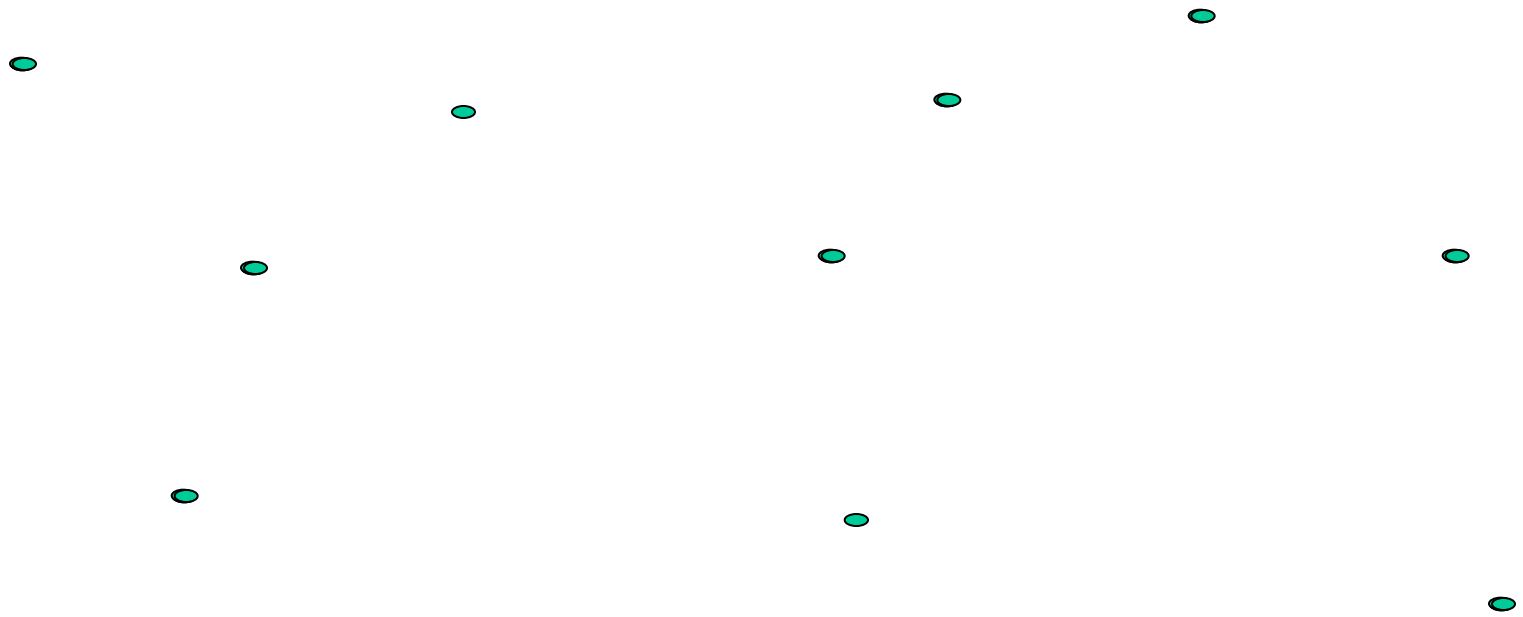
Problem 2: Constructing a simple polygon

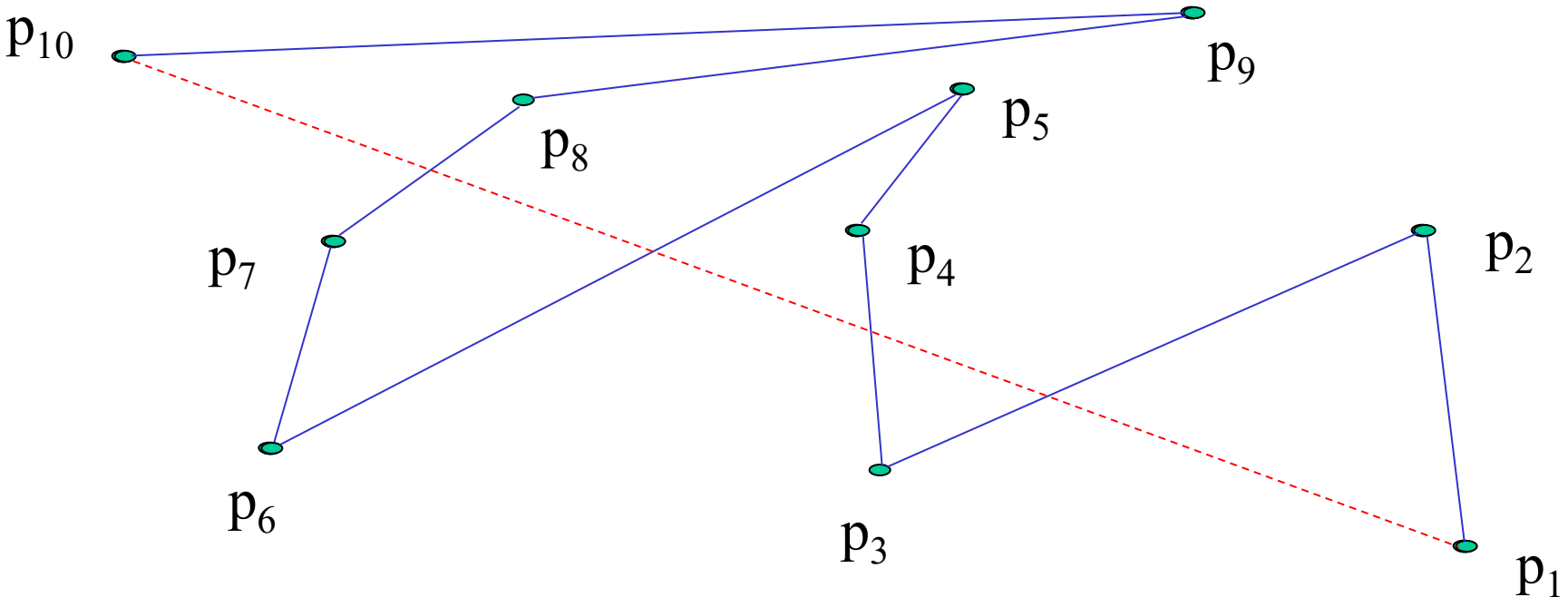
Problem: given a set of n points in the plane, connect them to form a simple polygon



The naïve method

Problem: given a set of n points in the plane, connect them to form a simple polygon

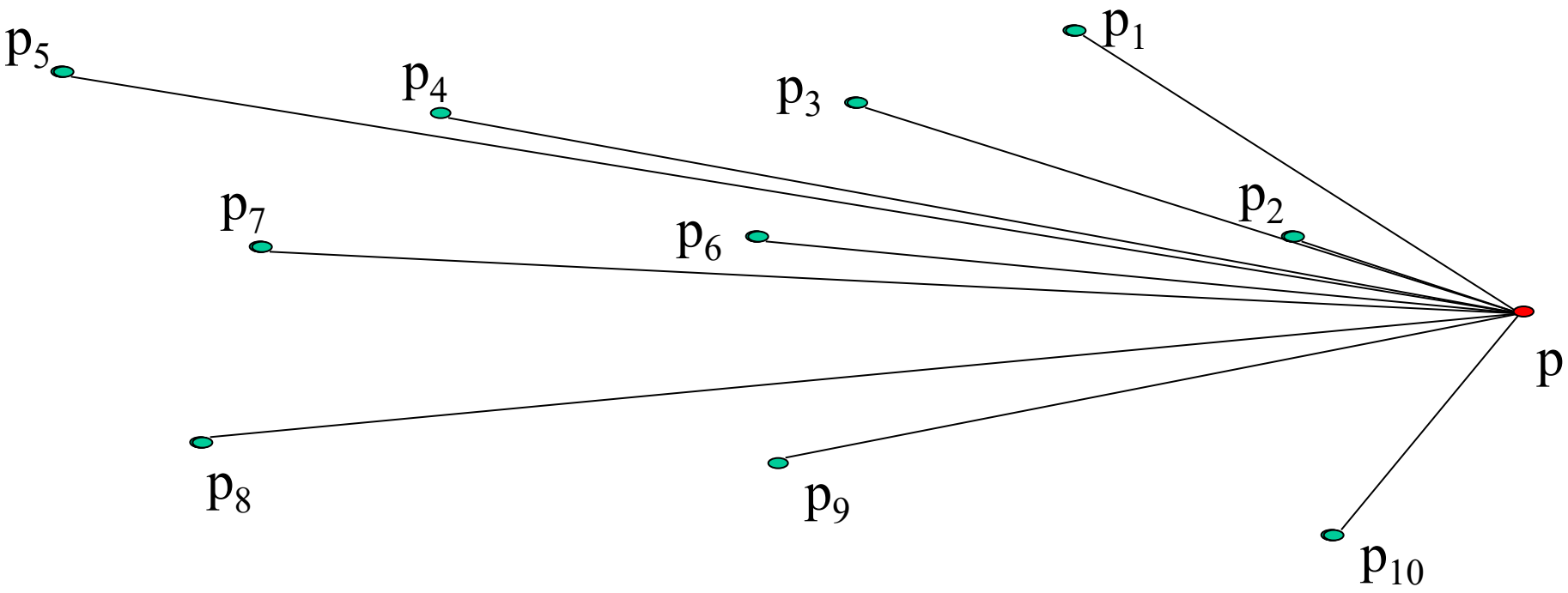
Naïve method - just join up the points in any order - won't work in general!



Idea (i) - use a circular scan

Problem: given a set of n points in the plane, connect them to form a simple polygon

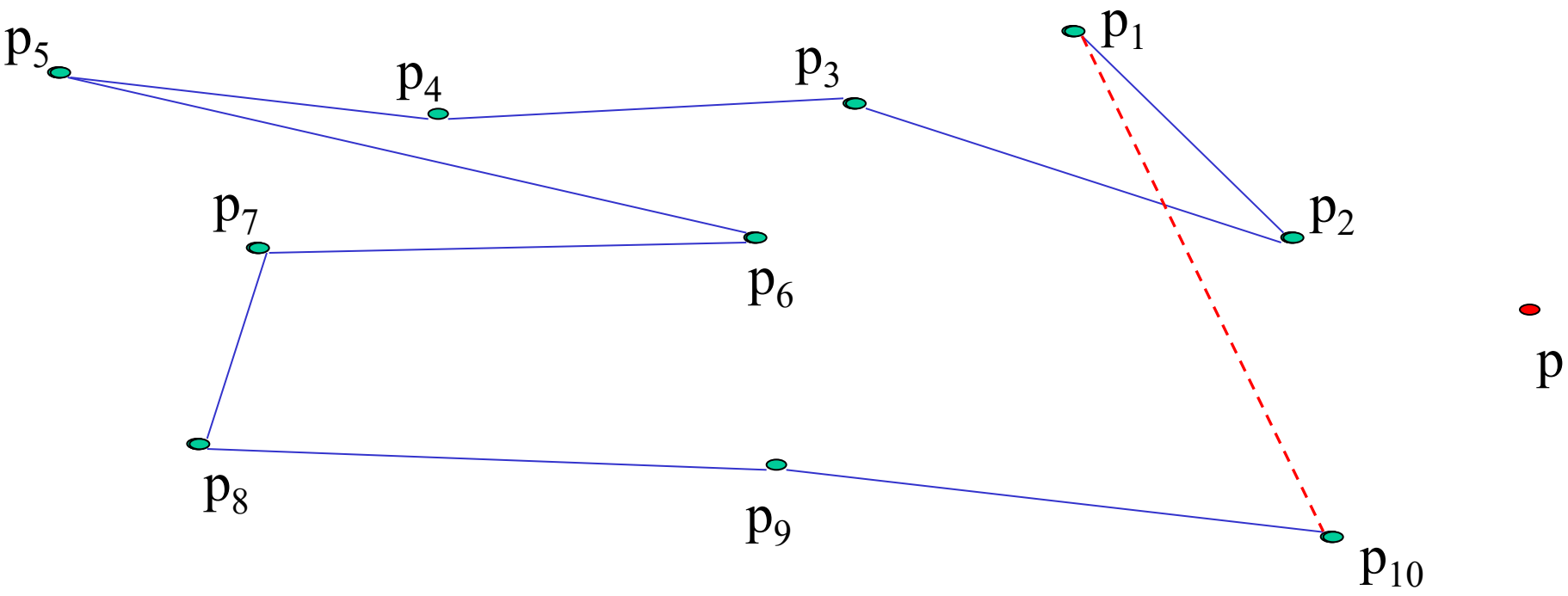
Idea (i): choose a point p as the *pivot* of a circular scan; connect the points in the order they are encountered.



