CSC 225 - Summer 2019

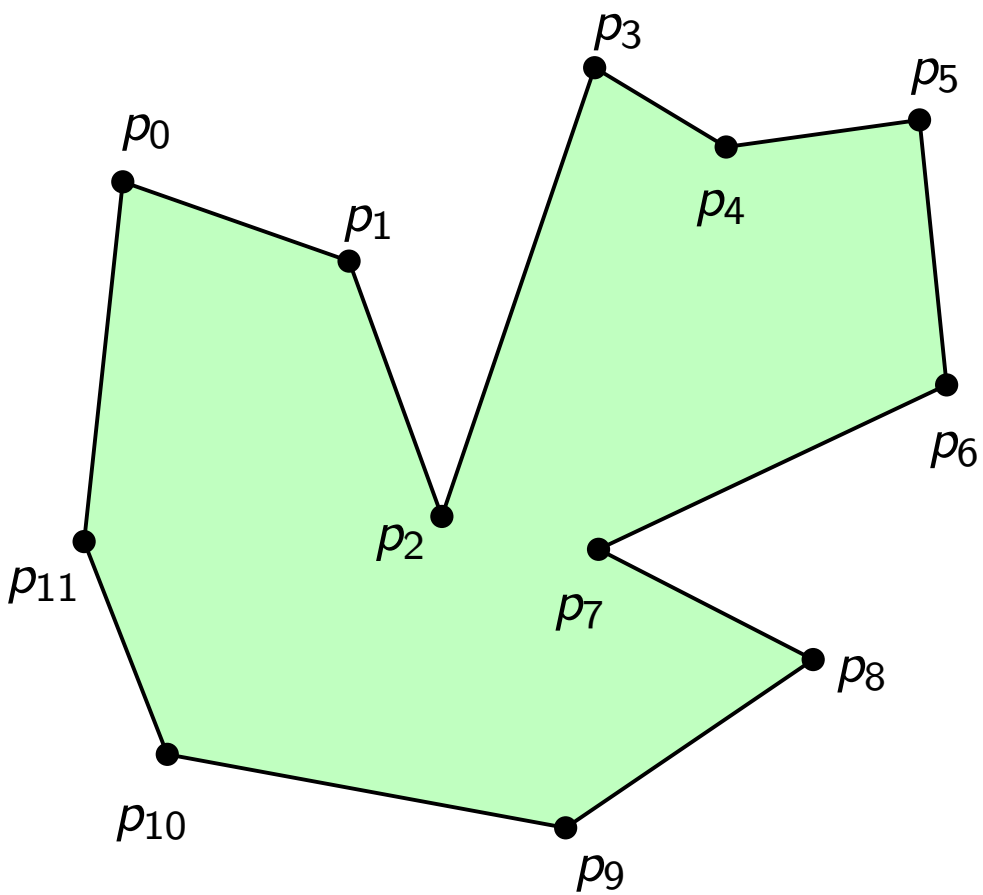
Algorithms for Convex Hulls

Bill Bird

Department of Computer Science
University of Victoria

May 23, 2019

Polygons (1)

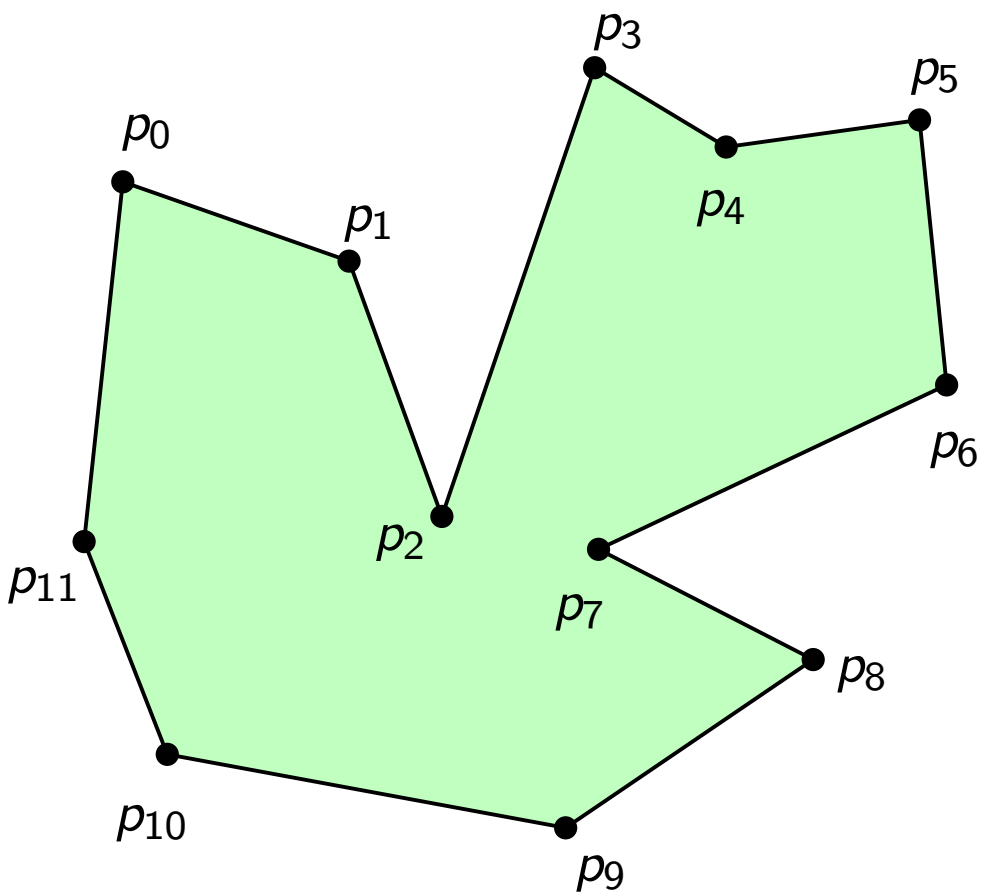


A **polygon** is a sequence of points

$$p_0, \quad p_1, \quad p_2, \quad \ldots, \quad p_k, \quad p_0$$

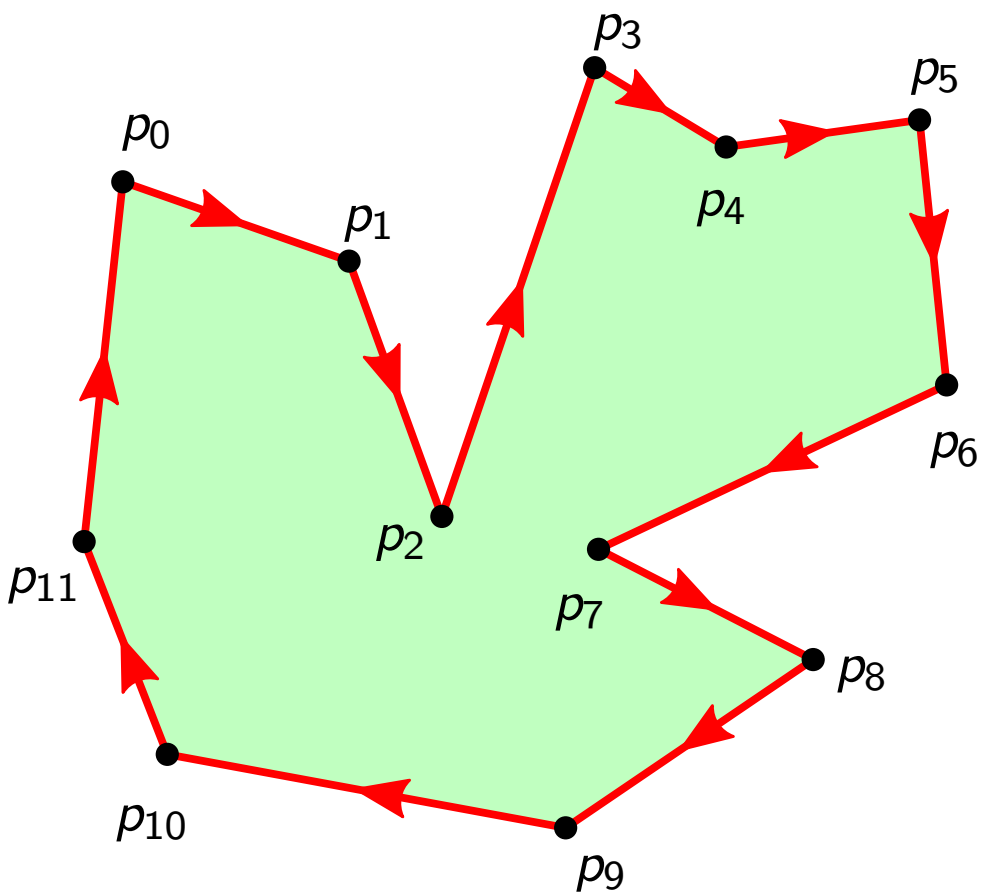
that describes a closed shape in 2d space.

Polygons (2)



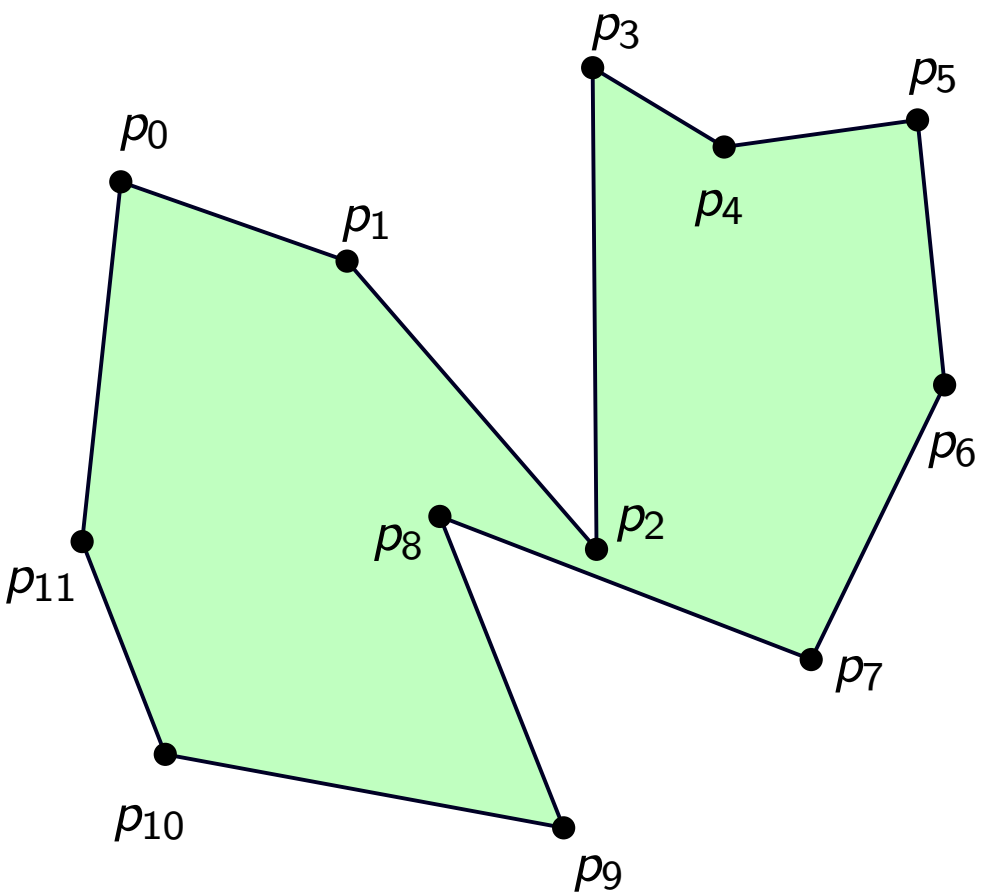
The ordering of the sequence is significant. Notice that, for the polygon above, the ordering p_0 through p_{11} corresponds to a walk around the outside of the polygon.

Polygons (3)



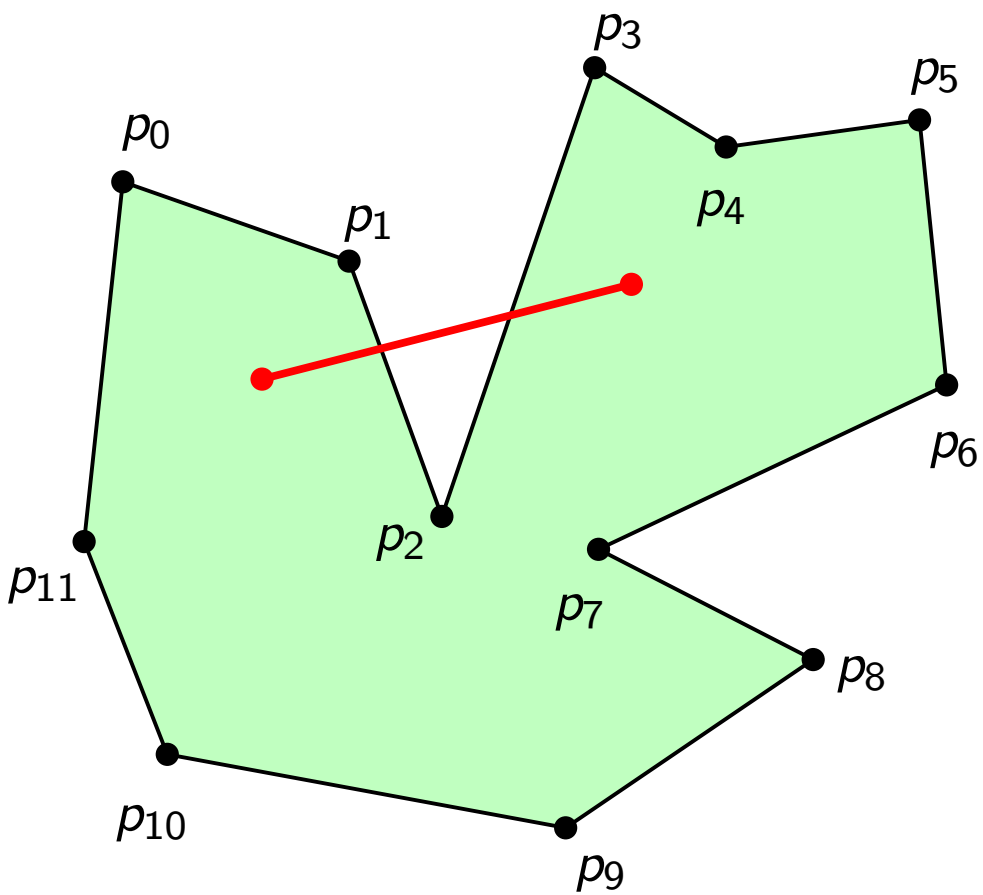
Notice that if we walk along the edges of the polygon in the order $p_0, p_1, \ldots, p_{11}, p_0$, the inside of the polygon is always to the **right** of the direction of travel.

Polygons (4)



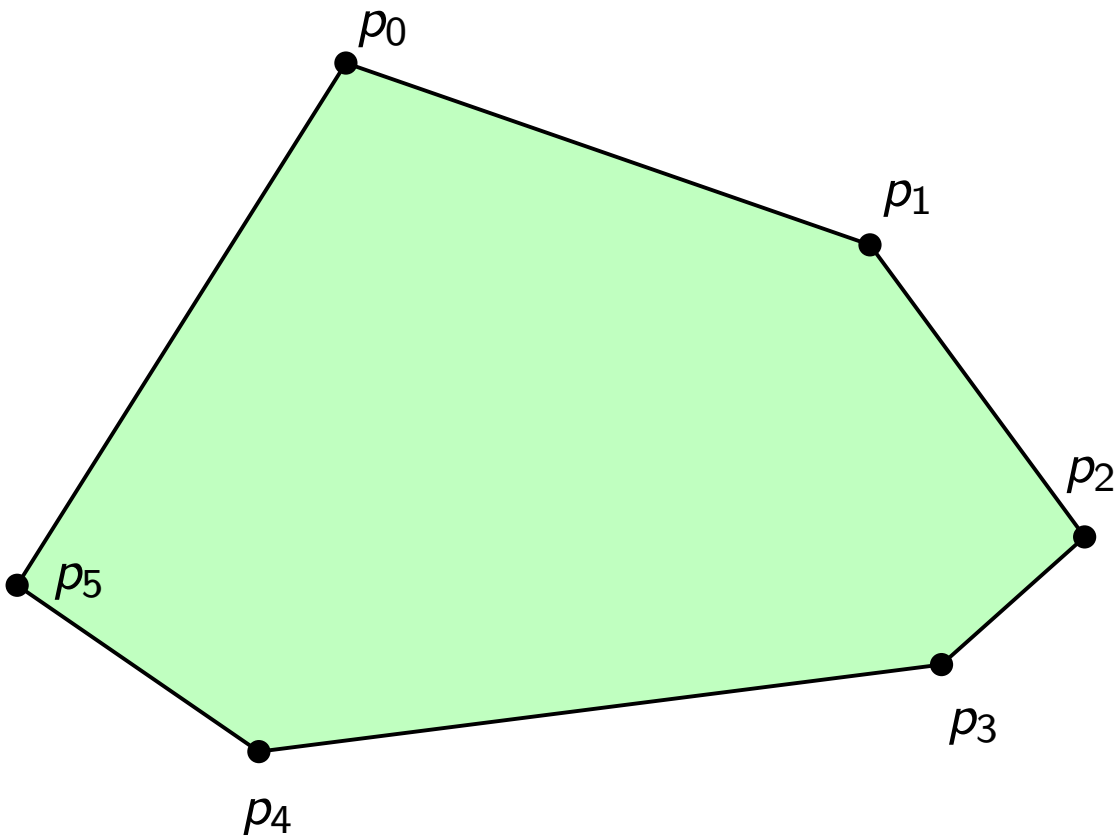
If the same set of points is numbered in a different order, a distinct shape will be produced.

Polygons (5)



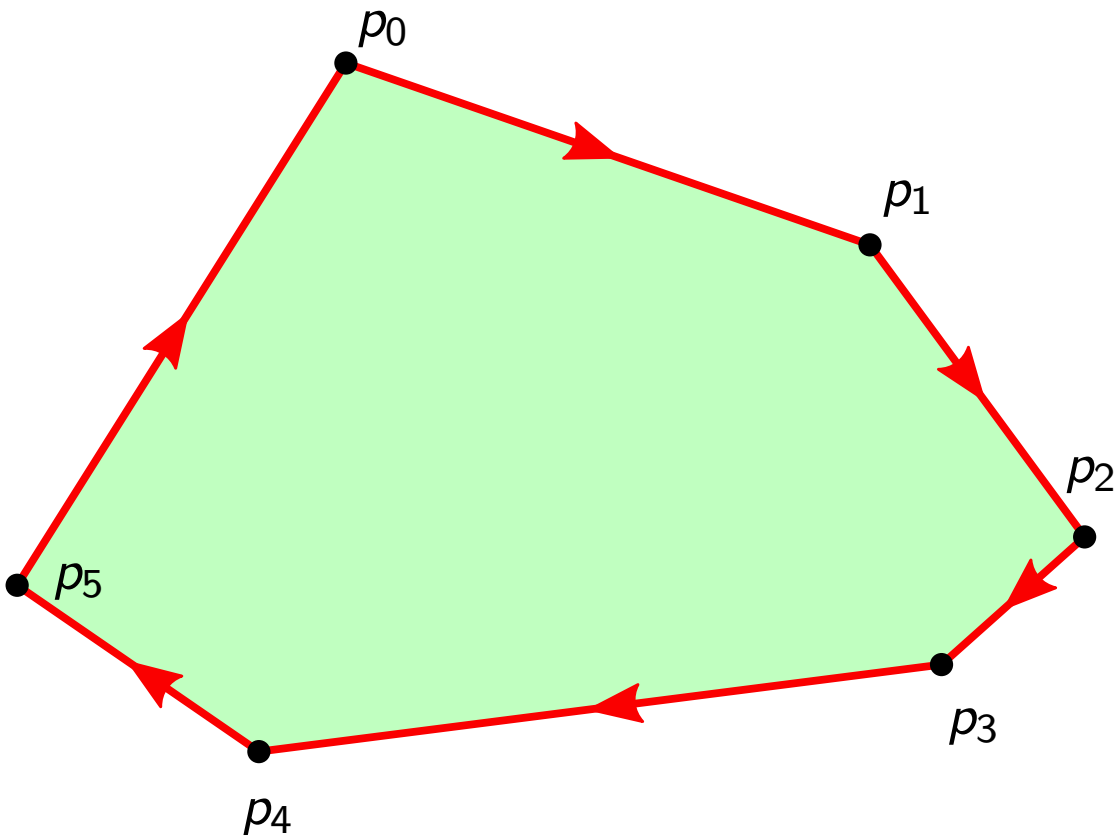
The polygon above is **concave**. Notice that the line connecting the two red points (which both lie inside the polygon's area) does not lie entirely inside the polygon.

Polygons (6)



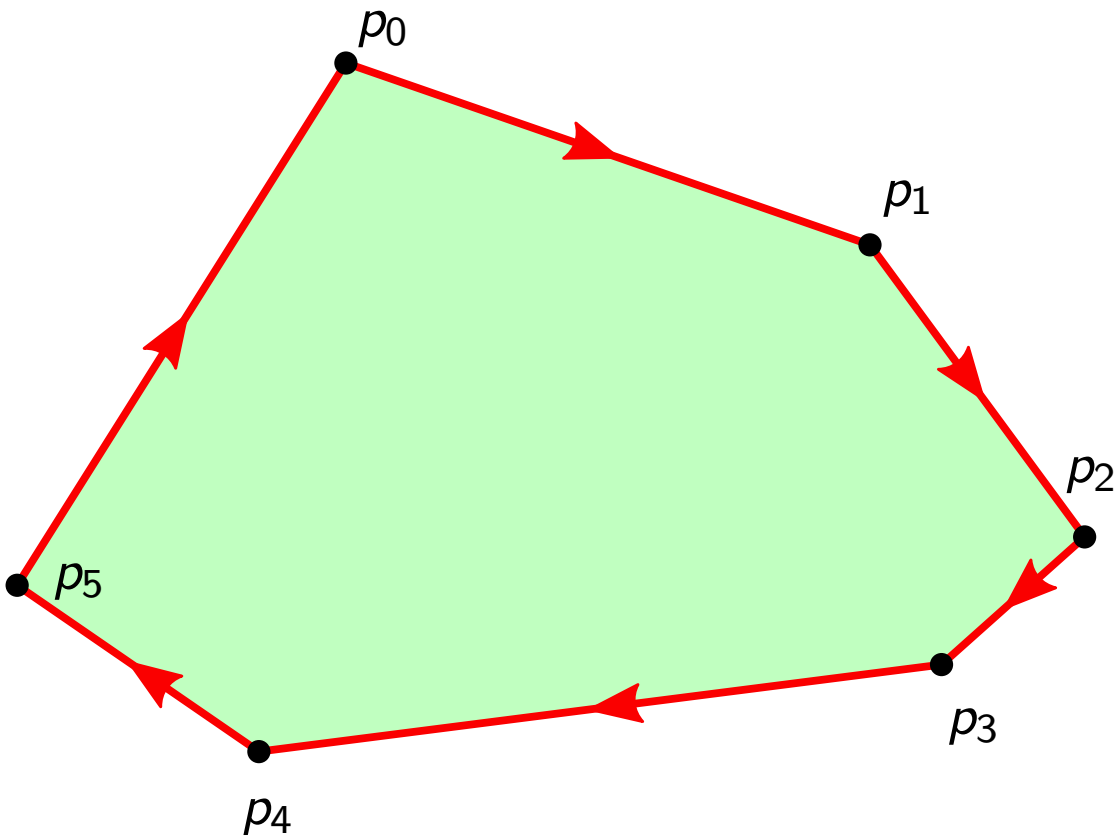
A **convex polygon** is a polygon with no concavities: For every pair of points q and r inside (or on the boundary of) the polygon, every point on the segment qr lies inside or on the boundary of the polygon.

Polygons (7)



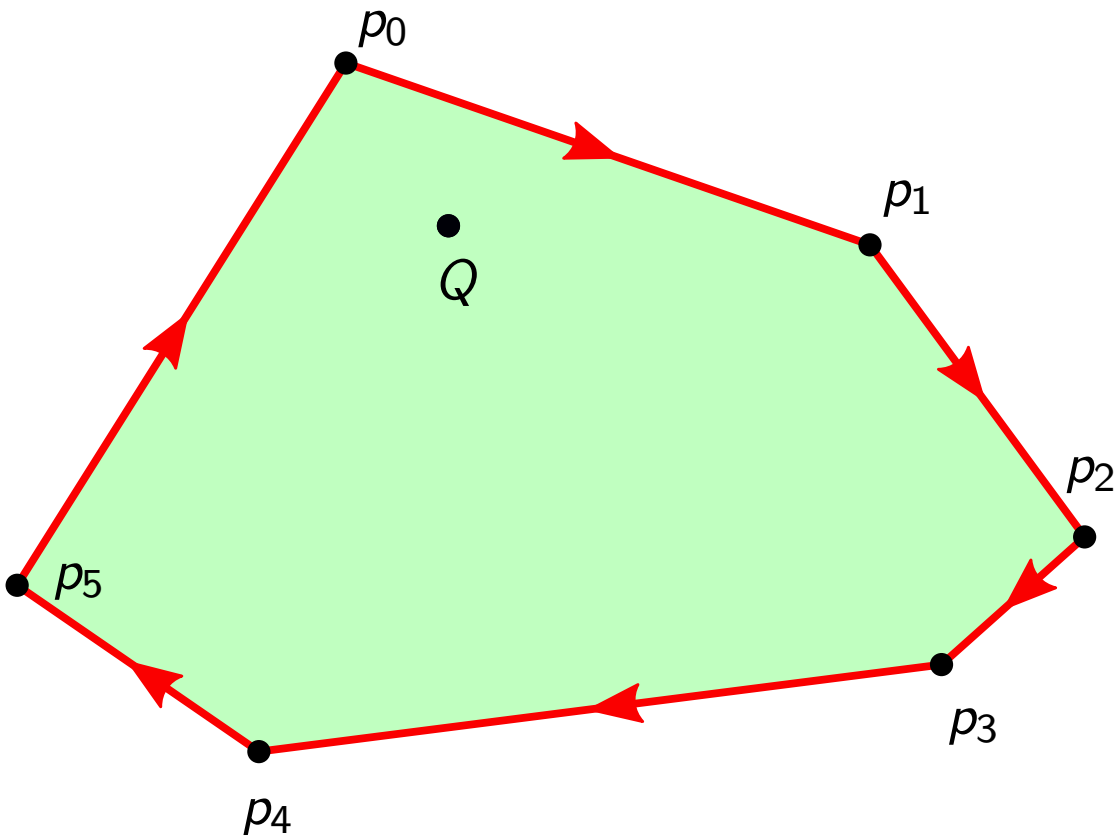
In the numbering given above, walking around the polygon in order produces a clockwise traversal (where the inside of the polygon is always on the right).

Polygons (8)



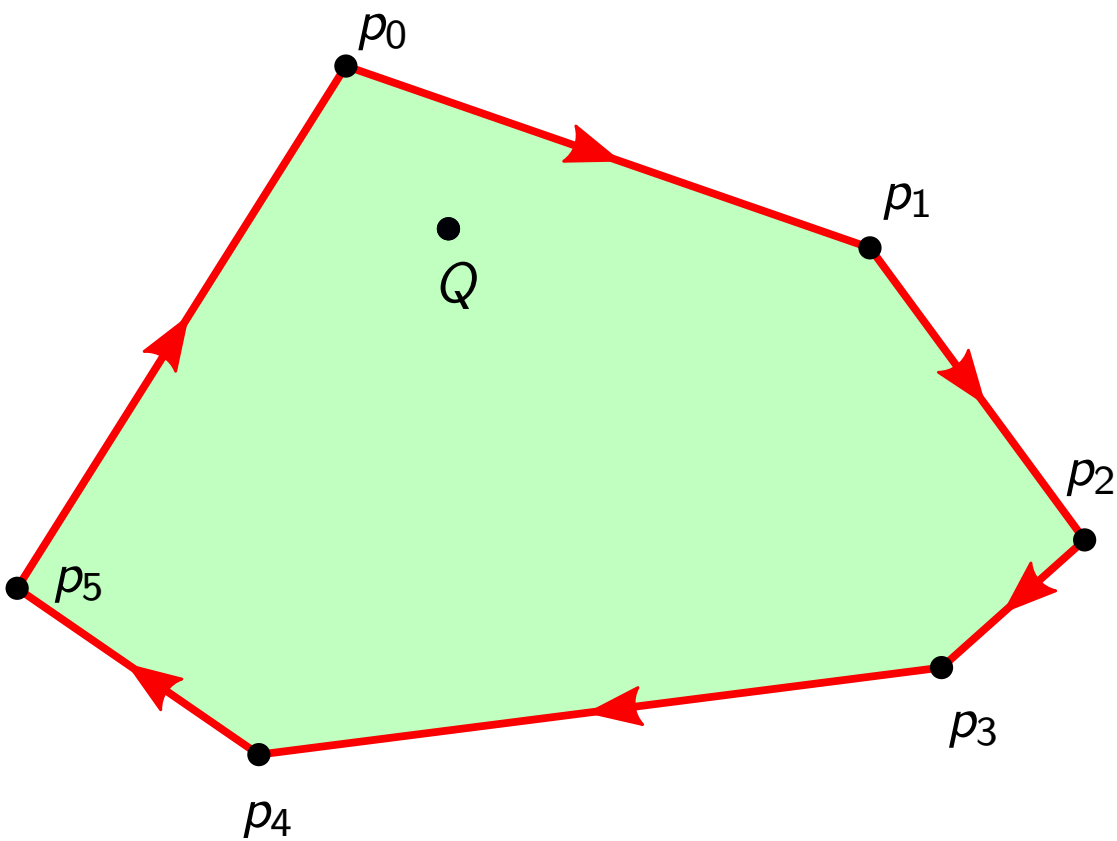
(Note that we could also have written the polygon with a reversed numbering where the traversal is counterclockwise and the inside is on the left)

Polygons (9)



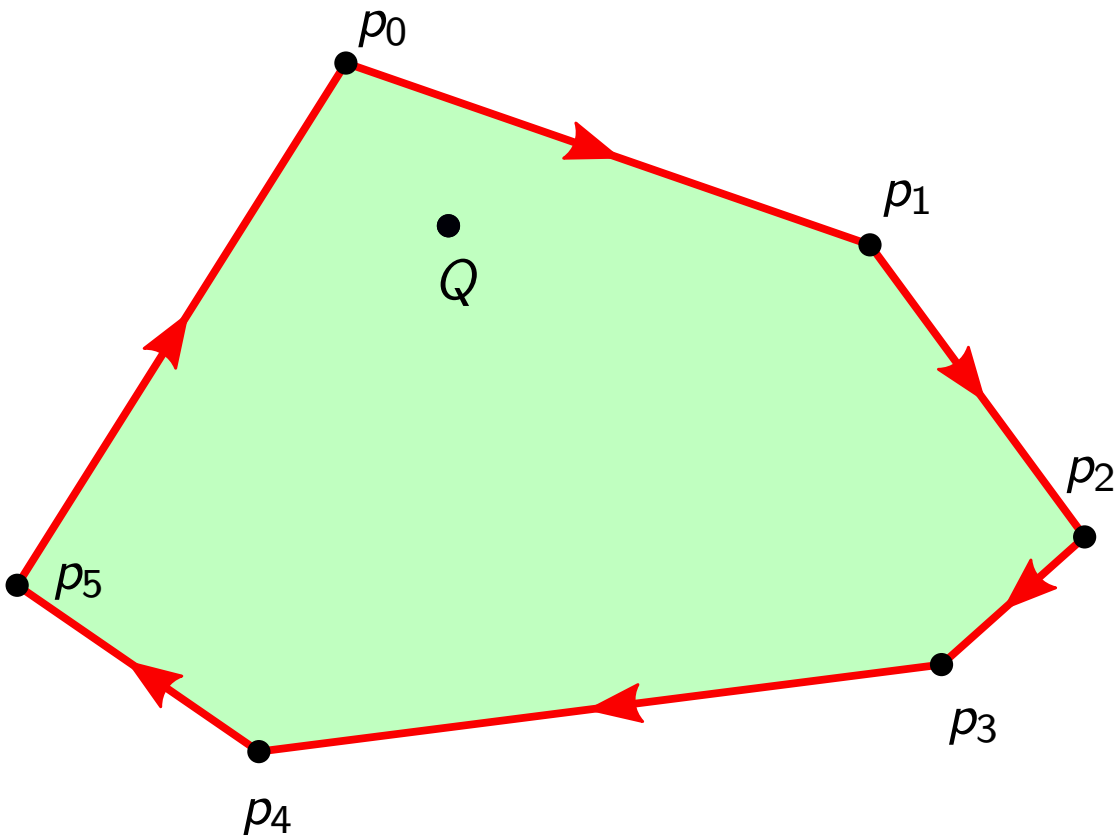
Question: Consider the point Q above. How can we verify that Q lies inside of the polygon (using only the points $p_0, p_1, \ldots, p_5$)?

