

Algorithms 2

Section 4: Geometric Algorithms

Polygons

Polygons are an ordered list of vertices.

Vertices are points (vectors) in some kind of 2D vector space.

We are mostly interested in planar, closed, simple polygons.

Our first problem is to work out whether a point is on the “inside” of a polygon.

Planar Polygons [1]

If the space in which the polygon exists is not planar, it can be tricky or impossible to define “inside” and “outside”.

Example: the Earth’s surface is (roughly) spherical; is Cambridge (assumed to be a point) “inside” the UK mainland (represented as a polygon), or “outside”? Neither label makes sense because the polygon boundary divides two finite areas and we could label either as “inside”.

Note that we cannot say the “smaller” area is “inside”: ask whether a container ship’s position is “inside” the ocean polygon or is on land.

Planar Polygons [2]

A planar space is 2D, flat, and infinite in the “horizontal” and “vertical” directions.

A polygon drawn on a planar surface separates a finite area from an infinite area: we refer to the finite area as “inside” and the infinite area as “outside”.

Closed Polygons

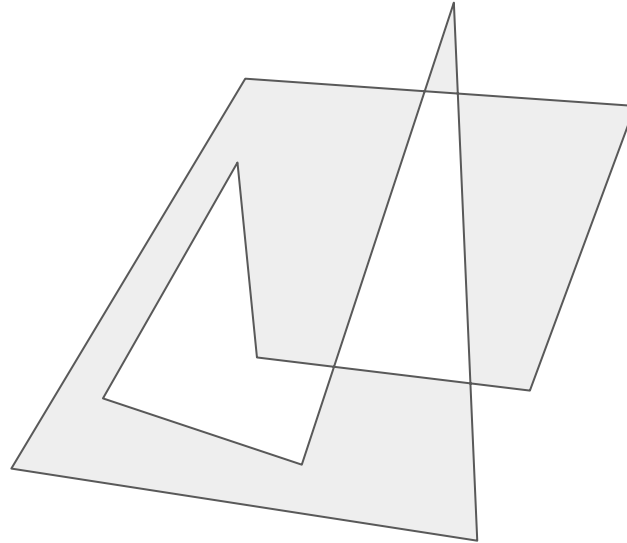
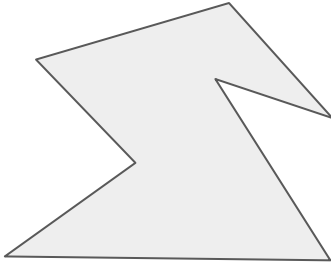
A closed polygon is one where there is an edge from its last vertex back to its first.

An open polygon does not (necessarily) enclose any area so we cannot define inside and outside.

Example: can you be “inside” the letter O? What about letter C?

Simple Polygons [1]

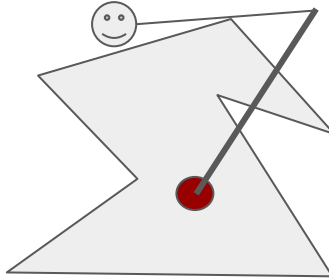
Simple polygons do not overlap themselves.



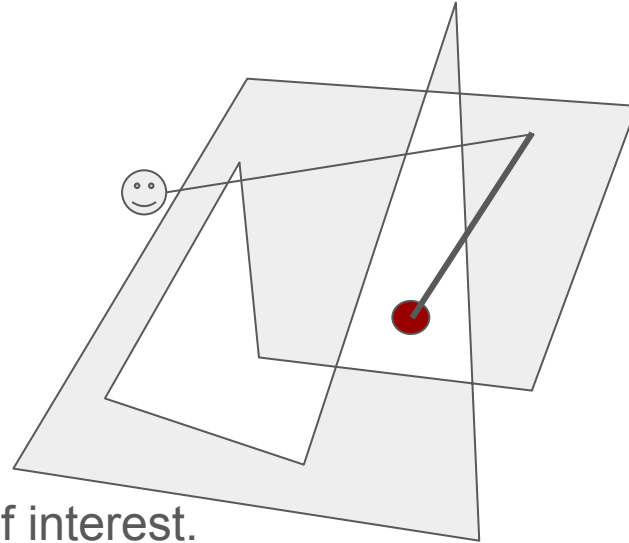
Winding Numbers [1]

How do we define what is “inside” a complex polygon?

One way is with the **winding number**.

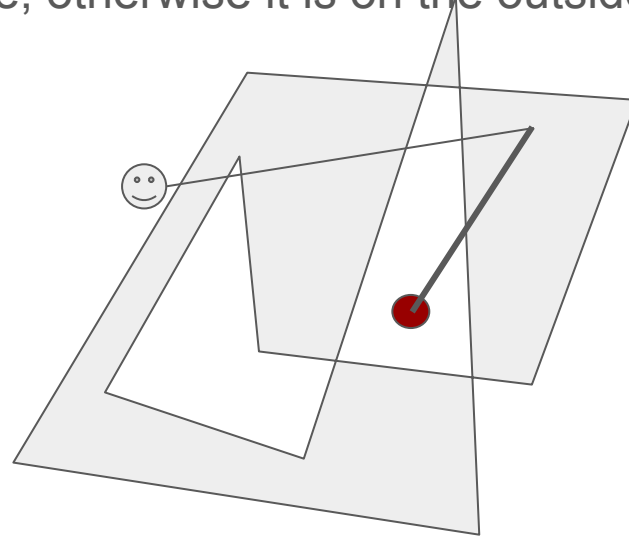
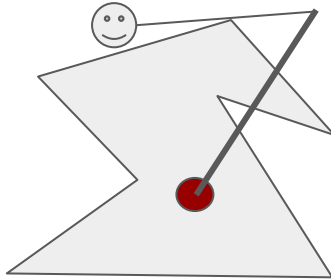


Walk around the perimeter with a piece of string attached to a post at the point of interest.



Winding Numbers [2]

When you get back to the start, if the string is wound around the post an odd number of times, the post is on the inside; otherwise it is on the outside.



Winding Numbers [3]

We can implement this algorithm on a computer:

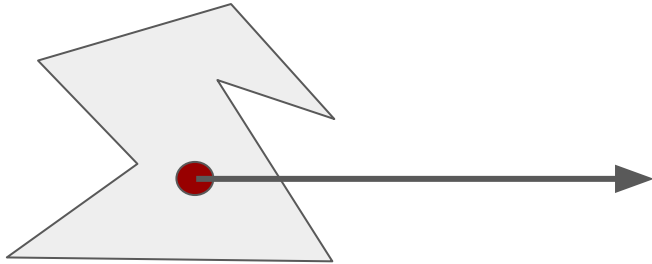
1. calculate angles subtended at the post by the two ends of each edge;
2. sum the angles
3. divide by 2π to get the winding number.

Problems: floating point inaccuracy; slow trigonometric functions.

Inclusion within Simple, Planar, Closed Polygons

Add a semi-line from the point of interest P , in any direction.