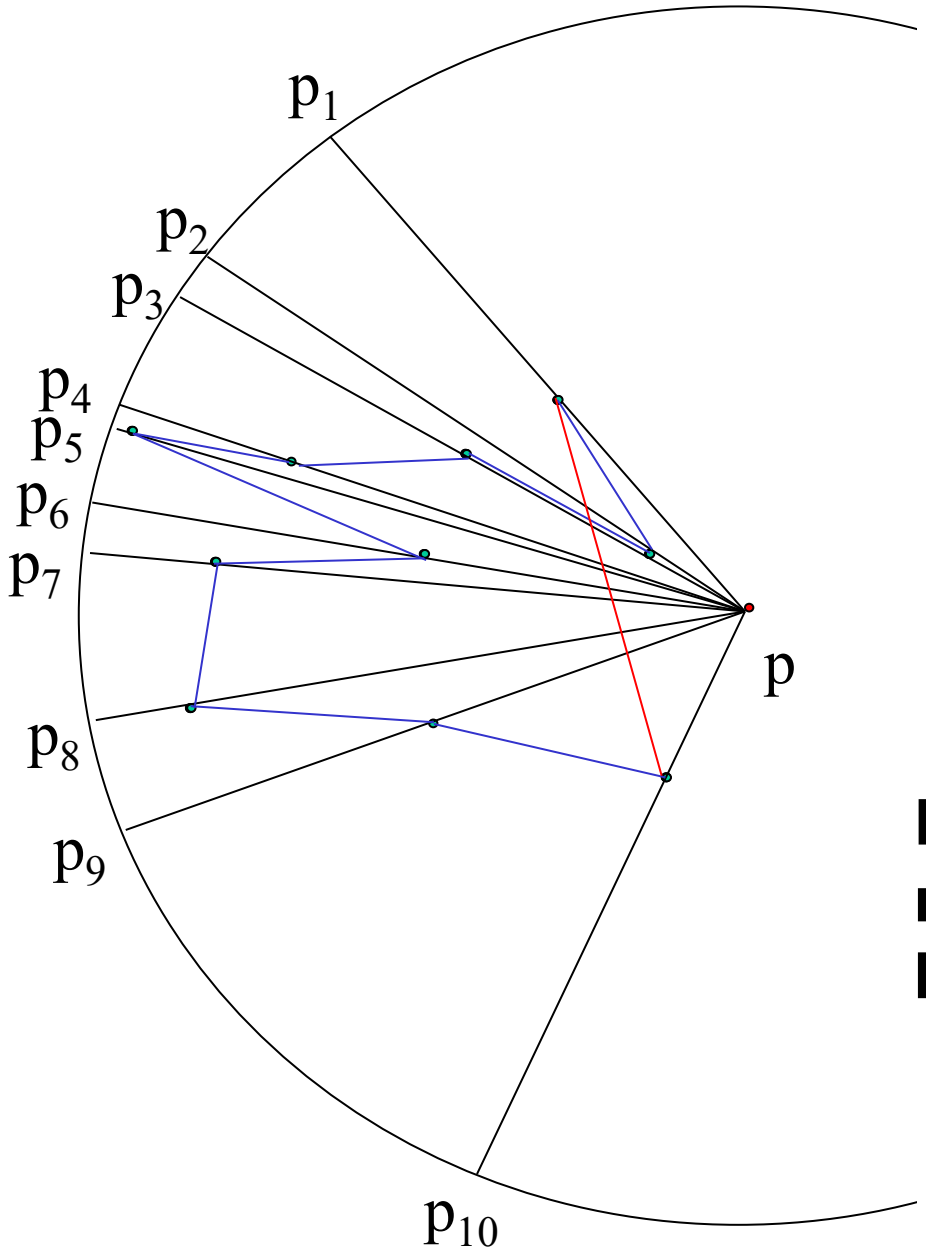
Idea (i) - use a circular scan

Problem: given a set of n points in the plane, connect them to form a simple polygon

Idea (i): choose a point p as the *pivot* of a circular scan; connect the points in the order they are encountered.



Regions in the circle corresponding to the circular scan

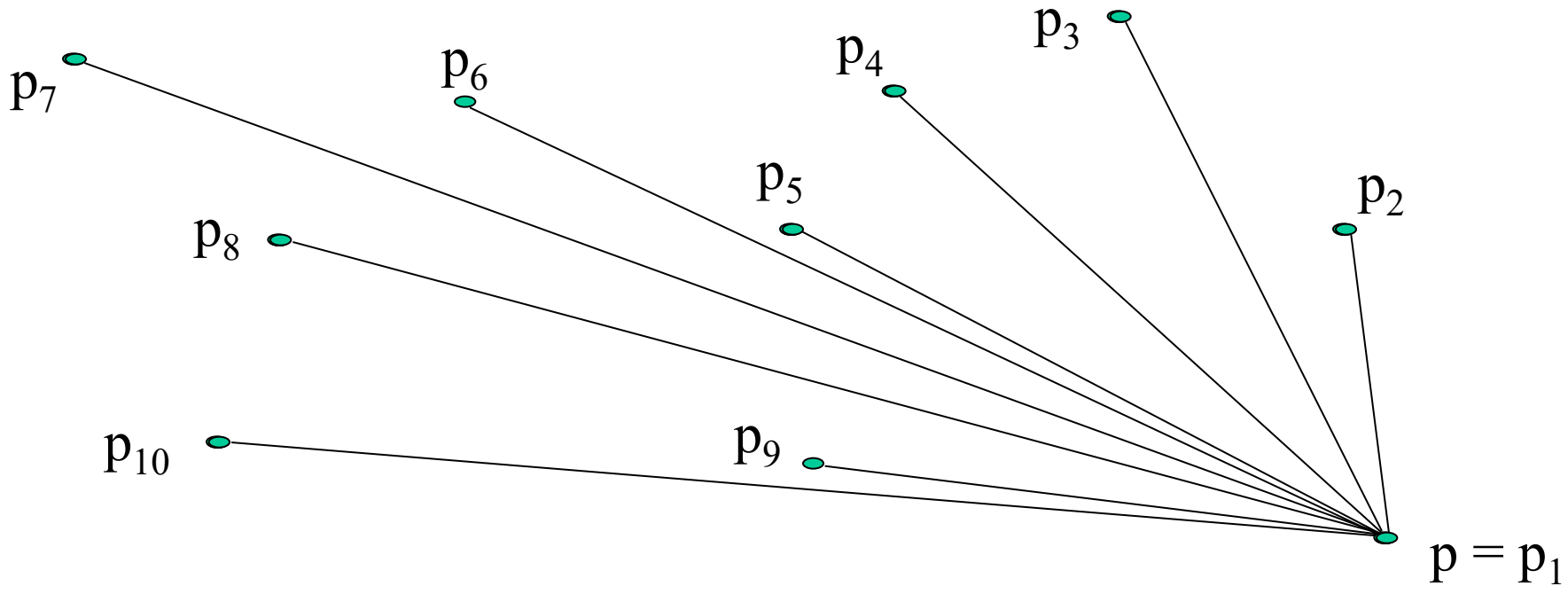


Line p_{10}—p_1 intersects regions of the circle belonging to other lines

Idea (ii) - choose a pivot for the scan

Problem: given a set of n points in the plane, connect them to form a simple polygon

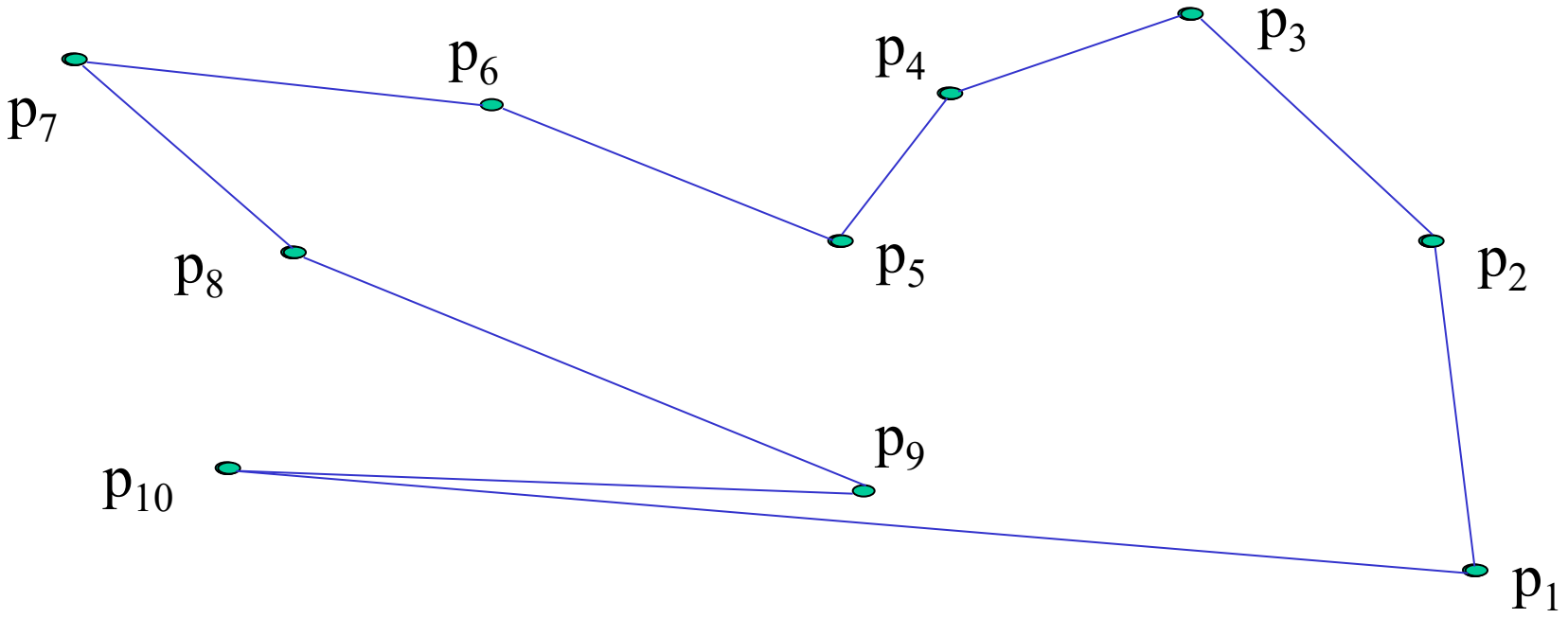
Idea (ii): choose p to be one of the points in the input point set, e.g. the point with the largest x-coordinate



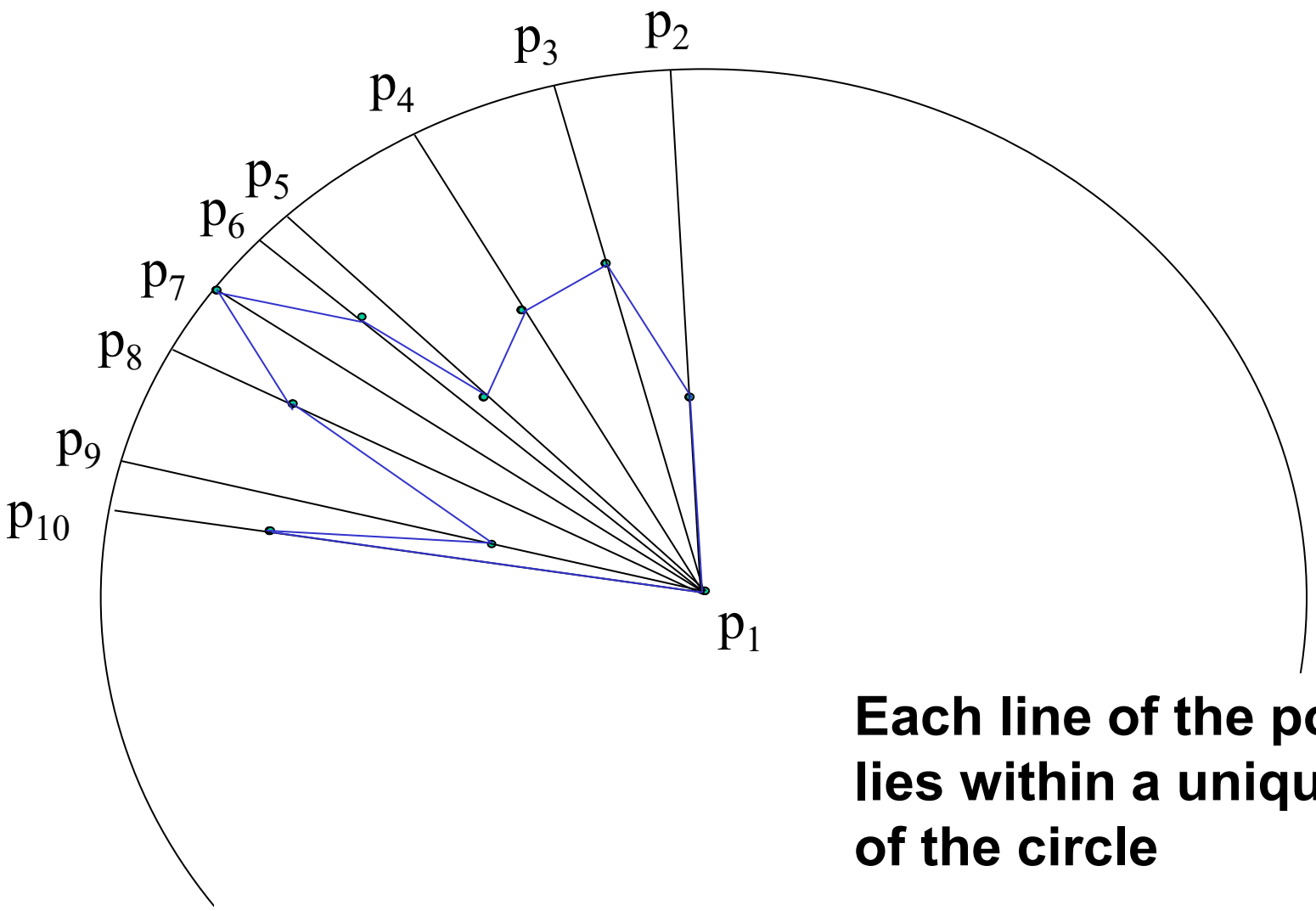
Idea (ii) - choose a pivot for the scan

Problem: given a set of n points in the plane, connect them to form a simple polygon