Polygons (10)



Answer: We know that the "inside" is always to the right of every line segment on the border, so if Q lies to the right of every line segment $\overline{p_i p_{i+1}}$ (and $\overline{p_5 p_0}$), Q must be an interior point.

Polygons (11)



Conversely, if Q lies to the left of any border segment, it must be outside the polygon. In the diagram above, Q lies to the right of every segment except $\overline{p_5 p_0}$.

Polygons (12)

Given three points $P = (P_x, P_y)$, $Q = (Q_x, Q_y)$ and $R = (R_x, R_y)$, define

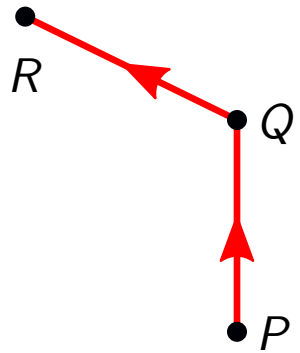
$$T(P, Q, R) = (Q_y - P_y)(R_x - P_x) + (P_x - Q_x)(R_y - P_y).$$

Then,

- $T(P, Q, R) < 0$ if the sequence P, Q, R makes a left turn.
- $T(P, Q, R) > 0$ if the sequence P, Q, R makes a right turn.
- $T(P, Q, R) = 0$ if the sequence P, Q, R is a straight line.

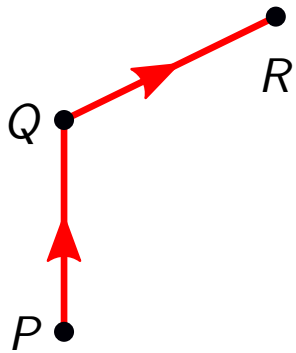
Polygons (13)

/



$T(P, Q, R) < 0$
(left turn)

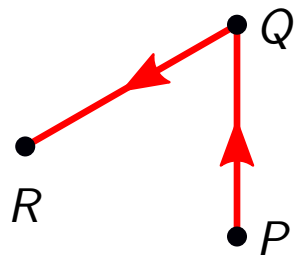
$T(P, Q, R) = 0$
(straight line)



$T(P, Q, R) > 0$
(right turn)

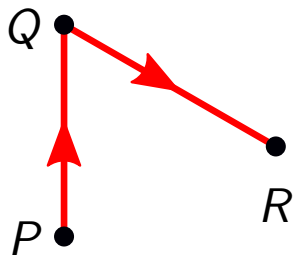
The sign of $T(P, Q, R)$ can be used to determine if R lies to the left or right of the directed segment $P \rightarrow Q$ (or if P, Q and R are co-linear).

Polygons (14)



$T(P, Q, R) < 0$
(left turn)

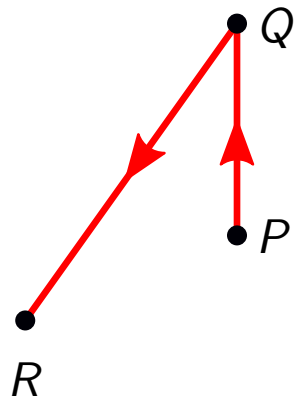
$T(P, Q, R) = 0$
(straight line)



$T(P, Q, R) > 0$
(right turn)

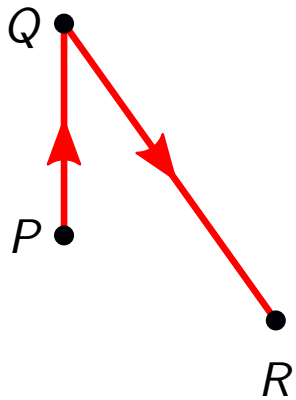
Note that, other than its relationship to the line segment $P \rightarrow Q$, the relative position of R is irrelevant.

Polygons (15)



$T(P, Q, R) < 0$
(left turn)

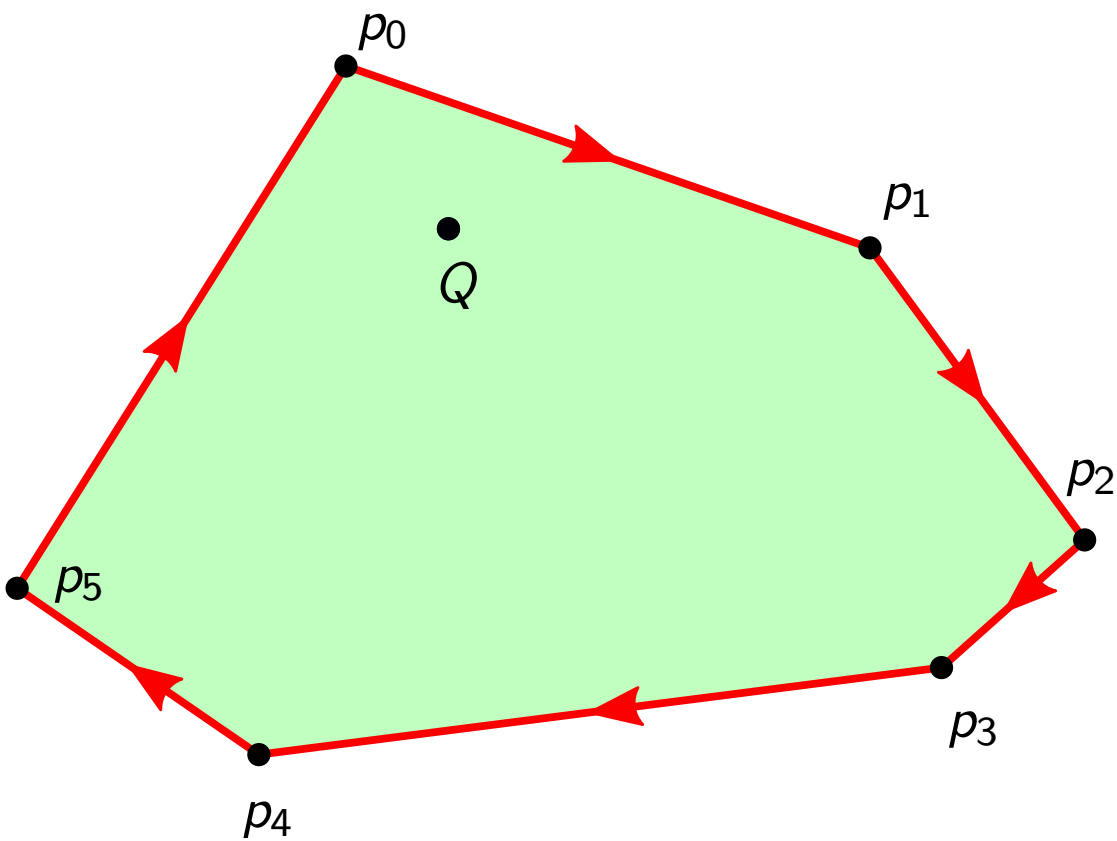
$T(P, Q, R) = 0$
(straight line)



$T(P, Q, R) > 0$
(right turn)

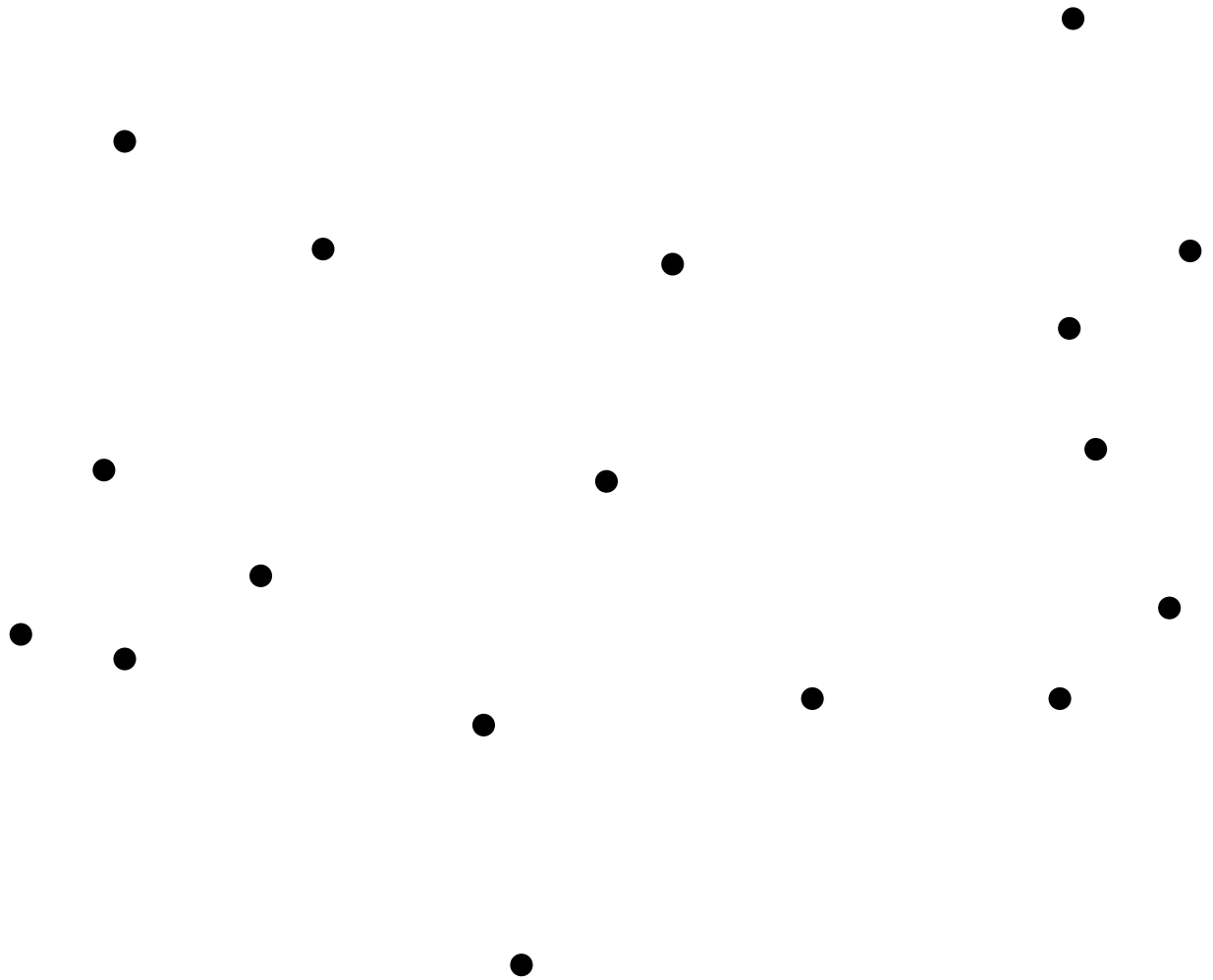
Also observe that the function T can be evaluated in constant time. (A pre-written version of T has been provided on the assignment for convenience as the `Point2d.chirality` method).

Polygons (16)



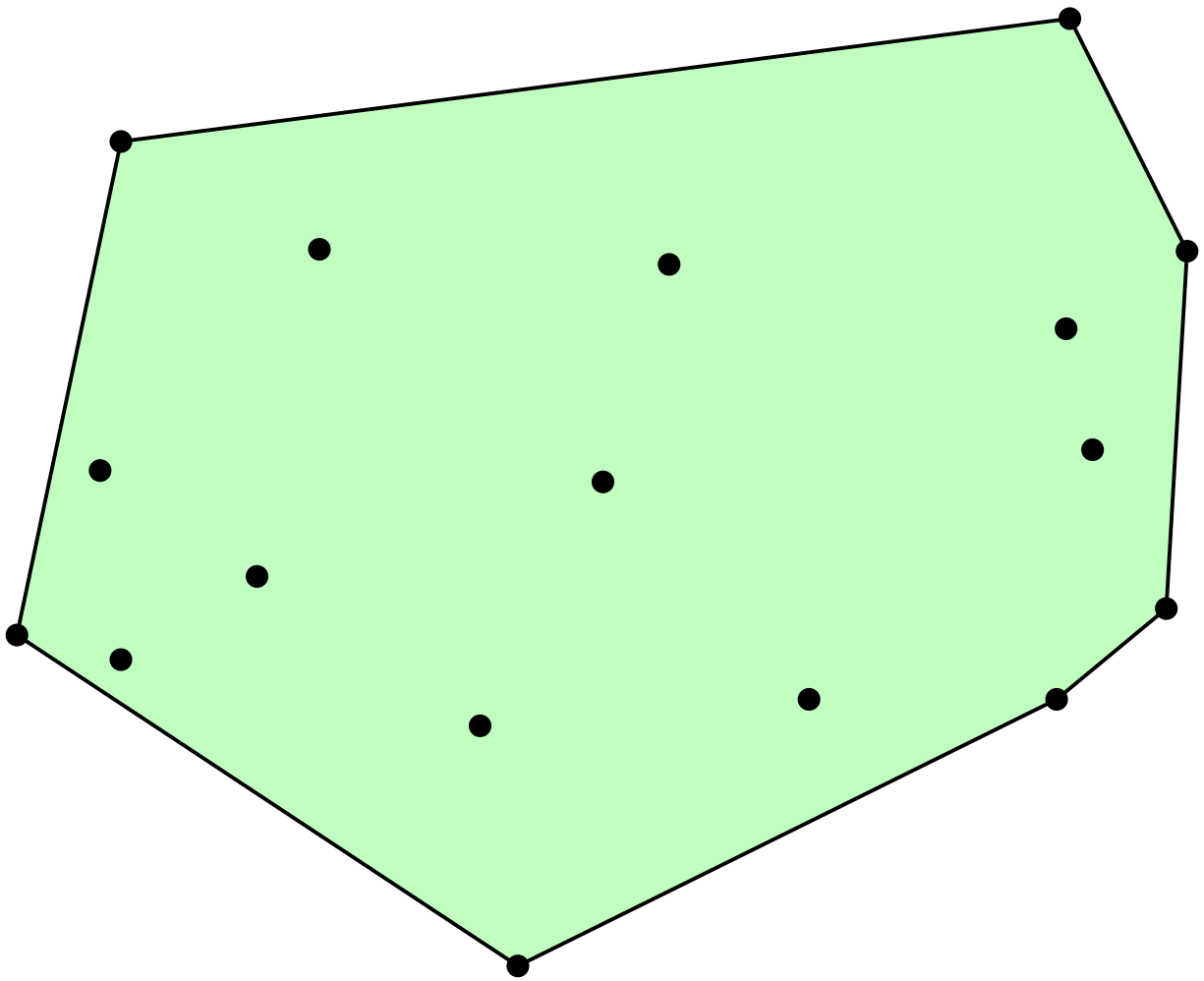
Therefore, given a list of k points comprising a convex polygon, we can test whether a point R lies inside or outside the polygon in $O(k)$ time by testing whether R lies on the same side of each line segment in the polygon.

Convex Hulls (1)



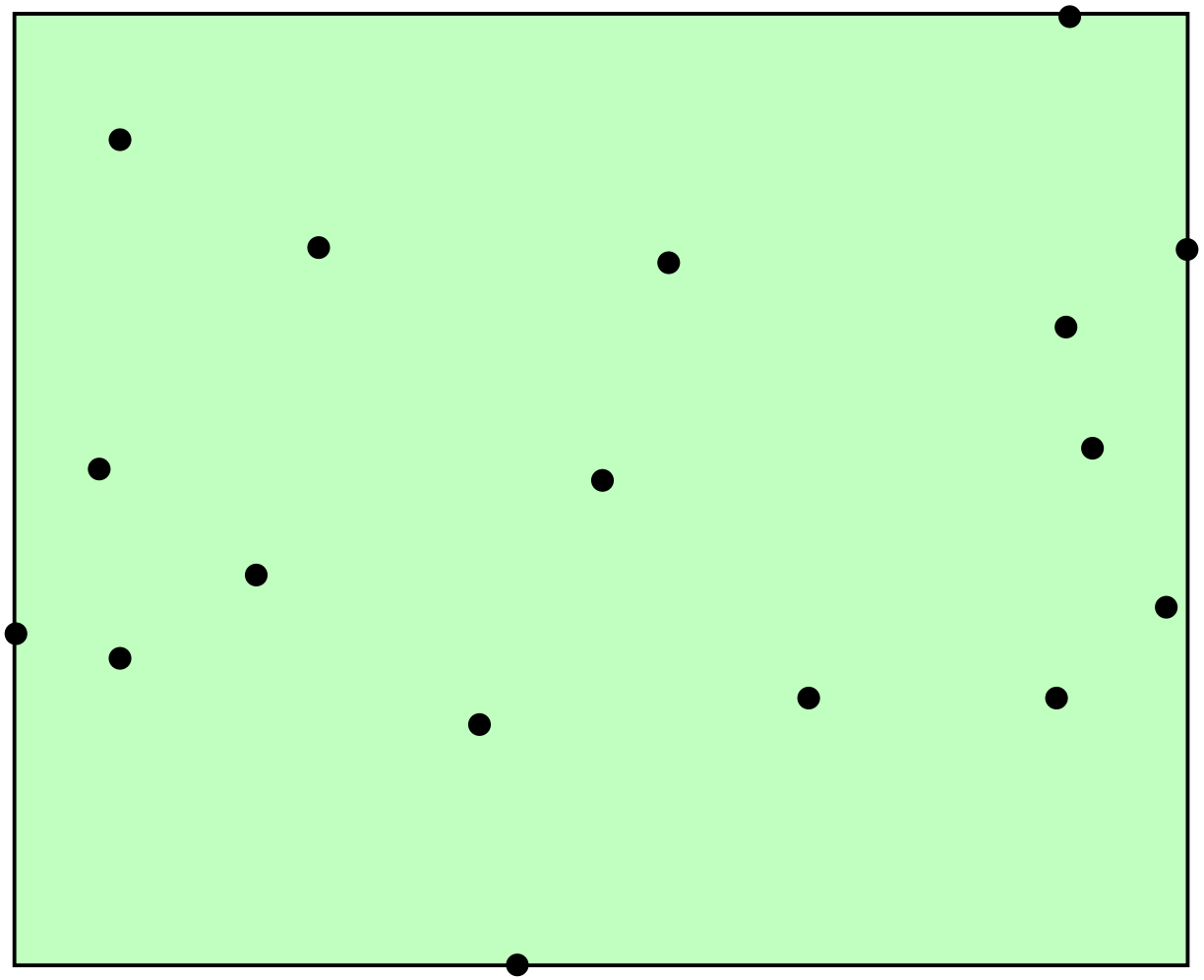
Consider the collection of points above.

Convex Hulls (2)



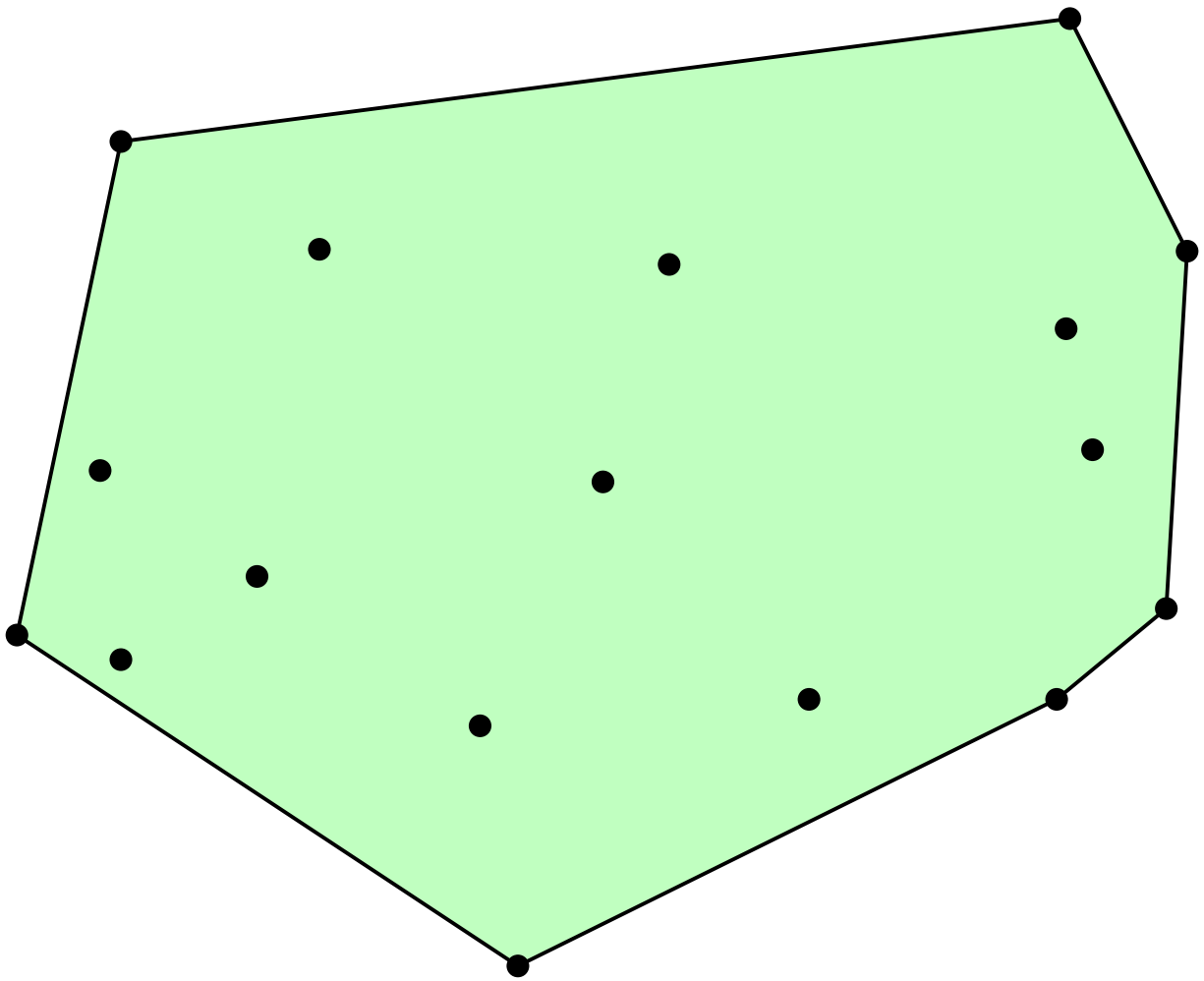
The **convex hull** of a set S of points in the plane is the **lowest-perimeter convex polygon** P such that every point $p \in S$ is either on the boundary or contained in the interior of P.

Convex Hulls (3)



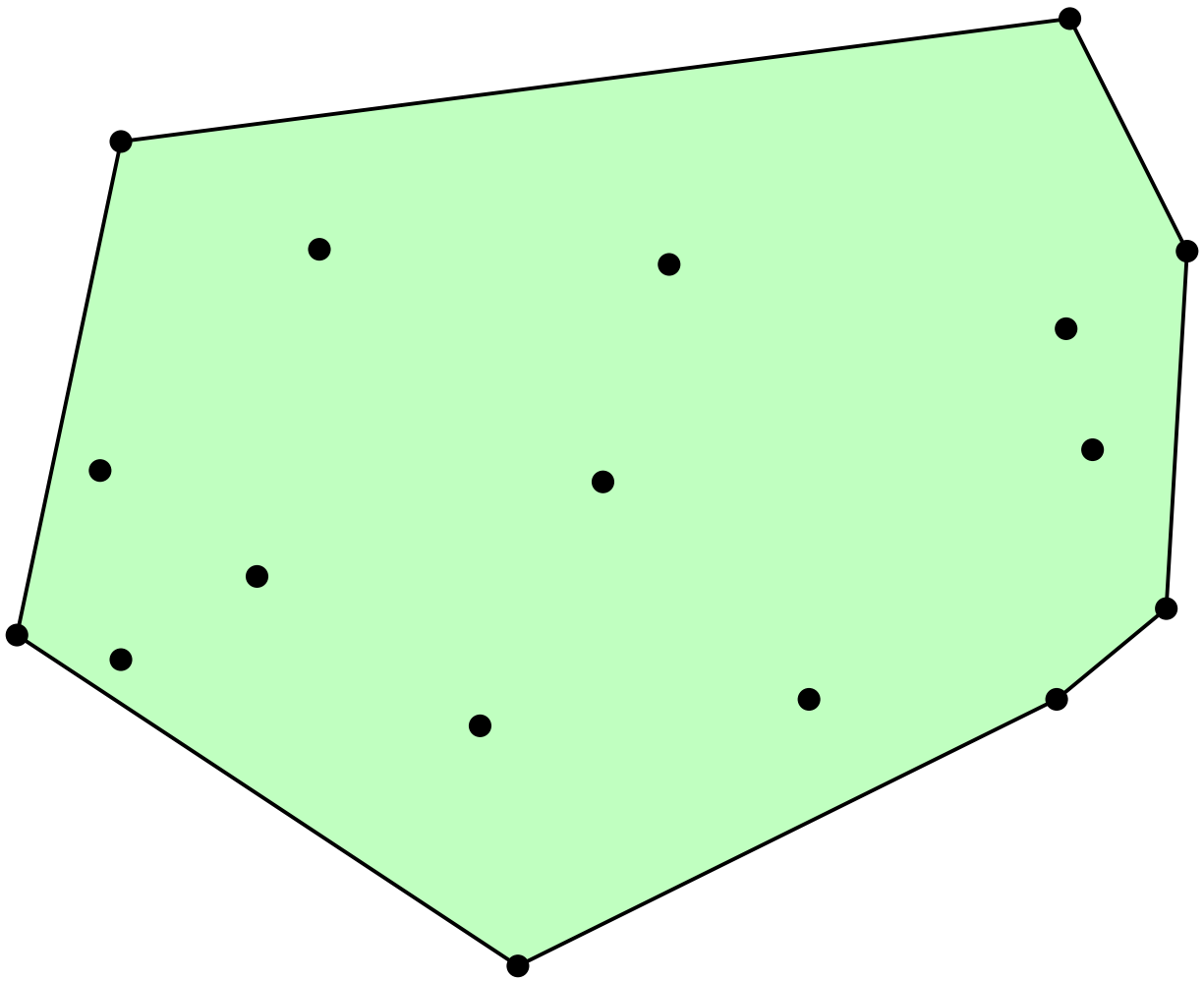
Note that we can easily create convex polygons that contain every point in S, but most polygons, like the rectangle above, will have 'wasted space' that does not contain any points.

Convex Hulls (4)



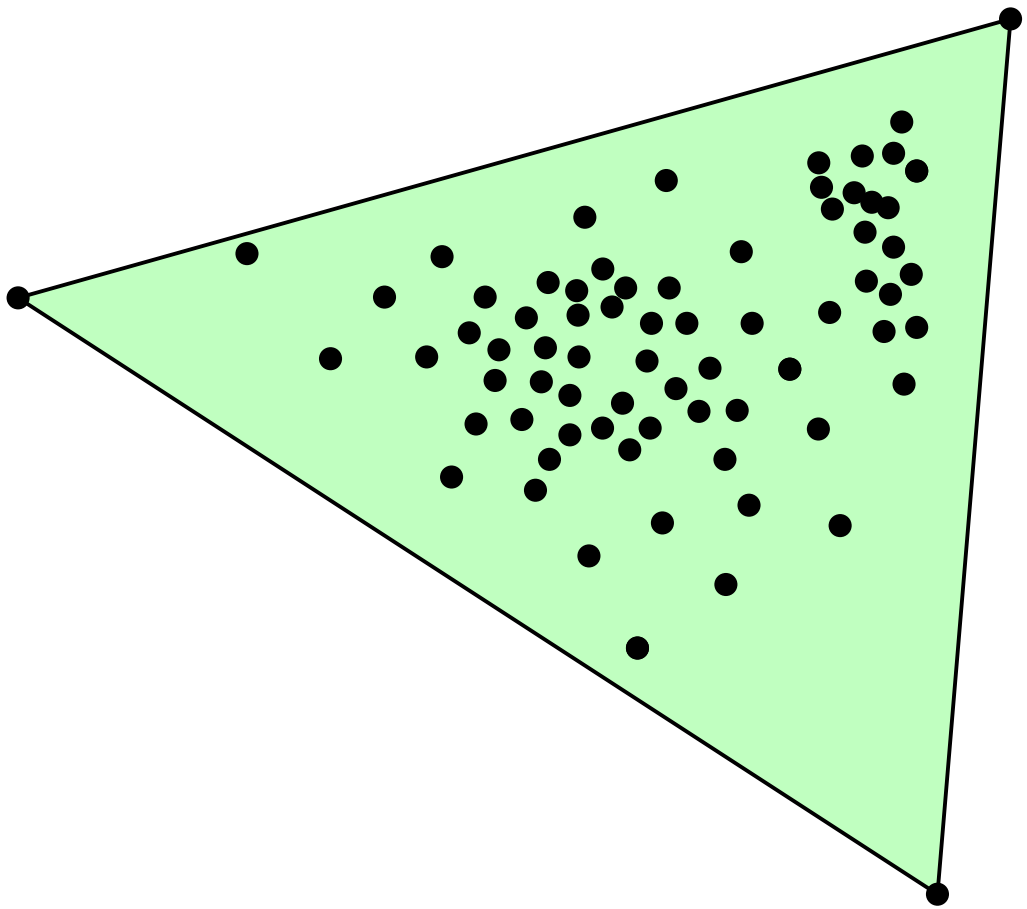
One defining feature of the convex hull P of a set S is that every vertex of the polygon must be a point in S.