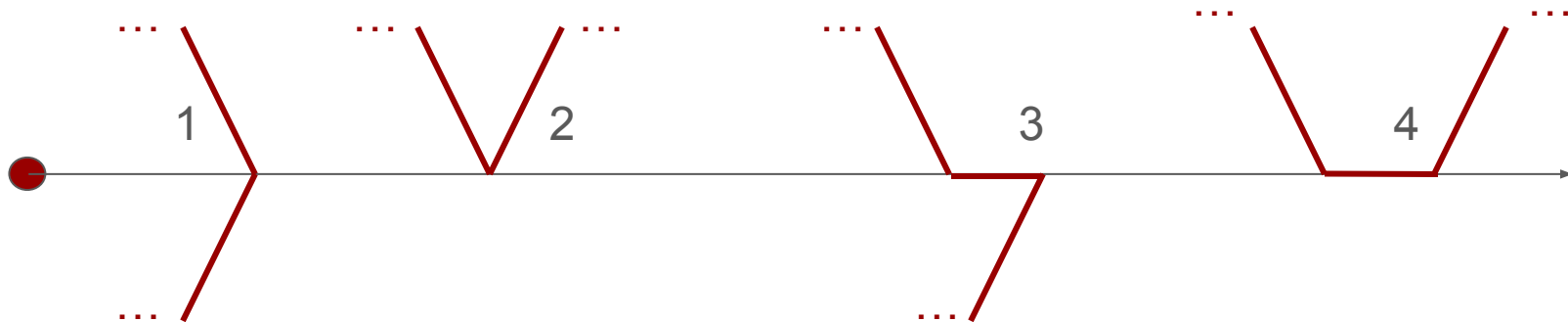
A semi-line is infinite in one direction. Because the coordinates of any vertex are finite values, a point at infinity must be on the “outside”. Because the polygon is simple, planar and closed, each edge separates a region of “inside” from a region of “outside” so we can count edge crossings.



Awkward cases

If the ray goes through a vertex, we could discard the ray and send one in a different direction; keep retrying until it doesn't hit any vertices.

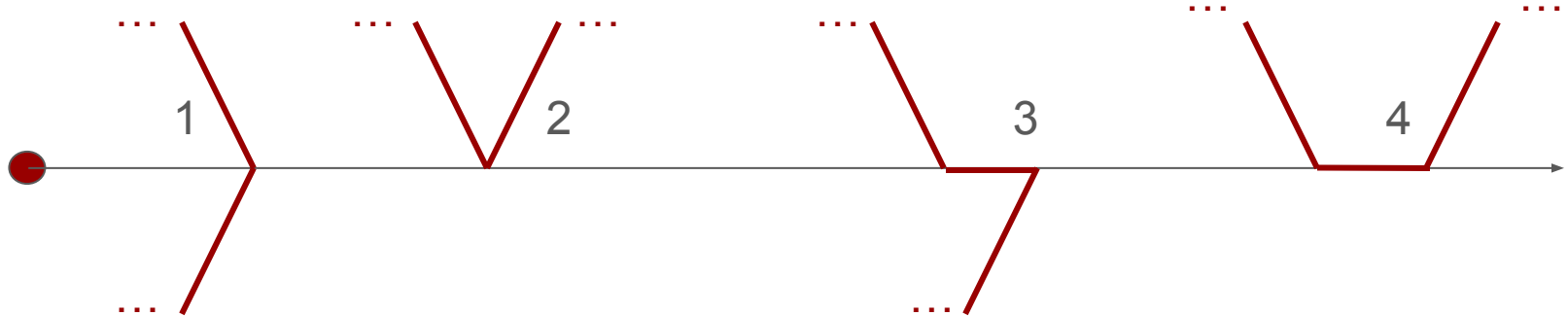
The horizontal ray avoids floating point error in calculations of whether we hit the vertex, were slightly above or were slightly below because (non-NaN) floats are totally ordered.



Handling the Awkward cases

If a vertex is on the ray, look at the neighbouring vertices. If they're on the same side (both above / both below) then the polygon's edge was *not* crossed (case 2); if they are on opposite sides then the edge was crossed (case 1).

If either neighbour is also on the ray, replace it with the next neighbour in the same direction around the polygon boundary.



Line segments

A **line segment** p_1p_2 is a straight line between two points p_1 and p_2 . We say that p_1 and p_2 are the **endpoints** and, if the line has a direction then we have a **directed segment** $p_1 \rightarrow p_2$.

These points might be adjacent vertices in a polygon or the test point and a point “at infinity”.

Convex combinations

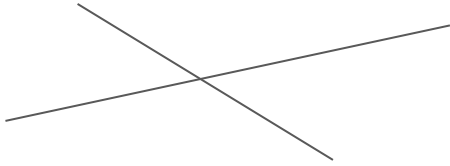
If $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, then we say that $p_3 = (x_3, y_3)$ is a **convex combination** of p_1 and p_2 if p_3 is on the line segment between p_1 and p_2 (including the endpoints).

Mathematically, $x_3 = \alpha x_1 + (1-\alpha) x_2$ and $y_3 = \alpha y_1 + (1-\alpha) y_2$. This is often written as the vector equation $p_3 = \alpha p_1 + (1-\alpha)p_2$. We require $0 \leq \alpha \leq 1$, to place p_3 between p_1 and p_2 inclusive of the endpoints.

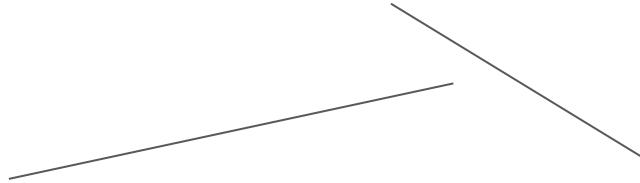
Intersection Determination Problem

Input: two line segments p_1p_2 and p_3p_4 .

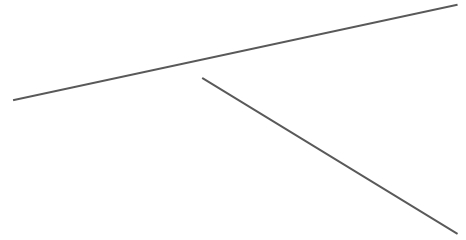
Output: true if the line segments intersect; false otherwise.



→ True



→ False

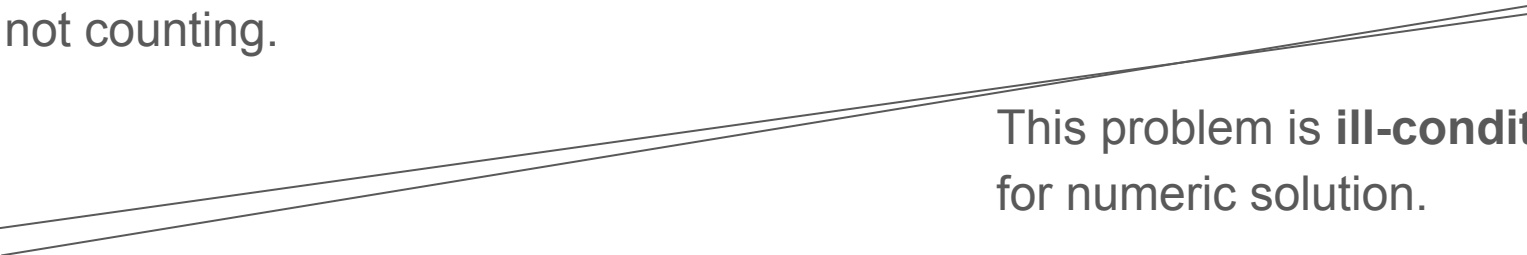


→ False

Intersection Determination

We would like to avoid trigonometry (slow).