Idea (ii): choose p to be one of the points in the input point set, e.g. the point with the largest x-coordinate



Regions in the circle corresponding to the circular scan



Each line of the polygon lies within a unique region of the circle

Algorithm `simplePolygon`

```
/** Input: an arbitrary polygon, points, of n points,
  * Output: points, sorted to give a simple polygon */
public void simplePolygon (PointSet points) {
   double num, denom;
   double [] angle = new double[points.length];
   select_largest_xcoord(points);
   /* modifies points so that points[0] contains the point with
    * largest x-coord (and smallest y-coord if a tie) */
   for (int i=1; i<points.length; i++) {
      num = points[0].x - points[i].x;
      denom = points[i].y - points[0].y;
      if (denom > 0.0)          // points[i] above points[0]
         angle[i] = Math.atan(num/denom);
      else if (denom == 0.0)    // points[i] level with points[0]
         angle[i] = Math.PI/2.0;
      else                      // points[i] below points[0]
         angle[i] = Math.atan(num/denom) + Math.PI;
   }
   sort(points, 1, points.length, angle);
   /* sorts points[1..n-1] using the values of angle[1..n-1] */
}
```

Analysis of algorithm for constructing simple polygon

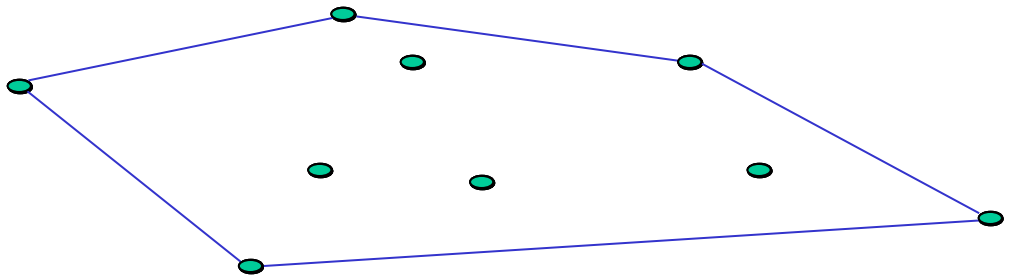
- **Complexity of the algorithm**
  - loop is O(n), where n is the number of points
  - sort is O(n log n)
  - so overall O(n log n)
- **Complication occurs if two or more points are collinear with the pivot (points[0])**
  - if so, further sort these points in increasing order of distance from pivot
- **Two aspects can be simplified:**
  - do not need to use arctan function to calculate angles – can sort according to gradients of lines
  - do not need to compute distances from the pivot
- **Must take account of a further special case (tutorial ex)**

Problem 3: Finding the Convex Hull

The *convex hull* of a set of points (in the plane) is the smallest convex polygon that contains the points.

Here, "smallest" refers to the total length of the edges (or, equivalently, the area).

e.g.

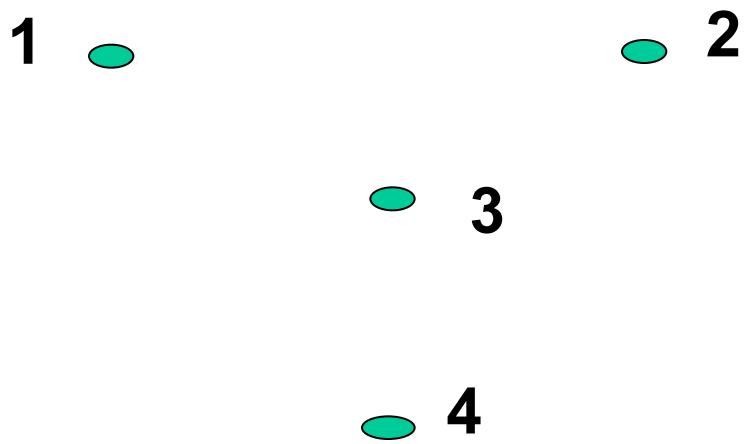


Problem: given a set P of n points in the plane, find their convex hull.

***Graham Scan* algorithm: O(n log n) time.**

A first example

The convex hull need not include all the points in P



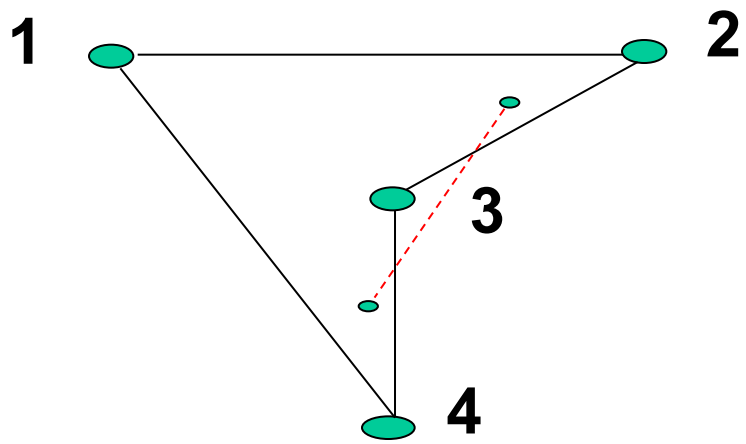
4!=24 ways to connect up points, but only 3 of these give distinct polygons:

- **1234**
- **1243**
- **1324**

None of these gives a convex polygon!

A first example

The convex hull need not include all the points in P



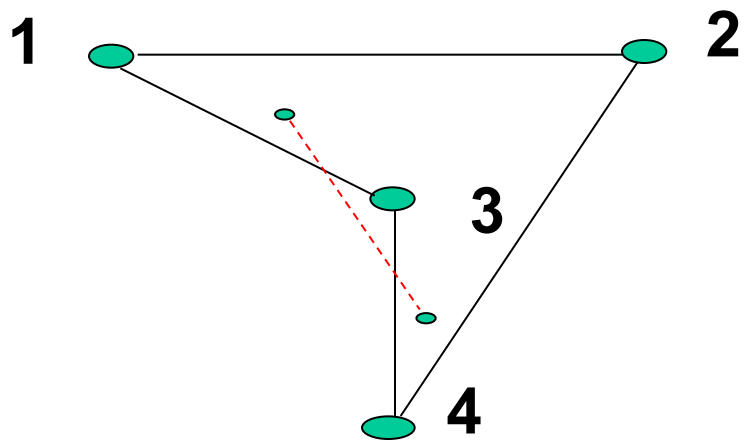
4!=24 ways to connect up points, but only 3 of these give distinct polygons:

- **1234**
- **1243**
- **1324**

None of these gives a convex polygon!

A first example

The convex hull need not include all the points in P



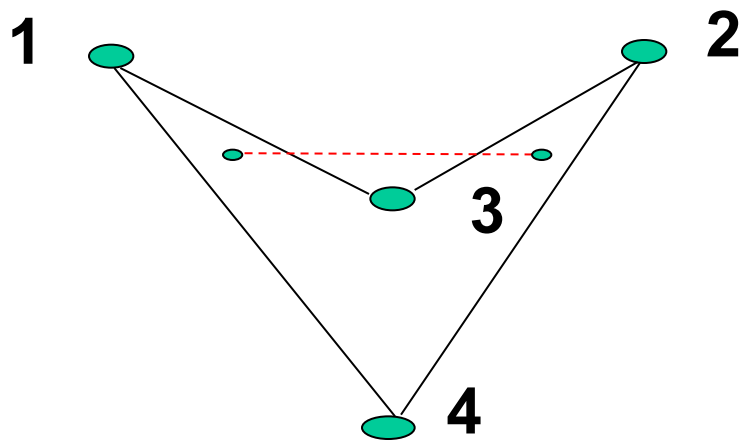
4!=24 ways to connect up points, but only 3 of these give distinct polygons:

- **1234**
- **1243**
- **1324**

None of these gives a convex polygon!

A first example

The convex hull need not include all the points in P



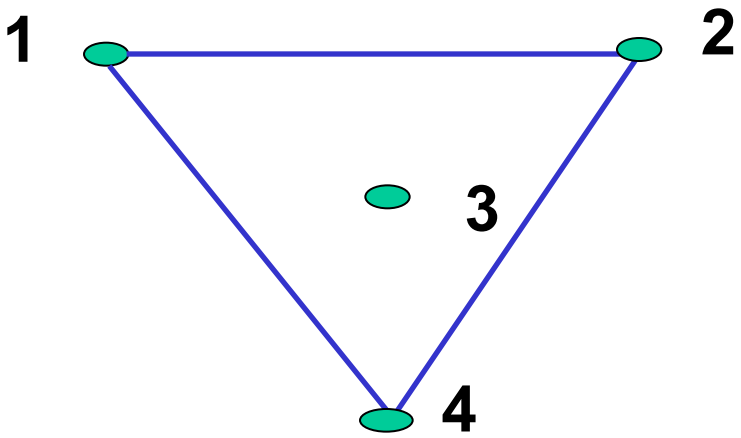
4!=24 ways to connect up points, but only 3 of these give distinct polygons:

- **1234**
- **1243**
- **1324**

None of these gives a convex polygon!

A first example

The convex hull need not include all the points in P

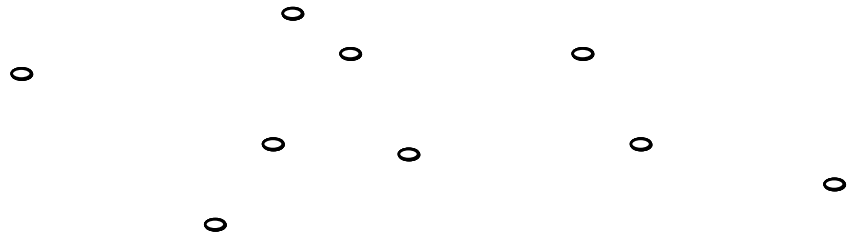


The convex hull

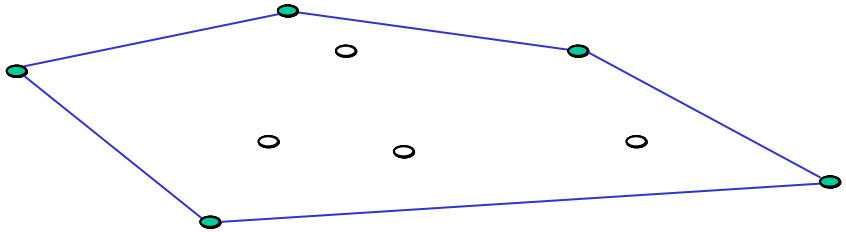
Properties of the convex hull

Let P be a set of points in the plane

- **Every point of the convex hull of P is itself a member of P**



The point set



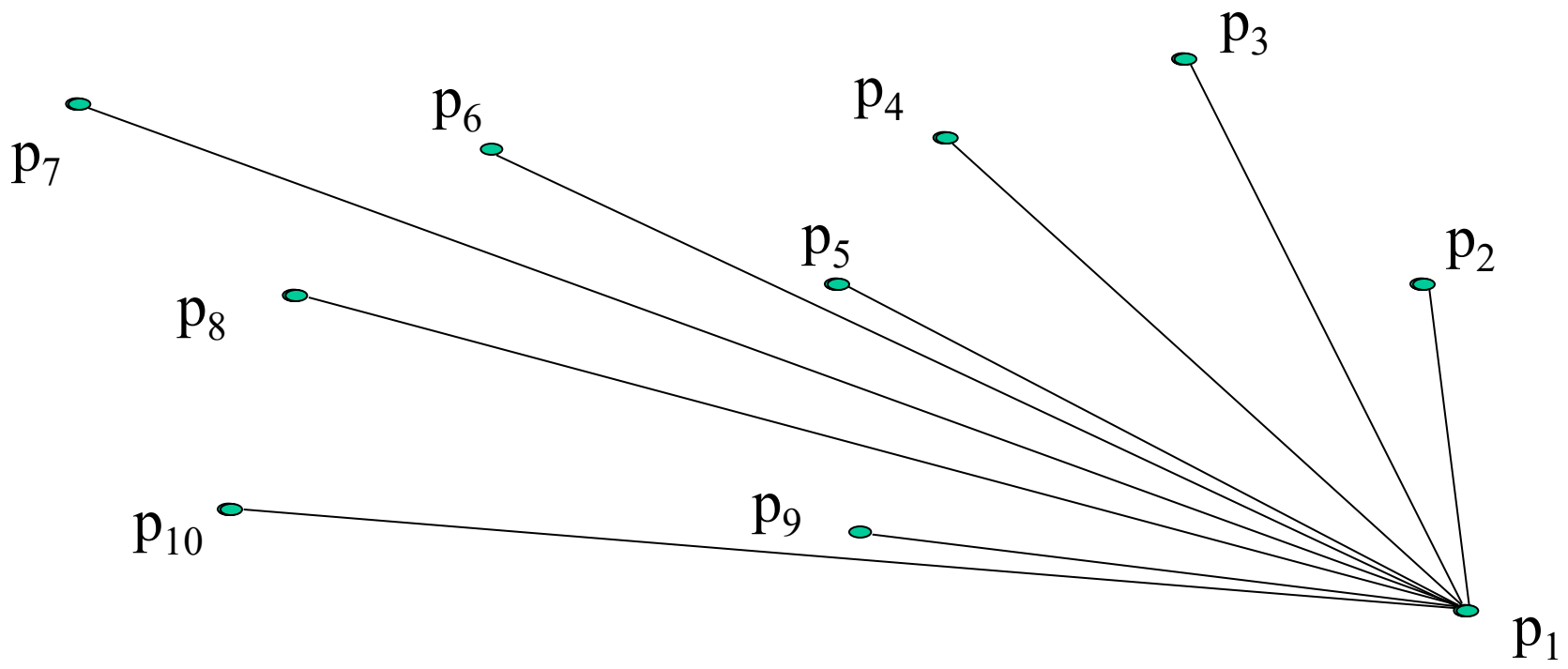
The convex hull

- **The points of P with the largest / smallest x / y coordinates are points of the convex hull of P**
- **The furthest pair of points in P are points of the convex hull of P**