Convex Hulls (5)



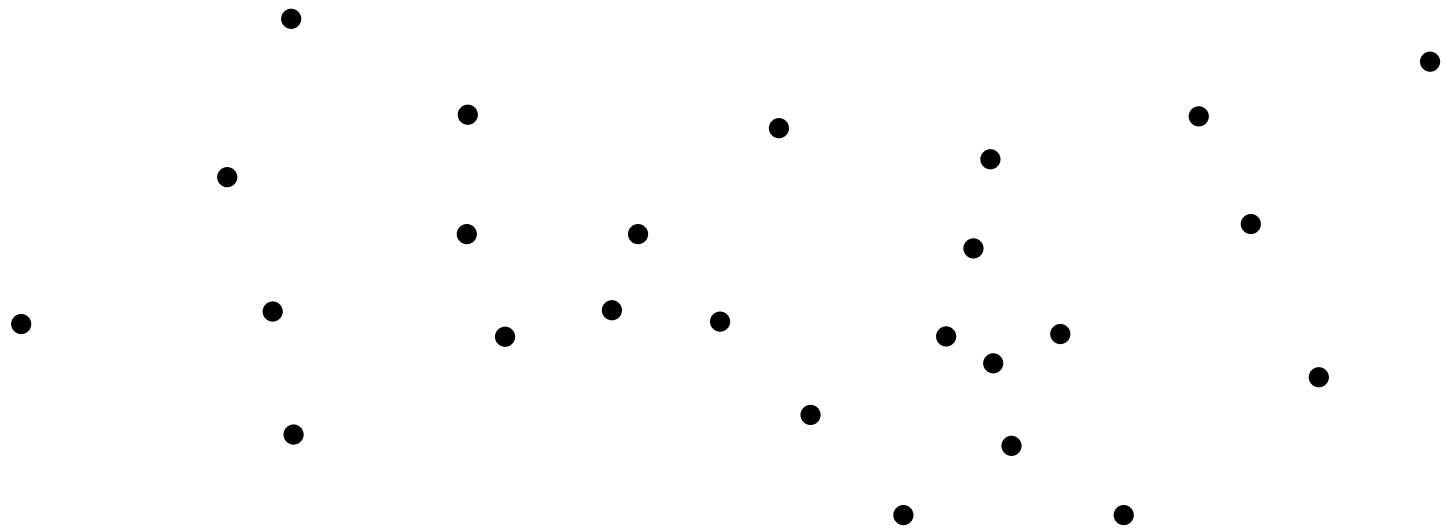
Question: Given a set of n points, what is the minimum number of points on the convex hull?

Convex Hulls (6)



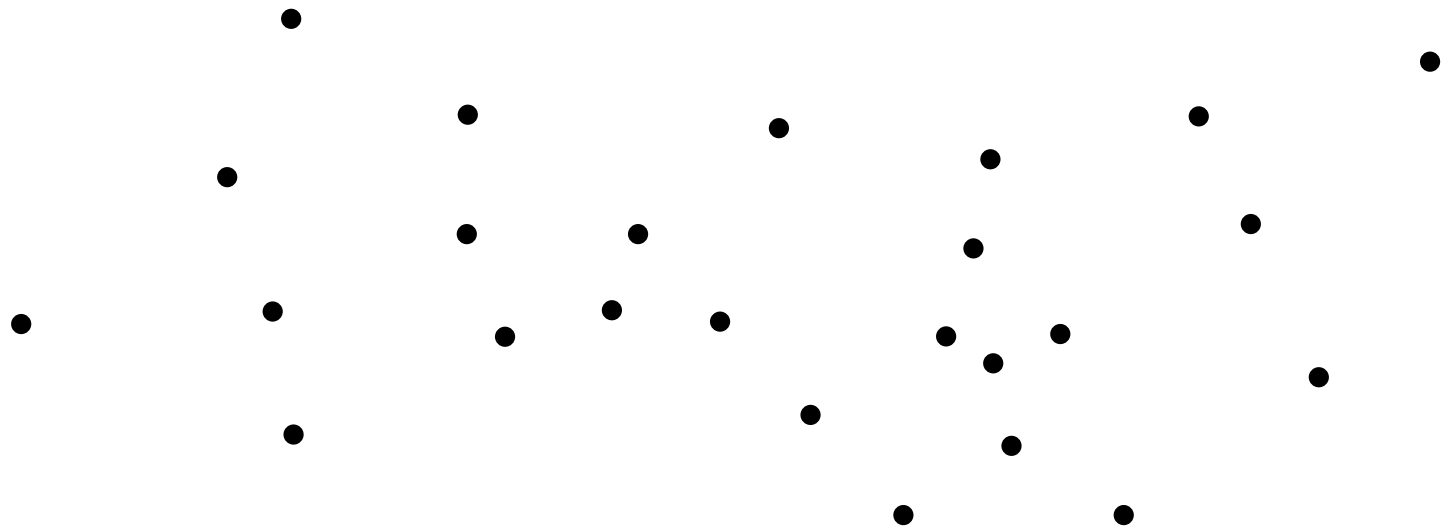
Even for massive point sets, the convex hull may have only 3 points (or only 2 in degenerate cases).

Basic Algorithms (1)



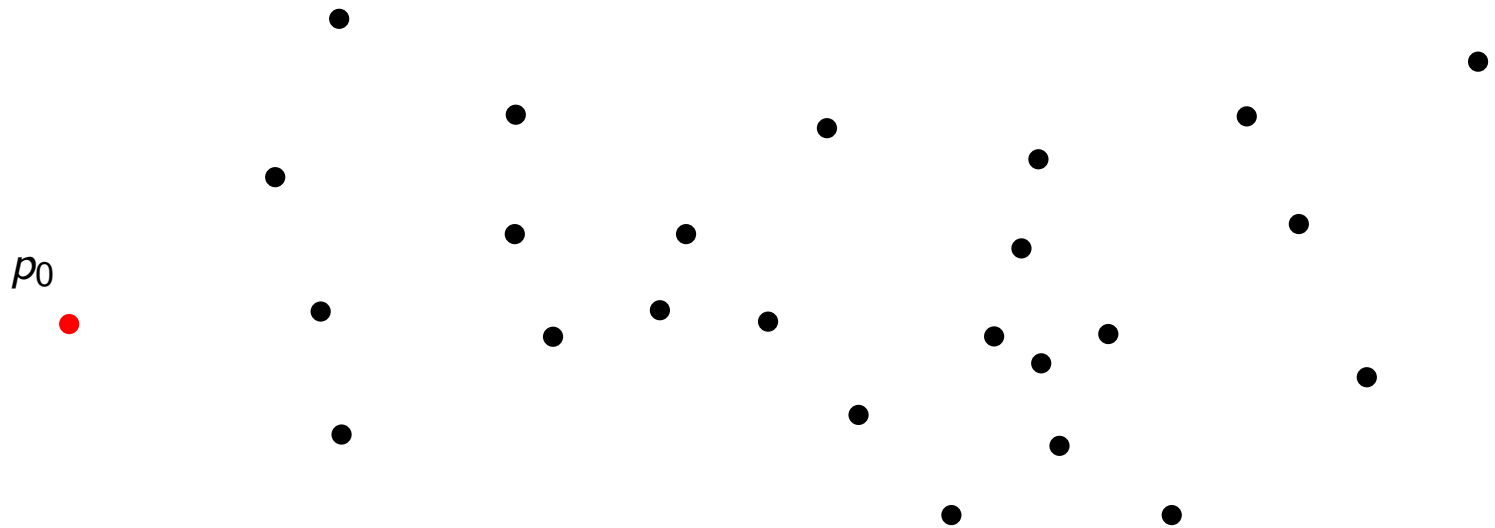
Task: Design an algorithm that takes a list L of n points and returns an ordered sequence containing the vertices of the convex hull of S.

Basic Algorithms (2)



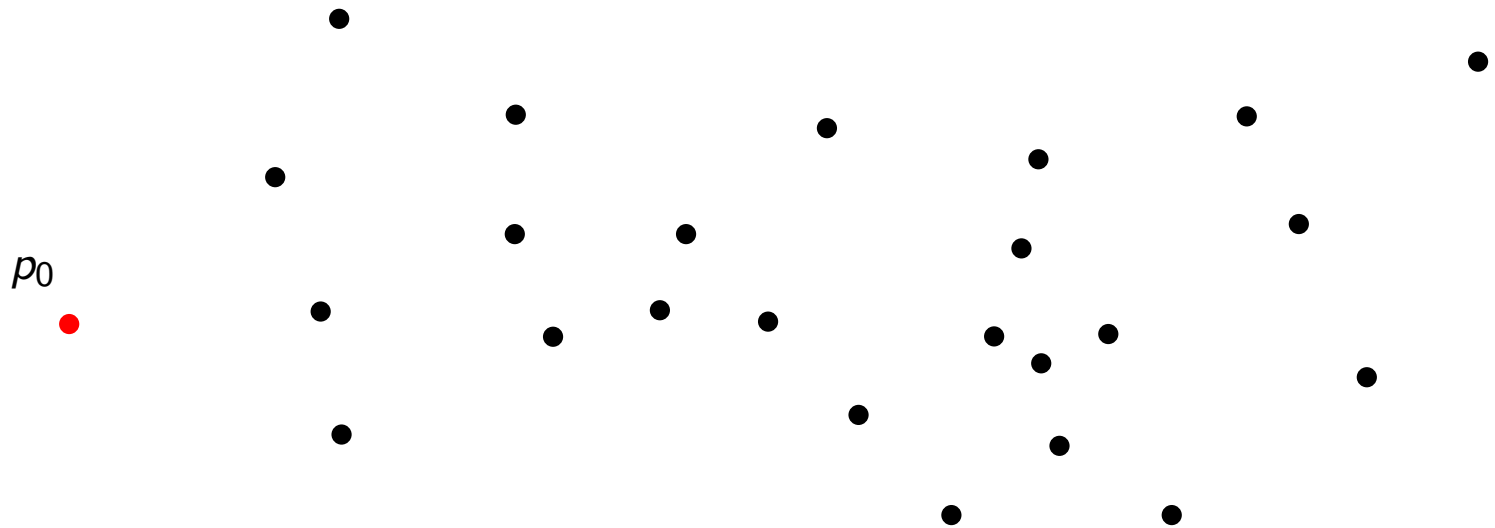
Observation: The point with the smallest x-coordinate (i.e. that left-most point) must always be part of the convex hull.

Basic Algorithms (3)



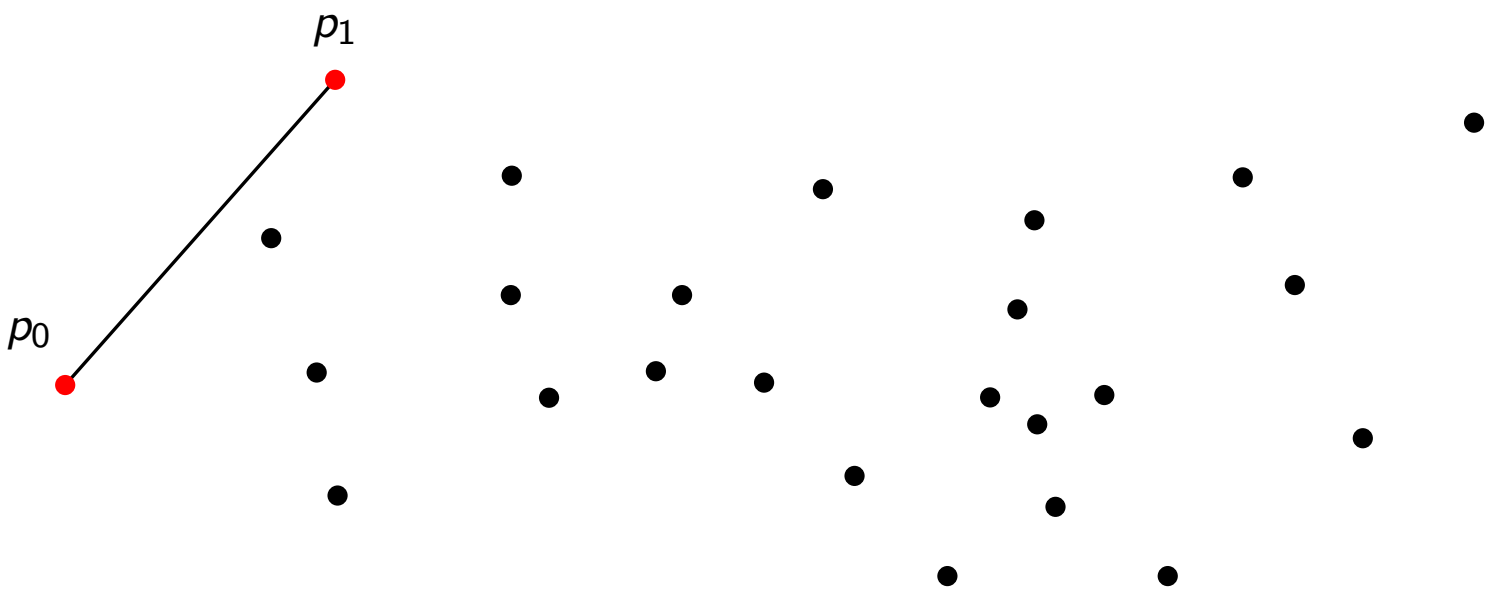
Idea: Start by finding the left-most point, then build the convex hull from there.

Basic Algorithms (4)



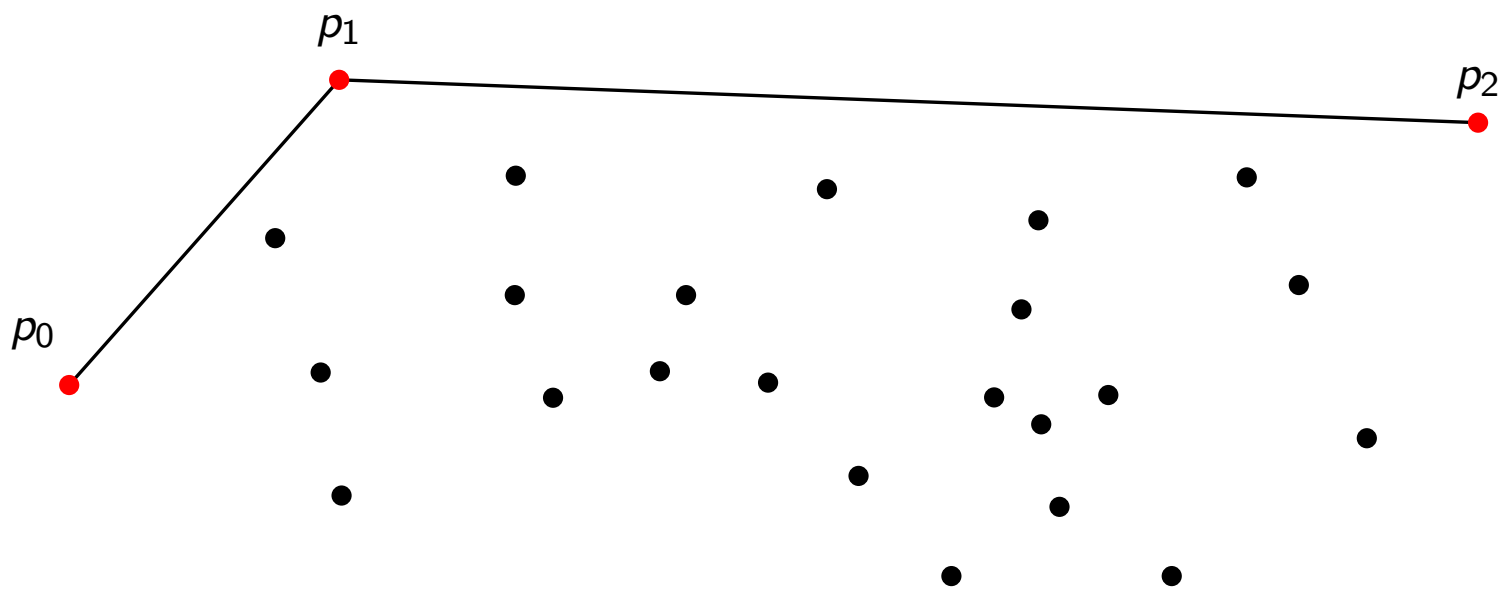
Observation: The next point p_1 needs to have the property that every point is on the right-hand side of $\overline{p_0 p_1}$. We can find such a point by trying all possibilities.

Basic Algorithms (5)



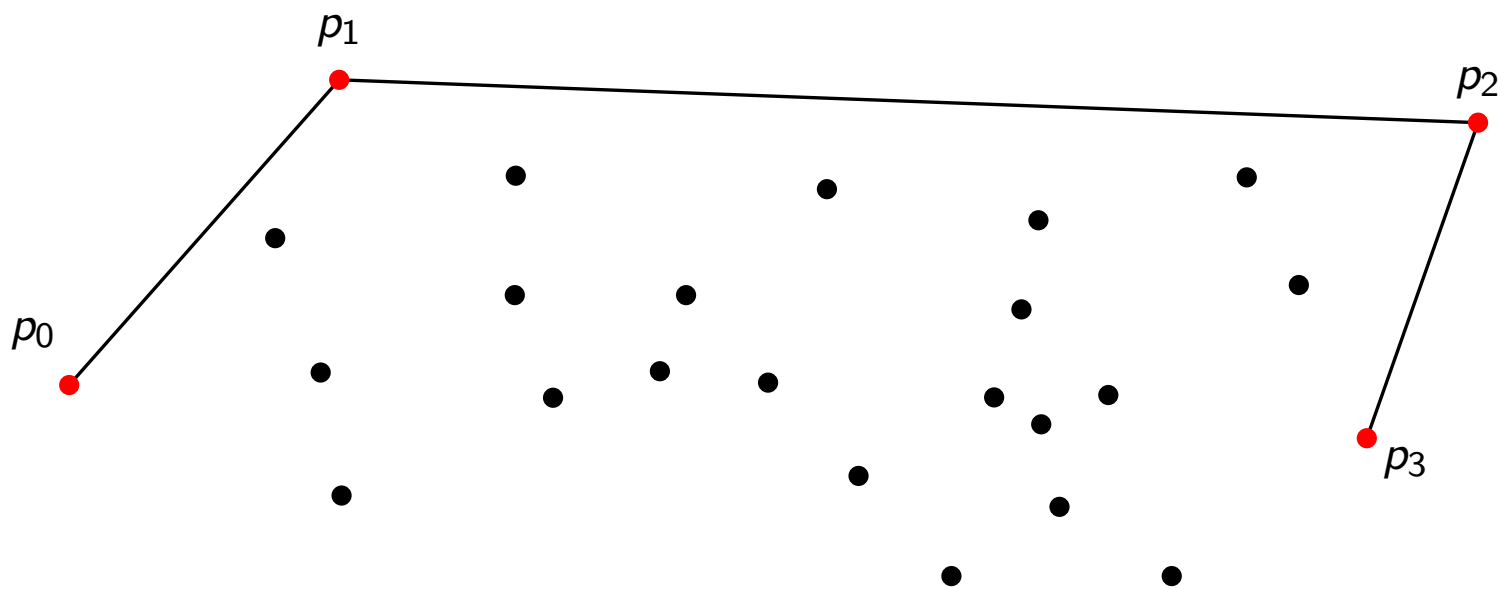
Once we find the point p_1, we continue the process, searching for a point p_2 such that all points (including p_0, p_1 and p_2 are on the right hand side of $\overline{p_1 p_2}$.

Basic Algorithms (6)



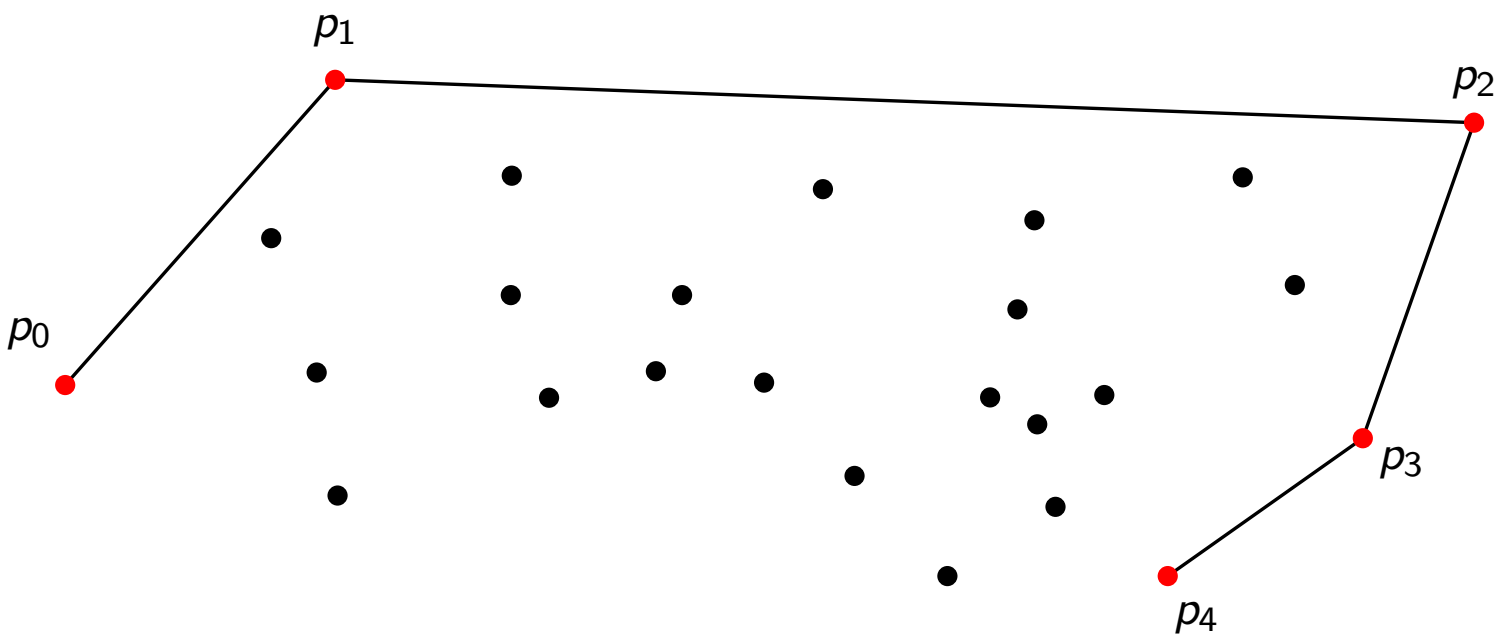
At each step, searching for the next point requires $O(n^2)$ time: For each possible next point, we have to evaluate $T(P, Q, R)$ on all n points.

Basic Algorithms (7)



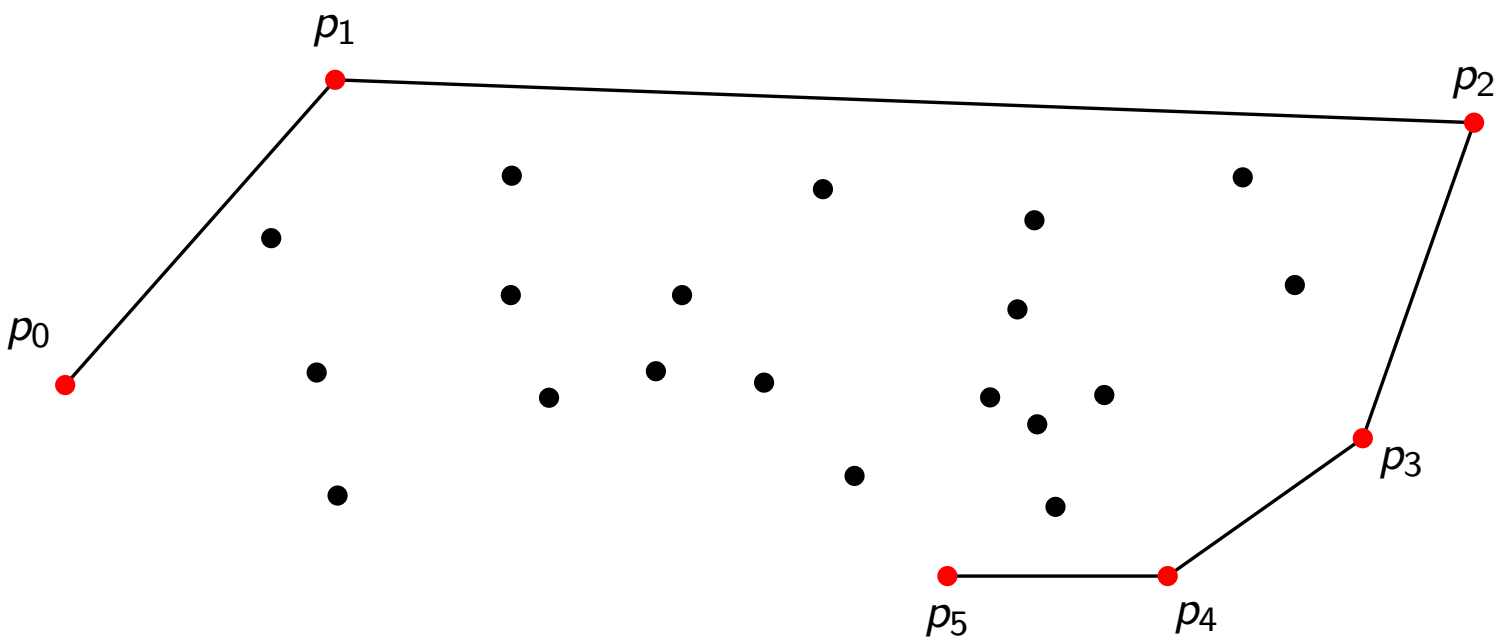
At each step, searching for the next point requires $O(n^2)$ time: For each possible next point, we have to evaluate $T(P, Q, R)$ on all n points.

Basic Algorithms (8)



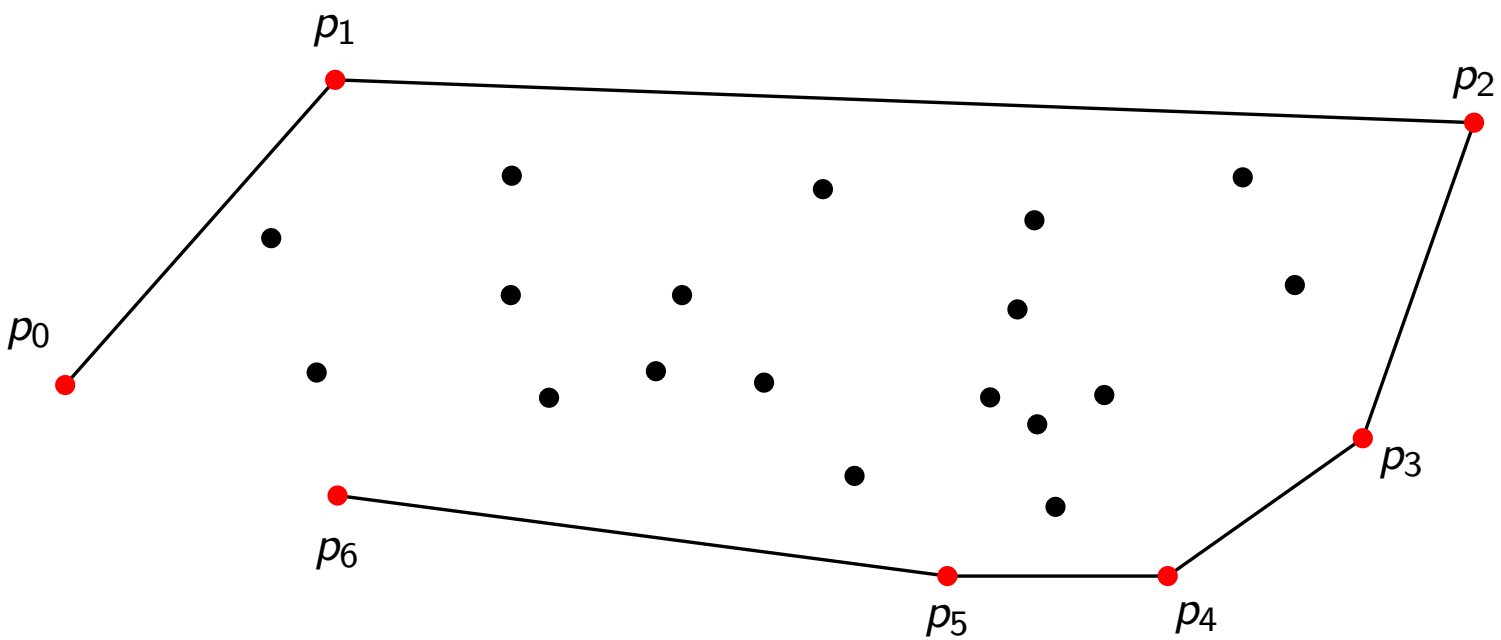
At each step, searching for the next point requires $O(n^2)$ time: For each possible next point, we have to evaluate $T(P, Q, R)$ on all n points.