The “high school maths” approach based on two equations of the form $y = mx + c$ leads to divisions, which are slow in floating point, and introduce error that cannot be managed as effectively as with addition and multiplication (a concept known as *infinite precision*). This can lead to incorrect answers: small floating point errors can lead to the intersection of these two lines being “off the end” of the segments so not counting.

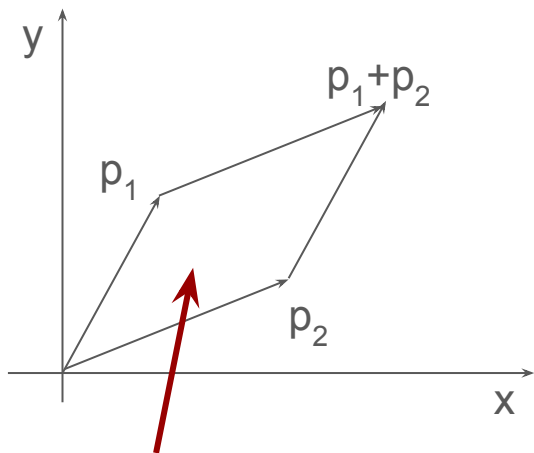


This problem is **ill-conditioned** for numeric solution.

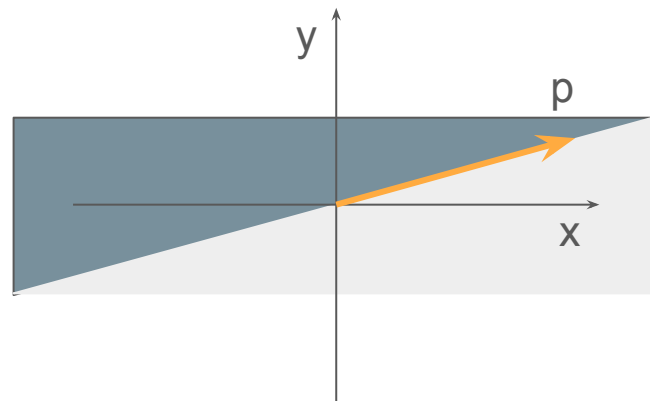
💡 Definition of an ill-conditioned problem: a small change in the input data can result in a large change in the output.

Cross Products

The vector cross product turns out to be very useful.



The cross product of p_1 and p_2 can be thought of as the signed area of the parallelogram.



The darker regions contains position vectors that are anticlockwise from p ; the lighter region contains vectors that are clockwise from p .



This is Figure 33.1 from CLRS3.

Matrix Determinants

$$\begin{aligned} p_1 \times p_2 &= \det \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1 \end{aligned}$$

If $p_1 \times p_2 > 0$ then p_1 is clockwise from p_2 with respect to the origin.

If $p_1 \times p_2 < 0$ then p_1 is anticlockwise from p_2 with respect to the origin.

If $p_1 \times p_2 = 0$ then p_1 and p_2 are collinear (parallel or antiparallel).

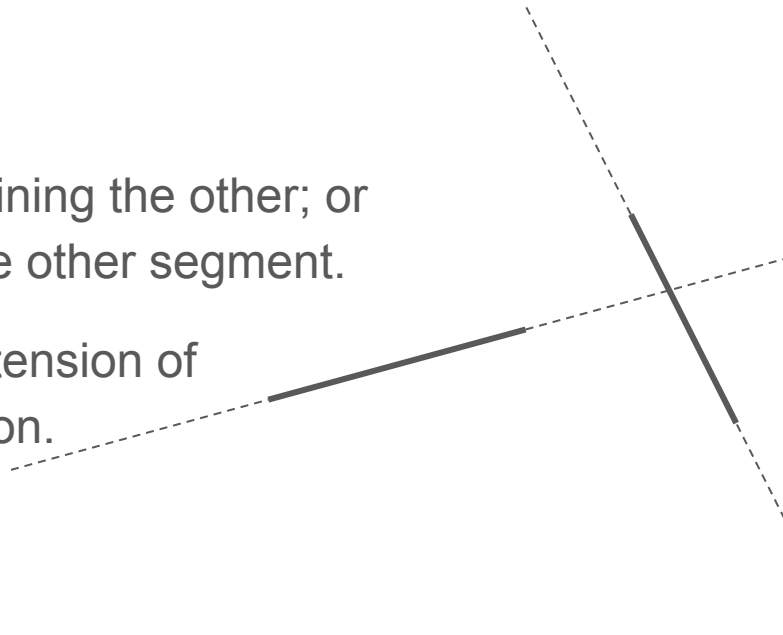
Line Segment Intersection

Check whether each line segment **straddles** the extension of the other. The **extension** of a line segment is the (infinite) line containing its two endpoints, i.e. drop the constraint that $0 \leq \alpha \leq 1$.

Two line segments intersect if and only if

- each segment straddles the line containing the other; or
- an endpoint of one segment lies on the other segment.

In this example, one segment crosses the extension of the other, but *not vice-versa*. No intersection.



SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4) [1]

```
1  d1 = DIRECTION(p3,p4,p1)           // Relative orientation of
2  d2 = DIRECTION(p3,p4,p2)           // each endpoint w.r.t. the
3  d3 = DIRECTION(p1,p2,p3)           // other segment
4  d4 = DIRECTION(p1,p2,p4)
5  if ((d1>0 && d2<0) || (d1<0 && d2>0)) &&
      ((d3>0 && d4<0) || (d3<0 && d4>0))
6      return true
```

💡 If $p_3 \rightarrow p_1$ and $p_3 \rightarrow p_2$ have opposite directions w.r.t. $p_3 \rightarrow p_4$ then $p_1 p_2$ straddles $p_3 p_4$.

💡 If $p_1 \rightarrow p_3$ and $p_1 \rightarrow p_4$ have opposite directions w.r.t. $p_1 \rightarrow p_2$ then $p_3 p_4$ straddles $p_1 p_2$.

SEGMENTS-INTERSECT(p_1, p_2, p_3, p_4) [2]

```
7  else if d1==0 && ON-SEGMENT(p3,p4,p1)  return true
8  else if d2==0 && ON-SEGMENT(p3,p4,p2)  return true
9  else if d3==0 && ON-SEGMENT(p1,p2,p3)  return true
10 else if d4==0 && ON-SEGMENT(p1,p2,p4)  return true
11 return false
```

$\text{DIRECTION}(p_i, p_j, p_k) = (p_k - p_i) \times (p_j - p_i)$

$\text{ON-SEGMENT}(p_i, p_j, p_k) = (\min(x_i, x_j) \leq x_k \leq \max(x_i, x_j)) \ \&\& \ (\min(y_i, y_j) \leq y_k \leq \max(y_i, y_j))$

💡 If p_1 or p_2 is on p_3p_4 then the segments intersect if that point is within the limits of the segment (L7,8).

💡 If p_3 or p_4 is on p_1p_2 then the segments intersect if that point is within the limits of the segment (L9,10)²⁸⁴

n-Segment Intersection Problem

Input: n line segments, each specified as pairs of endpoints, p_i for $1 \leq i \leq n$.

Output: true if any pair intersects; false otherwise.

Obvious solution: solve the segment intersection problem for all pairs, $O(n^2)$.

