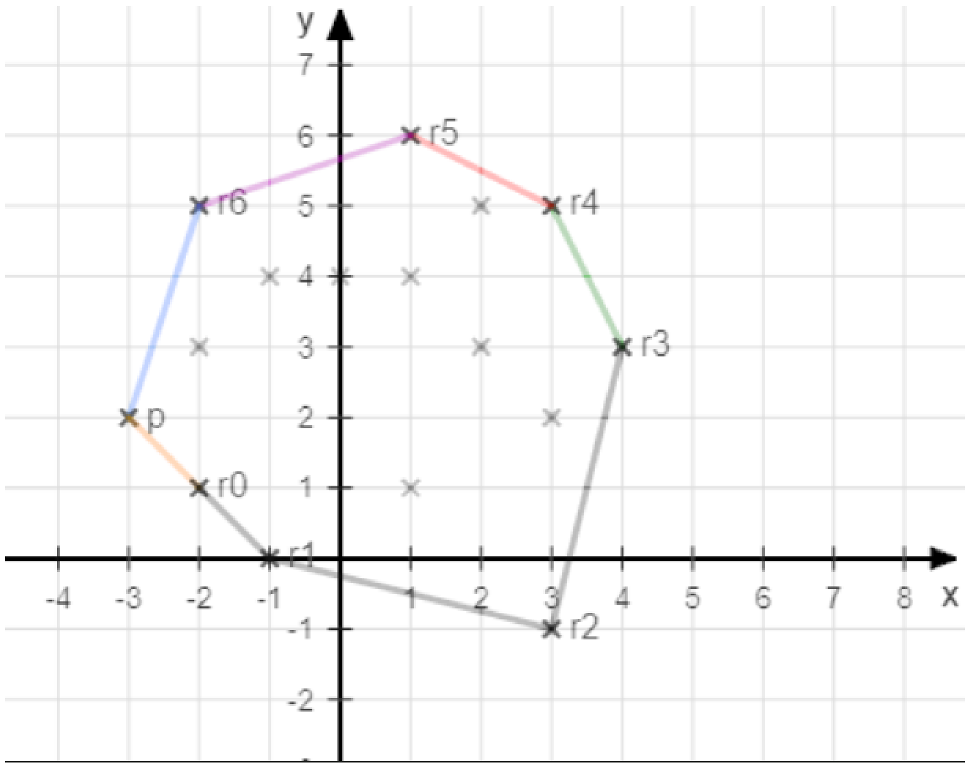


```
{
      {4,    3},
      {3,    5},
      {1,    6},
      {-2   , 5},
      {-3,     2}
}
```

Closed cases: A pre-completed shell, such as that eventually produced byquickHull() is to be returned does not refer to a straight line, but only has a starting point P. The hull is then represented again by listing all the points on it counterclockwise, but this time with P both as the first and as the last point:{P, . . . , P}. For example, the envelope shown in the following image would have a starting point p through the array{p, r0, r1, r2, r3, r4, r5, r6, p} being represented:

P.T.O

devsiddhartha@gmail.com



So explicitly through

```
{
        {-3,     2},
        {-2,     1},
        {-1,     0},
        {3,    - 1},
        {4,    3},
        {3,    5},
        {1,    6},
        {-2    , 5},
        {-3,     2}
}
```