Basic Algorithms (9)



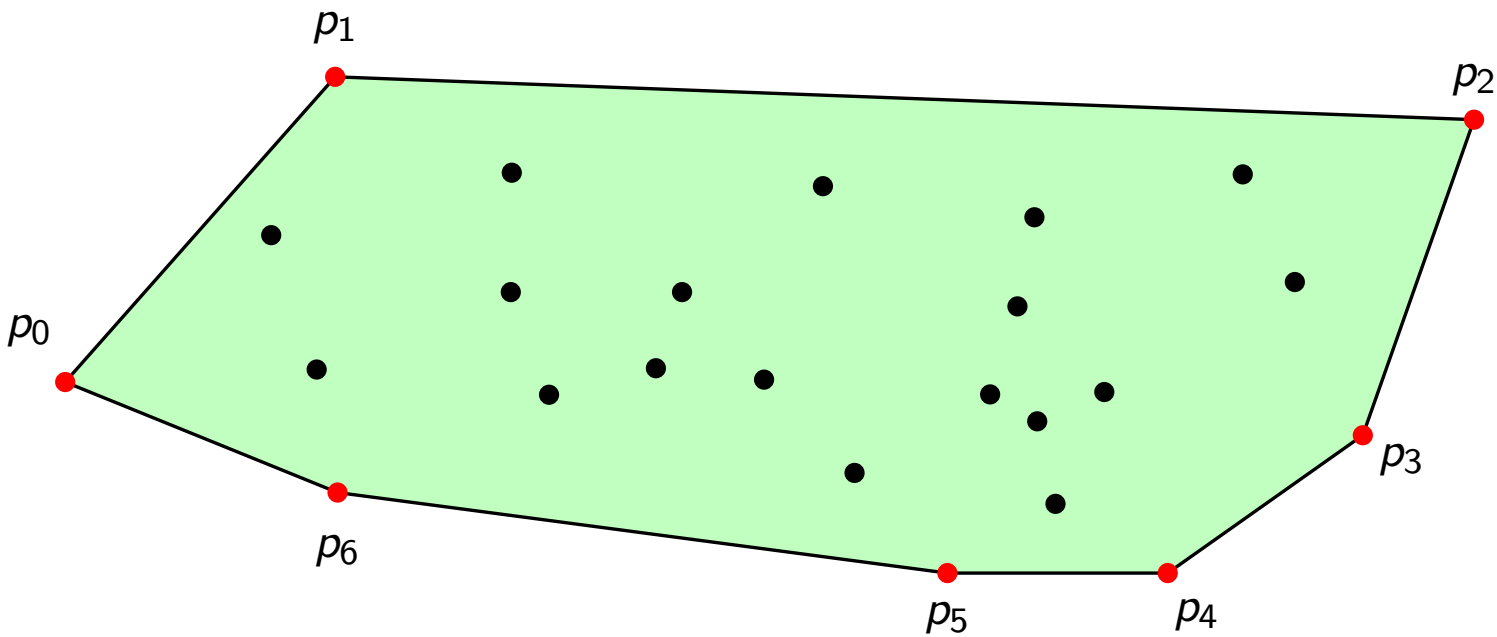
At each step, searching for the next point requires $O(n^2)$ time: For each possible next point, we have to evaluate $T(P, Q, R)$ on all n points.

Basic Algorithms (10)



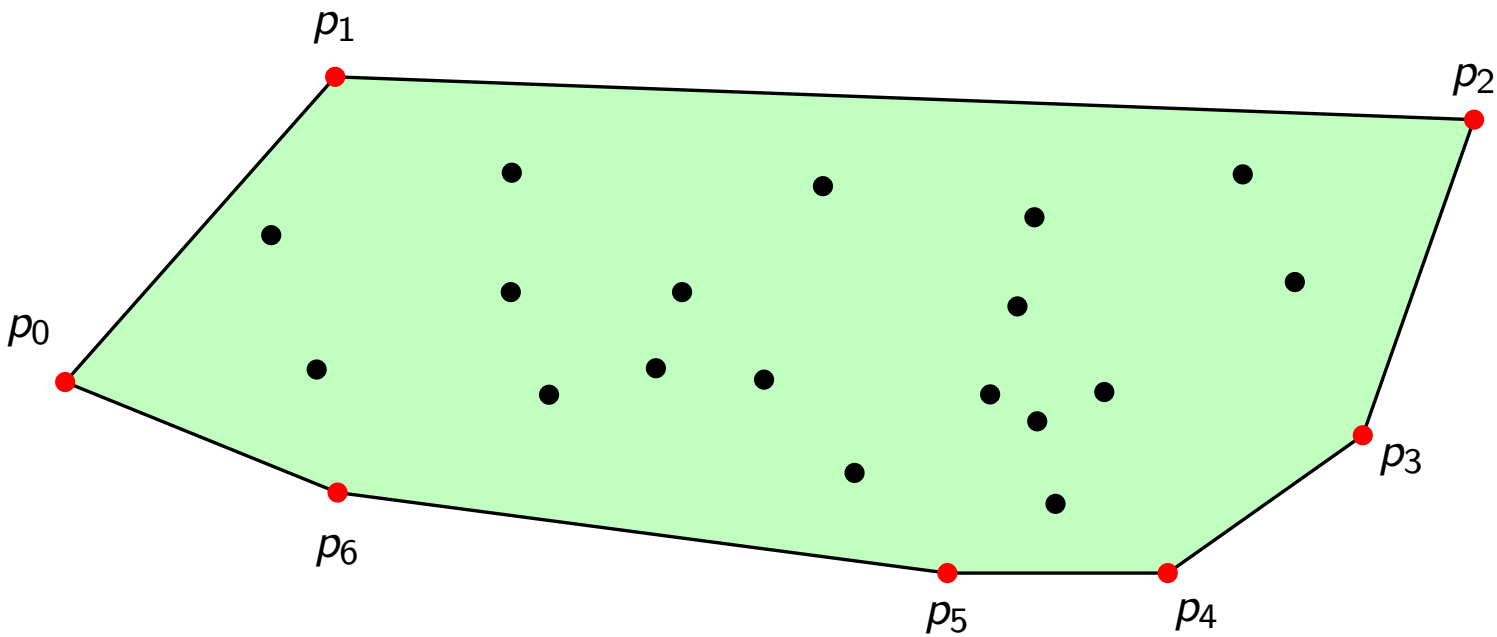
At each step, searching for the next point requires $O(n^2)$ time: For each possible next point, we have to evaluate $T(P, Q, R)$ on all n points.

Basic Algorithms (11)



Eventually, the next point we find will be the starting point p_0, and the convex hull will be complete.

Basic Algorithms (12)



Exercise: Prove that the only case where a previously used point is selected as the next point is the case where p_0 is selected (and the hull is complete).

Basic Algorithms (13)

```
1: procedure SlowHull(S)
2:     H ← Empty list.
3:     p_0 ← Point in S with smallest x-coordinate.
4:     Add p_0 to the end of H.
5:     while true do
6:         p ← Last element of H
7:         next ← null
8:         for each point q ∈ S do
9:             valid ← true
10:            for each point r ∈ S do
11:                if T(p, q, r) < 0 then
12:                    valid ← false
13:                    Break
14:                end if
15:            end for
16:            if valid = true then
17:                next ← q
18:            end if
19:        end for
20:        if next = p_0 then
21:            return H
22:        else
23:            Add next to the end of H
24:        end if
25:    end while
26: end procedure
```

Pseudocode for this approach is shown above. The running time is $O(hn^2)$ where h is the number of points in the final convex hull.

Basic Algorithms (14)

```
1: procedure SlowHull(S)
2:     H ← Empty list.
3:     p_0 ← Point in S with smallest x-coordinate.
4:     Add p_0 to the end of H.
5:     while true do
6:         p ← Last element of H
7:         next ← null
8:         for each point q ∈ S do
9:             valid ← true
10:            for each point r ∈ S do
11:                if T(p, q, r) < 0 then
12:                    valid ← false
13:                    Break
14:                end if
15:            end for
16:            if valid = true then
17:                next ← q
18:            end if
19:        end for
20:        if next = p_0 then
21:            return H
22:        else
23:            Add next to the end of H
24:        end if
25:    end while
26: end procedure
```

Question: What is the size h of the convex hull in the worst case?

Basic Algorithms (15)

```
1: procedure SlowHull(S)
2:     H ← Empty list.
3:     p_0 ← Point in S with smallest x-coordinate.
4:     Add p_0 to the end of H.
5:     while true do
6:         p ← Last element of H
7:         next ← null
8:         for each point q ∈ S do
9:             valid ← true
10:            for each point r ∈ S do
11:                if T(p, q, r) < 0 then
12:                    valid ← false
13:                    Break
14:                end if
15:            end for
16:            if valid = true then
17:                next ← q
18:            end if
19:        end for
20:        if next = p_0 then
21:            return H
22:        else
23:            Add next to the end of H
24:        end if
25:    end while
26: end procedure
```

Since $h = n$ in cases where the input set S is itself a convex hull, the algorithm is $O(n^3)$ in the worst case.

Basic Algorithms (16)

```
1: procedure SlowHull(S)
2:     H ← Empty list.
3:     p_0 ← Point in S with smallest x-coordinate.
4:     Add p_0 to the end of H.
5:     while true do
6:         p ← Last element of H
7:         next ← null
8:         for each point q ∈ S do
9:             valid ← true
10:            for each point r ∈ S do
11:                if T(p, q, r) < 0 then
12:                    valid ← false
13:                    Break
14:                end if
15:            end for
16:            if valid = true then
17:                next ← q
18:            end if
19:        end for
20:        if next = p_0 then
21:            return H
22:        else
23:            Add next to the end of H
24:        end if
25:    end while
26: end procedure
```

Question: Can we do better than $O(n^3)$ in the worst case?

Basic Algorithms (17)

```
1: procedure SlowHull(S)
2:     H ← Empty list.
3:     p_0 ← Point in S with smallest x-coordinate.
4:     Add p_0 to the end of H.
5:     while true do
6:         p ← Last element of H
7:         next ← null
8:         for each point q ∈ S do
9:             valid ← true
10:            for each point r ∈ S do
11:                if T(p, q, r) < 0 then
12:                    valid ← false
13:                    Break
14:                end if
15:            end for
16:            if valid = true then
17:                next ← q
18:            end if
19:        end for
20:        if next = p_0 then
21:            return H
22:        else
23:            Add next to the end of H
24:        end if
25:    end while
26: end procedure
```

The innermost loop (lines 10 - 15) can be combined with the loop over q to incrementally find a better choice for `next`.

Basic Algorithms (18)

```
1:  procedure SlightlyFasterHull(S)
2:      H ← Empty list.
3:      p_0 ← Point in S with smallest x-coordinate.
4:      Add p_0 to the end of H.
5:      while true do
6:          p ← Last element of H
7:          next ← First element of S
8:          if next = p then
9:              next ← Second element of S
10:         end if
11:         for each point q ∈ S do
12:             if T(p, next, q) < 0 then
13:                 next ← q
14:             end if
15:         end for
16:         if next = p_0 then
17:             return H
18:         else
19:             Add next to the end of H
20:         end if
21:     end while
22: end procedure
```

In this improved version, an arbitrary point is first chosen for `next`. As the inner loop checks each point $q \in S$, any time a point q that lies to the left of the line $p \rightarrow$ `next` is found, it replaces `next`.

Basic Algorithms (19)

```
1: procedure SlightlyFasterHull(S)
2:     H ← Empty list.
3:     p_0 ← Point in S with smallest x-coordinate.
4:     Add p_0 to the end of H.
5:     while true do
6:         p ← Last element of H
7:         next ← First element of S
8:         if next = p then
9:             next ← Second element of S
10:        end if
11:        for each point q ∈ S do
12:            if T(p, next, q) < 0 then
13:                next ← q
14:            end if
15:        end for
16:        if next = p_0 then
17:            return H
18:        else
19:            Add next to the end of H
20:        end if
21:    end while
22: end procedure
```

(Consider this as the analogue of finding the maximum of an array by iterating through and updating the maximum as larger values are found)

Basic Algorithms (20)

```
1: procedure SlightlyFasterHull(S)
2:     H ← Empty list.
3:     p_0 ← Point in S with smallest x-coordinate.
4:     Add p_0 to the end of H.
5:     while true do
6:         p ← Last element of H
7:         next ← First element of S
8:         if next = p then
9:             next ← Second element of S
10:        end if
11:        for each point q ∈ S do
12:            if T(p, next, q) < 0 then
13:                next ← q
14:            end if
15:        end for
16:        if next = p_0 then
17:            return H
18:        else
19:            Add next to the end of H
20:        end if
21:    end while
22: end procedure
```

This version has running time $O(hn)$ on n input points where the eventual hull has h points.

Basic Algorithms (21)

```
1:  procedure SlightlyFasterHull(S)
2:      H ← Empty list.
3:      p_0 ← Point in S with smallest x-coordinate.
4:      Add p_0 to the end of H.
5:      while true do
6:          p ← Last element of H
7:          next ← First element of S
8:          if next = p then
9:              next ← Second element of S
10:         end if
11:         for each point q ∈ S do
12:             if T(p, next, q) < 0 then
13:                 next ← q
14:             end if
15:         end for
16:         if next = p_0 then
17:             return H
18:         else
19:             Add next to the end of H
20:         end if
21:     end while
22: end procedure
```

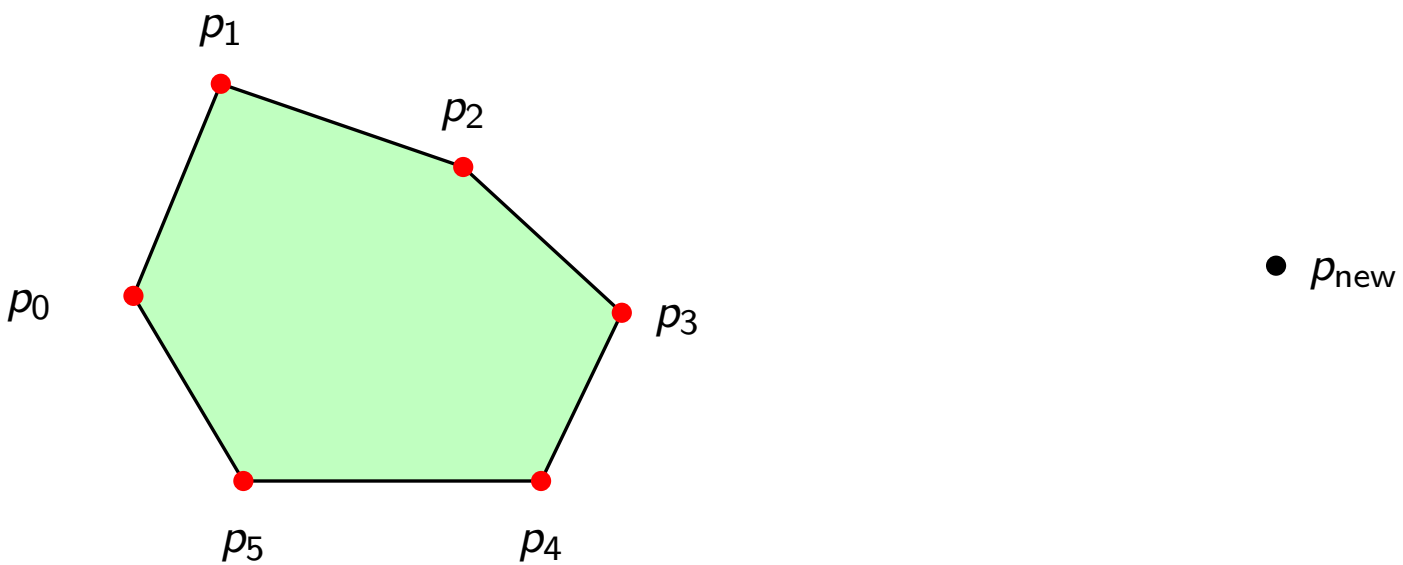
In the worst case this is $O(n^2)$.

Basic Algorithms (22)

```
1:  procedure SlightlyFasterHull(S)
2:      H ← Empty list.
3:      p_0 ← Point in S with smallest x-coordinate.
4:      Add p_0 to the end of H.
5:      while true do
6:          p ← Last element of H
7:          next ← First element of S
8:          if next = p then
9:              next ← Second element of S
10:         end if
11:         for each point q ∈ S do
12:             if T(p, next, q) < 0 then
13:                 next ← q
14:             end if
15:         end for
16:         if next = p_0 then
17:             return H
18:         else
19:             Add next to the end of H
20:         end if
21:     end while
22: end procedure
```

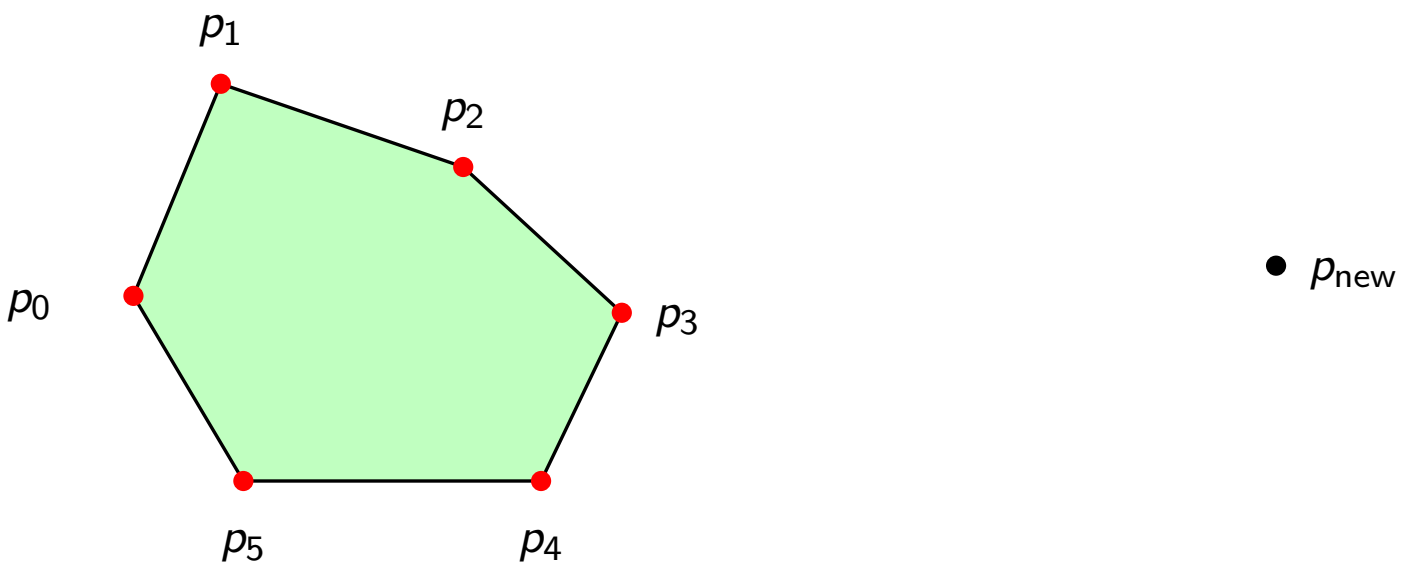
Question: Can we do better?

Upper and Lower Hulls (1)



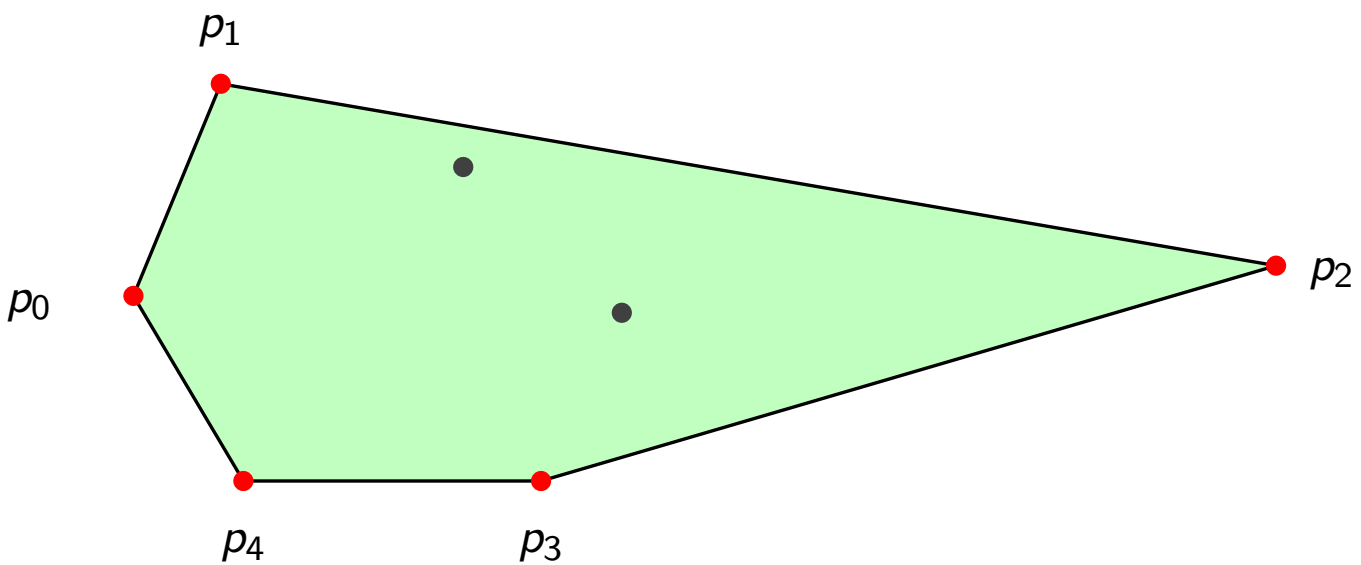
Suppose the convex hull on the left is constructed around a set of points S. Now consider a new point p_{new} (which is not part of S) whose x-coordinate is larger than anything in S.

Upper and Lower Hulls (2)



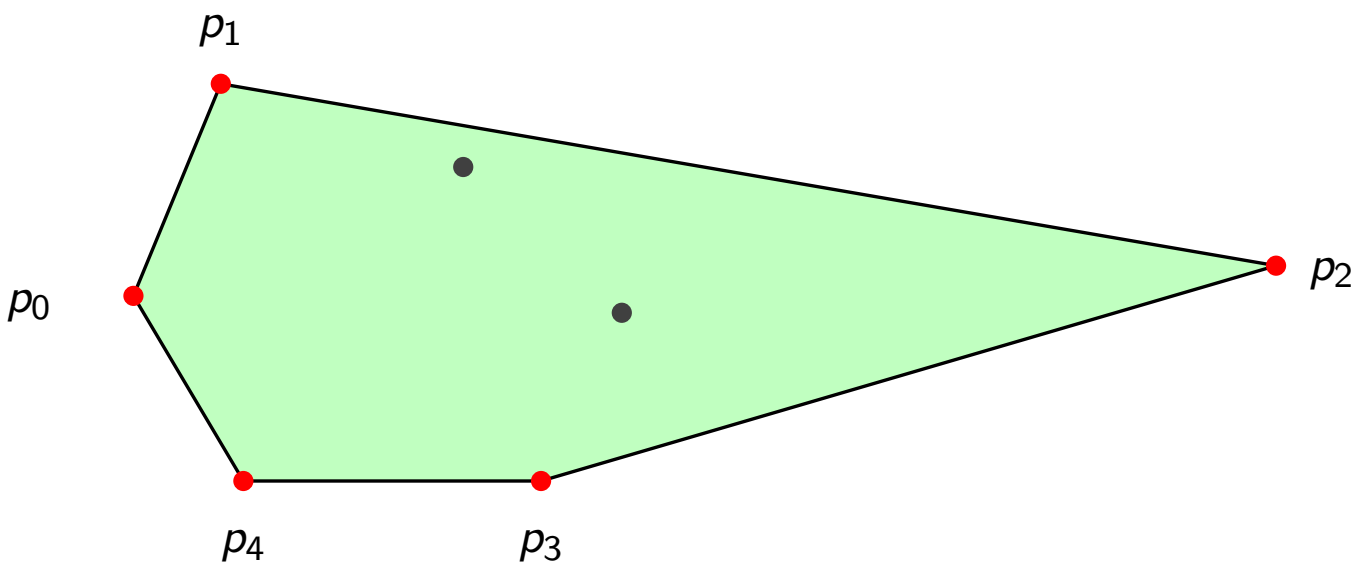
Observation: If p_{new} is added to S, it **must** be one of the points on the resulting convex hull.

Upper and Lower Hulls (3)



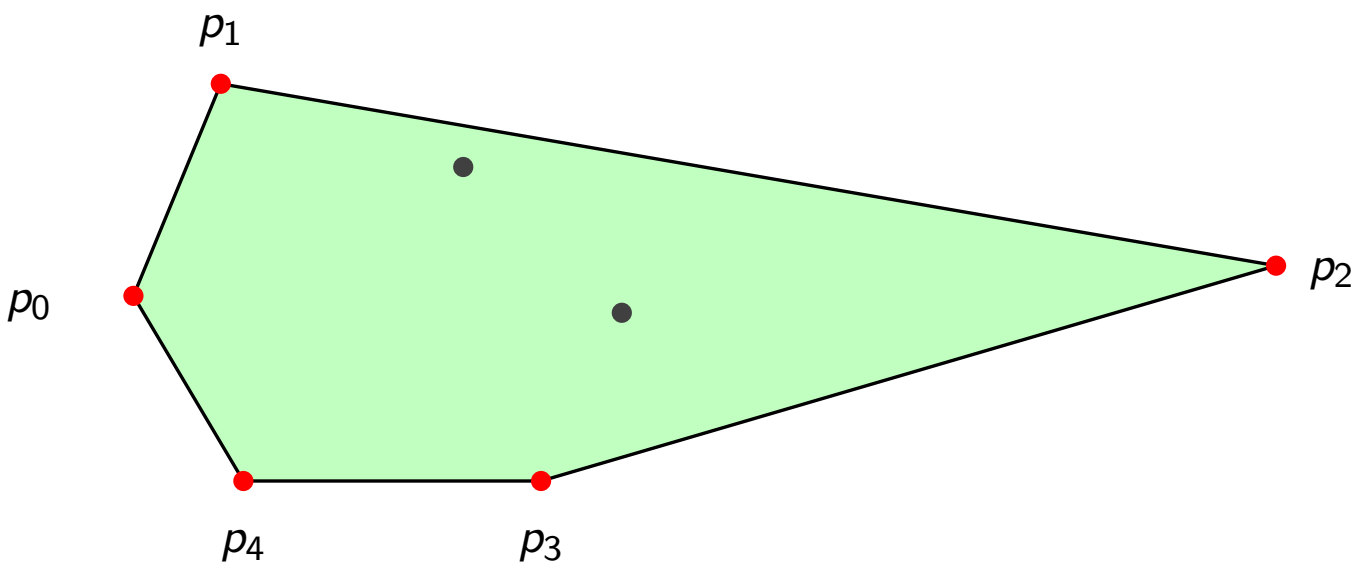
Since p_{new} will have the largest x-coordinate of any point in the set, it must appear on the boundary of the convex hull.

Upper and Lower Hulls (4)



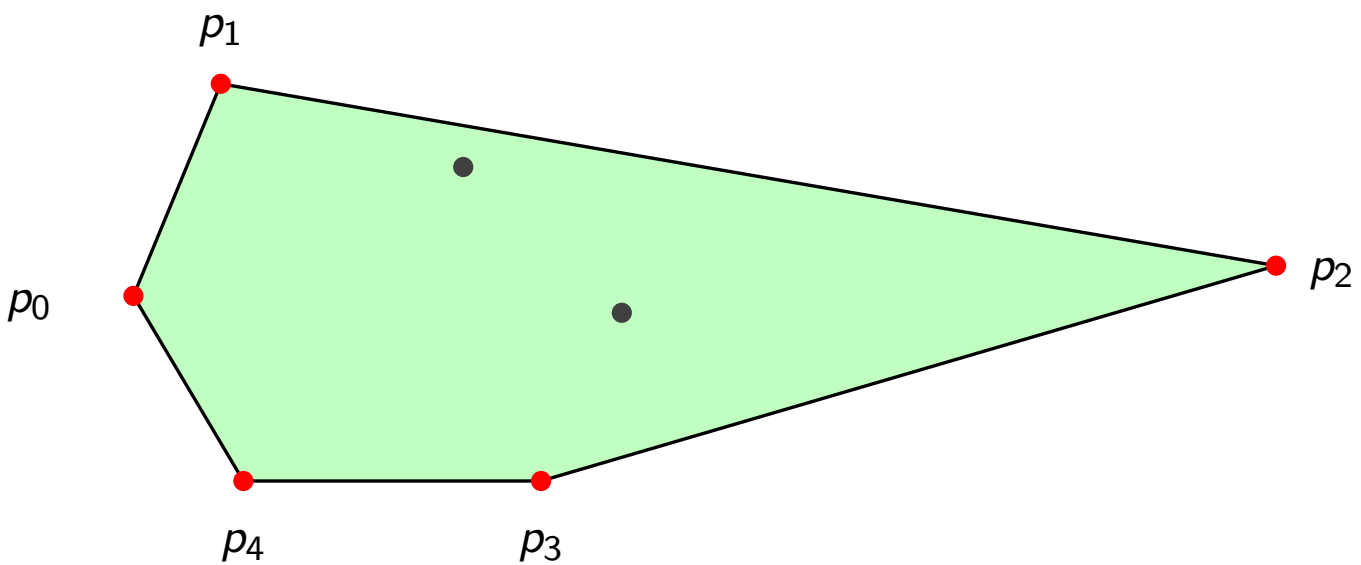
Notice that when the convex hull is adjusted to accommodate p_{new}, some of the points in the previous convex hull are removed.

Upper and Lower Hulls (5)



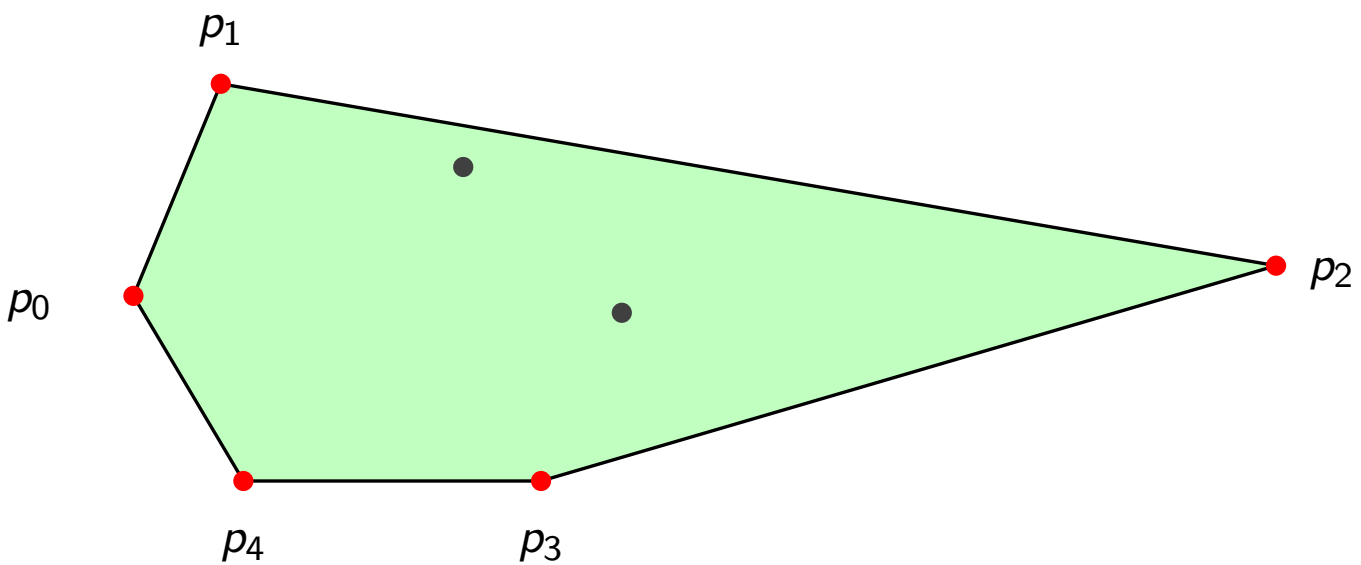
The ultimate effect of adding p_{new} to the set is that p_{new} will be spliced into the convex hull at some location, possibly resulting in other points being removed.