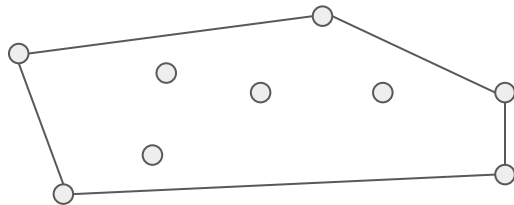
There is a smarter solution called **sweeping** with running time $O(n \lg n)$ that exploits the geometry of lines in a plane to constrain the cases that must be considered. Supervision exercise!

Convex Hull Problem

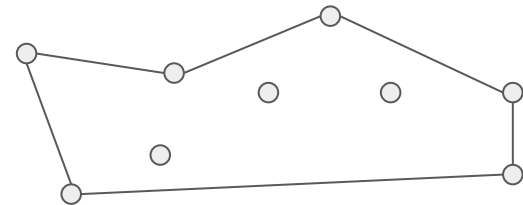
Input: a set of $n > 2$ points p_i for $1 \leq i \leq n$. At least 3 points are not collinear (so the polygon is not a zero-area line).

Output: an ordered list of points forming a convex hull for the input points.

The furthest-apart of a set of points in a plane are both on the convex hull. The convex hull of a set of points is a minimal subset that forms a convex polygon with none of the points outside the polygon (i.e. either inside or on the edge).



✓ Convex Hull



✗ Not convex!

Five Solutions

1. Rotational Sweeps
 - a. Graham's Scan $O(n \lg n)$
 - b. Jarvis's March $O(n h)$
2. Incremental $O(n \lg n)$
3. Divide and Conquer $O(n \lg n)$
4. Prune and Search $O(n \lg h)$

n : number of points in the input data

h : number of points on the convex hull produced

Prune-and-Search is asymptotically fastest since $h \leq n$.

Graham's Scan [1]

- Start at the left-most of the bottom-most points.
- Sort the points by increasing polar angle relative to a horizontal line through this point.
 - Resolve tie-breaks by retaining only the point farthest from the start point.
- Push the first three points onto an initially empty stack.
- For each of the other points, p , taken in the sorted order:
 - Pop off the stack until the directed segment from the next-to-top vertex on the stack to the top vertex on the stack forms a (strictly) left turn with the directed segment from top vertex to p
 - Push p onto the stack.
- The points on the stack are the convex hull.

Graham's Scan [2]

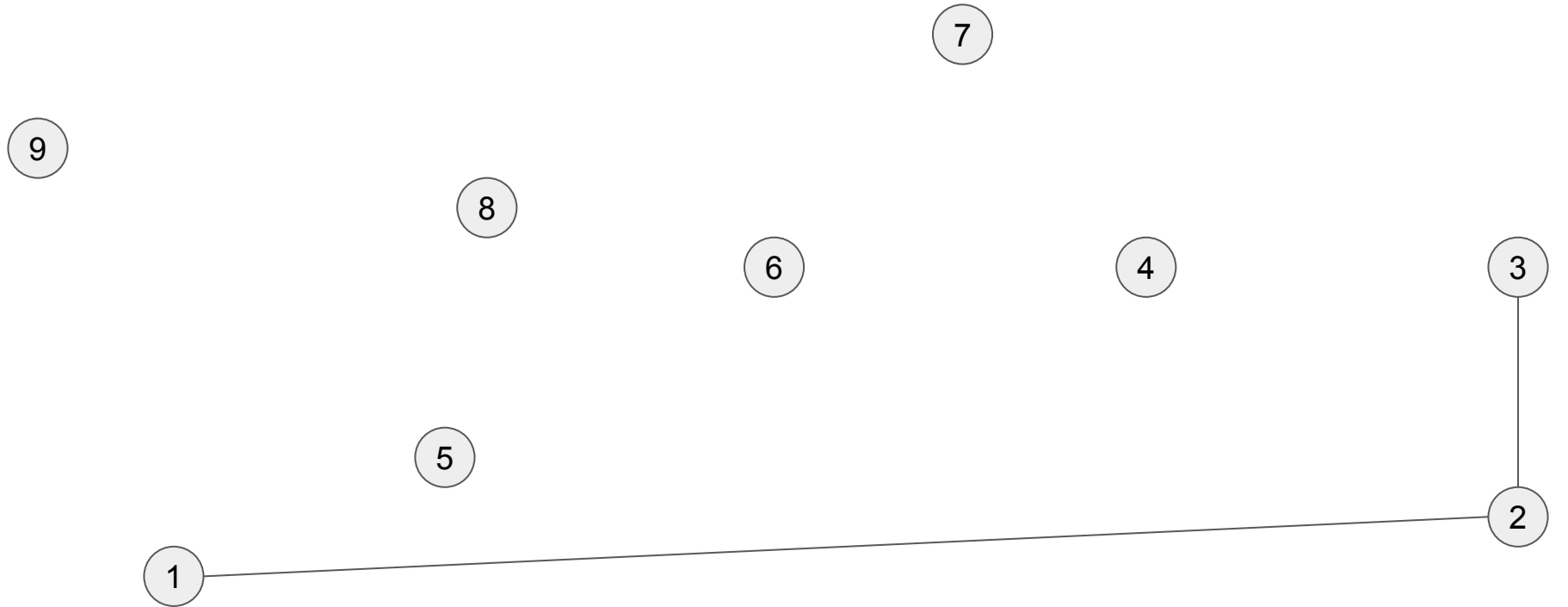
To sort by polar angle, we do not need to compute the angles!

The cross product $a \times b = |a| |b| \sin \theta$, where θ is the angle between the vectors a and b .

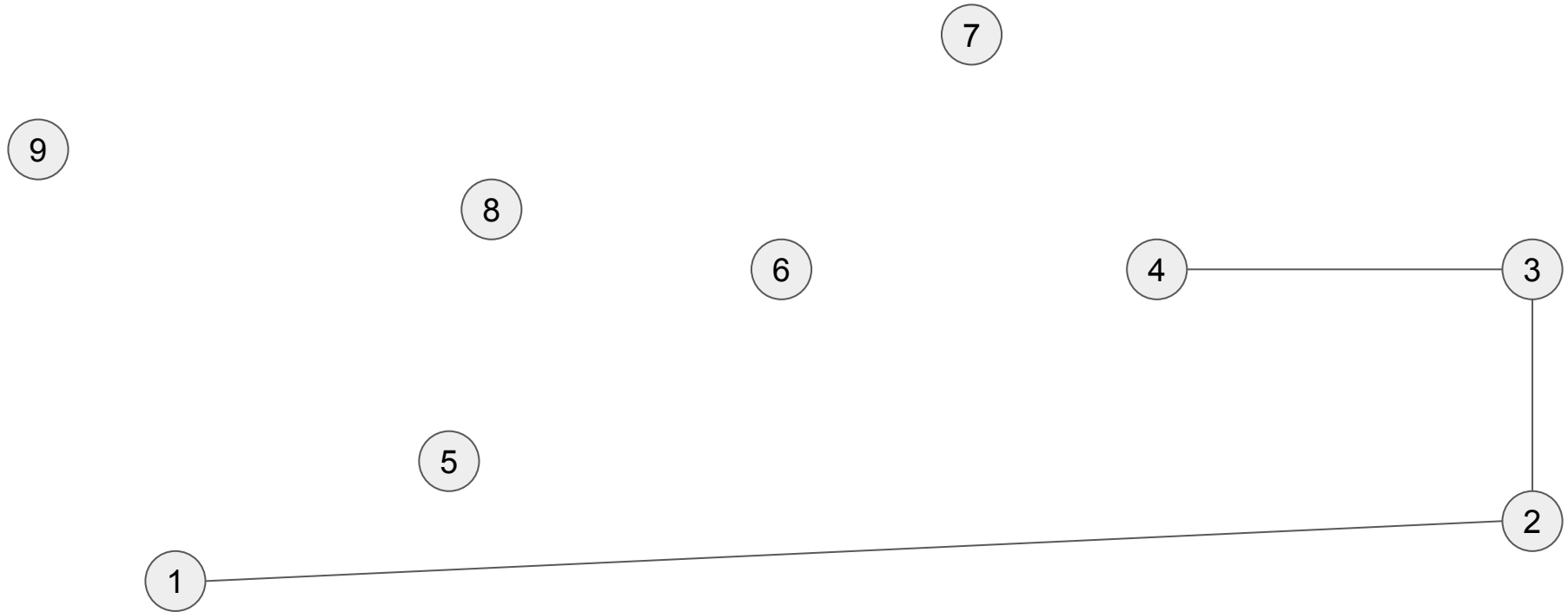
If a and b are unit vectors, sorting by the value of the cross product is the same as a sort by θ because $\sin \theta$ is monotonic with θ for $-\pi/2 \leq \theta < \pi/2$.