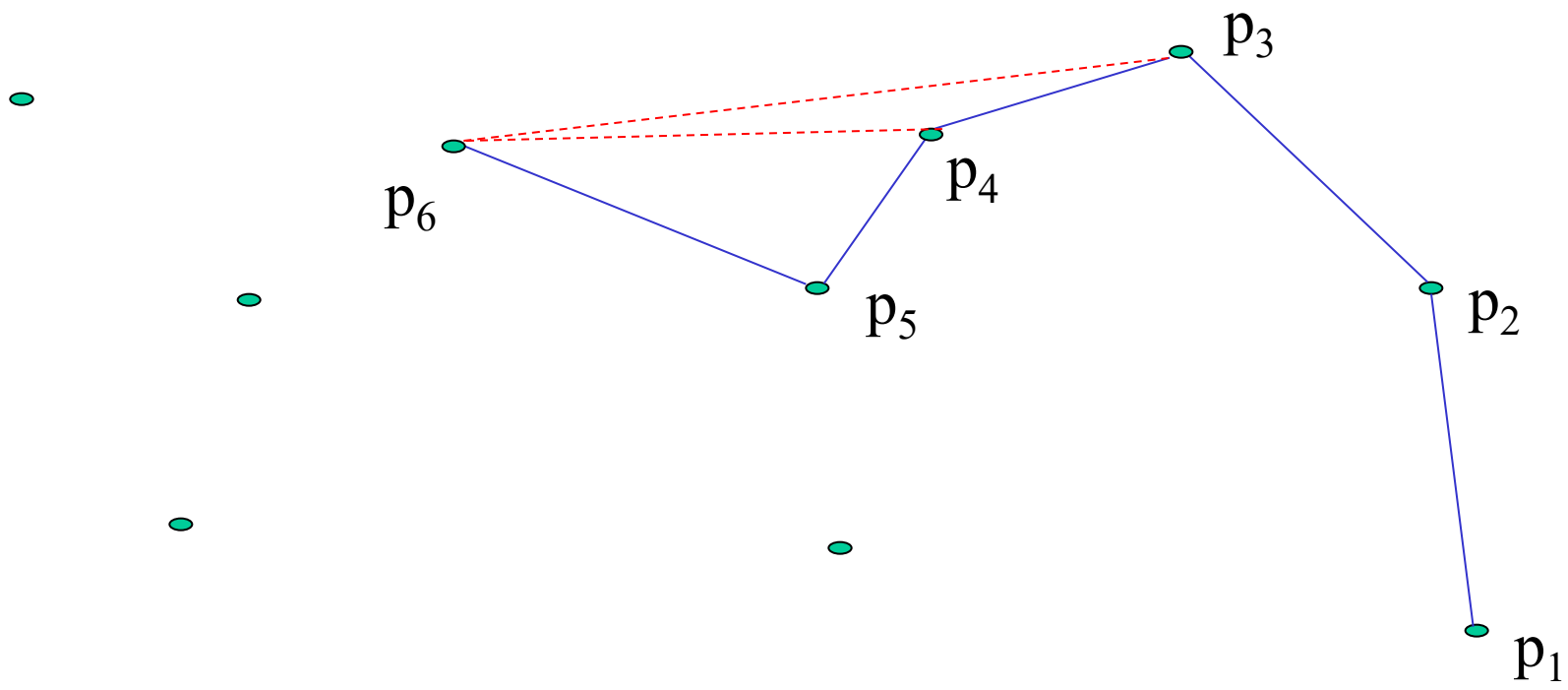
Graham scan algorithm – the basic idea:

- Choose a point known to be on the hull as *pivot* – say the point with the largest x coordinate (using the smallest y coordinate to break a tie if necessary)
- Scan the remaining points in order of angle to the pivot
- For each point, include it in temporary hull, possibly excluding one or more of its predecessors

Graham scan algorithm

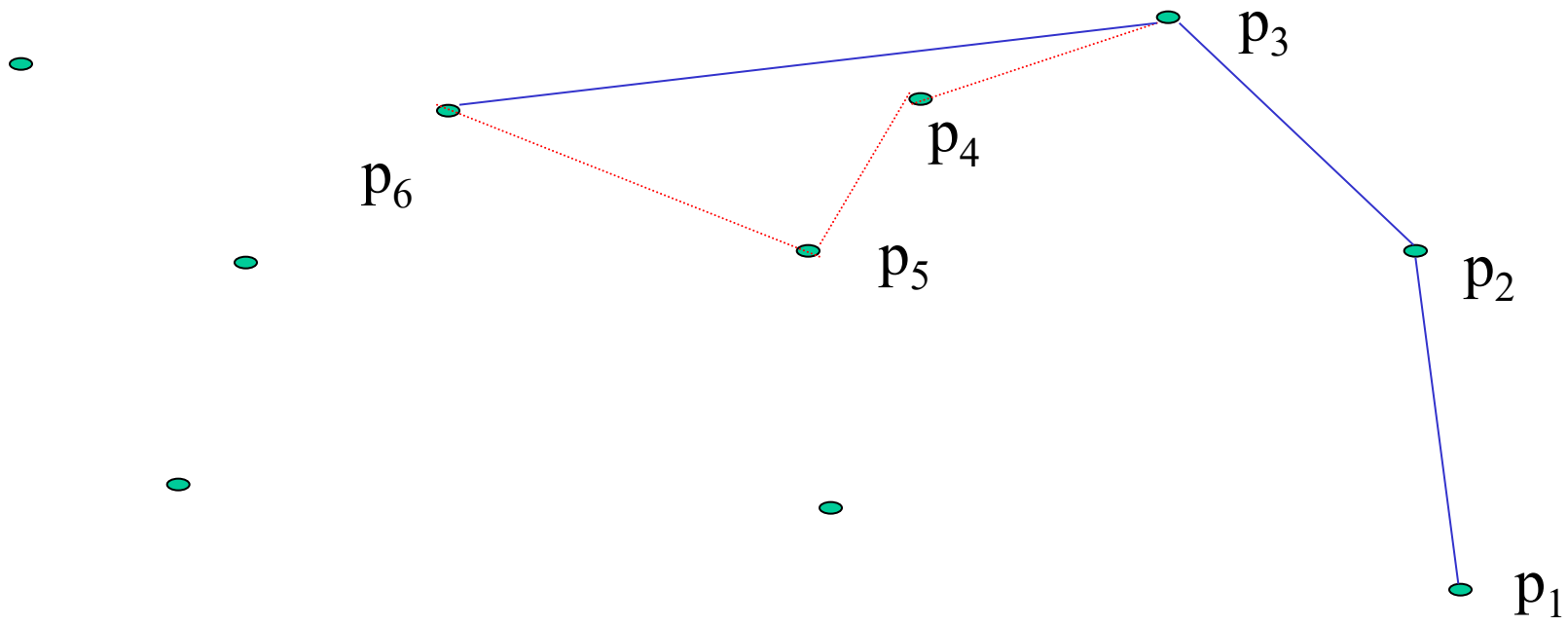


Graham scan algorithm



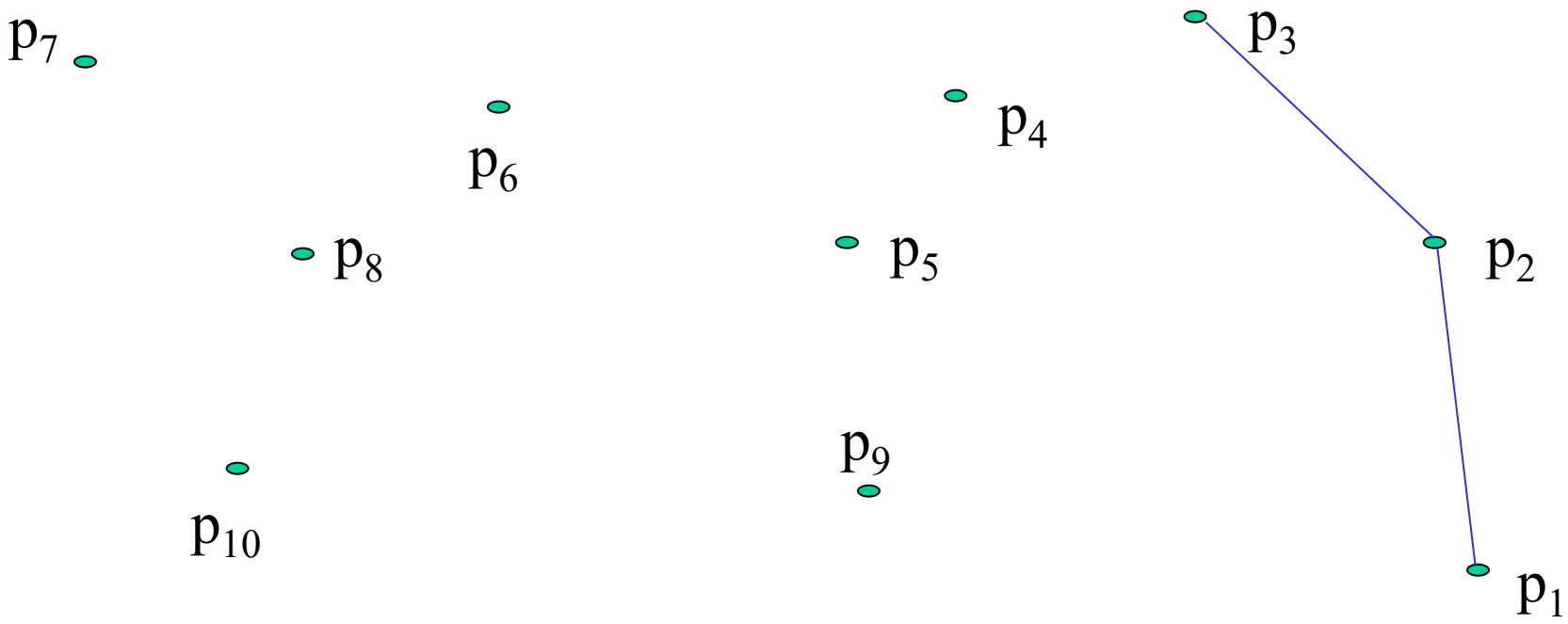
p_5 must be excluded

p_4 must be excluded

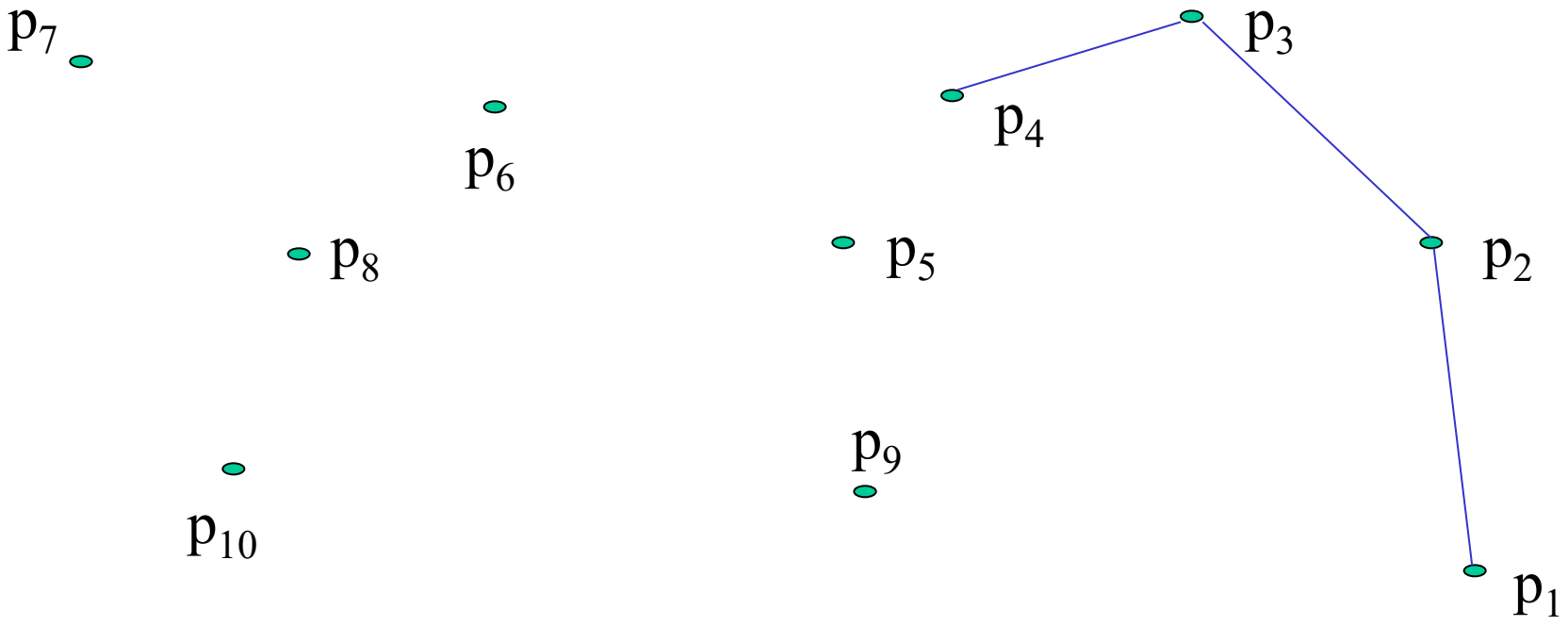


- **As we trace out the potential hull, we expect to 'turn left'**
- **If we 'turn right' we must exclude one or more points**
- **p_5 must be excluded because the angle between $-p_4 — p_5-$ and $-p_5 — p_6-$ is $\geq 180°$**
- **p_4 must be excluded because the angle between $-p_3 — p_4-$ and $-p_4 — p_6-$ is $\geq 180°$**