Upper and Lower Hulls (6)



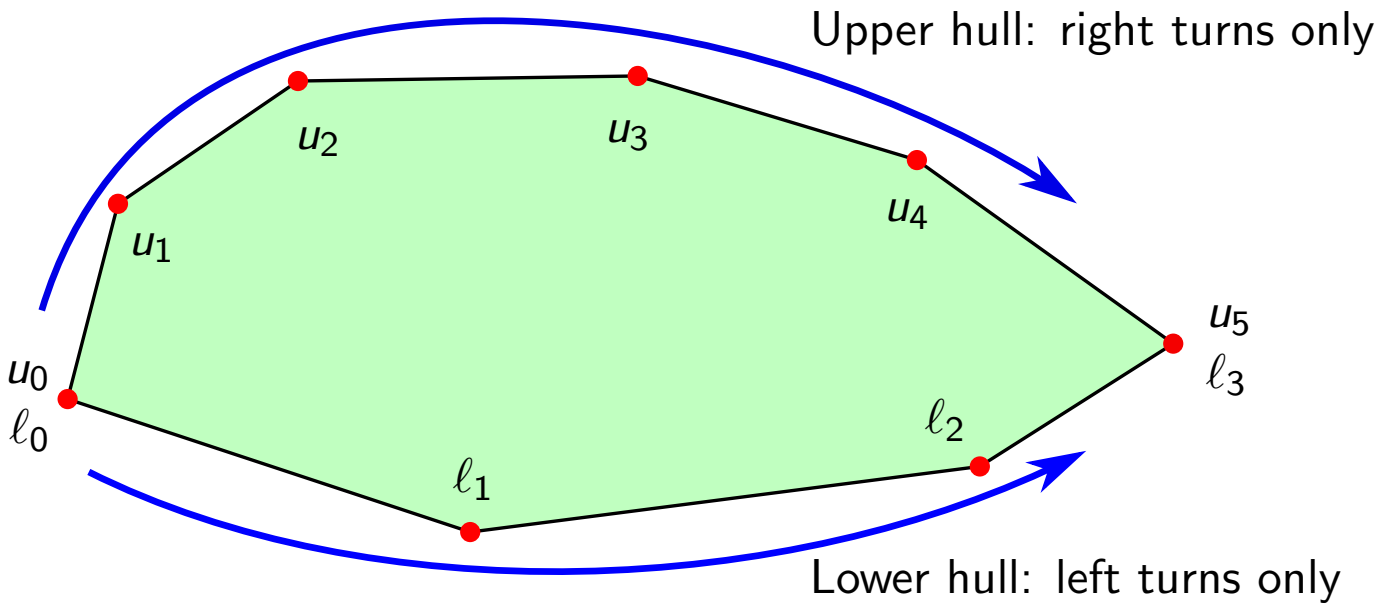
Idea: We can build a convex hull by adding new points in left-to-right order (such that each new point leverages observation above).

Upper and Lower Hulls (7)



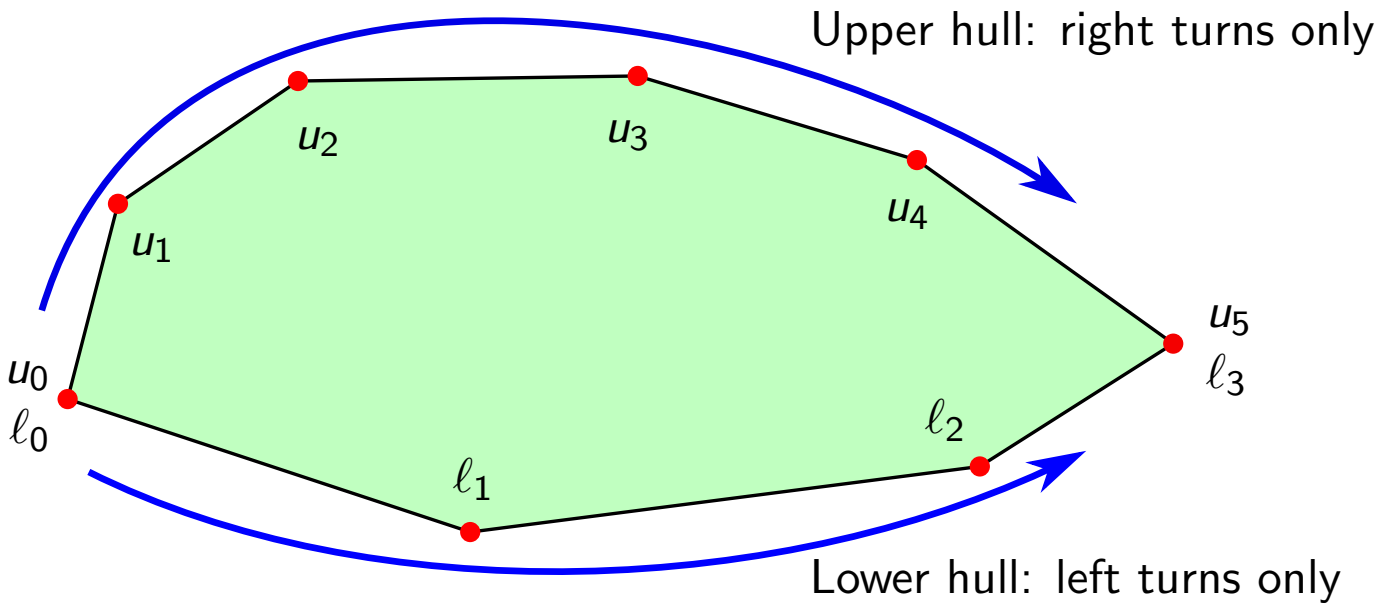
Determining where to splice the new point into the hull might still be difficult, though, since the normal polygon ordering does not indicate exactly where the point with the largest x-coordinate is.

Upper and Lower Hulls (8)



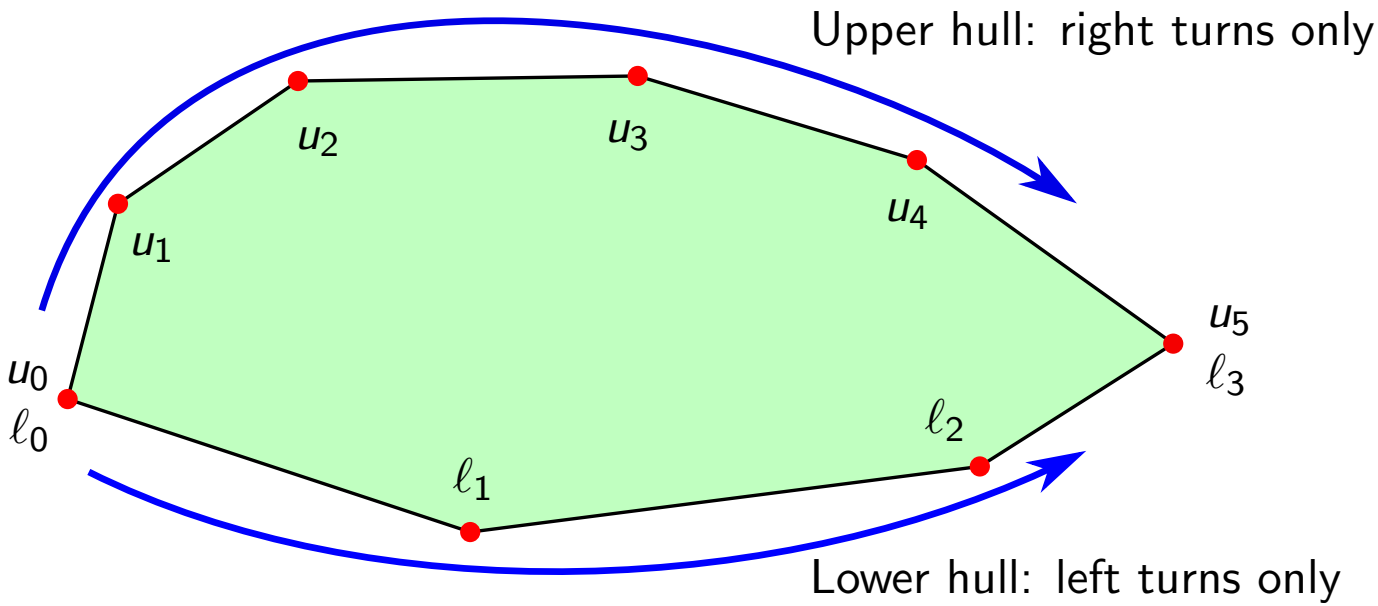
To remedy this, we can split the convex hull into an **upper hull** and a **lower hull**. The upper and lower hulls are sequences of points which always start with the leftmost (lowest x-coordinate) point and end with the rightmost (highest x-coordinate) point.

Upper and Lower Hulls (9)



Iterating over the upper hull in order will produce only **right turns**.
Iterating over the lower hull in order will produce only **left turns**.

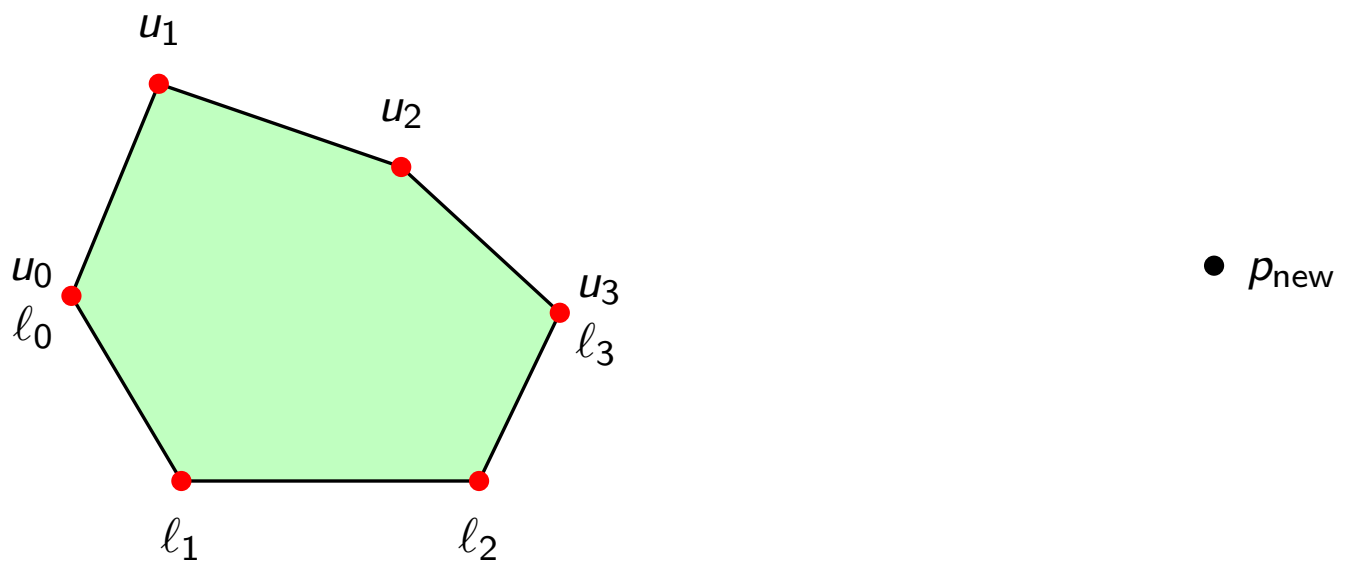
Upper and Lower Hulls (10)



Given an upper hull $u_0, u_1, \ldots, u_n$ and a lower hull $\ell_0, \ell_1, \ldots, \ell_m$, the full convex hull will be

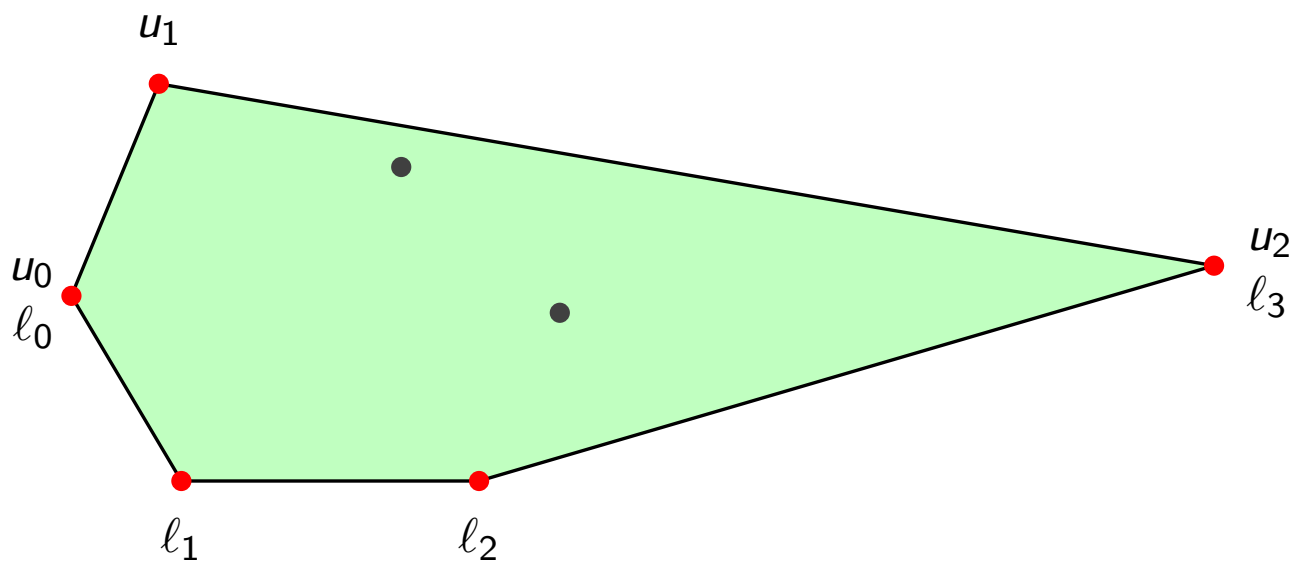
$$u_0, u_1, \ldots, u_n, \ell_{m-1}, \ell_{m-2}, \ldots, \ell_1.$$

Upper and Lower Hulls (11)



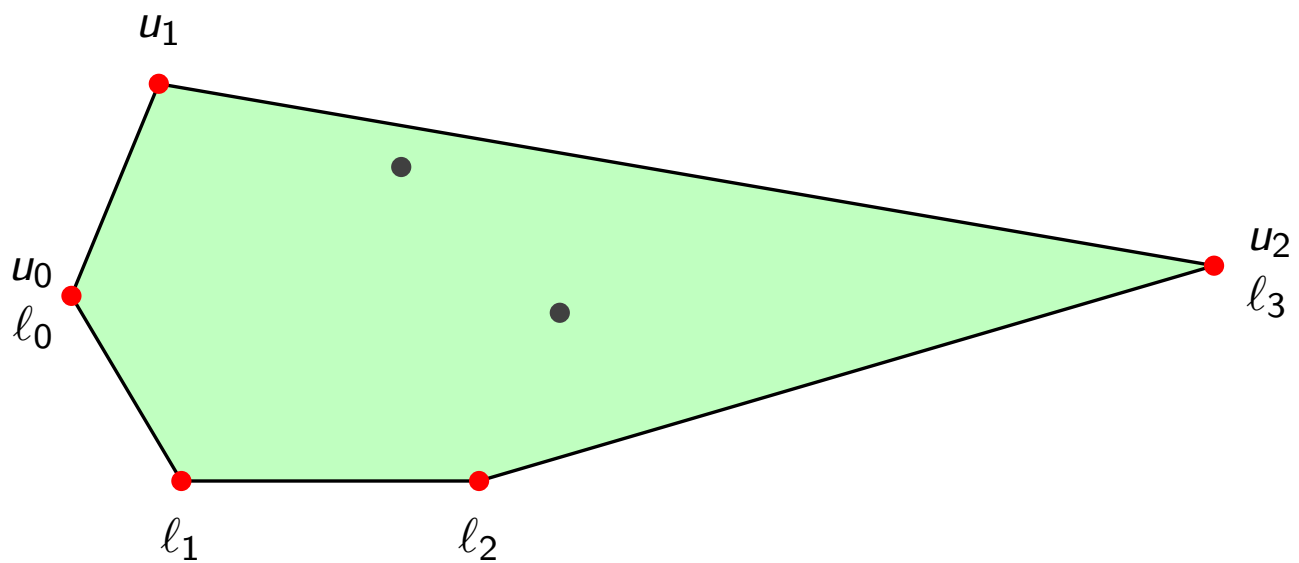
Since the upper and lower hull sequences are ordered from left to right, we know that the point p_{new} from our earlier observation will always be placed at the end of each sequence.

Upper and Lower Hulls (12)



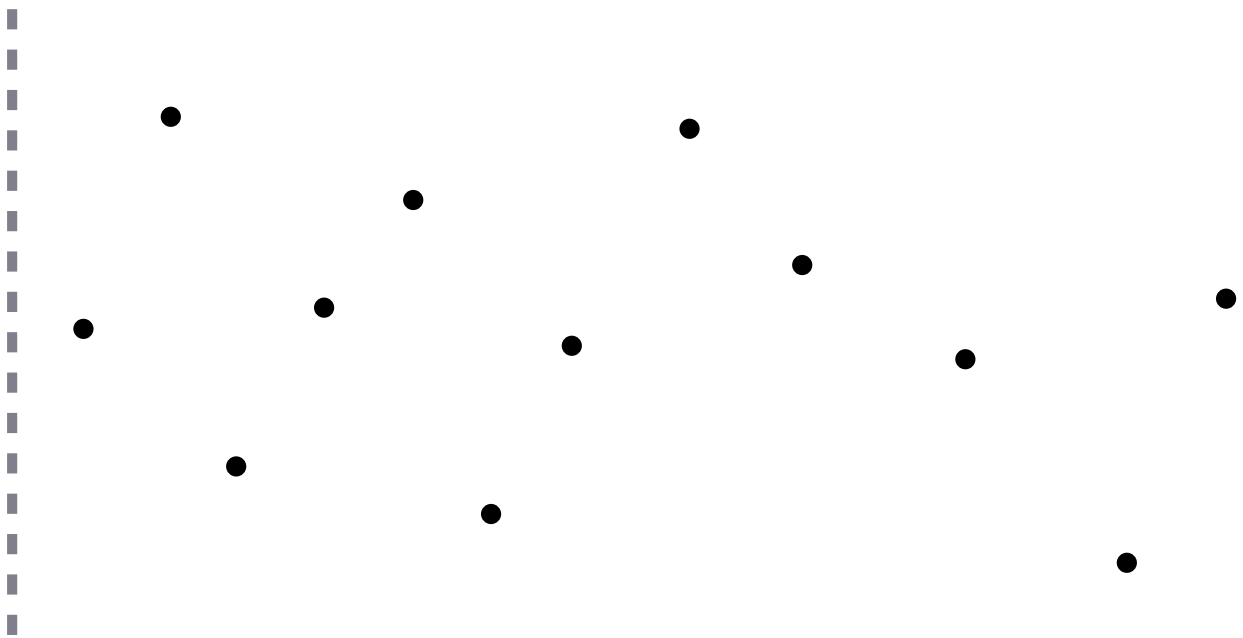
Additionally, the only points that may need to be removed from the upper or lower hulls after inserting p_{new} will be those points at the end of each sequence.

Upper and Lower Hulls (13)



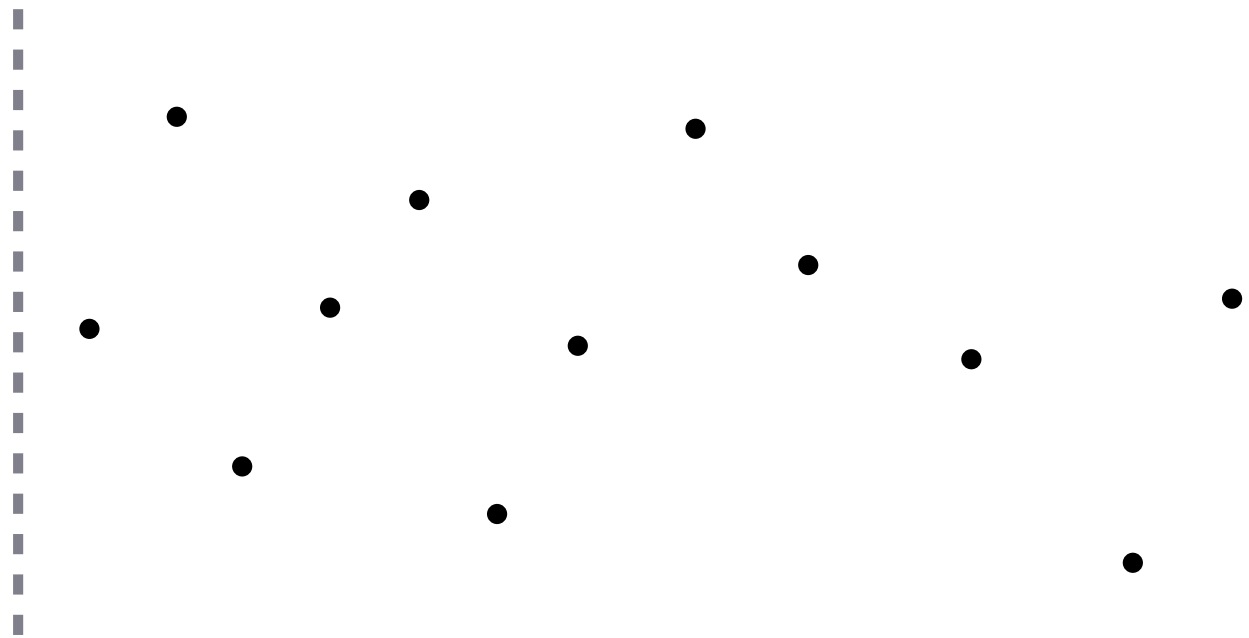
To update the upper hull to accommodate p_{new}, delete points from the end of the sequence until the last pair of points and p_{new} makes a right turn. Similarly, delete points from the lower hull until the last pair of points and p_{new} forms a left turn.

Monotone Scan Algorithm (1)



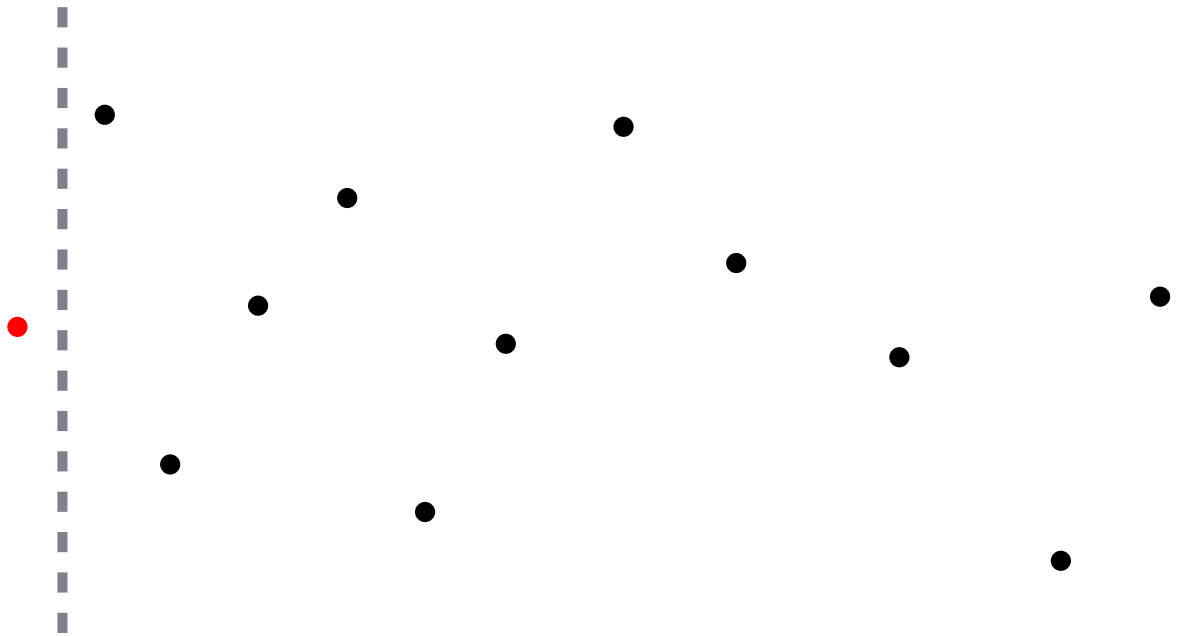
These observations can be used to formulate a new algorithm which builds the upper and lower hulls (separately) in left-to-right order, simulating the process of progressively adding new points to the right of the previous point.

Monotone Scan Algorithm (2)



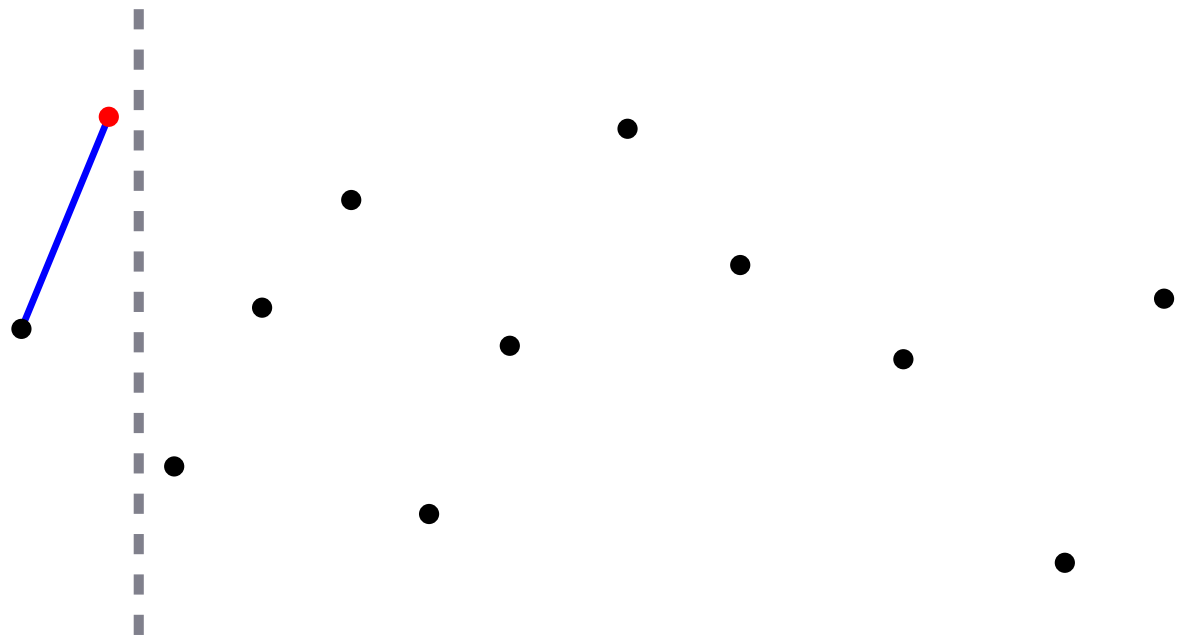
(At each step, only points to the left of the dotted line are included in the convex hull).

Monotone Scan Algorithm (3)



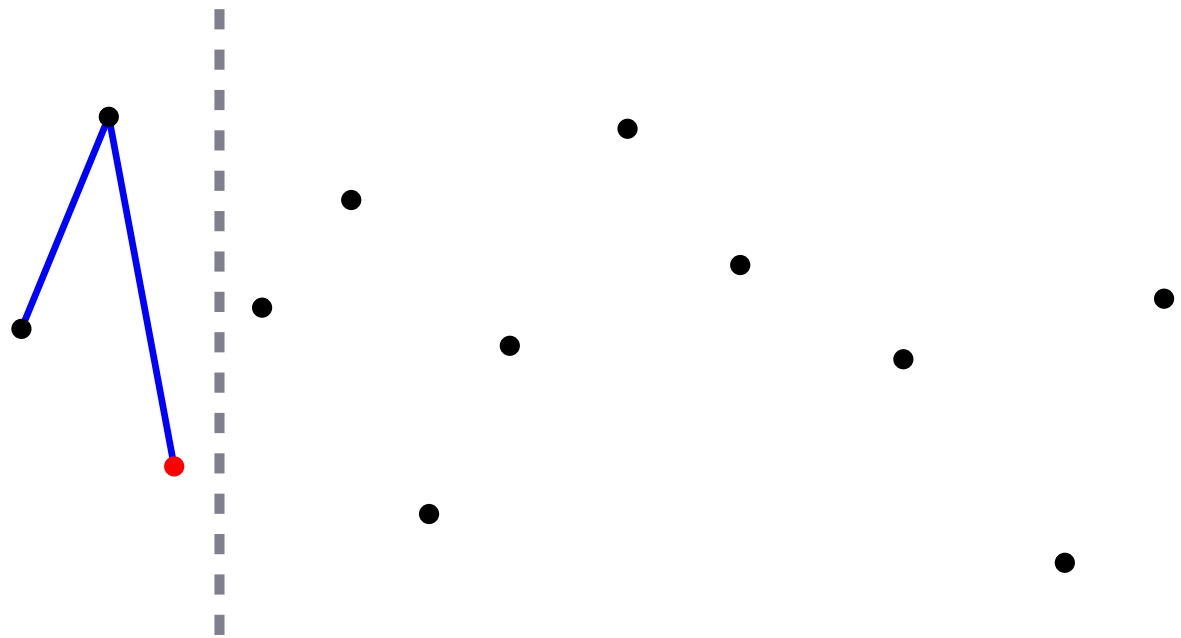
The upper hull is built by iterating over the points in left-to-right order and adding each point to the upper hull sequence.

Monotone Scan Algorithm (4)



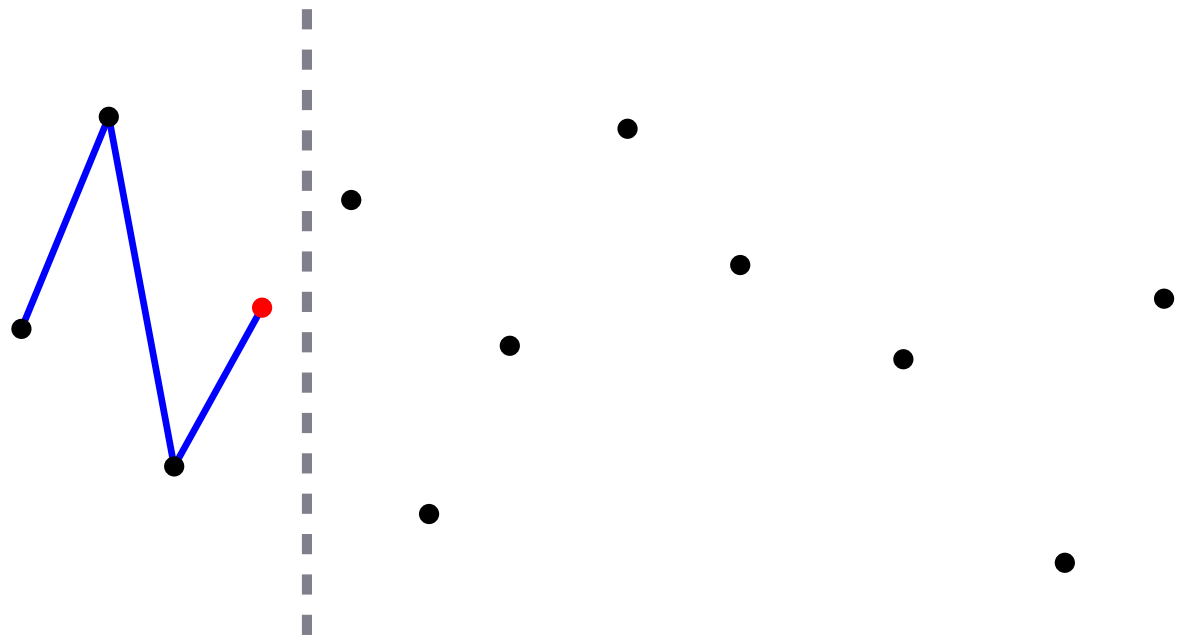
The upper hull is built by iterating over the points in left-to-right order and adding each point to the upper hull sequence.