Normalising the vectors is often quicker than trigonometry.

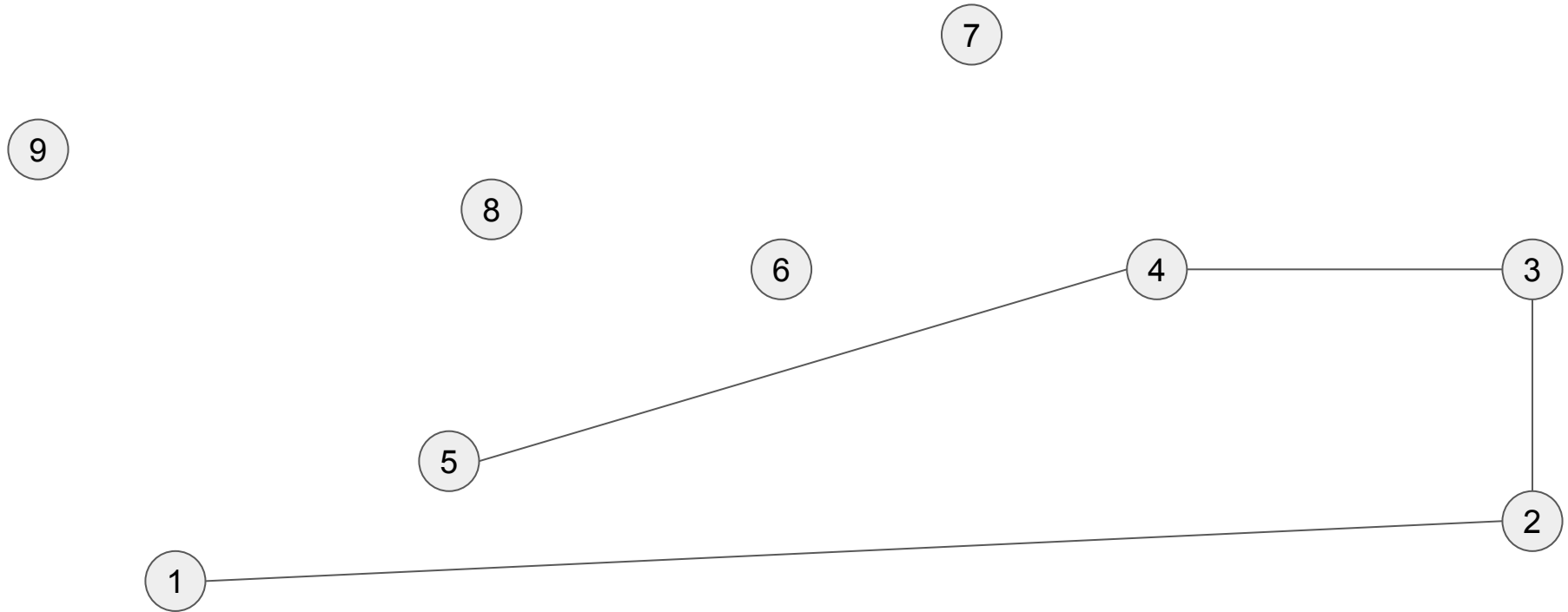
Graham's Scan [3]



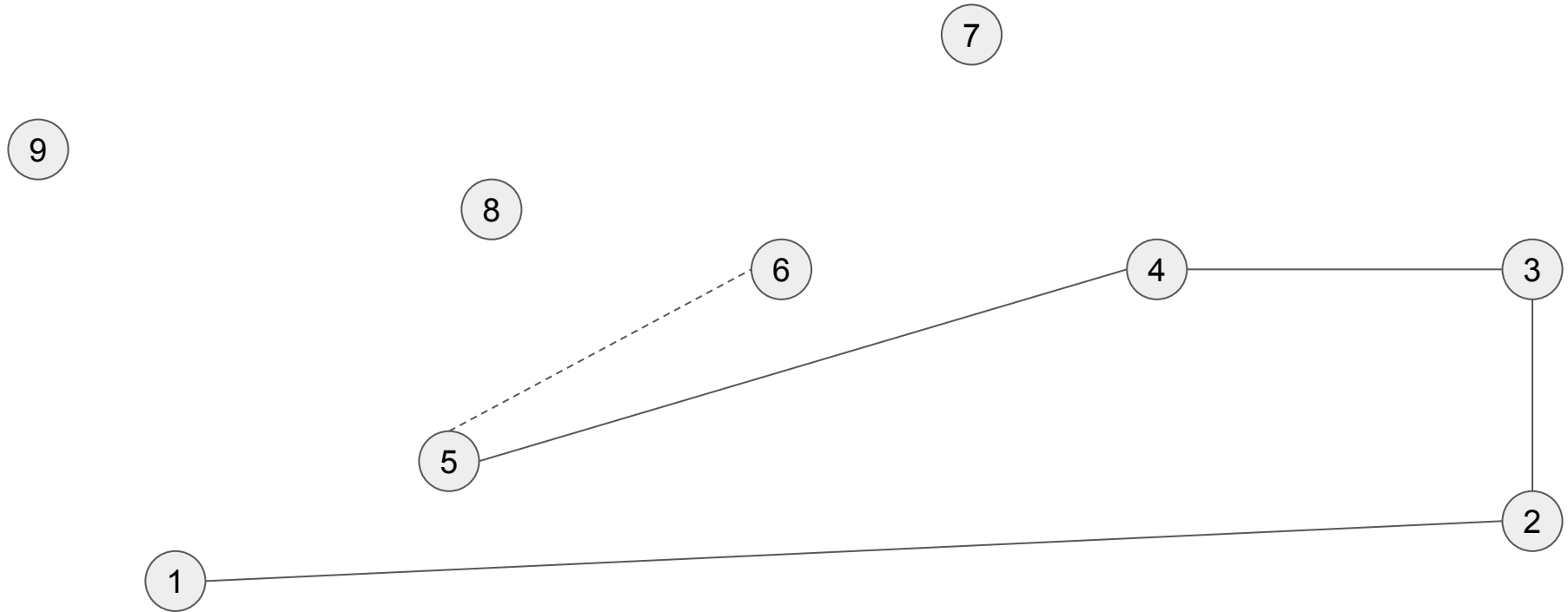
Graham's Scan [4]



