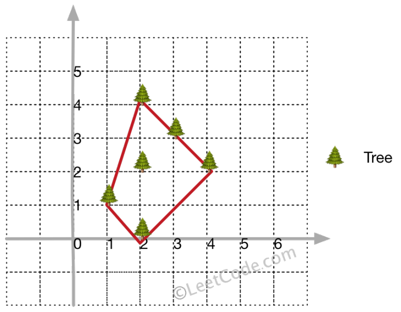
Erect the Fence

There are some trees, where each tree is represented by (x,y) coordinate in a two-dimensional garden. Your job is to fence the entire garden using the **minimum length** of rope as it is expensive. The garden is well fenced only if all the trees are enclosed. Your task is to help find the coordinates of trees which are exactly located on the fence perimeter.

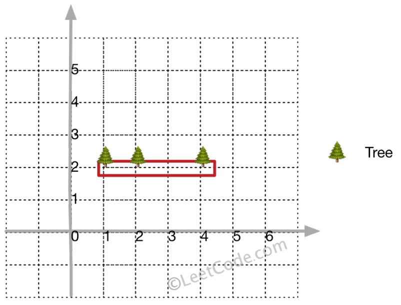
Example 1:

```
Input: [[1,1],[2,2],[2,0],[2,4],[3,3],[4,2]]
Output: [[1,1],[2,0],[4,2],[3,3],[2,4]]
Explanation:
```



Example 2:

```
Input: [[1,2],[2,2],[4,2]]
Output: [[1,2],[2,2],[4,2]]
Explanation:
```



```
Even you only have trees in a line, you need to use rope to enclose them.
```

Note:

1. All trees should be enclosed together. You cannot cut the rope to enclose trees that will separate them in more than one group.
2. All input integers will range from 0 to 100.
3. The garden has at least one tree.
4. All coordinates are distinct.
5. Input points have **NO** order. No order required for output.

Solution 1

There are couple of ways to solve Convex Hull problem. https://en.wikipedia.org/wiki/Convex_hull_algorithms
The following code implements `Gift wrapping aka Jarvis march` algorithm https://en.wikipedia.org/wiki/Gift_wrapping_algorithm and also added logic to handle case of `multiple Points in a line` because original Jarvis march algorithm assumes `no three points are collinear`.
It also uses knowledge in this problem https://leetcode.com/problems/convex-polygon . Disscussion: https://discuss.leetcode.com/topic/70706/beyond-my-knowledge-java-solution-with-in-line-explanation

```
public class Solution {
    public List<Point> outerTrees(Point[] points) {
        Set<Point> result = new HashSet<>();

        // Find the leftmost point
        Point first = points[0];
        int firstIndex = 0;
        for (int i = 1; i < points.length; i++) {
            if (points[i].x < first.x) {
                first = points[i];
                firstIndex = i;
            }
        }
        result.add(first);

        Point cur = first;
        int curIndex = firstIndex;
        do {
            Point next = points[0];
            int nextIndex = 0;
            for (int i = 1; i < points.length; i++) {
                if (i == curIndex) continue;
                int cross = crossProductLength(cur, points[i], next);
                if (nextIndex == curIndex || cross > 0 ||
                        // Handle collinear points
                        (cross == 0 && distance(points[i], cur) > distance(next, cu
r))) {
                    next = points[i];
                    nextIndex = i;
                }
            }
            // Handle collinear points
            for (int i = 0; i < points.length; i++) {
                if (i == curIndex) continue;
                int cross = crossProductLength(cur, points[i], next);
                if (cross == 0) {
                    result.add(points[i]);
                }
            }

            cur = next;
            curIndex = nextIndex;
        } while (curIndex != firstIndex);

        return new ArrayList<Point>(result);
    }

    private int crossProductLength(Point A, Point B, Point C) {
        // Get the vectors' coordinates.
        int BAx = A.x - B.x;
        int BAy = A.y - B.y;
        int BCx = C.x - B.x;
        int BCy = C.y - B.y;

        // Calculate the Z coordinate of the cross product.
        return (BAx * BCy - BAy * BCx);
    }

    private int distance(Point p1, Point p2) {
        return (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
    }
}
```

written by shawngao original link here

Solution 2

The trick is that once all points are sorted by polar angle with respect to the reference point:

- For collinear points in the begin positions, make sure they are sorted by distance to reference point in **ascending** order.
- For collinear points in the end positions, make sure they are sorted by distance to reference point in **descending** order.