Monotone Scan Algorithm (5)



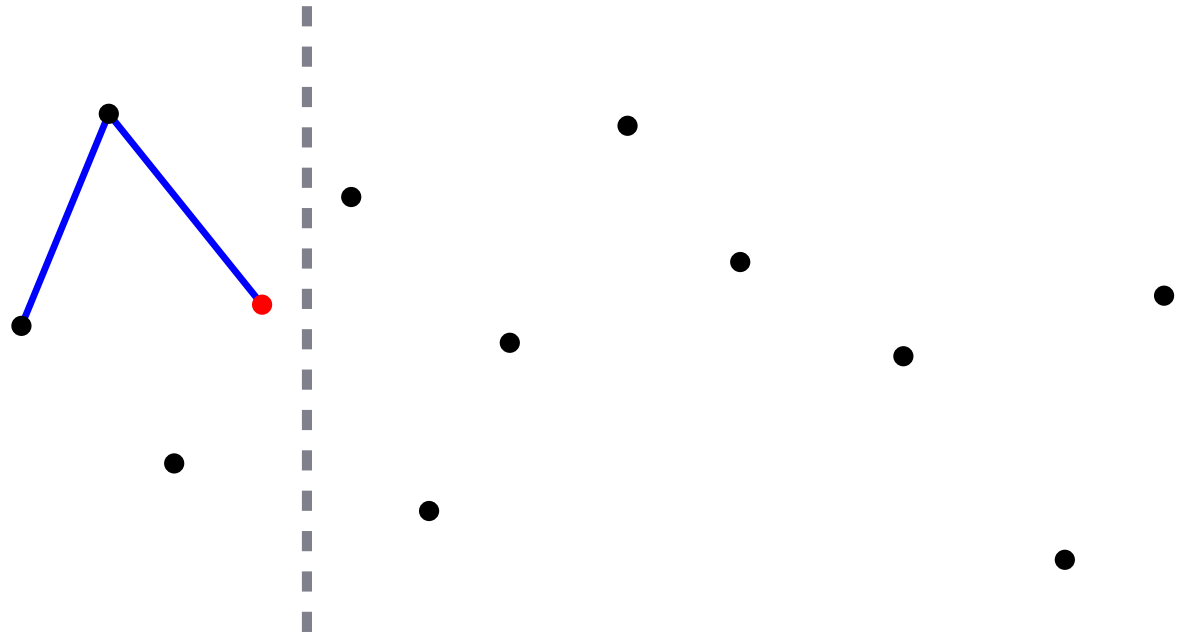
The upper hull is built by iterating over the points in left-to-right order and adding each point to the upper hull sequence.

Monotone Scan Algorithm (6)



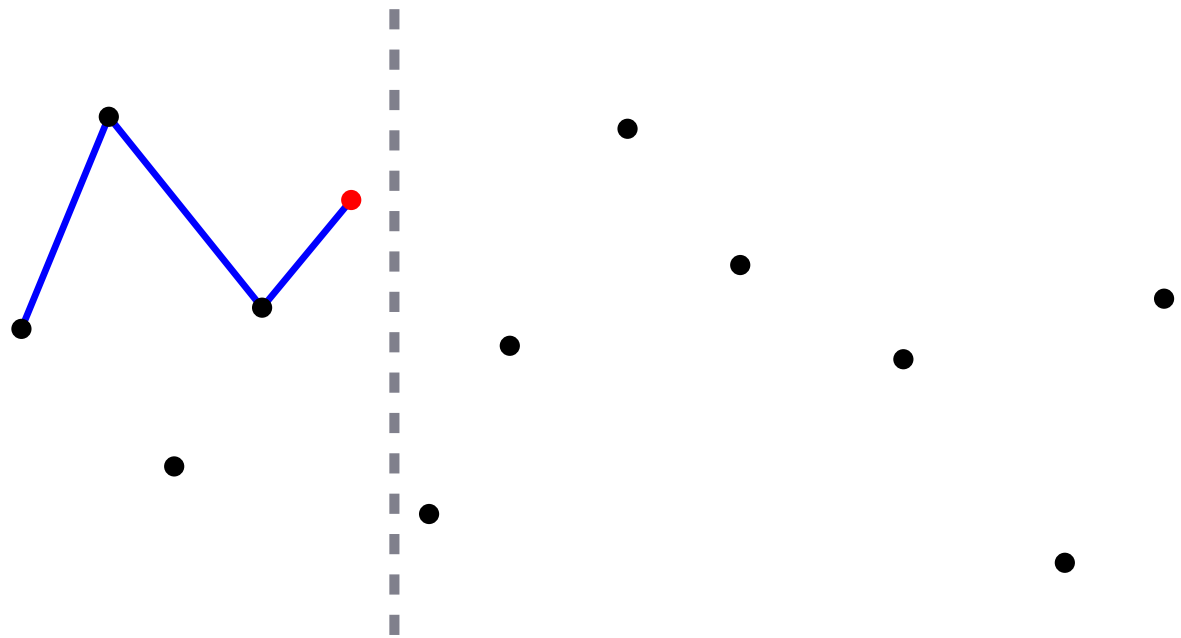
If, after adding any point, the upper hull does not consist entirely of right turns, points are removed from the end of the sequence (before the new point) until no right turns exist.

Monotone Scan Algorithm (7)



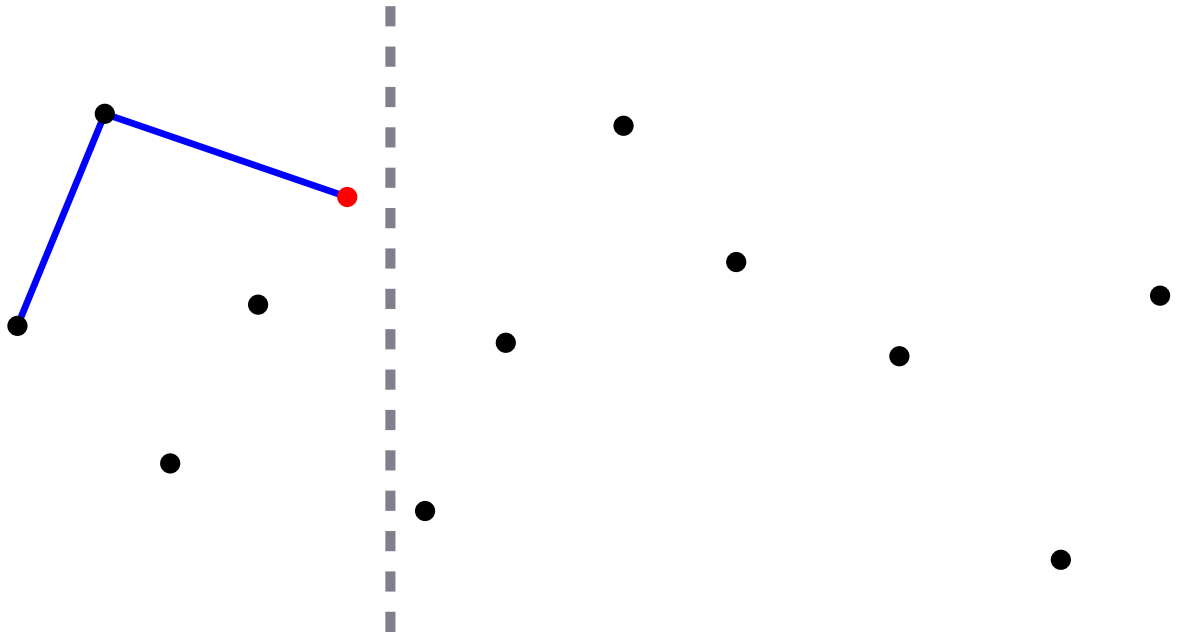
In this case, only one point needed to be removed (but in general, it may be necessary to remove several).

Monotone Scan Algorithm (8)

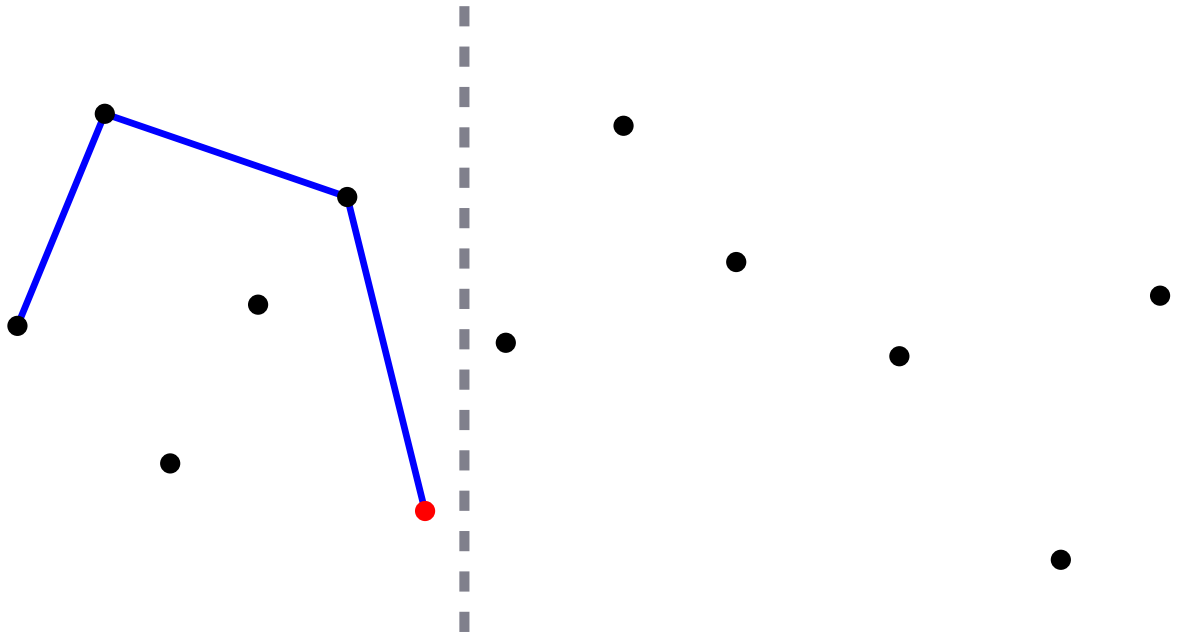


Adding the red point above also creates a left turn and requires removing a point.

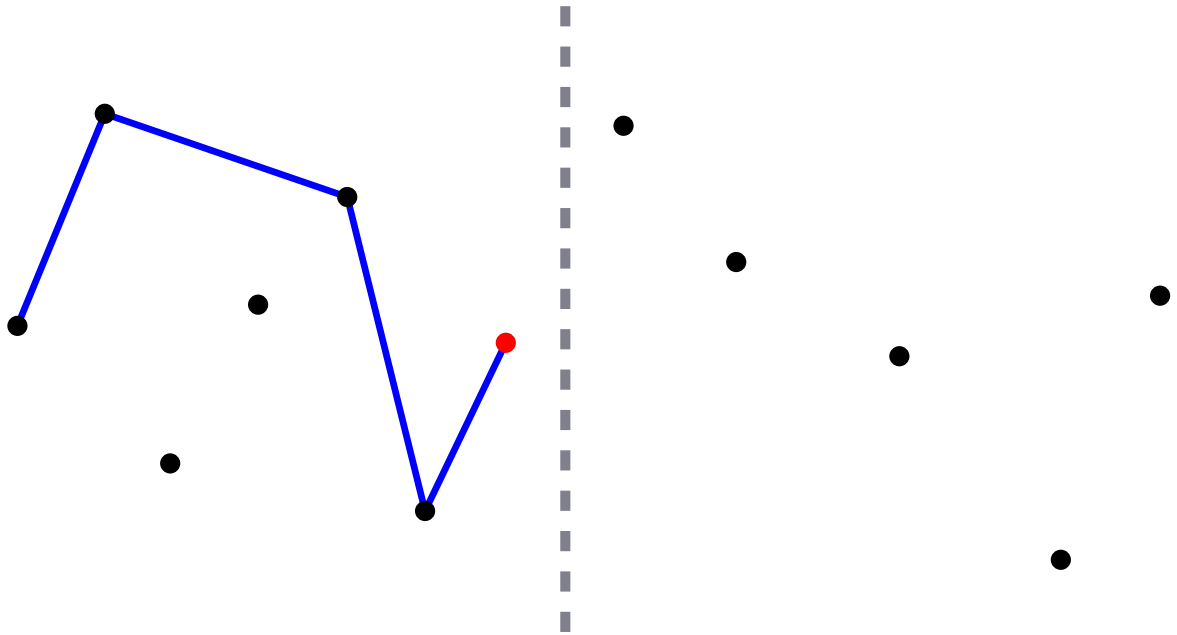
Monotone Scan Algorithm (9)



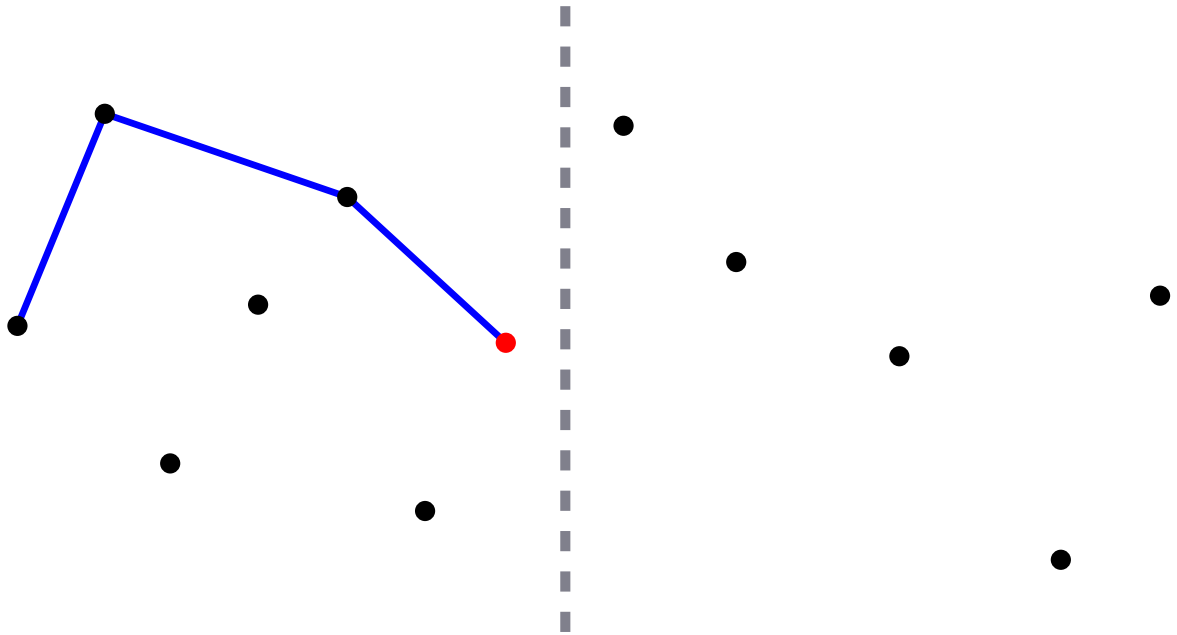
Monotone Scan Algorithm (10)



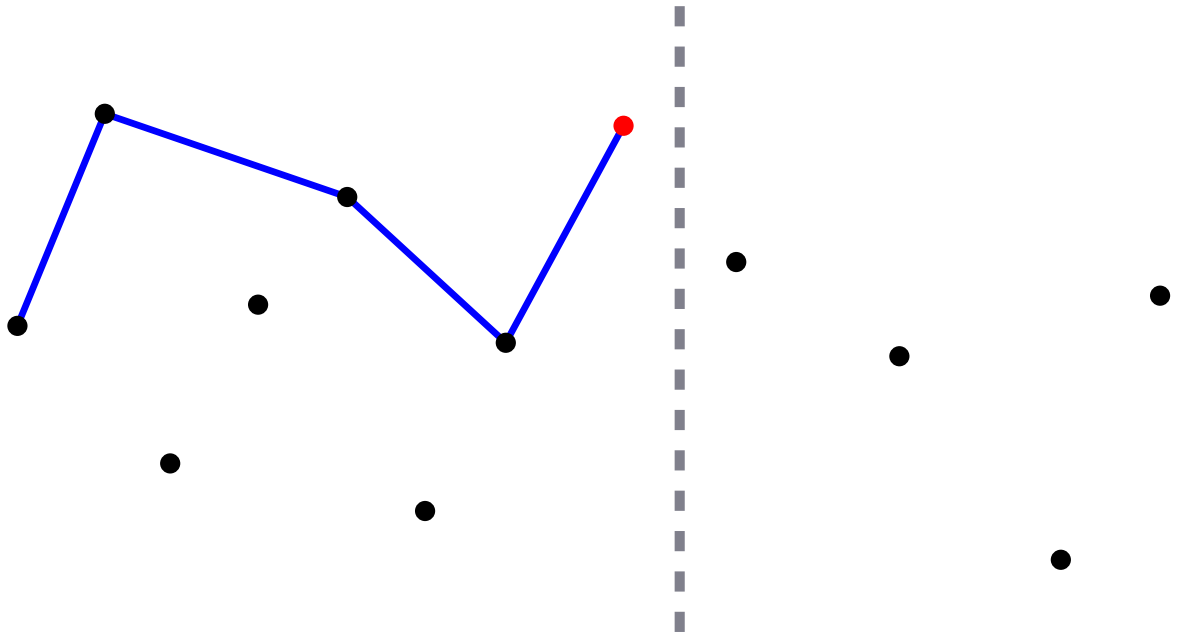
Monotone Scan Algorithm (11)



Monotone Scan Algorithm (12)

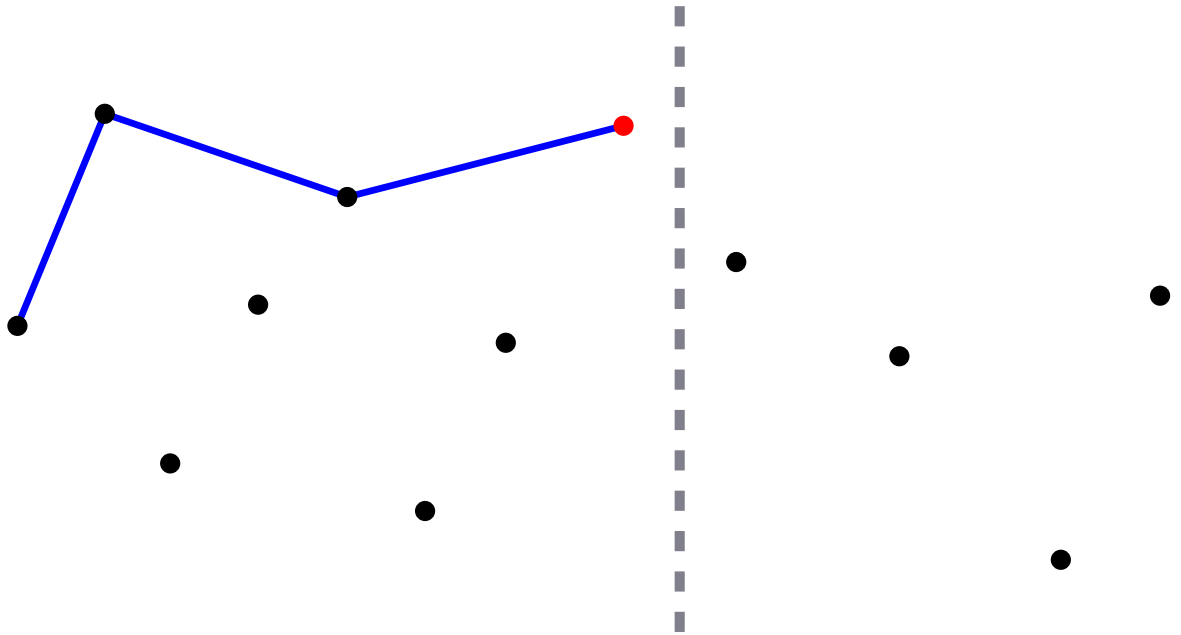


Monotone Scan Algorithm (13)



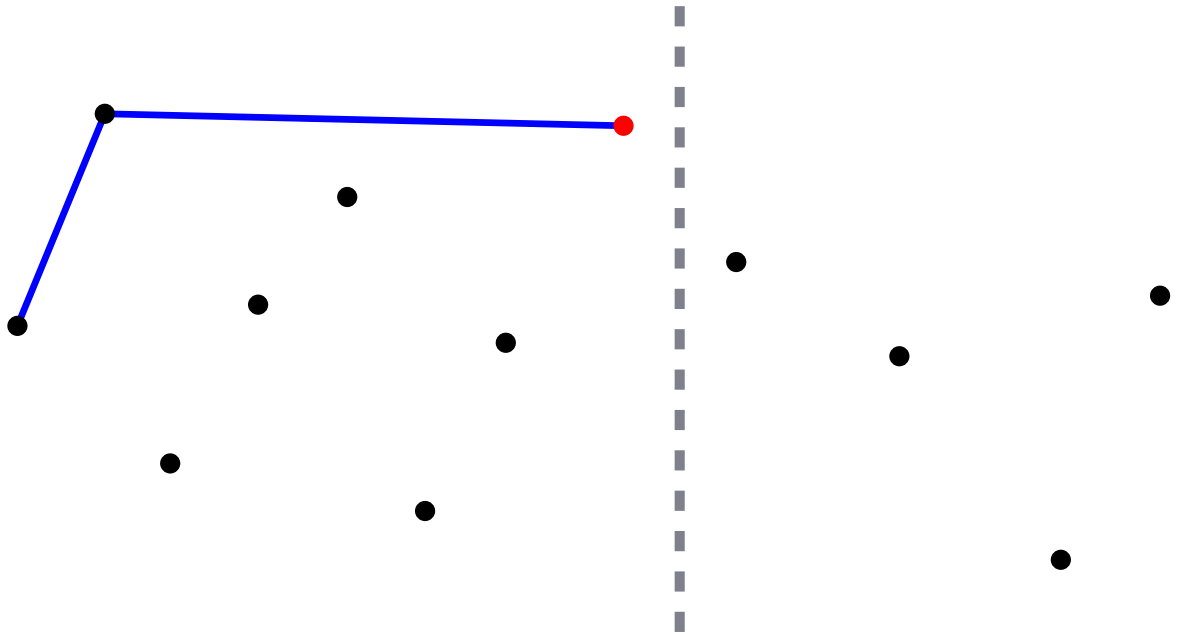
After adding the point above, it is necessary to remove the previous two points to remove left turns.

Monotone Scan Algorithm (14)



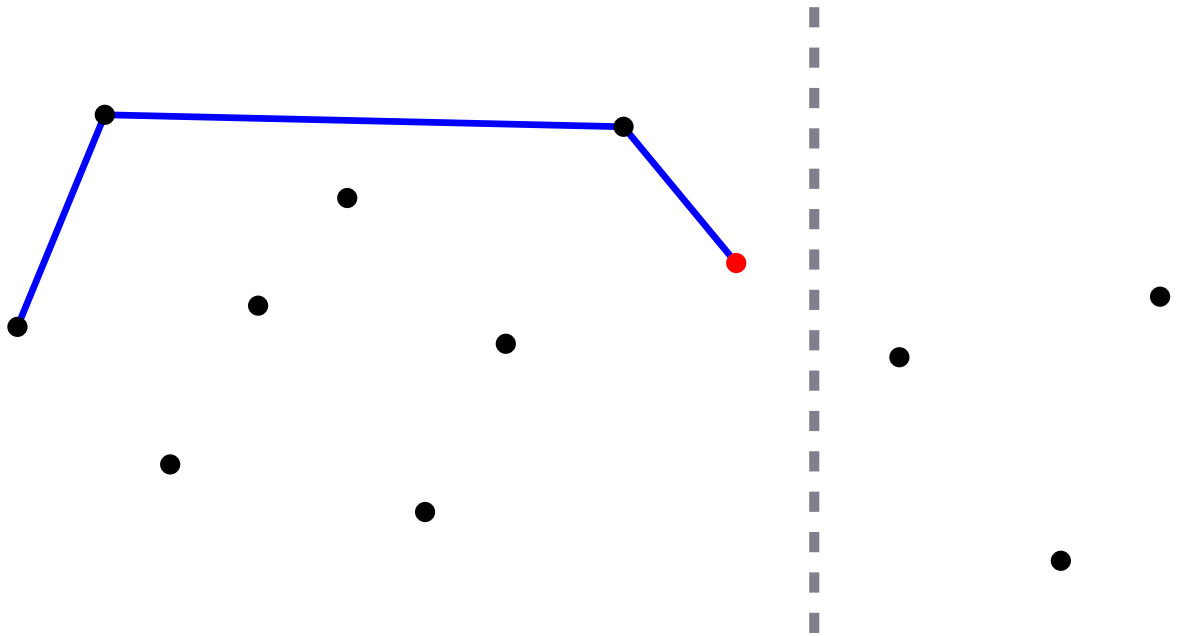
After adding the point above, it is necessary to remove the previous two points to remove left turns.

Monotone Scan Algorithm (15)



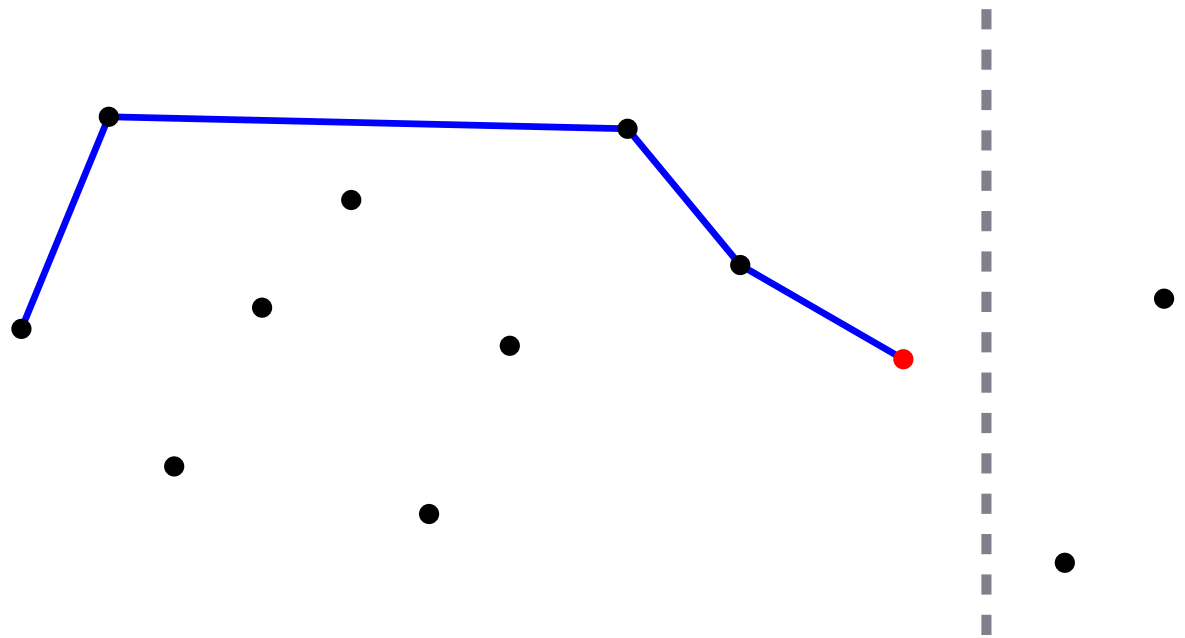
After adding the point above, it is necessary to remove the previous two points to remove left turns.

Monotone Scan Algorithm (16)



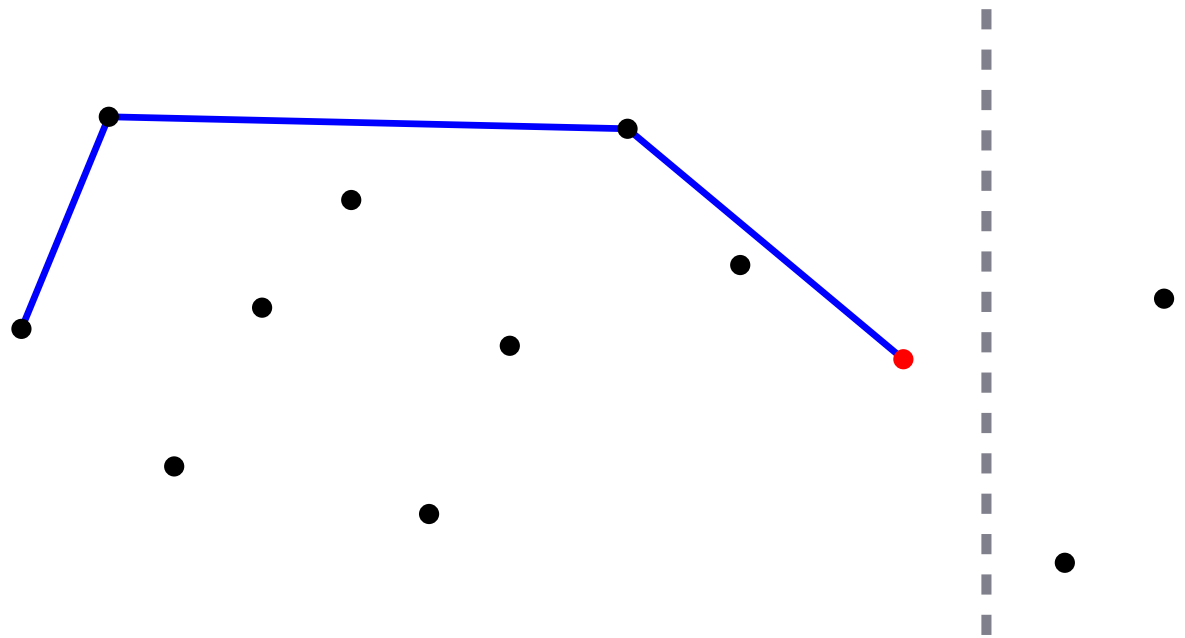
This process continues until the rightmost point is added to the upper hull.