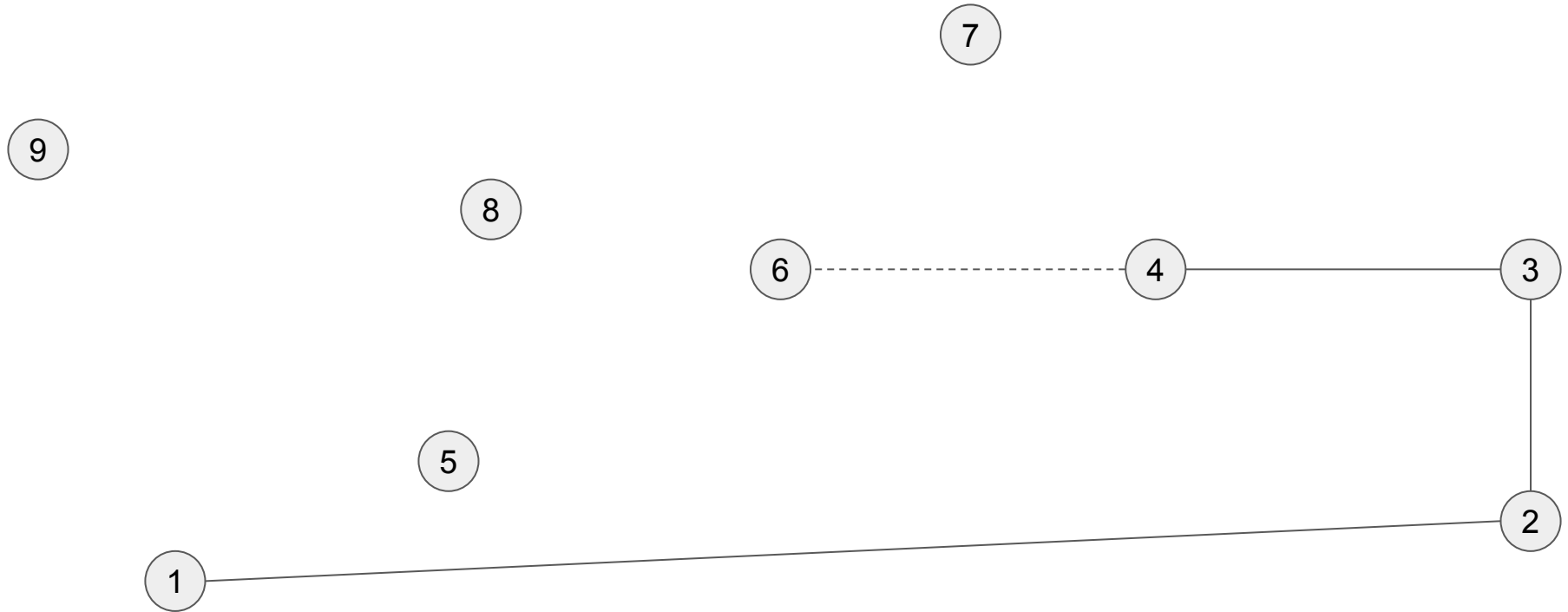
Graham's Scan [5]



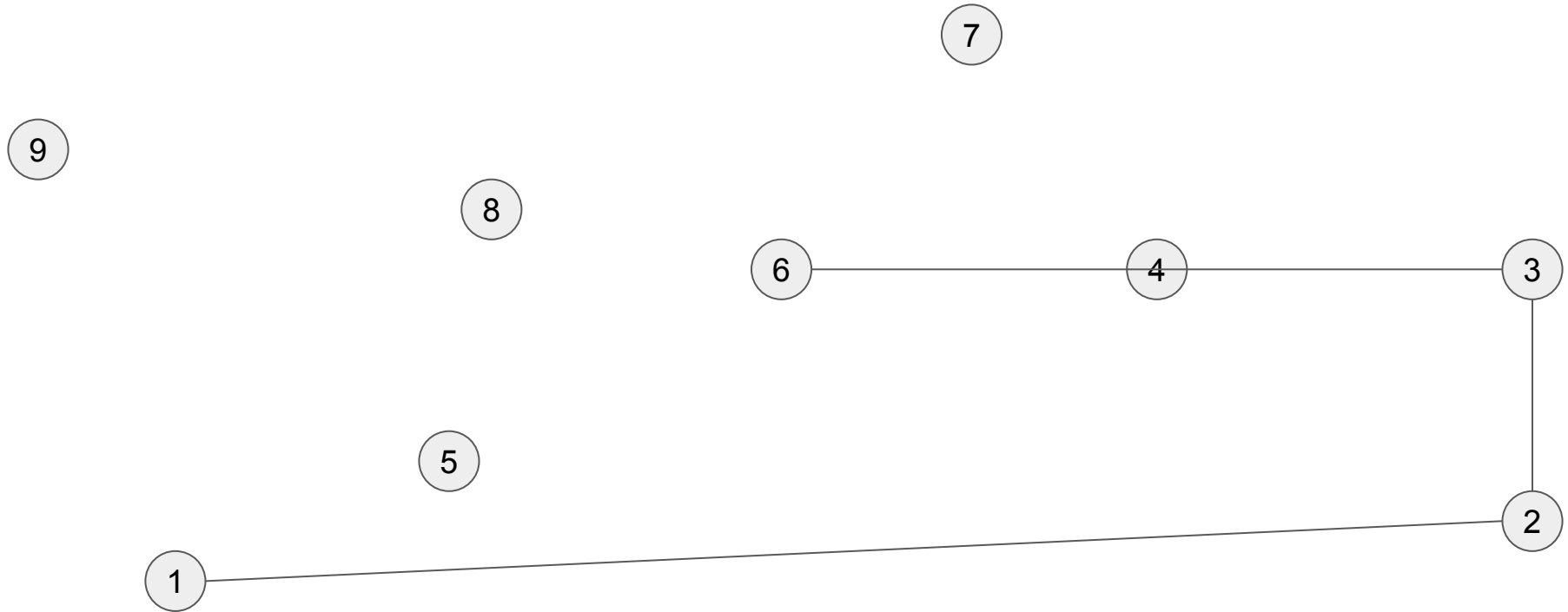
Graham's Scan [6]