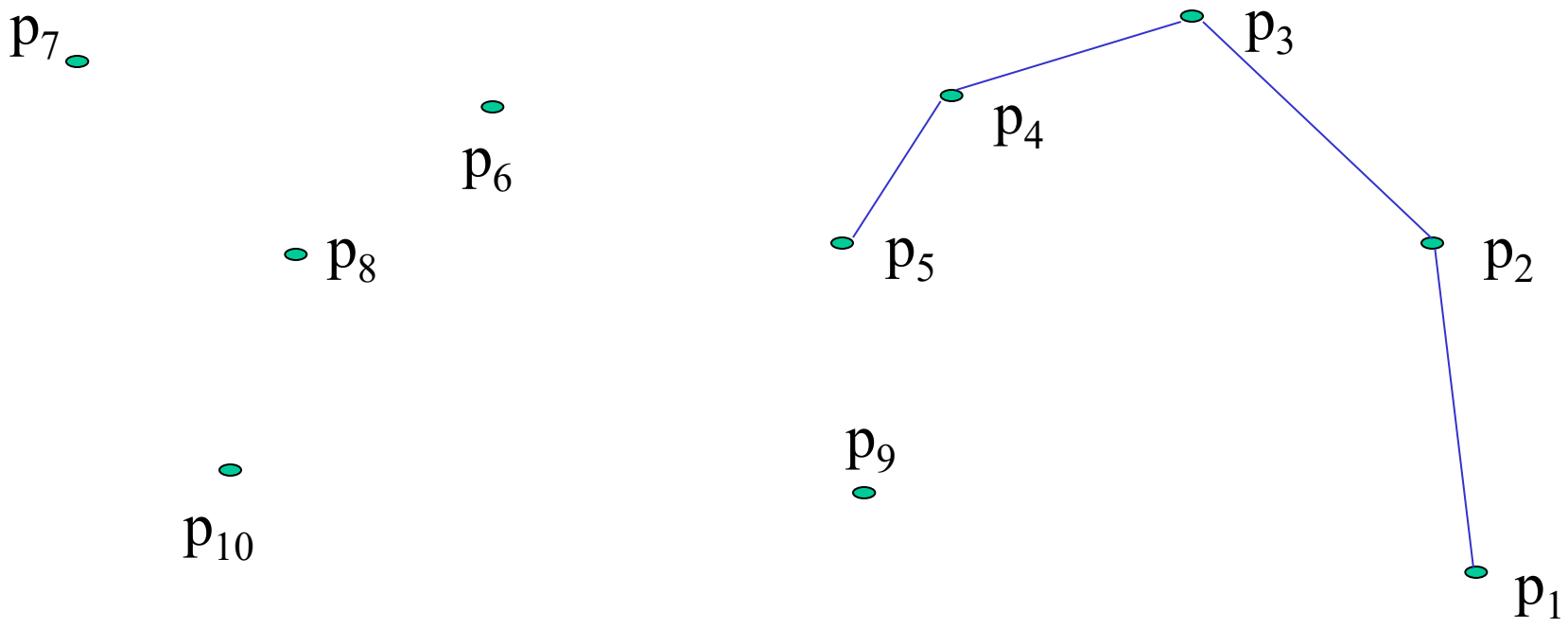
Graham scan algorithm



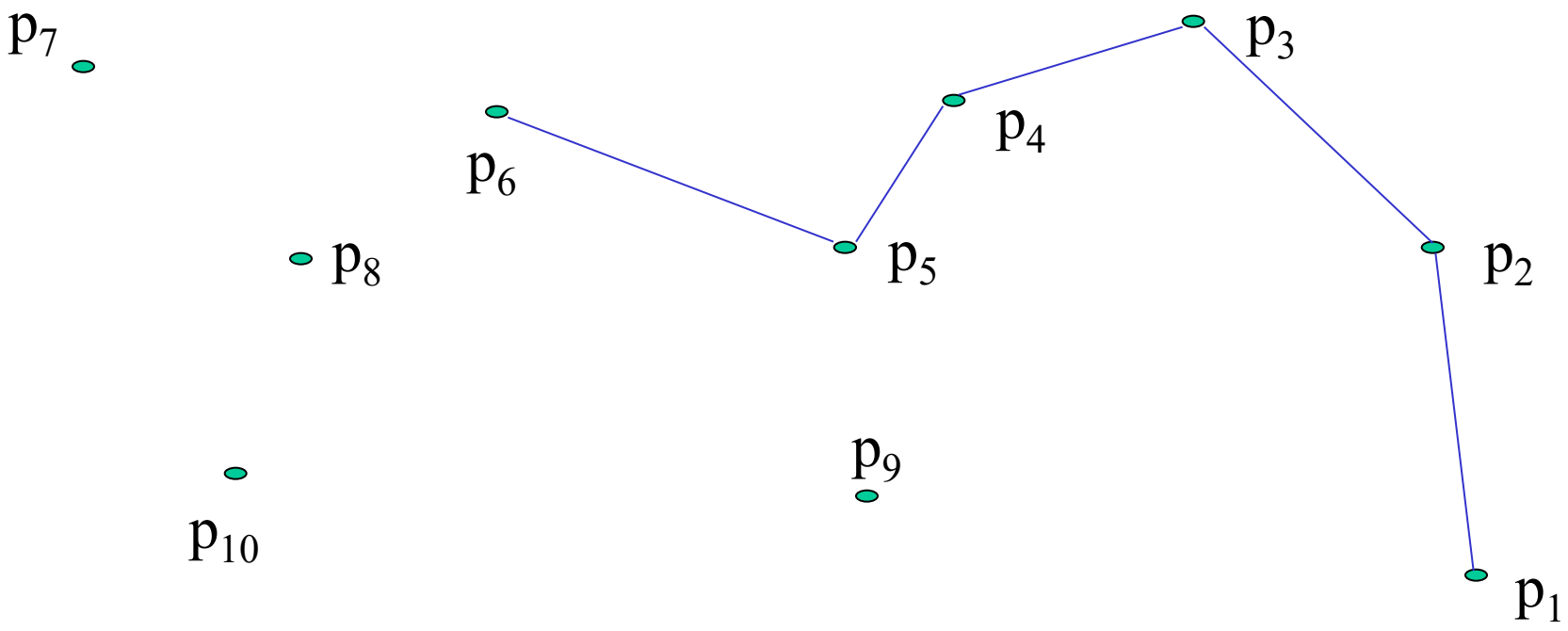
Graham scan algorithm



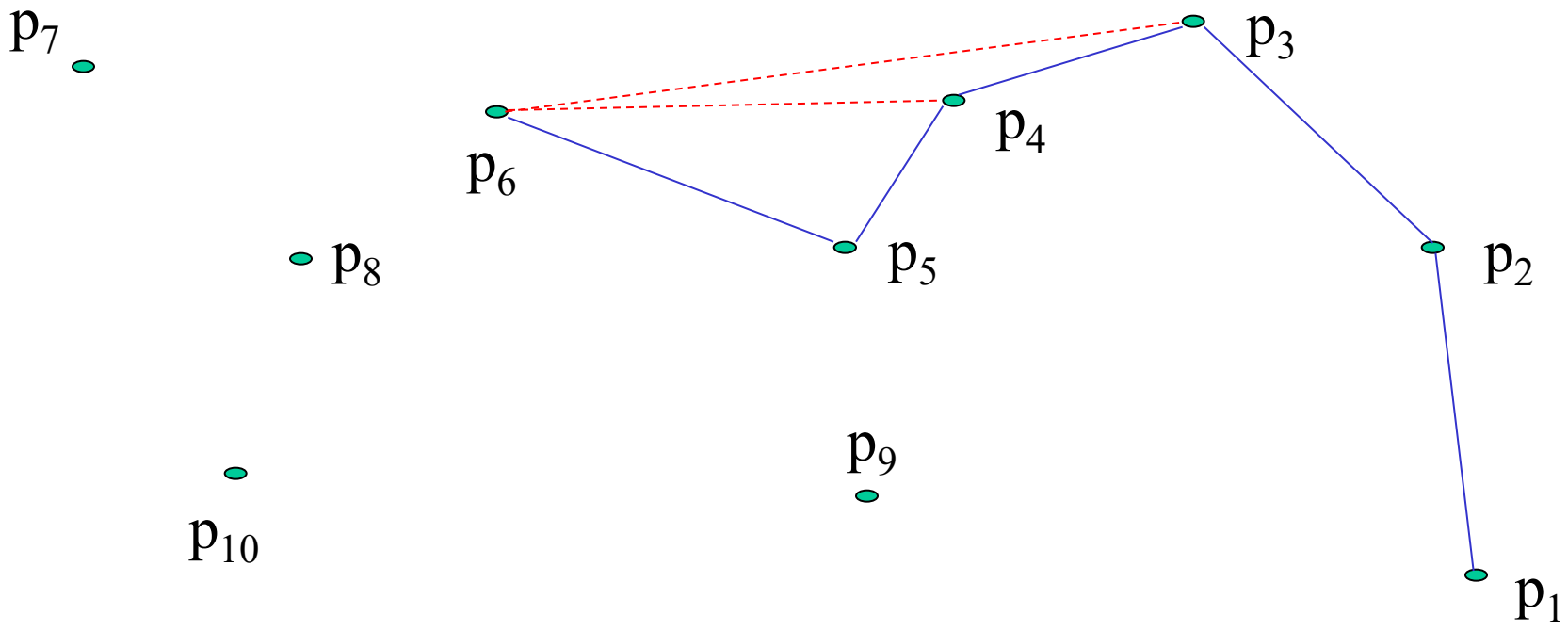
Graham scan algorithm



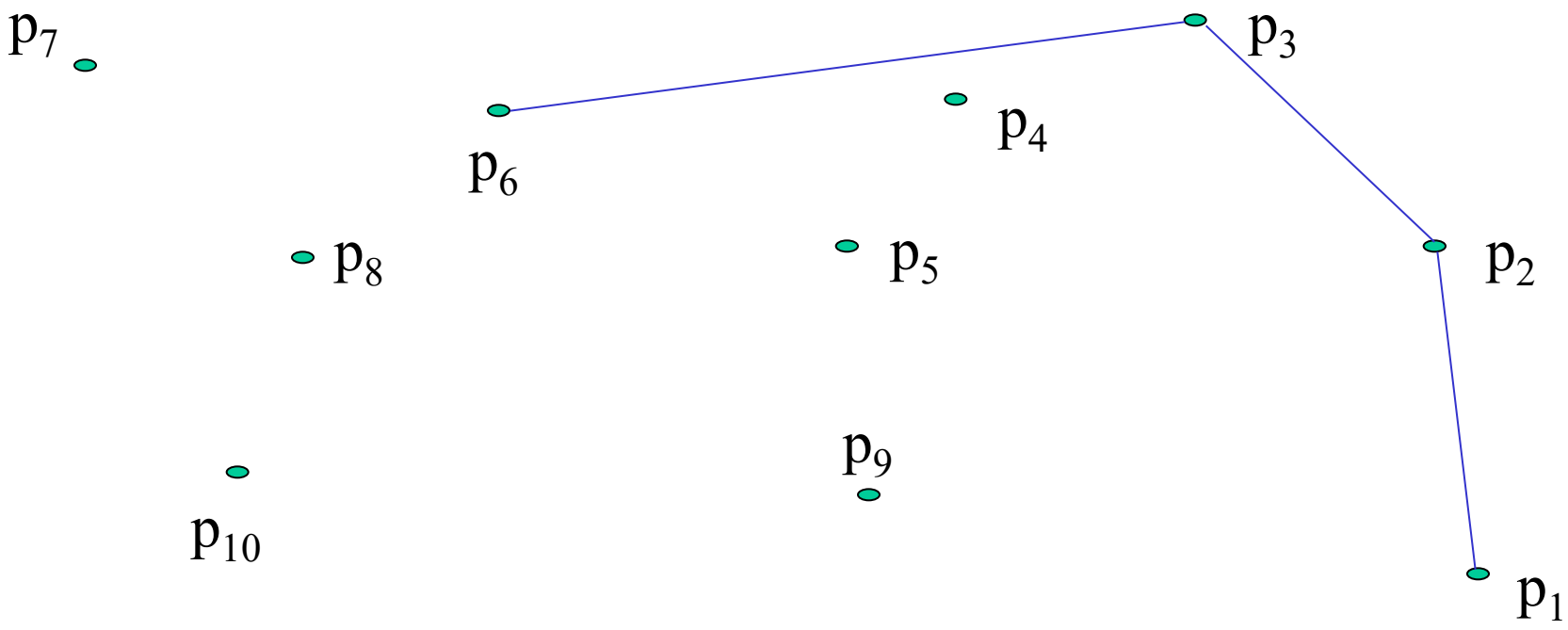
Graham scan algorithm



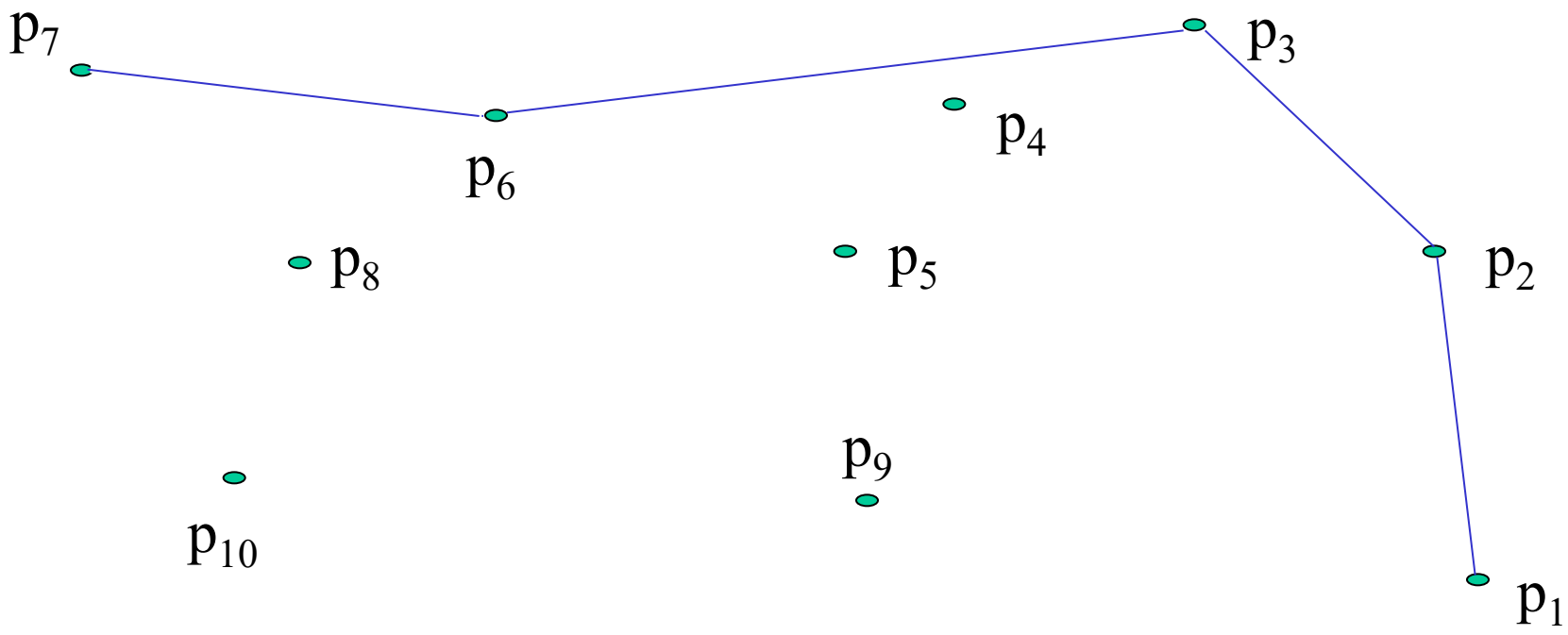
Graham scan algorithm



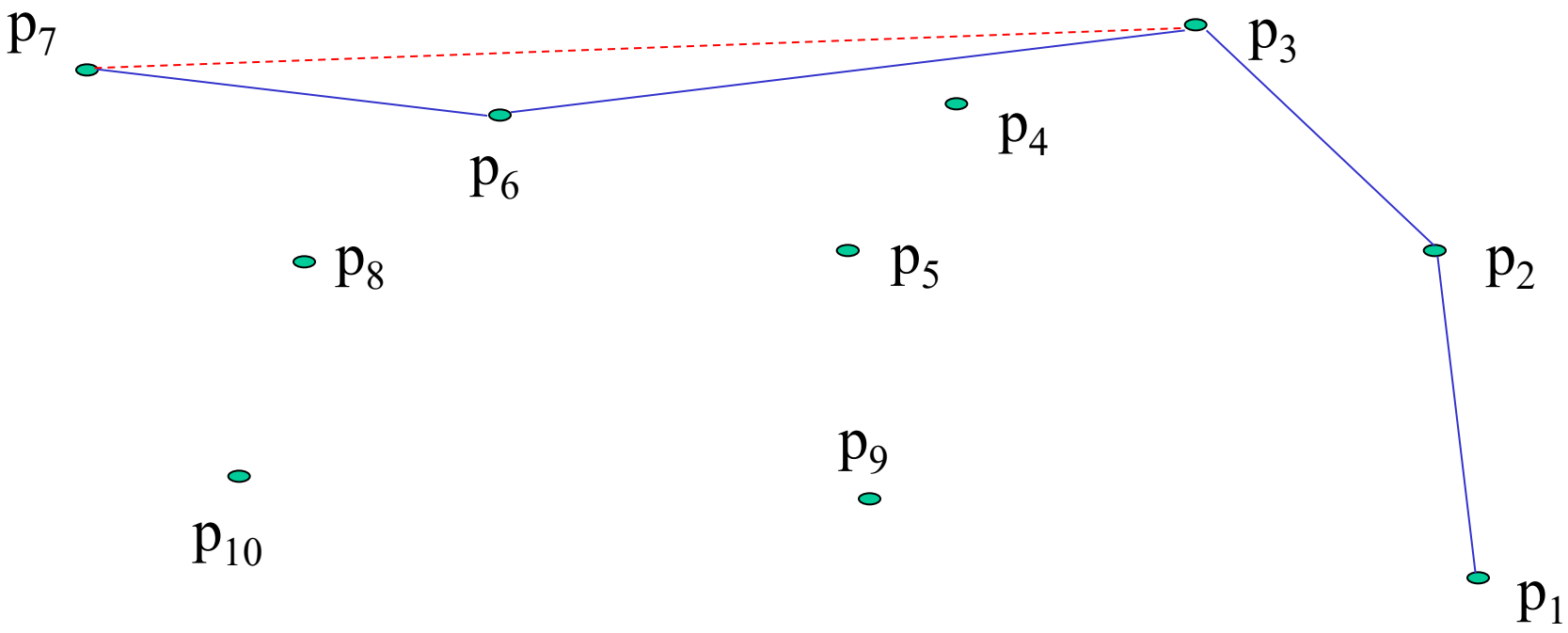
Graham scan algorithm



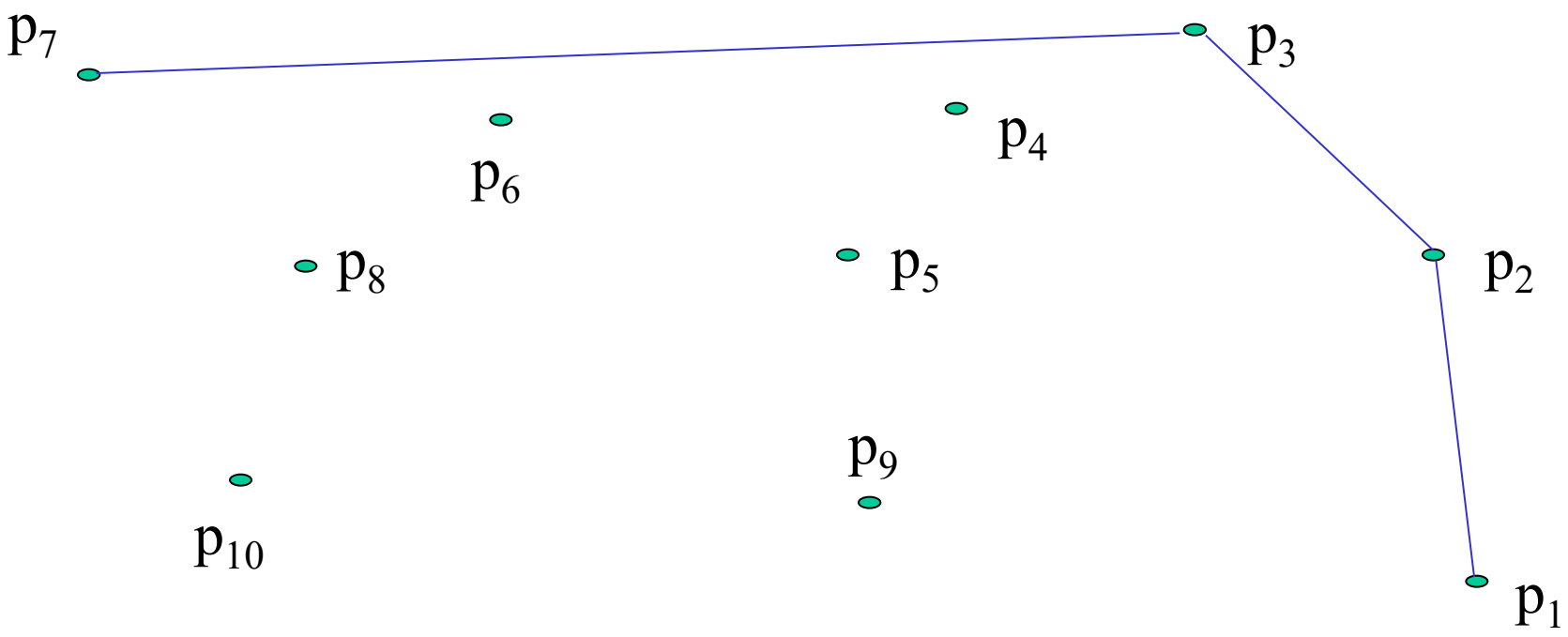
Graham scan algorithm



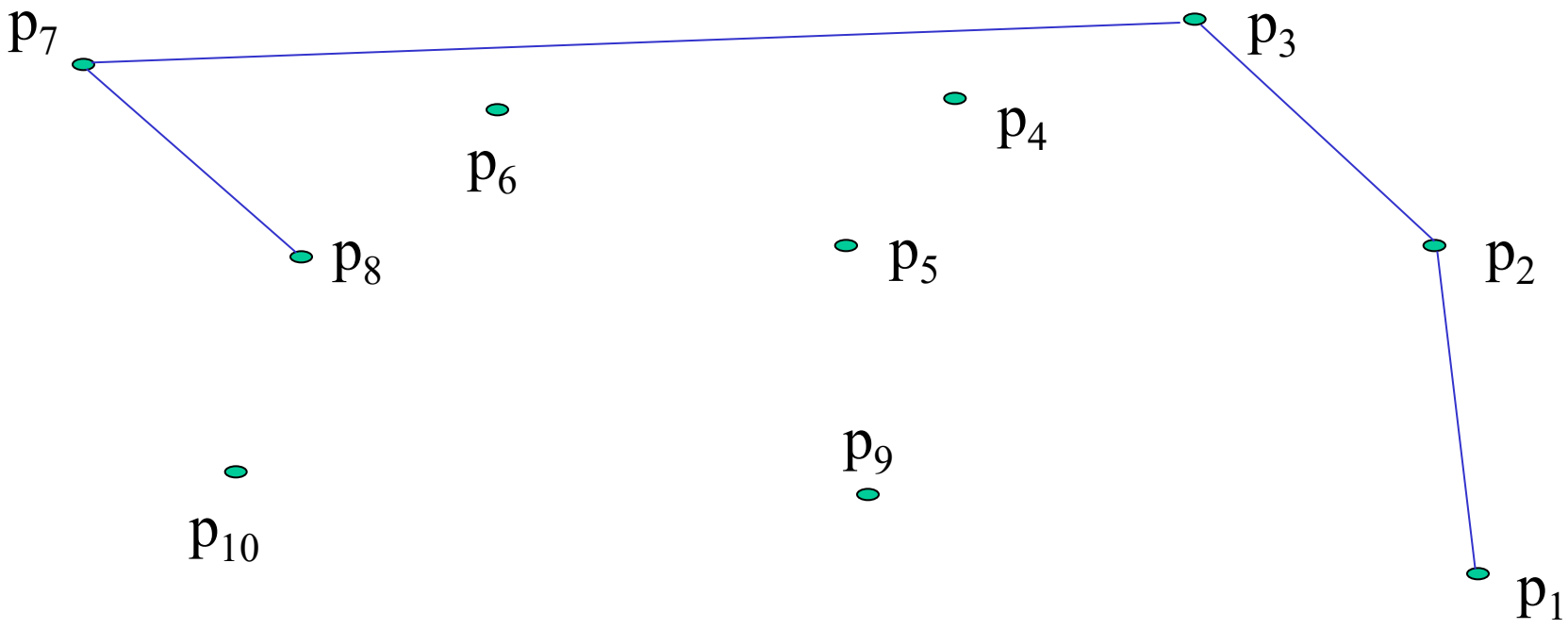
Graham scan algorithm



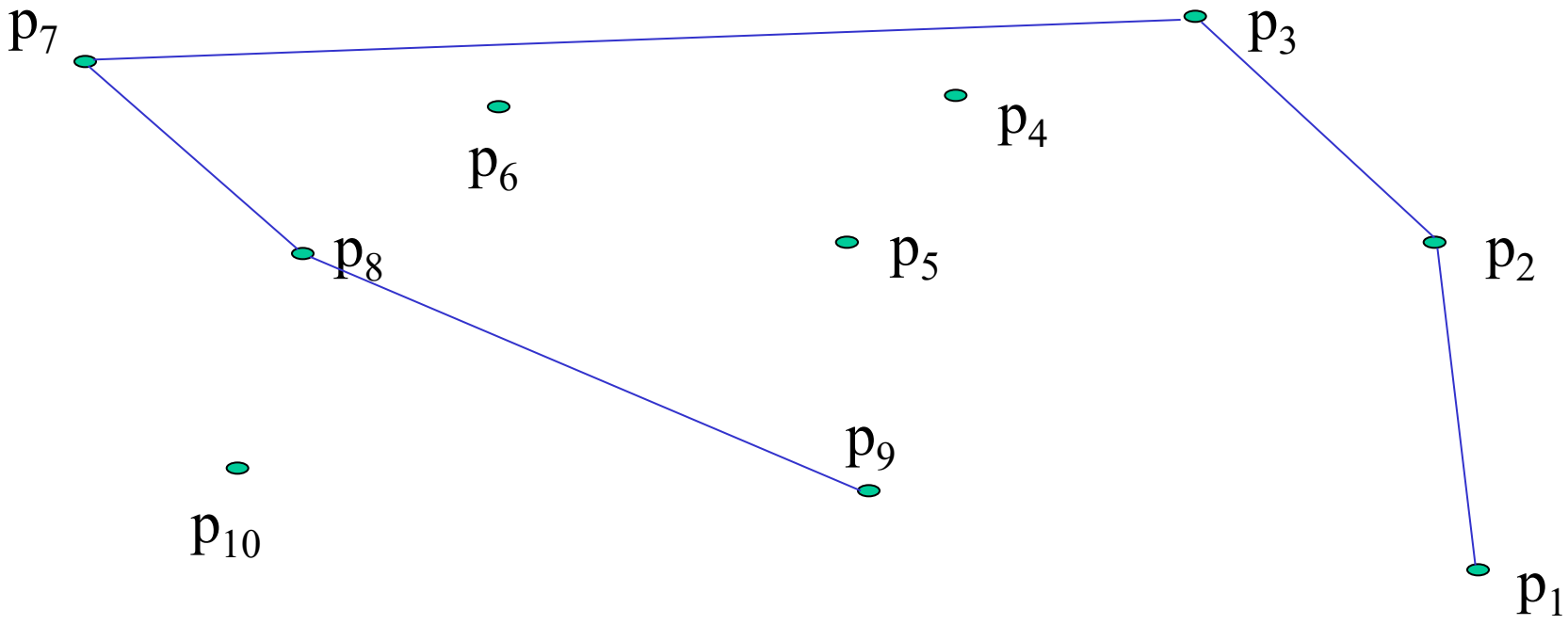
Graham scan algorithm



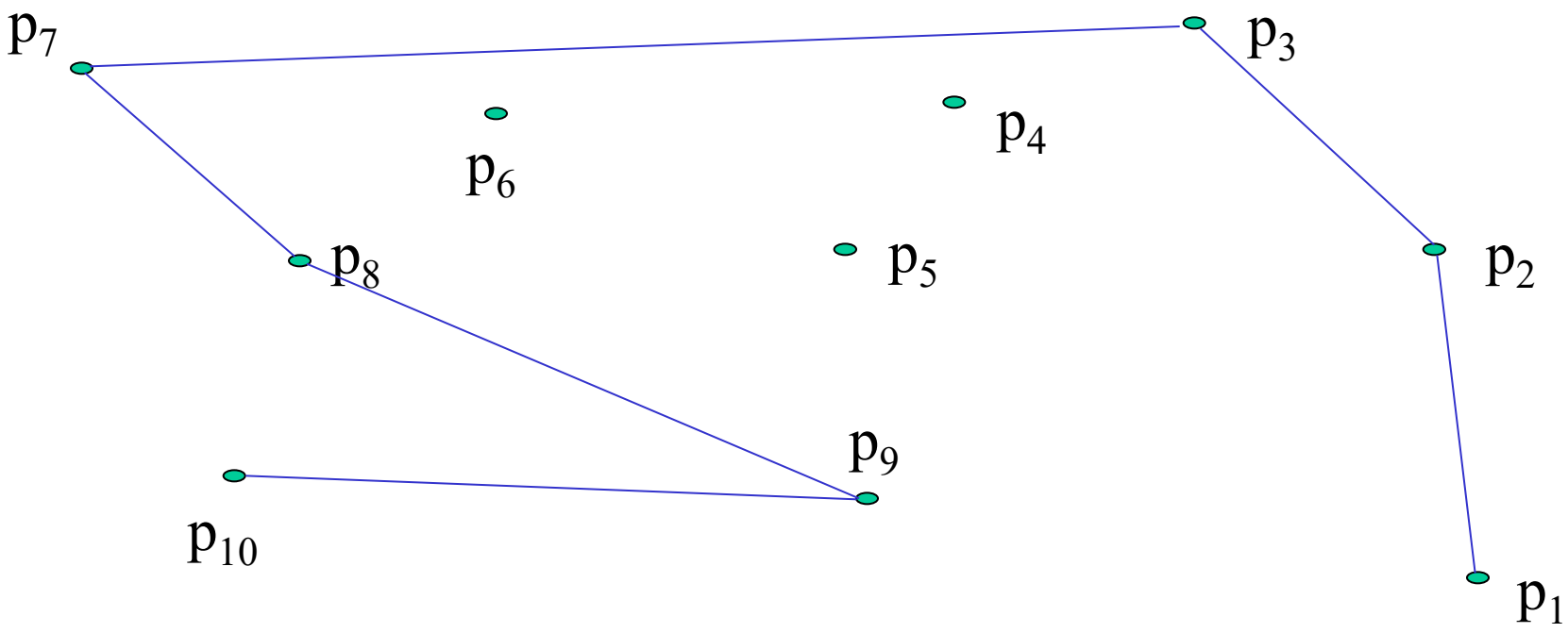
Graham scan algorithm



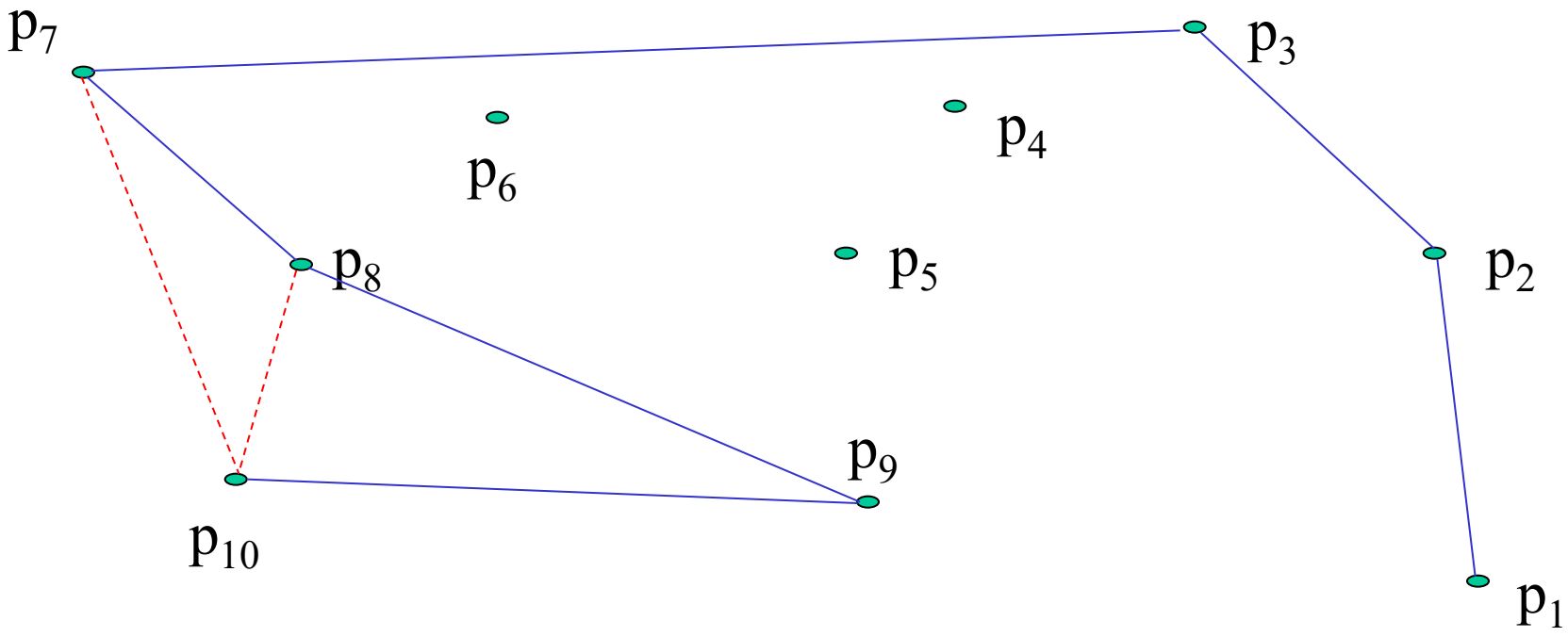
Graham scan algorithm



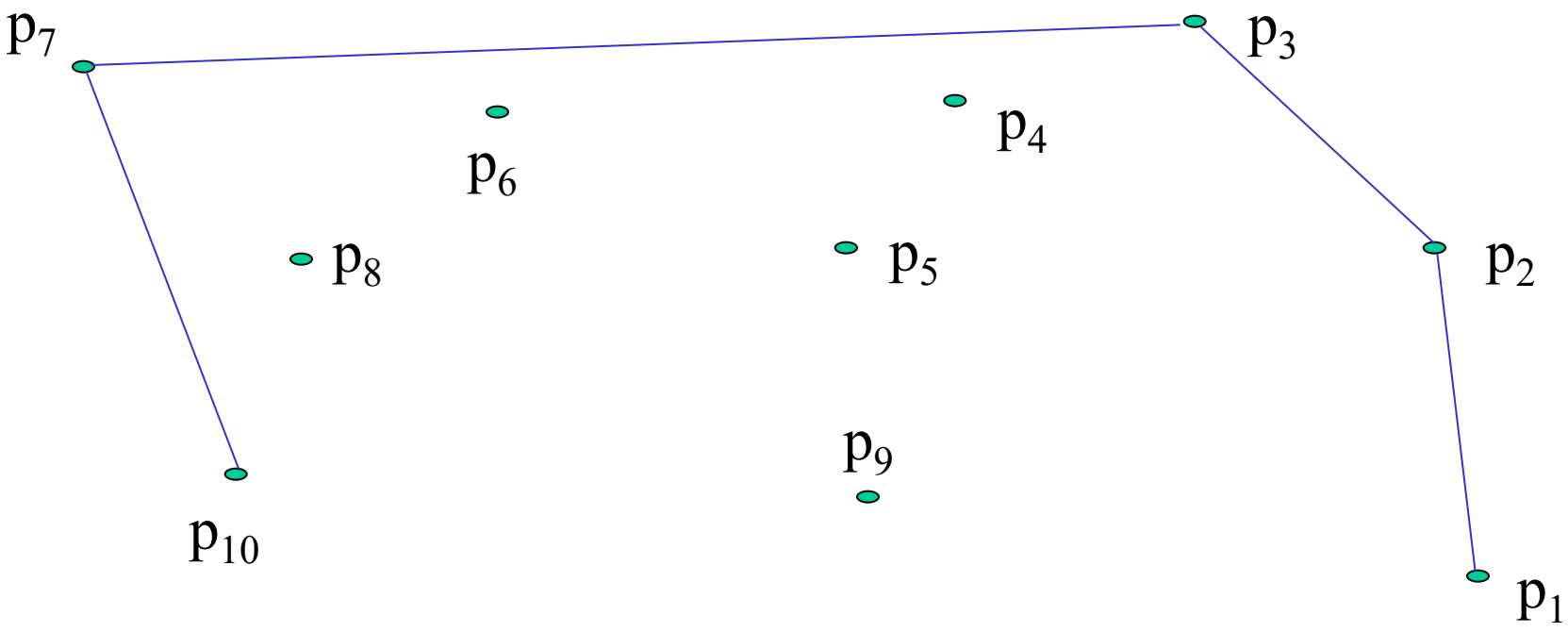
Graham scan algorithm



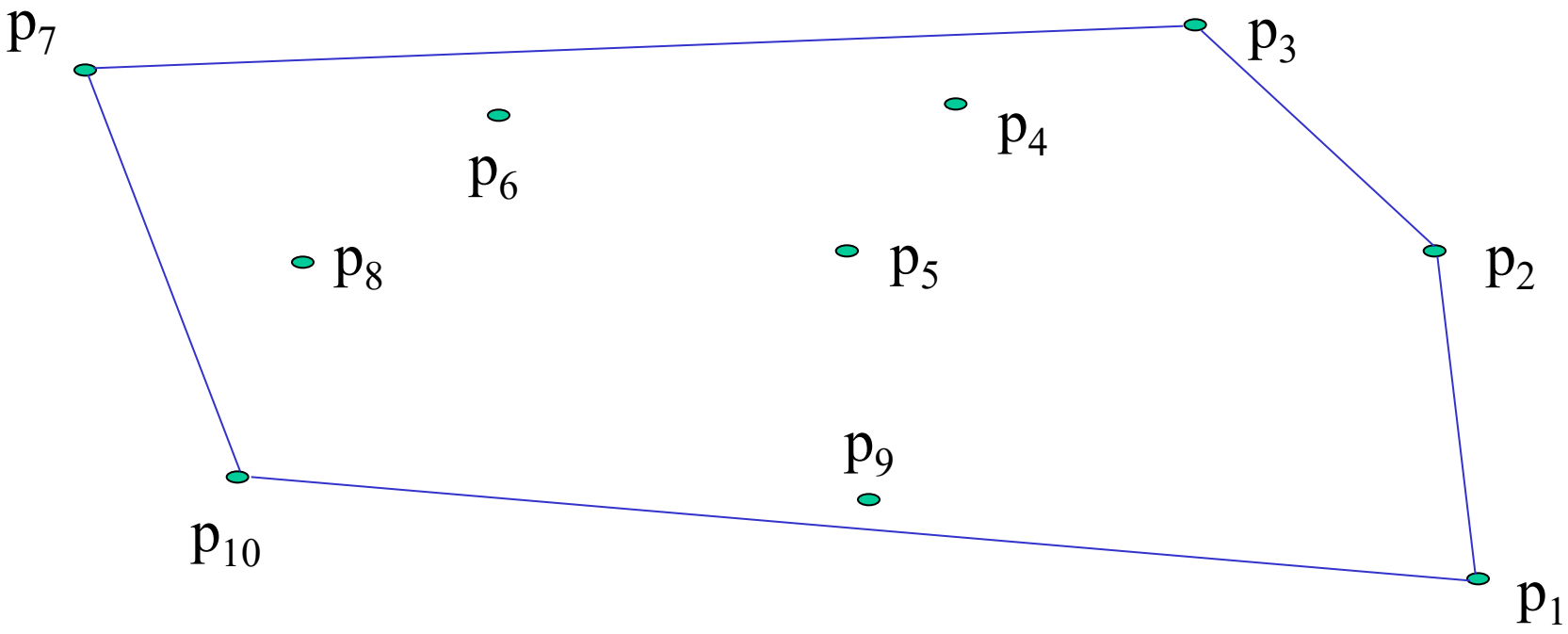
Graham scan algorithm



Graham scan algorithm



Graham scan algorithm



Graham Scan algorithm for the convex hull

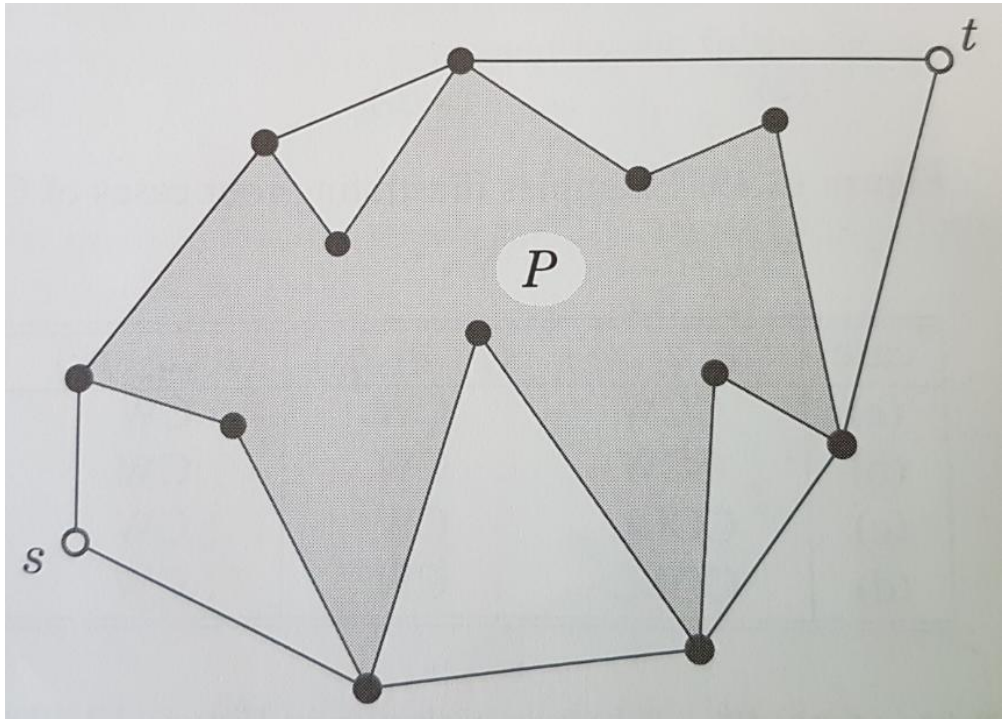
```
/** Input: an arbitrary polygon p of n points,