Monotone Scan Algorithm (17)



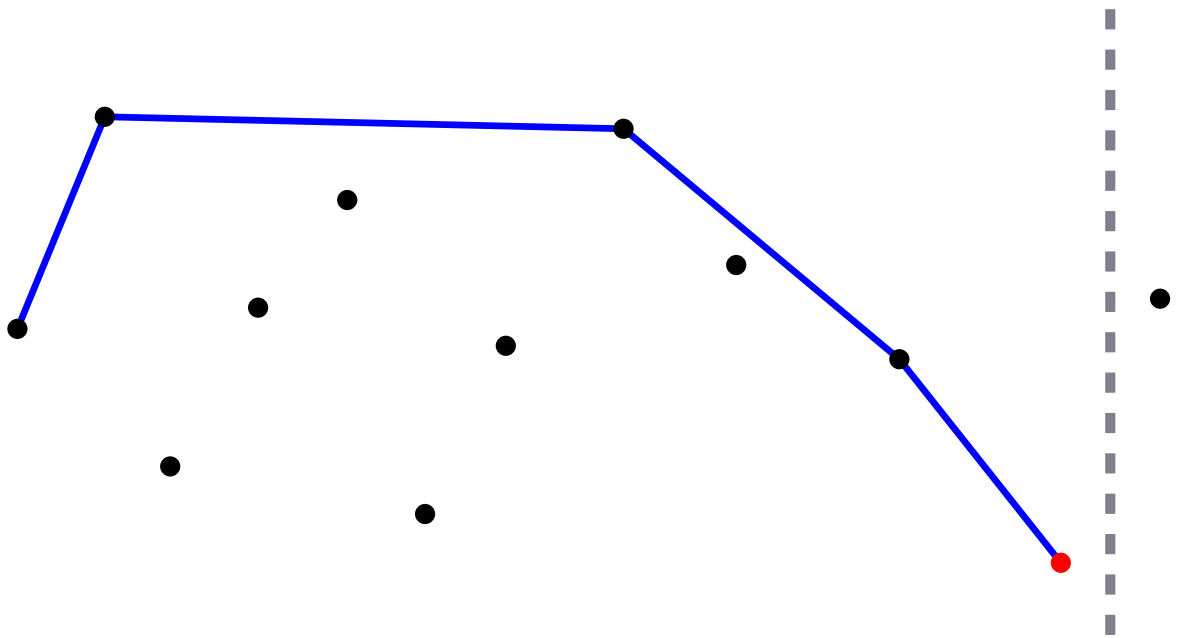
This process continues until the rightmost point is added to the upper hull.

Monotone Scan Algorithm (18)



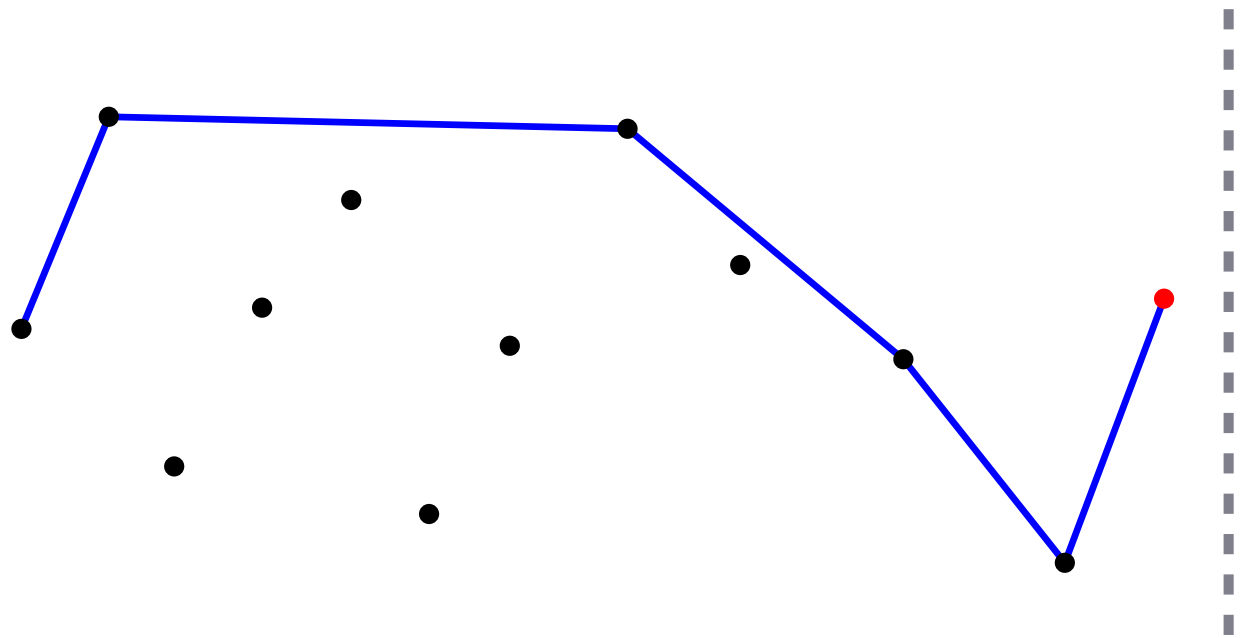
This process continues until the rightmost point is added to the upper hull.

Monotone Scan Algorithm (19)



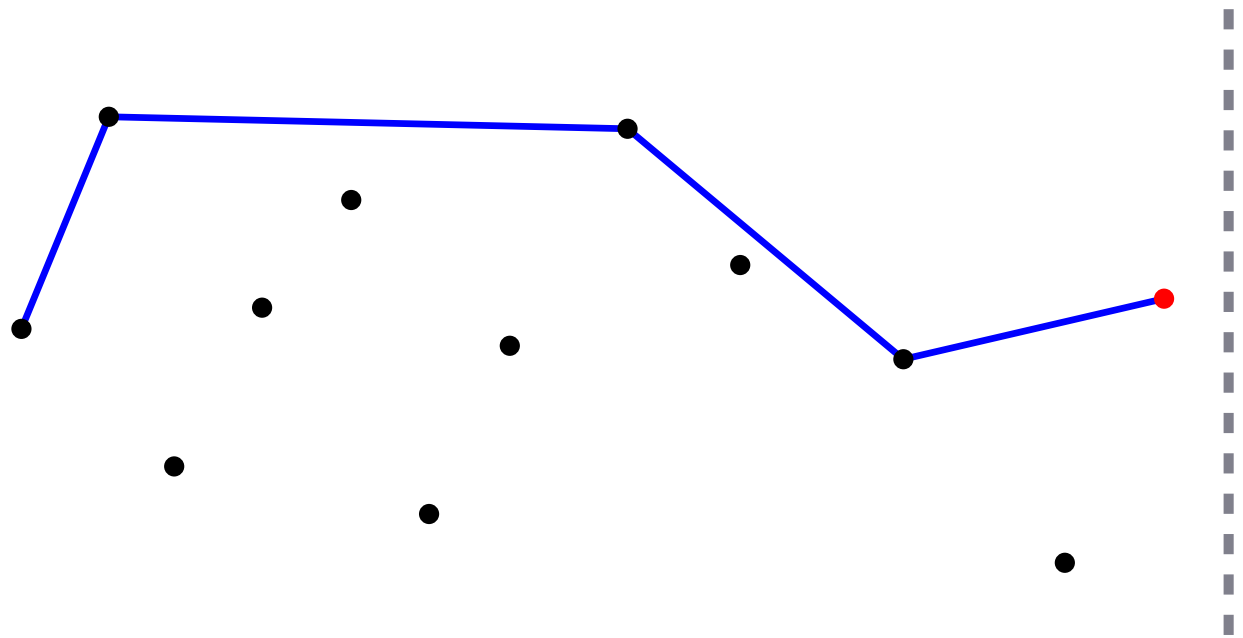
This process continues until the rightmost point is added to the upper hull.

Monotone Scan Algorithm (20)



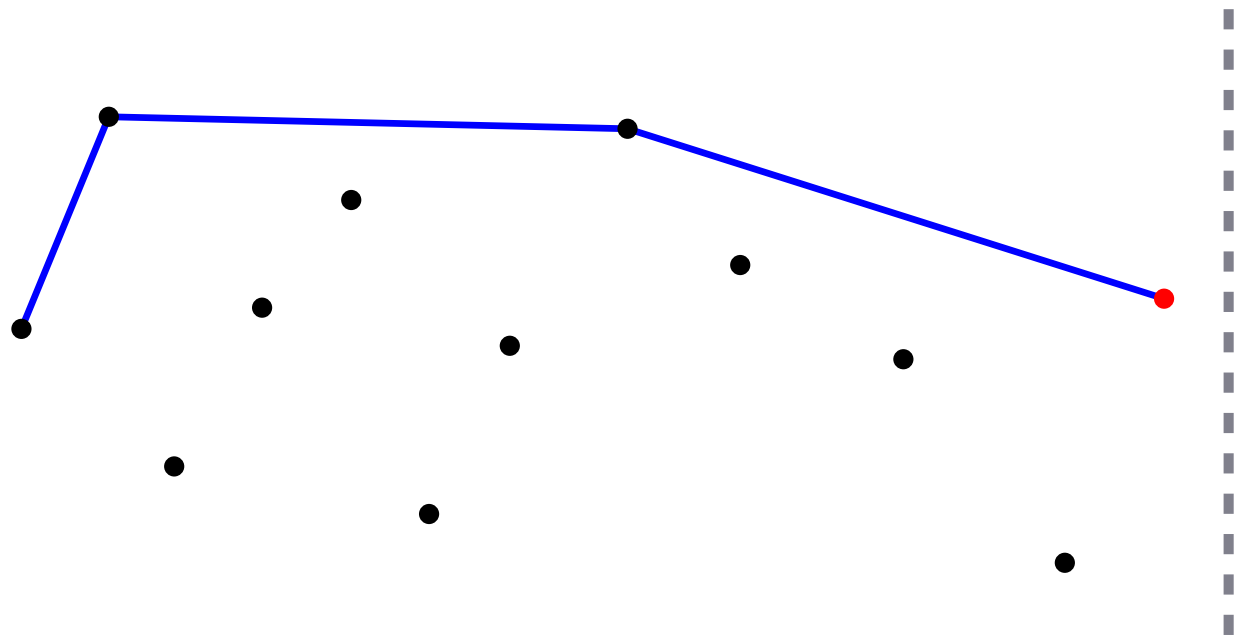
This process continues until the rightmost point is added to the upper hull.

Monotone Scan Algorithm (21)



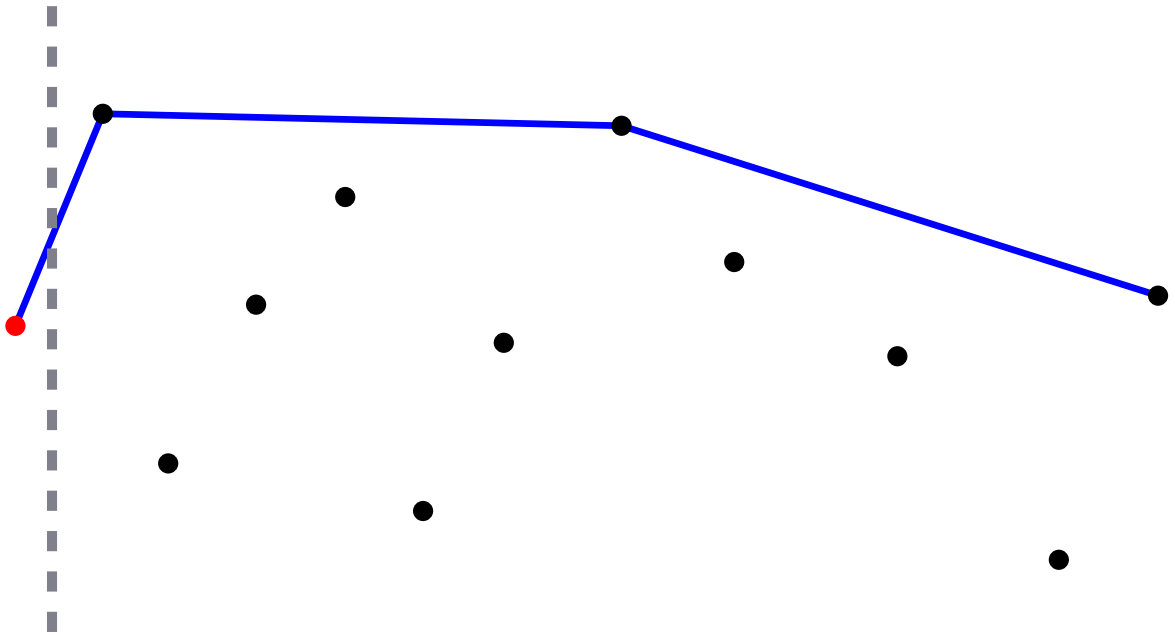
This process continues until the rightmost point is added to the upper hull.

Monotone Scan Algorithm (22)



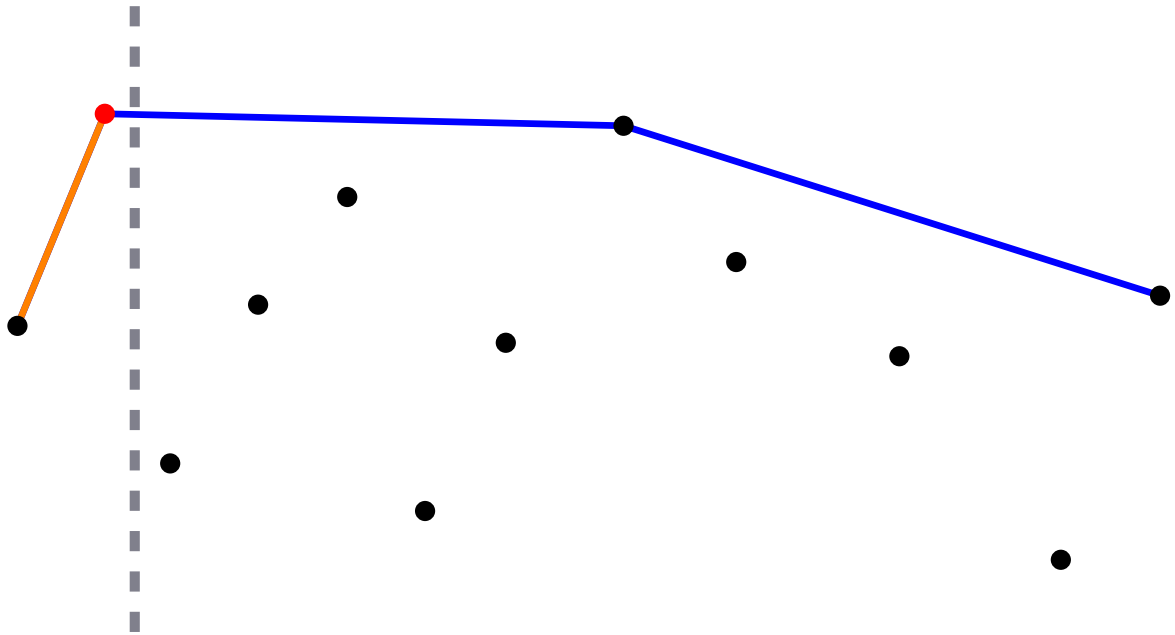
This process continues until the rightmost point is added to the upper hull.

Monotone Scan Algorithm (23)



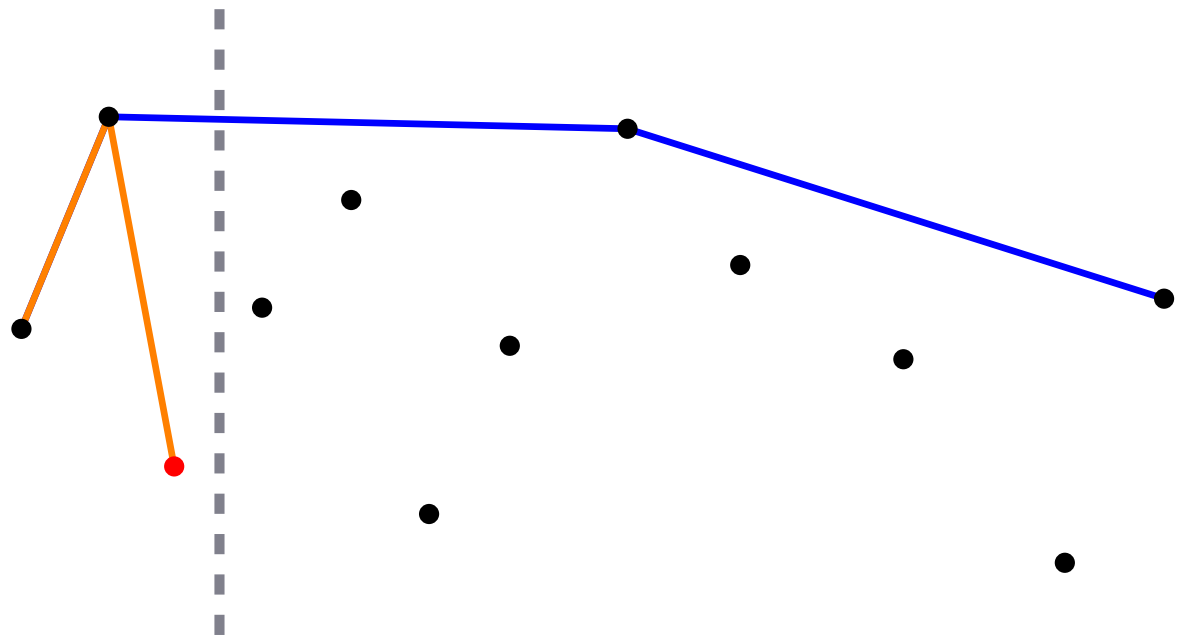
After the upper hull is finished, the lower hull is constructed separately, by again iterating through the points in left-to-right order.

Monotone Scan Algorithm (24)



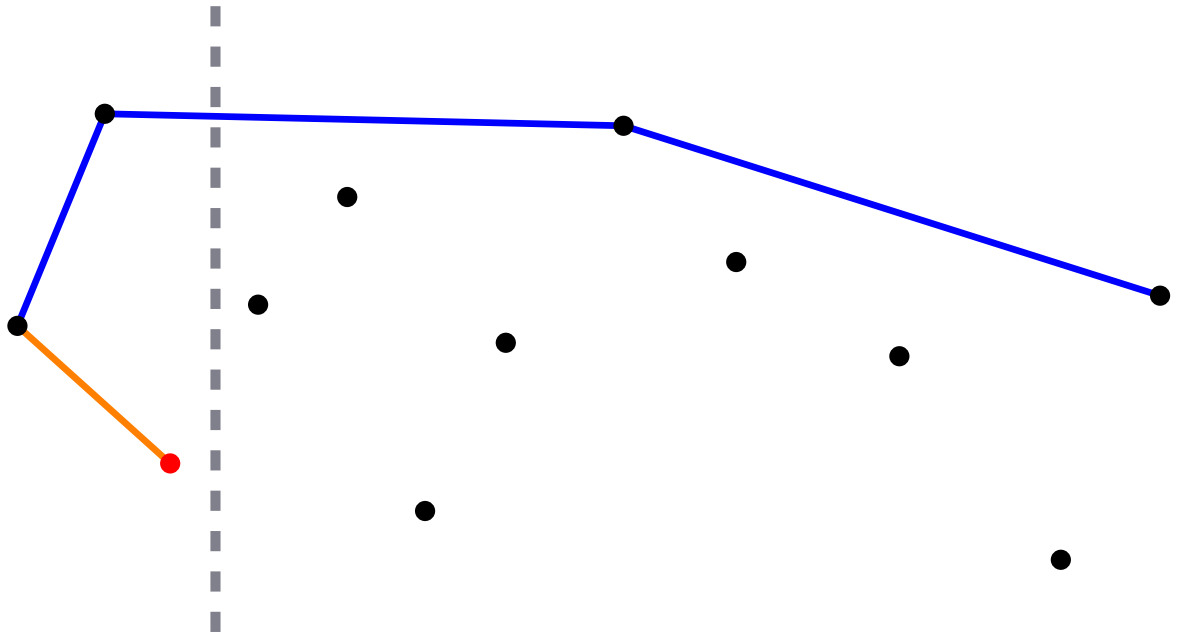
For the lower hull, every three points must form a **left turn**, so points forming right turns are removed.

Monotone Scan Algorithm (25)



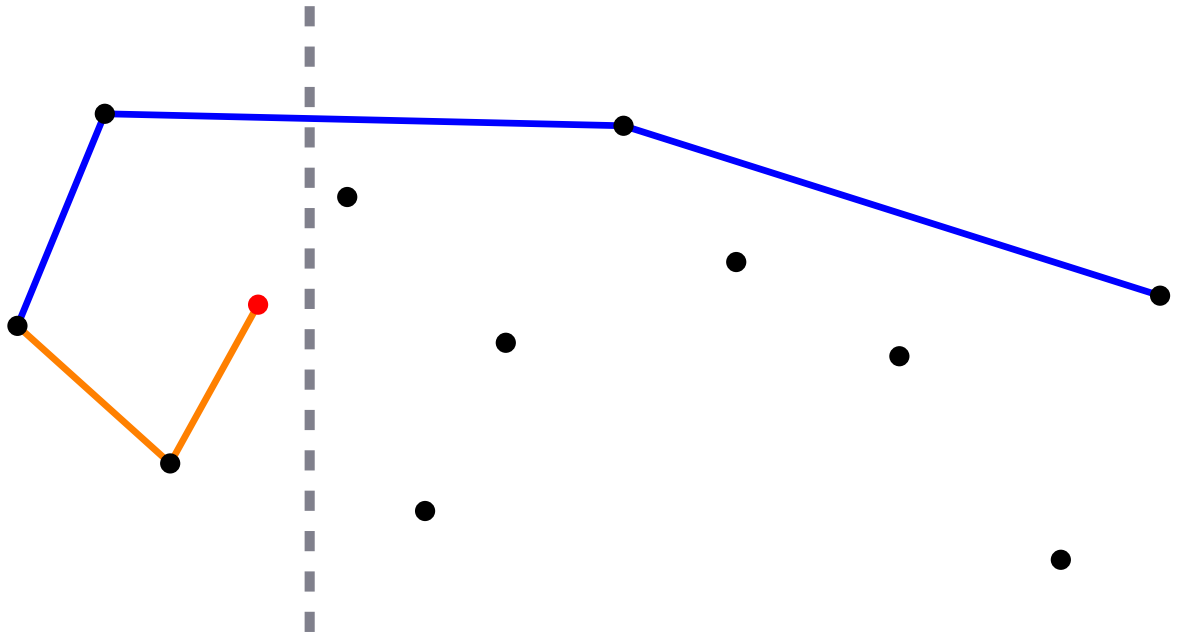
For example, the case above is a right turn, so the middle point is removed.

Monotone Scan Algorithm (26)

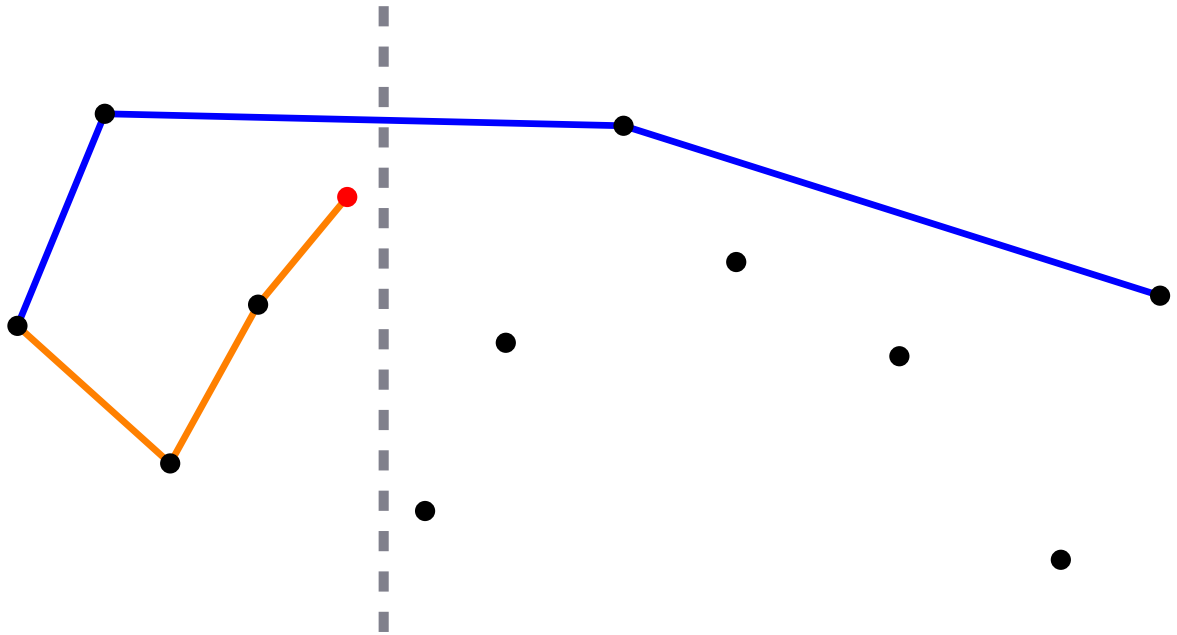


For example, the case above is a right turn, so the middle point is removed.

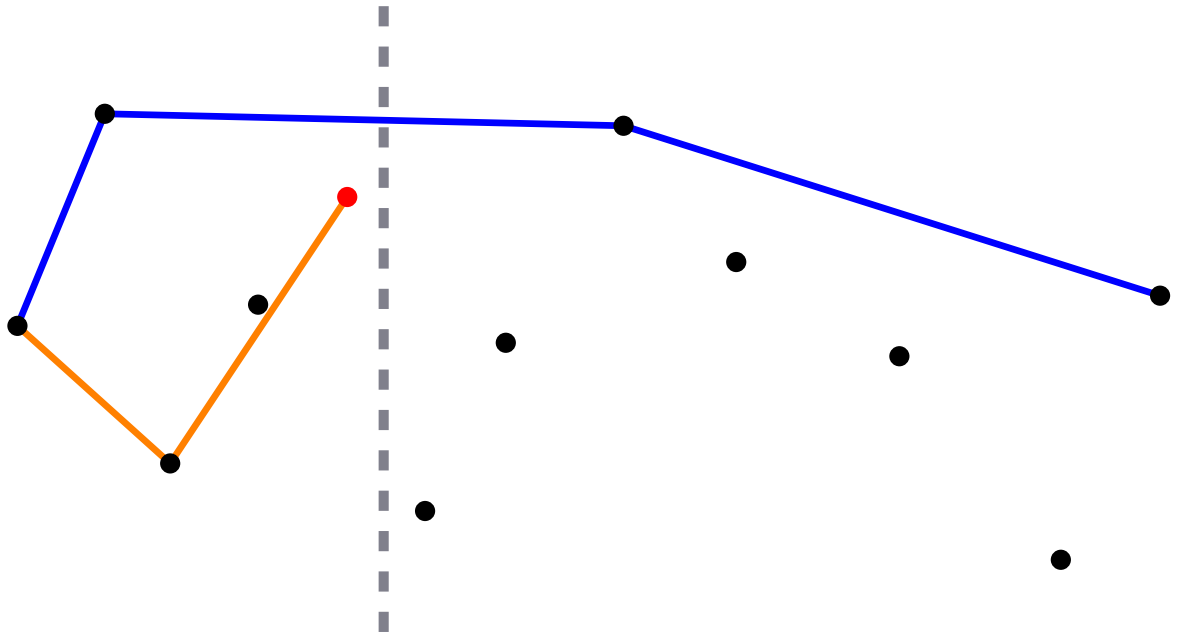
Monotone Scan Algorithm (27)



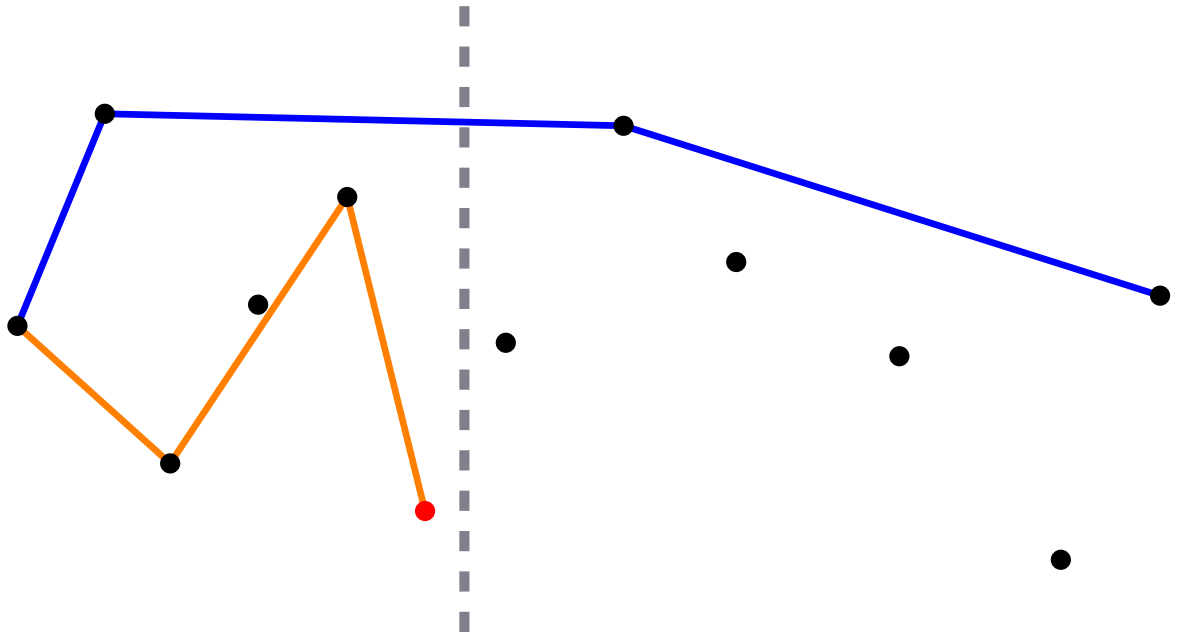
Monotone Scan Algorithm (28)