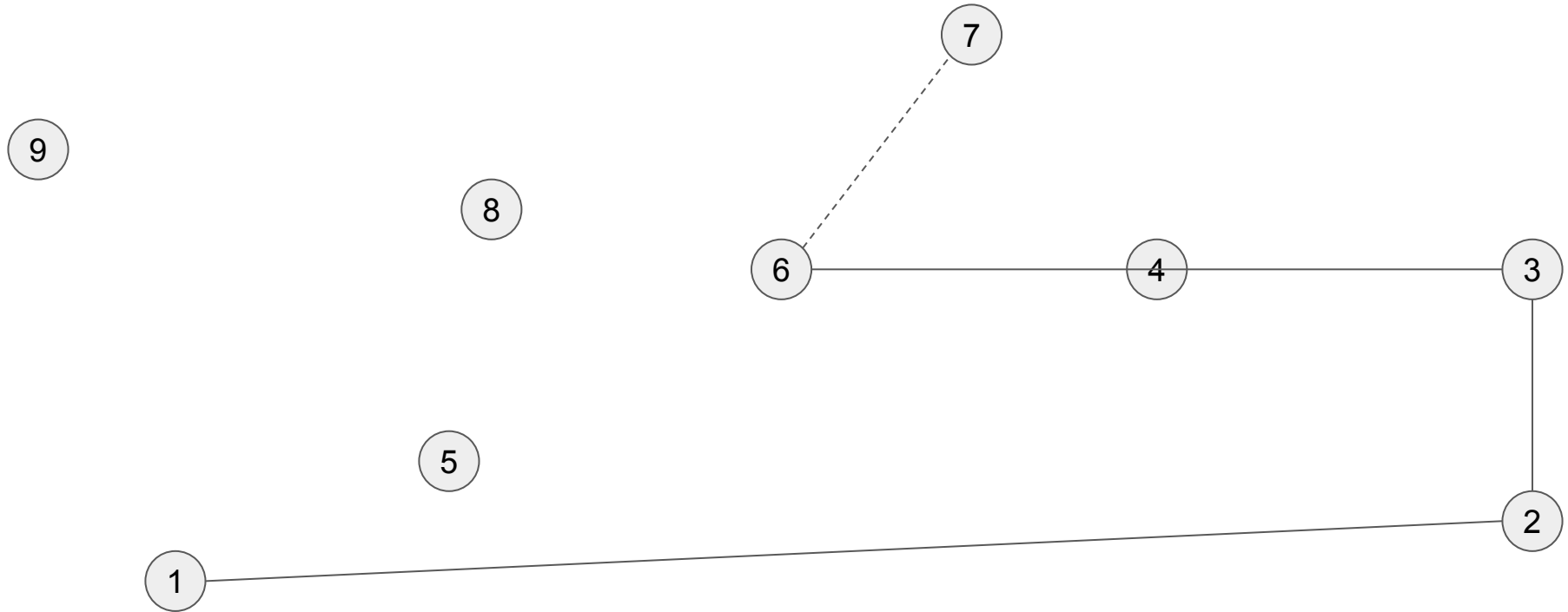
Graham's Scan [7]



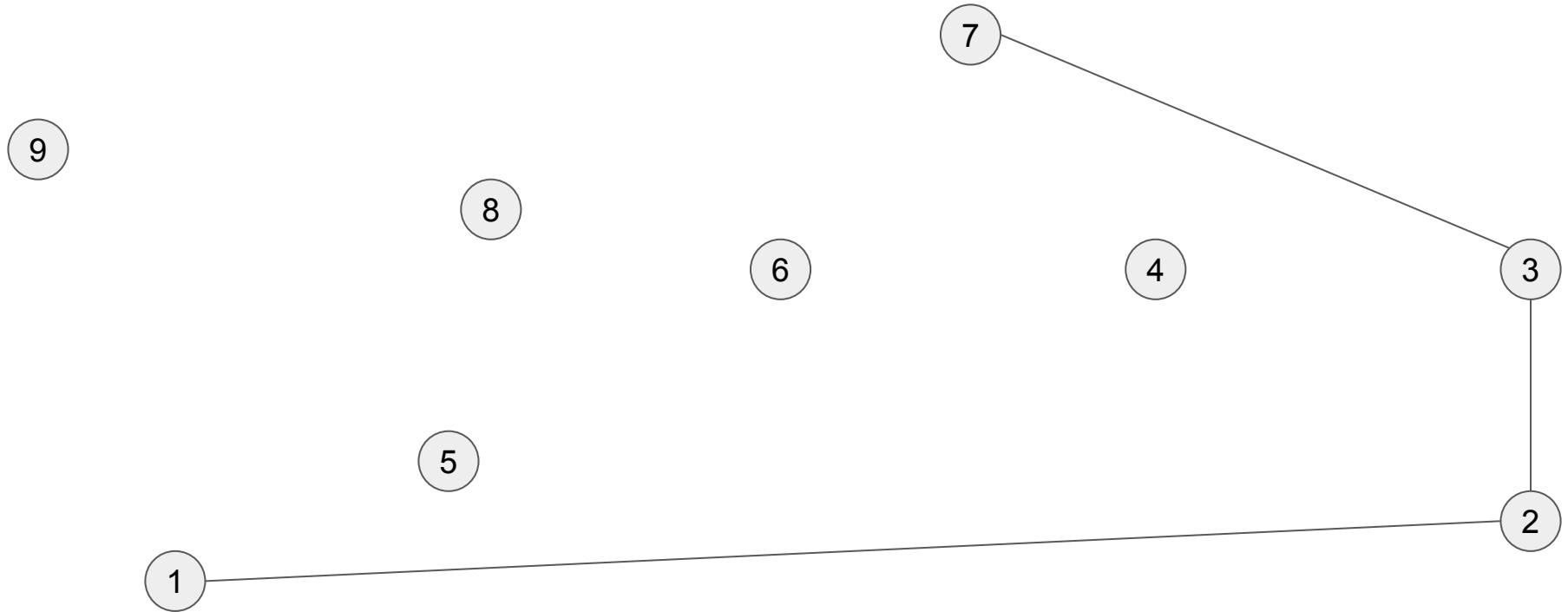
Graham's Scan [8]



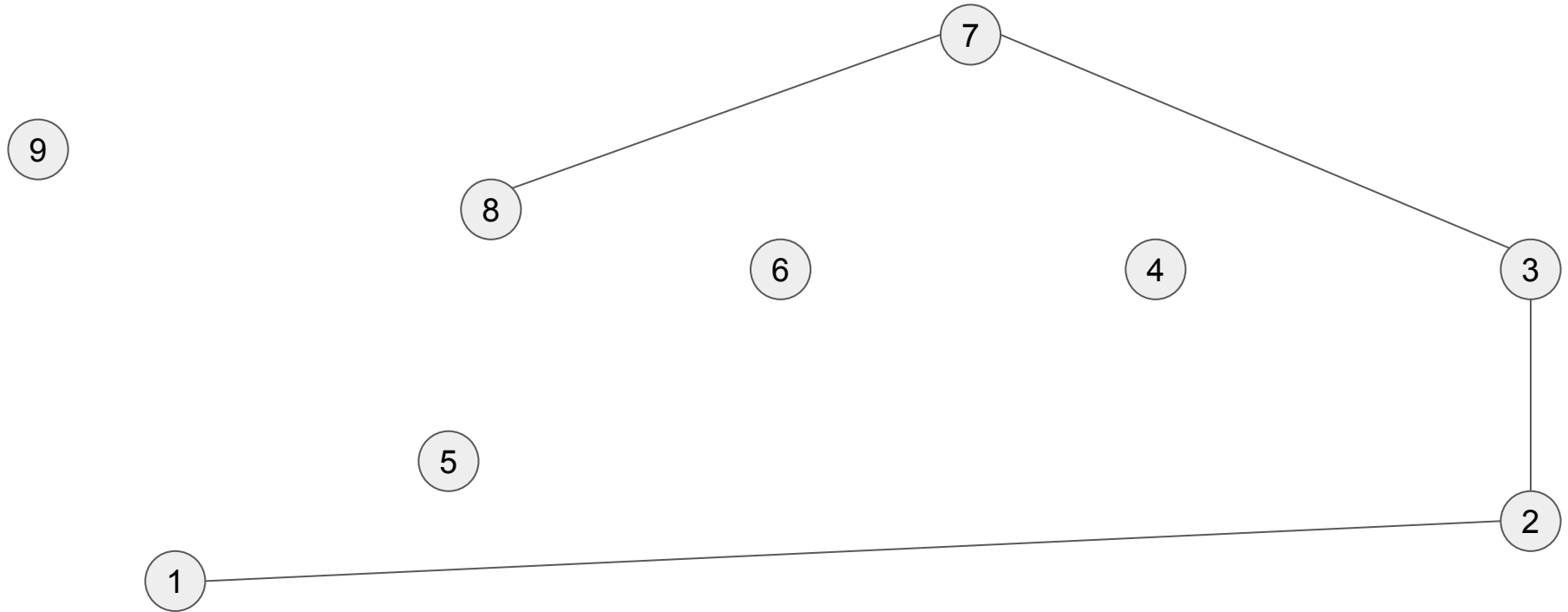
Graham's Scan [9]