  * i.e., a sequence of n points
  * Output: q, a simple polygon that is the convex hull
  * of the points in p */

public PointSet grahamScan(PointSet p) {
  simplePolygon(p); // earlier algorithm
  PointSet q = new PointSet(p.length);
  q[0..2] = p[0..2]; // deep copy
  int m = 2; // m points other than pivot in current hull
  for (int k = 3; k < p.length; k++) {
    /* before adding p[k] to the current hull, exclude
     * appropriate previous points, if any */
    while (angle(-q[m-1]—q[m]-,-q[m]—p[k]-)>= Math.PI)
       m--; // exclude q[m]
    q[++m] = p[k]; // add p[k] to the current hull
  }
  return q[0..m];
}
```

Complexity of `grahamScan`

- `simplePolygon` is O(n log n)
- `angle` function has O(1) complexity
  - gives the anticlockwise angle between the positive direction of `-q[m-1]–q[m]-` and the positive direction of `-q[m]–p[k]-`
  - in fact only need to know whether angle is $\geq \pi$
  - can be implemented without using trig functions
  - see Goodrich & Tamassia 12.5.2
- Main loop has O(n) complexity (a point is eliminated at most once)
- So overall O(n log n)
  - consequence of sorting

Application 1: Robotics

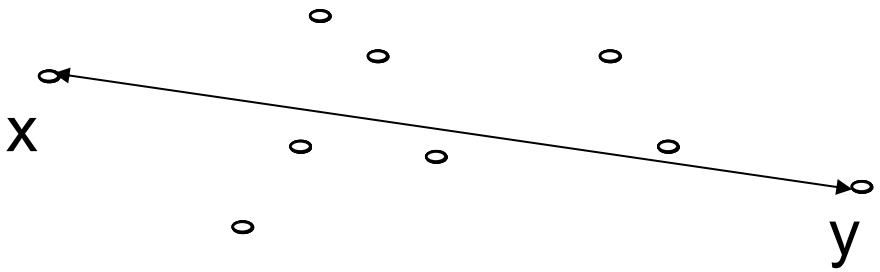
- Robot wants to travel from s to t, avoiding obstacle P

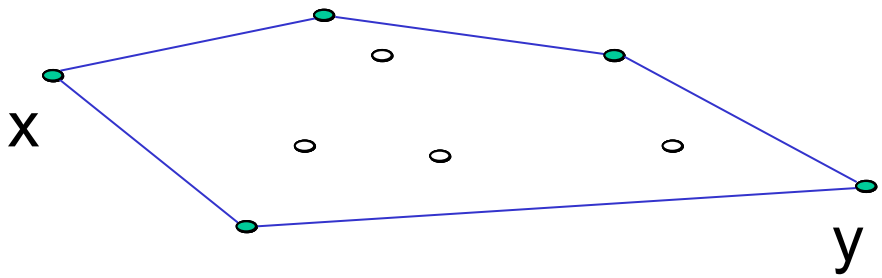


- Check whether s–t intersects P
- If so, compute convex hull of s, t and points of P's boundary
- Shortest path from s to t is shorter of:
  - Clockwise distance along convex hull from s to t
  - Anti-clockwise distance along convex hull from s to t

Application 2: Furthest Pair of Points

- Let P be a set of points in the plane, where n=|P|
- Naïve method for finding furthest pair of points: check each pair - $O(n^2)$ time
- Faster method:
  - A furthest pair of points in P are points of the convex hull of P





  - The convex hull may be constructed in O(n log n) time
  - Once the convex hull of P has been constructed, a furthest pair of points in P can be found in O(n) time (next slide)
  - Overall O(n log n) time

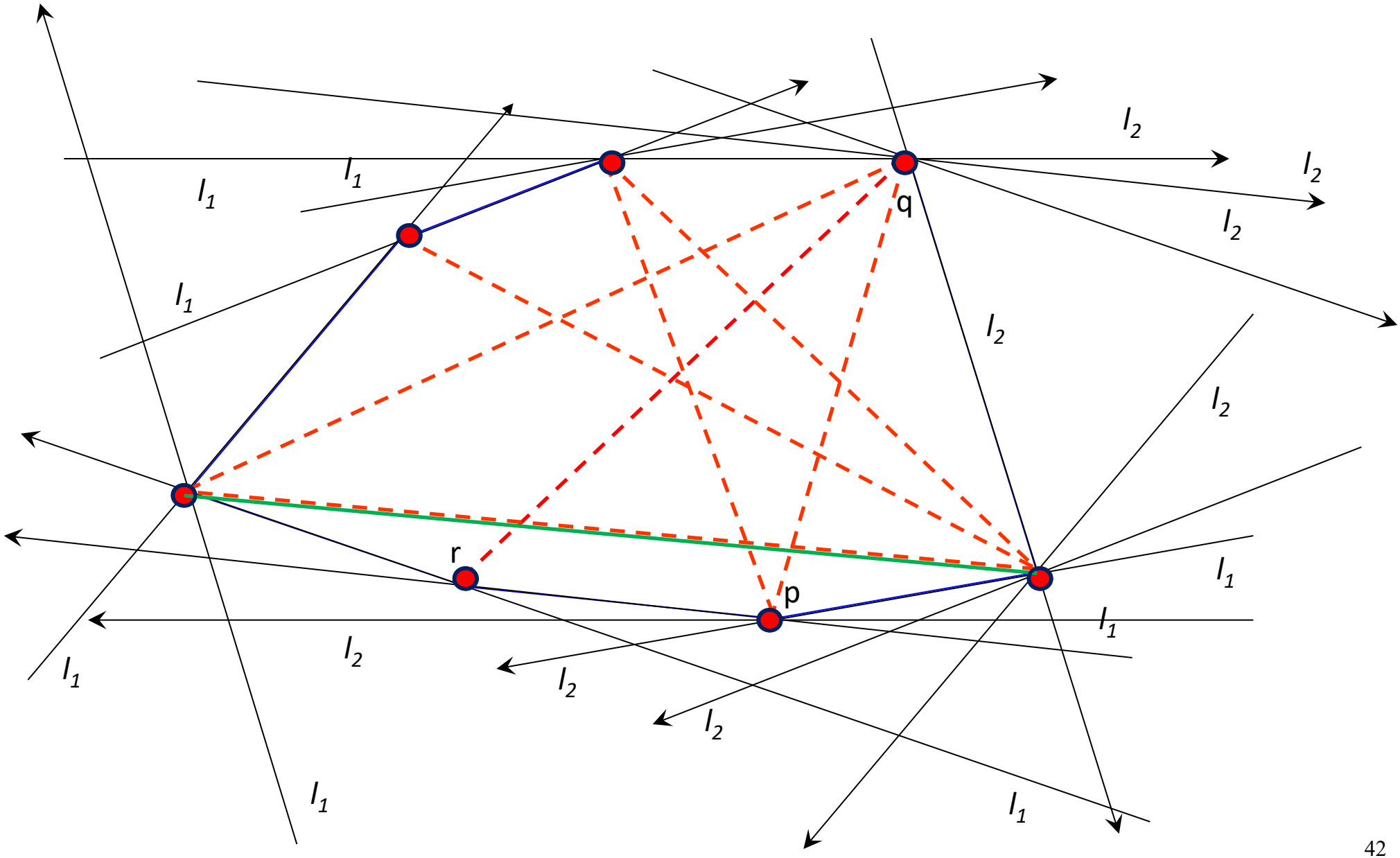
Furthest pair of points

- **Algorithm for finding the distance between a furthest pair of points in P**

"Rotating calipers" method:

1. Find two points p, q, on the convex hull with minimum and maximum y-coordinate respectively.
2. Draw two horizontal lines (the "calipers") l_1 and l_2 through each of p and q, and record the distance $d(p, q)$.
3. Rotate l_1 clockwise around p, and rotate l_2 clockwise around q, keeping l_1 parallel to l_2 at all times, until one of the calipers $l \in \{l_1, l_2\}$ reaches the next point (call it r) on the convex hull.
4. Repeat steps 2-3 with caliper l rotated clockwise around the next pivot r and with the other caliper rotated around its old pivot until l_1 reaches q and l_2 reaches p (half circle). (Note, the perpendicular distance between the calipers may change during the course of their rotation.)
5. Return the largest distance $d(p, q)$ measured at Step 2 as the furthest distance between points on the convex hull.

“Rotating calipers” method: example



Furthest pair of points – a footnote

A furthest pair of points don't have to have smallest or largest x or y coordinate