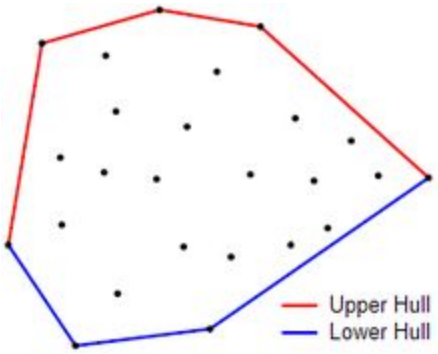
Convex hull

Description

Given a set of N points in a cartesian plane, we want to find the convex hull of those points. A convex hull of a set of points is defined as the convex polygon with minimum area that contains all the points.



Input from file

```
N - the total number of points
x[0], y[0]
x[1], y[1]
…
x[N - 1], y[N - 1]
```

Output in file

```
P - the total number of points that define the convex hull
x[0], y[0]
x[1], y[1]
…
x[P - 1], y[P - 1]
```

!! The points will be shown in clockwise order !!

Algorithm - Quickhull (similar to Quicksort)

1. Find the points with minimum and maximum x coordinates, as these will always be part of the convex hull.
2. Use the line formed by the two points to divide the set in two subsets of points, which will be processed recursively.
3. Determine the point, on one side of the line, with the maximum distance from the line. The two points found before along with this one form a triangle.
4. The points lying inside of that triangle cannot be part of the convex hull and can therefore be ignored in the next steps.

5. Repeat the previous two steps on the two lines formed by the triangle (not the initial line).
6. Keep on doing so on until no more points are left, the recursion has come to an end and the points selected constitute the convex hull.

Example

Input:

```
8
2.0 0.0
0.0 2.0
1.0 3.0
0.0 4.0
3.0 3.0
2.0 6.0
4.0 2.0
4.0 4.0
```

Output:

```
6
2.0 0.0
4.0 2.0
4.0 4.0
2.0 6.0
0.0 4.0
0.0 2.0
```

Projection and analysis

For the **sequential case**, it is `O(N * log(N))` on average case and `O(N^2)` worst case.

For the **parallel case**, when we divide the plane in two with the line, we delegate different threads to process the points.

We start with a fixed number of threads and we use Divide and Conquer paradigm, splitting the number of threads at each step. When we run out of threads, we just run the function on the current thread.

To be more clear, on each step we have a vector of points. We have to divide these points in two parts, and then call the same function recursively on these two parts - similar approach to QuickSort.
On the parallel case, we use different threads to do this.

In theory, according to this link, we get a reduction to `O(N)` on the average case using this Divide and Conquer approach.
In practice, on tests with a large number of points, using 8 threads is ~2 times faster than the sequential implementation.