For example:
`(0, 0), (2, 0), (3, 0), (3, 1), (3, 2), (2, 2), (1, 2), (0, 2), (0, 1)`
These points are sorted by polar angle
The reference point (bottom left point) is `(0, 0)`

- In the begin positions `(0, 0)` collinear with `(2, 0), (3, 0)` sorted by distance to reference point in **ascending** order.
- In the end positions `(0, 0)` collinear with `(0, 2), (0, 1)` sorted by distance to reference point in **descending** order.

Now we can run the standard Graham scan to give us the desired result.

```
public class Solution {

    public List<Point> outerTrees(Point[] points) {
        if (points.length <= 1)
            return Arrays.asList(points);
        sortByPolar(points, bottomLeft(points));
        Stack<Point> stack = new Stack<>();
        stack.push(points[0]);
        stack.push(points[1]);
        for (int i = 2; i < points.length; i++) {
            Point top = stack.pop();
            while (ccw(stack.peek(), top, points[i]) < 0)
                top = stack.pop();
            stack.push(top);
            stack.push(points[i]);
        }
        return new ArrayList<>(stack);
    }

    private static Point bottomLeft(Point[] points) {
        Point bottomLeft = points[0];
        for (Point p : points)
            if (p.y < bottomLeft.y || p.y == bottomLeft.y && p.x < bottomLeft.x)
                bottomLeft = p;
        return bottomLeft;
    }

    /**
     * @return positive if counter-clockwise, negative if clockwise, 0 if collinear
     */
    private static int ccw(Point a, Point b, Point c) {
        return a.x * b.y - a.y * b.x + b.x * c.y - b.y * c.x + c.x * a.y - c.y * a.
x;
    }

    /**
     * @return distance square of |p - q|
     */
    private static int dist(Point p, Point q) {
        return (p.x - q.x) * (p.x - q.x) + (p.y - q.y) * (p.y - q.y);
    }

    private static void sortByPolar(Point[] points, Point r) {
        Arrays.sort(points, (p, q) -> {
            int compPolar = ccw(p, r, q);
            int compDist = dist(p, r) - dist(q, r);
            return compPolar == 0 ? compDist : compPolar;
        });
        // find collinear points in the end positions
        Point p = points[0], q = points[points.length - 1];
        int i = points.length - 2;
        while (i >= 0 && ccw(p, q, points[i]) == 0)
            i--;
        // reverse sort order of collinear points in the end positions
        for (int l = i + 1, h = points.length - 1; l < h; l++, h--) {
            Point tmp = points[l];
            points[l] = points[h];
            points[h] = tmp;
        }
    }
}
```

written by yuxiangmusic original link here

Solution 3

Based on the formula for the signed area of a triangle, we can find whether a triangle PQR has vertices which are counter-clockwise (sign 1), collinear (sign 0), or clockwise (sign -1).

We will now perform the AM-Chain algorithm for finding the lower and upper hulls which together form the convex hull of these points.

To find the lower hull of points, we process the points in sorted order. Focus on the function `drive`. Our loop invariant is that we started the function `drive` with a lower hull, and we return a lower hull. This answer must include the new right-most point `r` as it cannot be contained by some points below it. During the while loop, whenever our last turn XYZ was clockwise, the middle point Y cannot be part of the lower hull, as it is contained by WXZ (where W is the point in the hull before X.)

We can do this process again with the points sorted in reverse to find the upper hull. Both hulls combined is the total convex hull as desired.

```
def outerTrees(self, A):
    def sign(p, q, r):
        return cmp((p.x - r.x)*(q.y - r.y), (p.y - r.y)*(q.x - r.x))

    def drive(hull, r):
        hull.append(r)
        while len(hull) >= 3 and sign(*hull[-3:]) < 0:
            hull.pop(-2)
        return hull

    A.sort(key = lambda p: (p.x, p.y))
    lower = reduce(drive, A, [])
    upper = reduce(drive, A[::-1], [])
    return list(set(lower + upper))
```

written by awice original link here