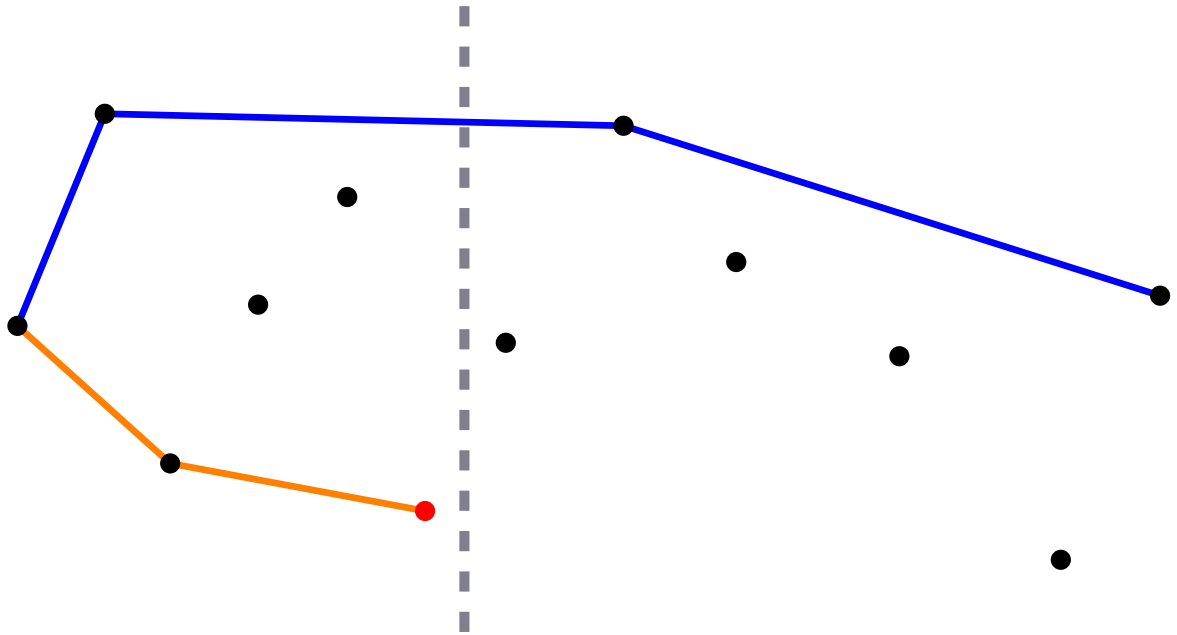
Monotone Scan Algorithm (29)



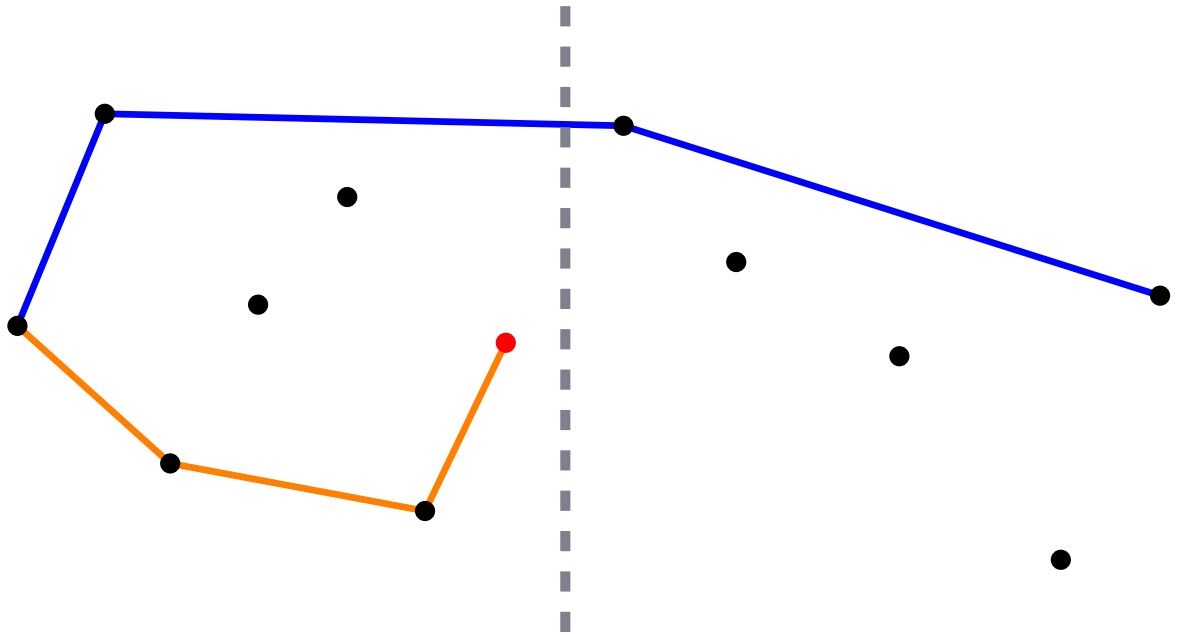
Monotone Scan Algorithm (30)



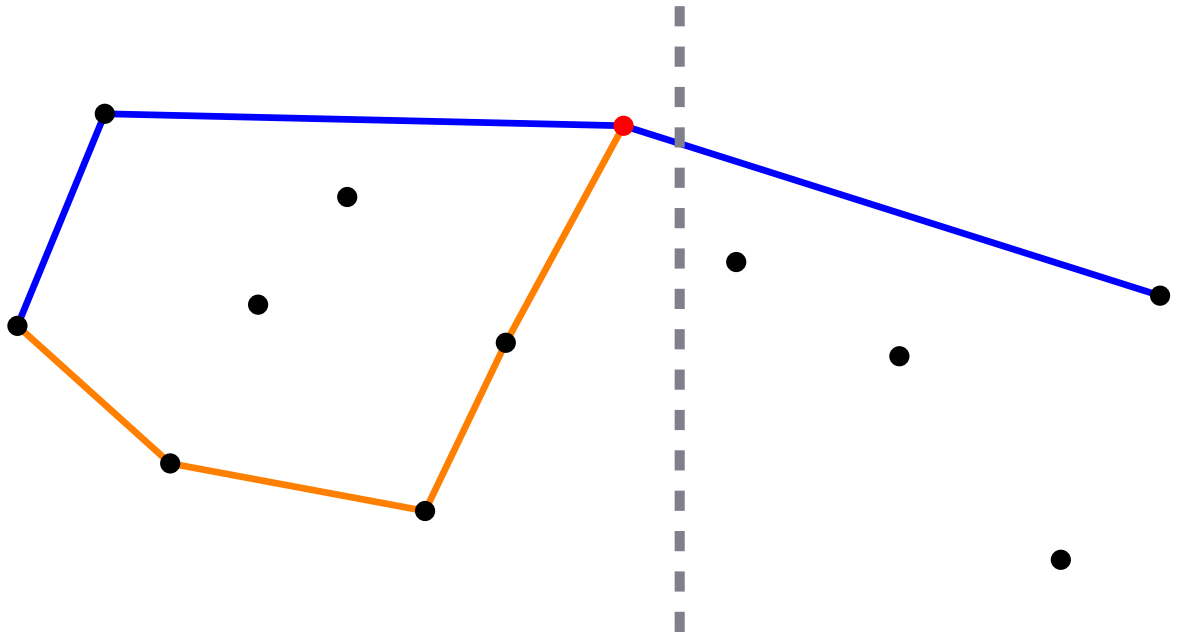
Monotone Scan Algorithm (31)



Monotone Scan Algorithm (32)

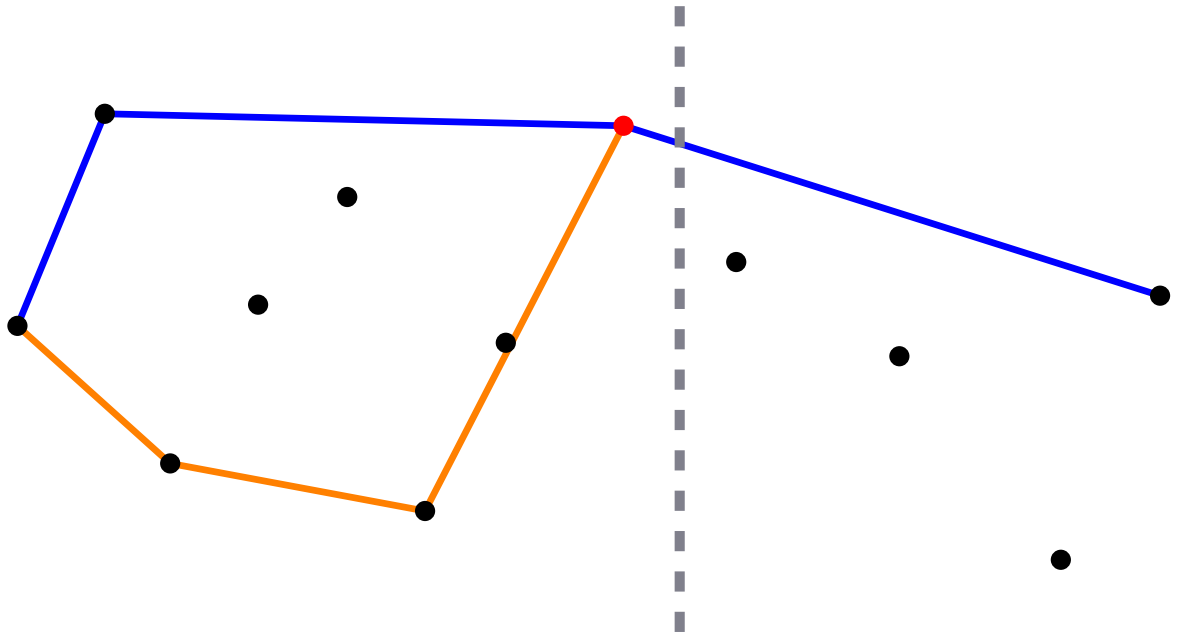


Monotone Scan Algorithm (33)

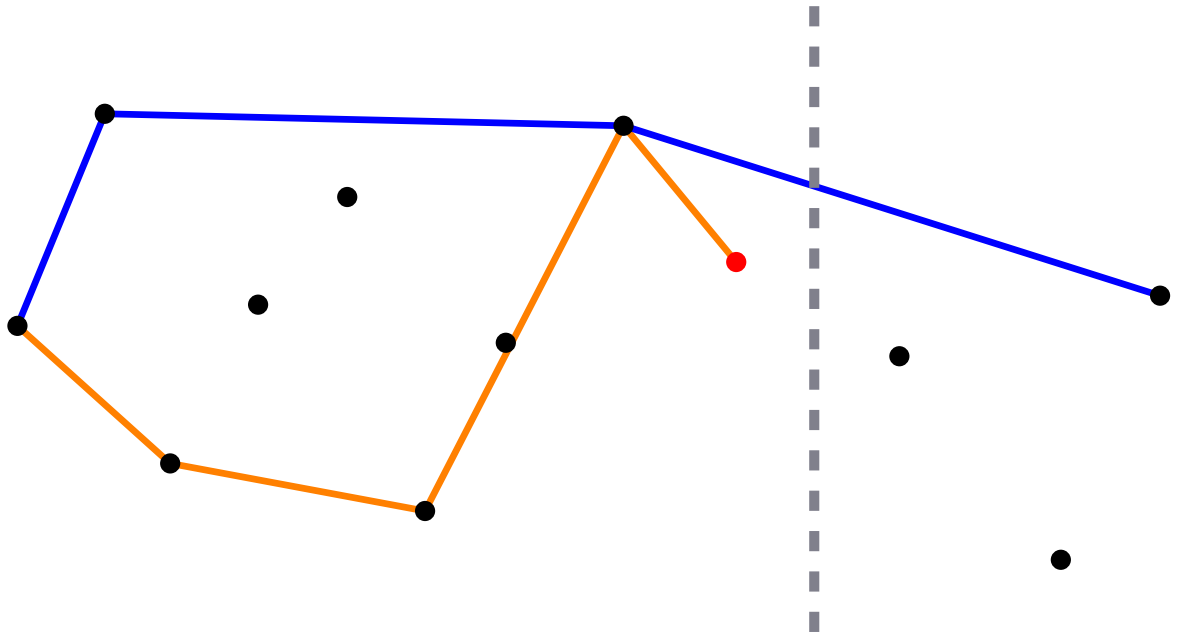


Notice that at this step, the lower and upper hulls intersect. Since there are still points left to add, the algorithm is not finished yet.

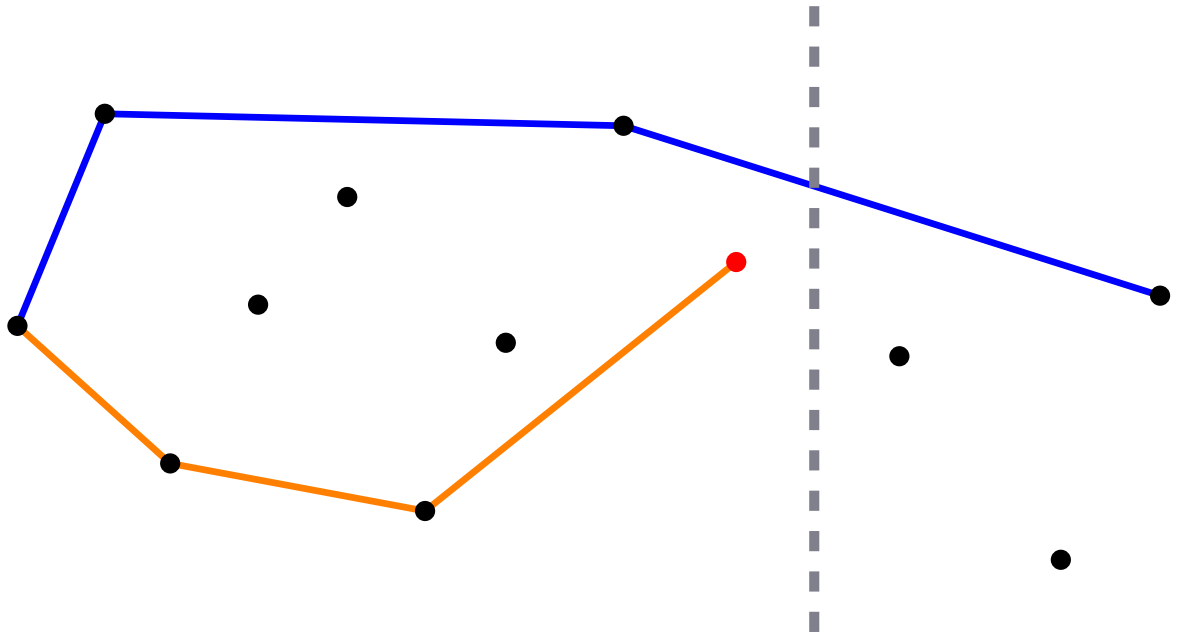
Monotone Scan Algorithm (34)



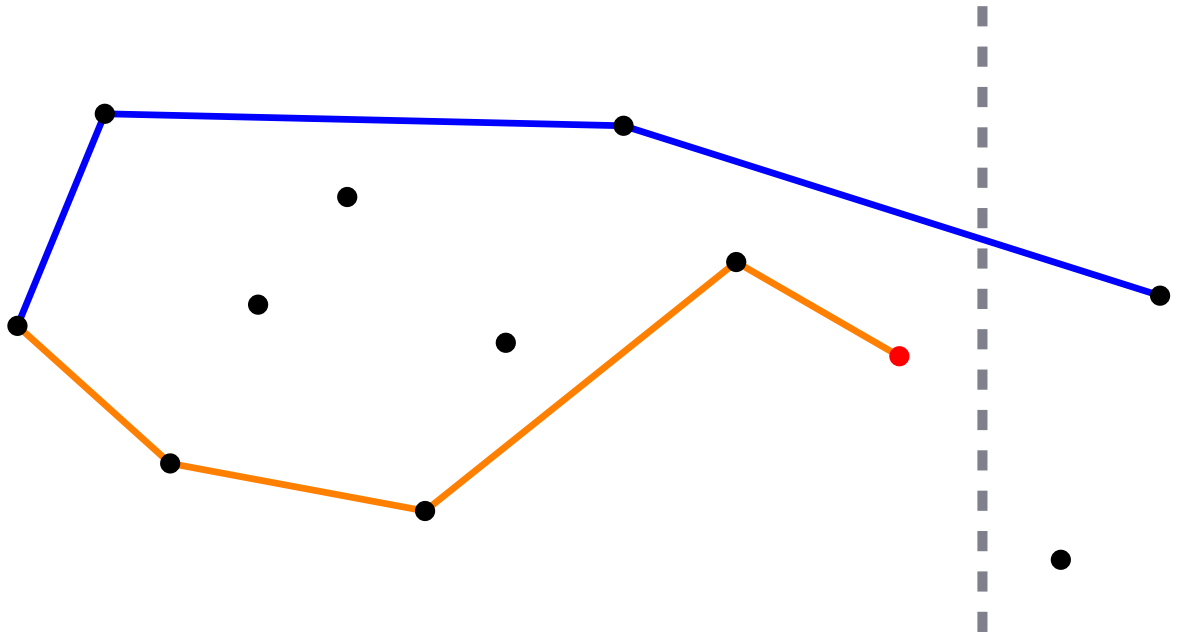
Monotone Scan Algorithm (35)



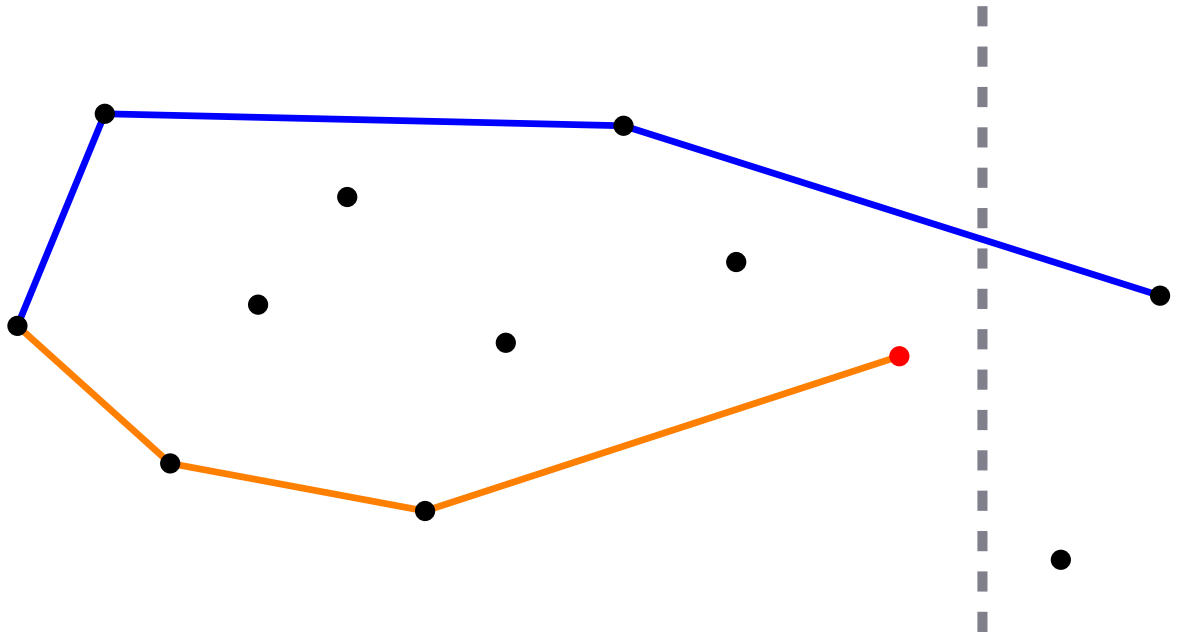
Monotone Scan Algorithm (36)



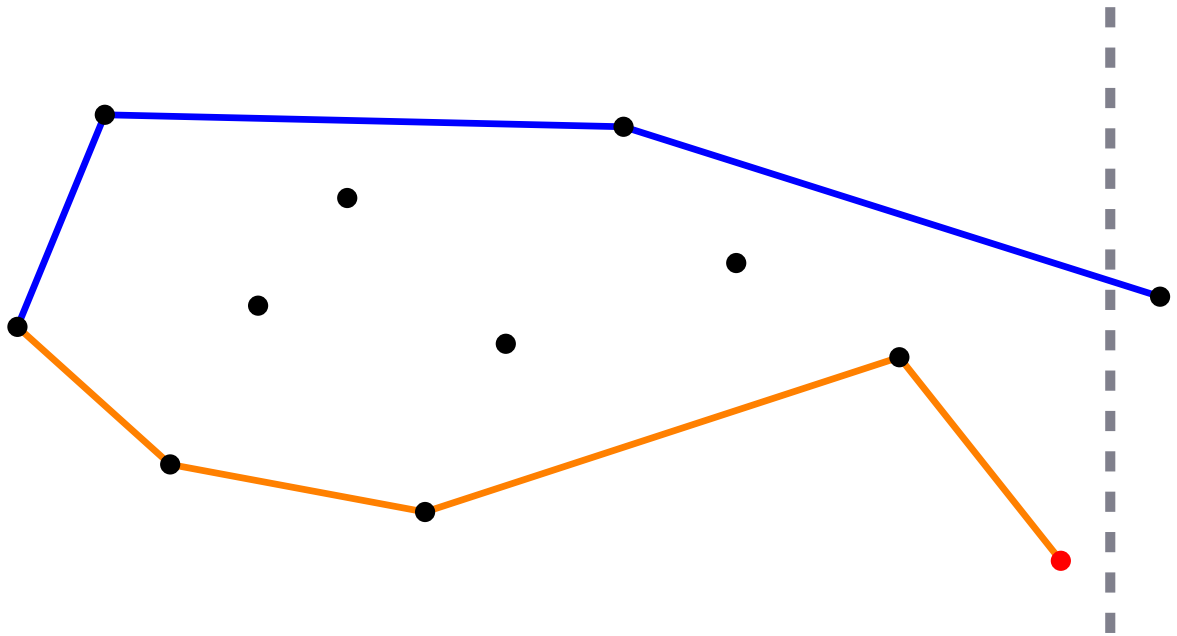
Monotone Scan Algorithm (37)



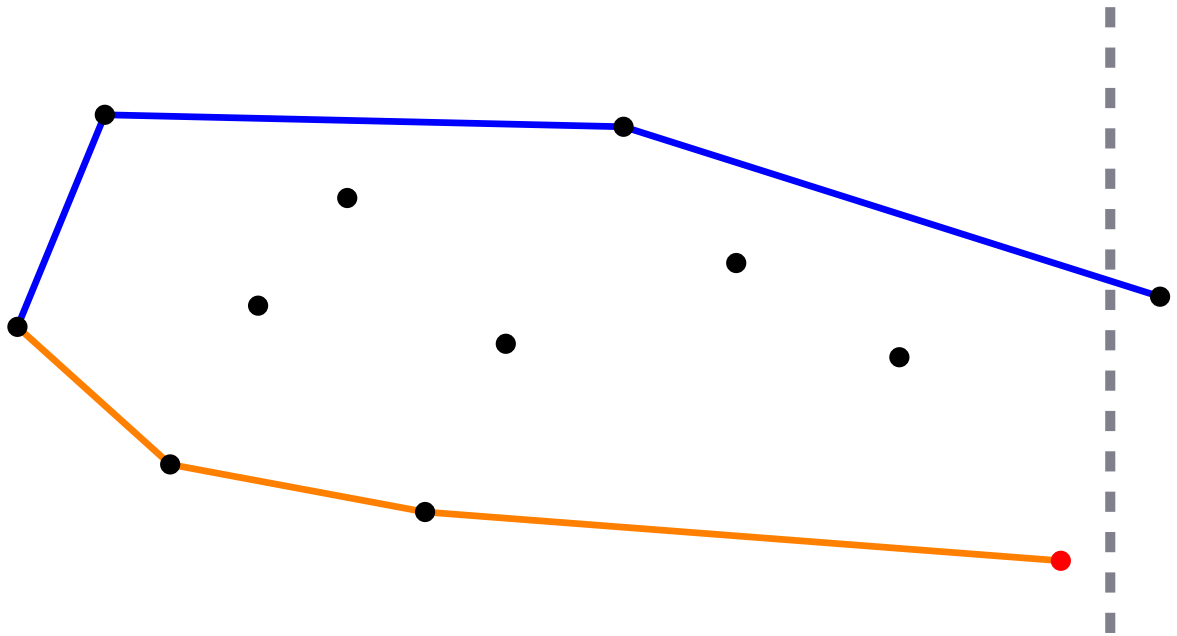
Monotone Scan Algorithm (38)



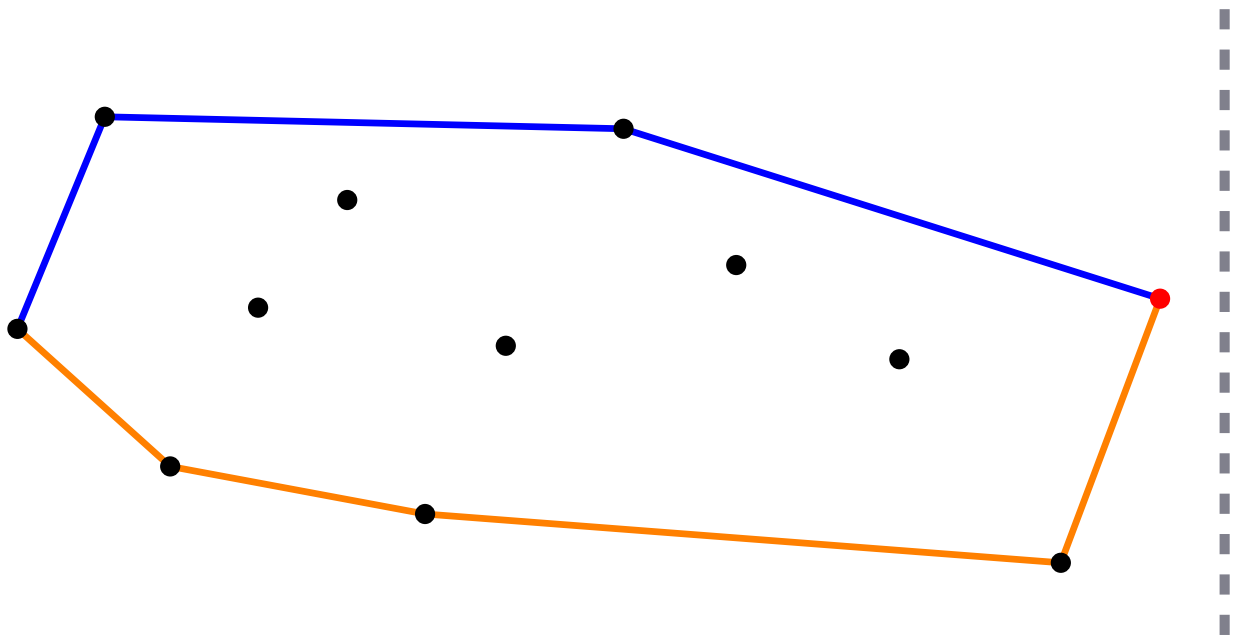
Monotone Scan Algorithm (39)



Monotone Scan Algorithm (40)

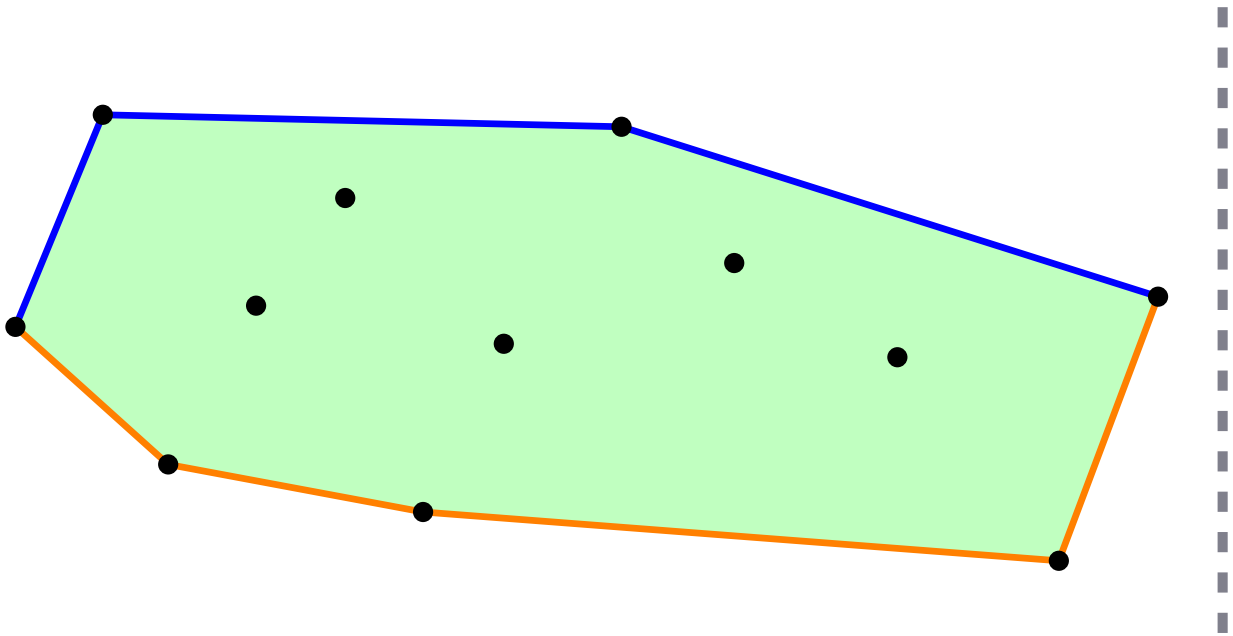


Monotone Scan Algorithm (41)



When the rightmost point is processed, the lower hull will be complete. By the observations in the previous example, we know that the upper and lower hulls will always have the same first and last points.

Monotone Scan Algorithm (42)



The complete convex hull can be obtained by combining the upper hull with the lower hull (where the lower hull is traversed in reverse order).

Monotone Scan Algorithm (43)

```
procedure MonotoneScan(S)
    n ← Size of S
    Sort the points in S by x-coordinate (using y-coordinate in case of ties)
    U, L ← Empty lists.
    Add S[0] and S[1] to U.
    Add S[0] and S[1] to L.
    for each point p = S[1], S[2], . . . , S[n − 1] do
        while |U| ≥ 2 do
            a, b ← last two points in U
            if T(a, b, p) > 0 then
                Break //Right turn (upper hull is consistent)
            else
                Delete last point in U //Delete left turn
            end if
        end while
        Add p to U
    end for
    for each point p = S[1], S[2], . . . , S[n − 1] do
        while |L| ≥ 2 do
            a, b ← last two points in L
            if T(a, b, p) < 0 then
                Break //Left turn (lower hull is consistent)
            else
                Delete last point in L //Delete right turn
            end if
        end while
        Add p to L
    end for
    H ← U[0], U[1], . . . , U[length(U) − 2], L[length(L) − 1], L[length(L) − 2], . . . , L[1]
    return H
end procedure
```

Pseudocode for this approach is shown above.

Monotone Scan Algorithm (44)

On a point set S containing n points, sorting S by x-coordinate requires $\Theta(n \log_2 n)$ time.

The loops to compute U and L each require $\Theta(n)$ time: Each point in S is added to U and L exactly once, and therefore can be deleted from U or L only once (at most). Each iteration of the inner loop either ends the loop or deletes a point from U or L, and each iteration of the outer loop adds a point to U or L. Therefore each pair of nested loops requires at most $O(2n)$ time.

Therefore, the complete algorithm has $\Theta(n \log_2 n)$ running time in the worst case. This can be reduced to $\Theta(n)$ if the point set is pre-sorted.