The task

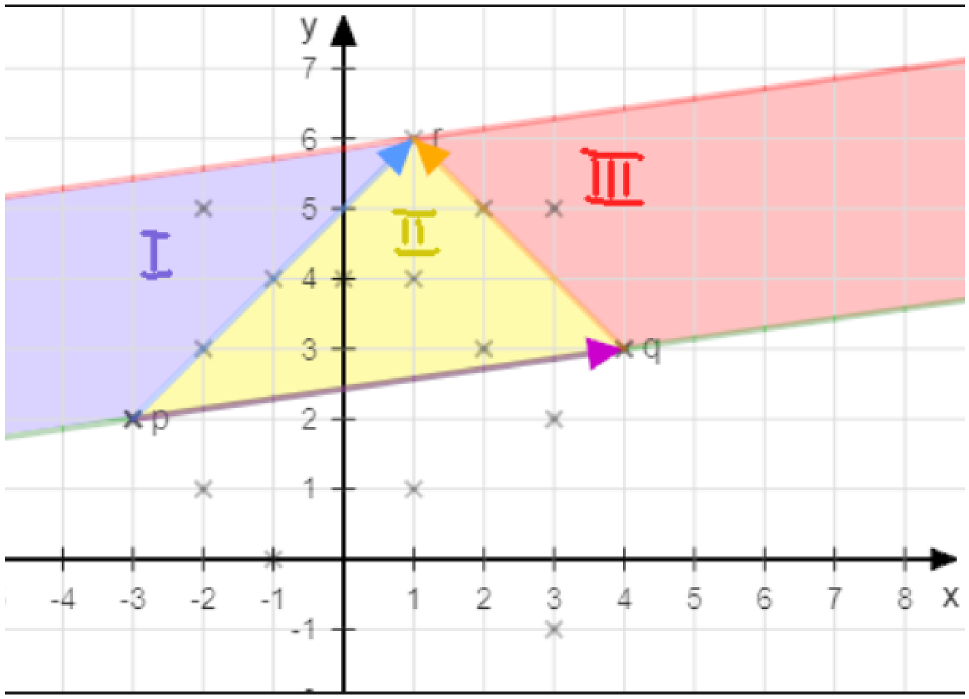
A notice: You can assume that all point listings passed in the hidden tests points not only contain points with the same x-coordinate. In other words: The absolute left and the absolute right point are actually separate from each other in the x-direction. They don't have the same x-coordinate.

1. The auxiliary method

The method int[][] combineHulls(int[][] firstHull, int[][] secondHull) accepts two open shells where it can be assumed that the end point firstHull[firstHull.length - 1] the first equal to the starting pointsecondHull[0] the second is. She combines them into a shell. This combined hull can then itself be open if the endpoint is in secondHull not equal to the starting point in firstHull is, or closed if both are the same. The combined case will be returned. From a shell of the form{p, s0, s1, s2, s3, q}and another of the shape{q, t0, t1, t2, r} would, for example, be the overall shell{p, s0, s1, s2, s3, q, t0, t1, t2, r} be made.

2. The heart

Implement the method int[][] quickHullLeftOf(int[][] points, int[] p, int[] q). This method recursively computes the convex hull of all points to the left of the through p and q spanned straight lie, as well p and q itself. It is therefore an open envelope related to the pair of points p and q. The procedure is as follows: We choose the point right out points to the left of the straight p and q, which has the largest (absolute) distance to this line. Since there are no points further away, we know that this point is also an outermost point of the enclosure (point of the convex hull). If such a point exists, we can recursively break down our problem with the following idea: The points p, q and right span a triangle that has the points in points to the left of the straight p and q partitioned into three disjoint sets: The points to the left of per, the dots to the right of qr and the points that are in or on the triangle Δpqr lie. The points in the triangle would already all be inside the hull{q,r,p}, but on the first two partitions we still have to expand our envelope. shout for its quickHullLeftOf() each with the correct parameters again. Do also think about whether the order of the parameters p and q plays a role and how the recursion is terminated.



The picture shows an illustration of the call quickHullLeftOf(points, p, q). There should be an open shell (relative to the straight line through p and q, which then closes it from the right (or below) around all points above the straight-line p and q be placed. For this purpose, the leftmost point in the viewing direction of the straight line is used right searched. Now you can see a parallel to the original straight line right introduce. All other points must (in the direction of view p after q) to the right of this one, there right yes is the leftmost point. All points around which an envelope is placed in this method call lie between the two straight lines. This area is now divided into three parts:

- (I): Left of per. An (open) convex hull has yet to be found in this region.
- (II): Inside the triangle Δpqr. Located within{q,r,p}.
- (III): Right of qr. An (open) convex hull has yet to be found in this region.

A notice: You should solve this method recursively. Ie in particular that loops, ie the keywords for and while (and streams) are prohibited in this subtask. If you still use some, we will deduct the points awarded by the tests. In the other two methods quickHull() and combineHulls() however, you can use them.

3. The launch

Implement the method int[][] quickHull(int[][] points). This method receives an array of the coordinates of all posts ({{x0, y0}, {x1, y1}, ...}) with which you should delimit the area. Returns the convex hull with the leftmost point in points(the one from findLeftmostPoint(points) is returned) as the starting point. So, this is a closed shell. Also look for the rightmost point in points and then start the algorithm with appropriate calls to quickHullLeftOf().

***** END *****