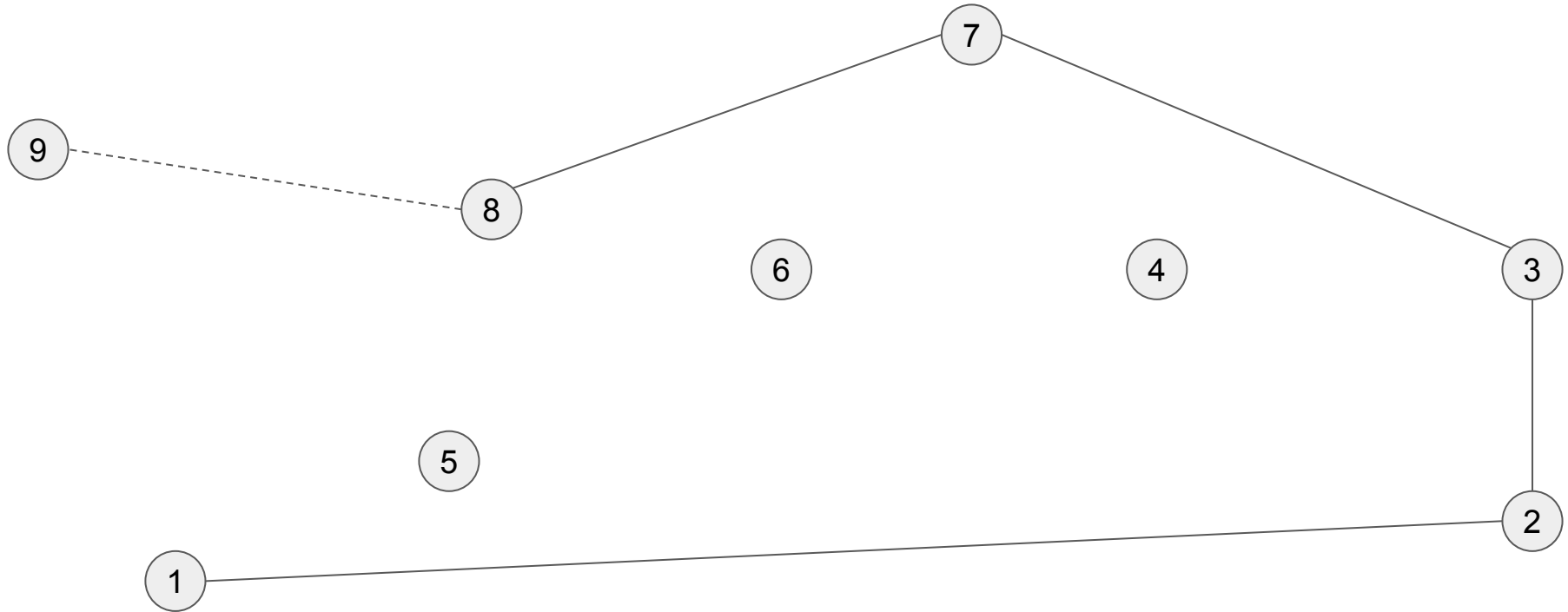
Graham's Scan [10]



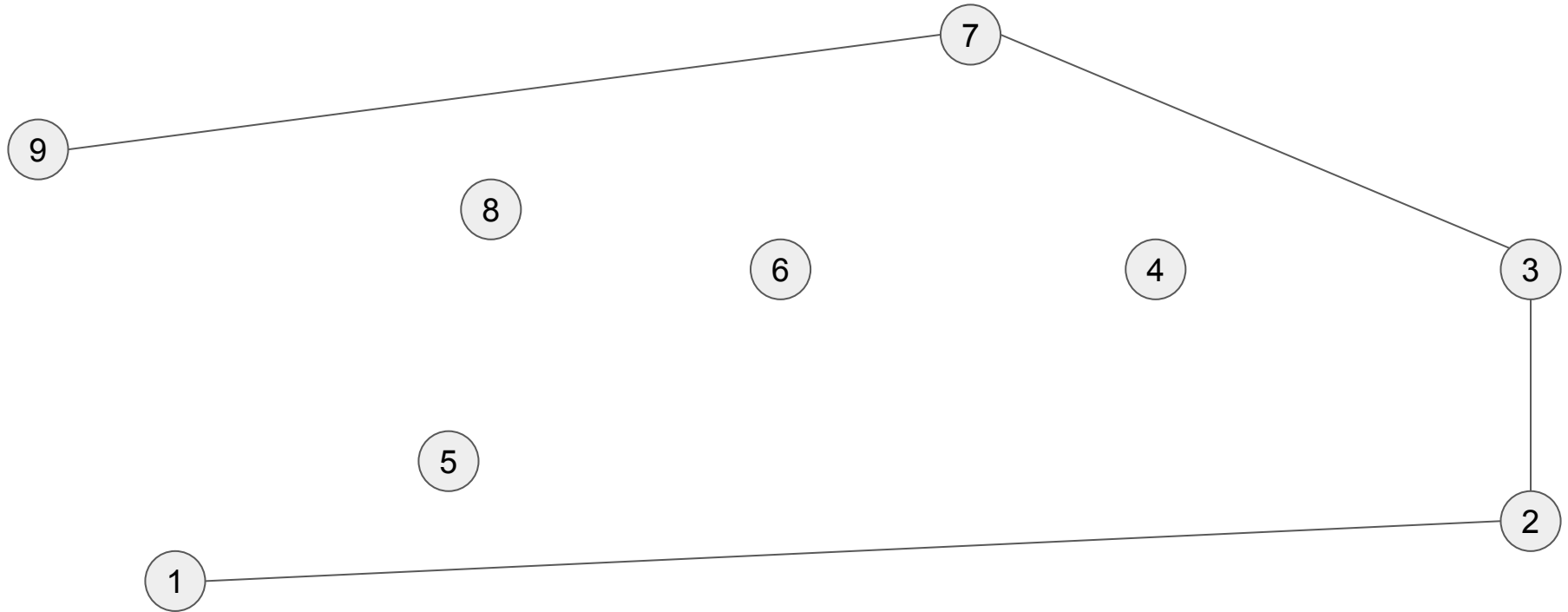
Graham's Scan [11]



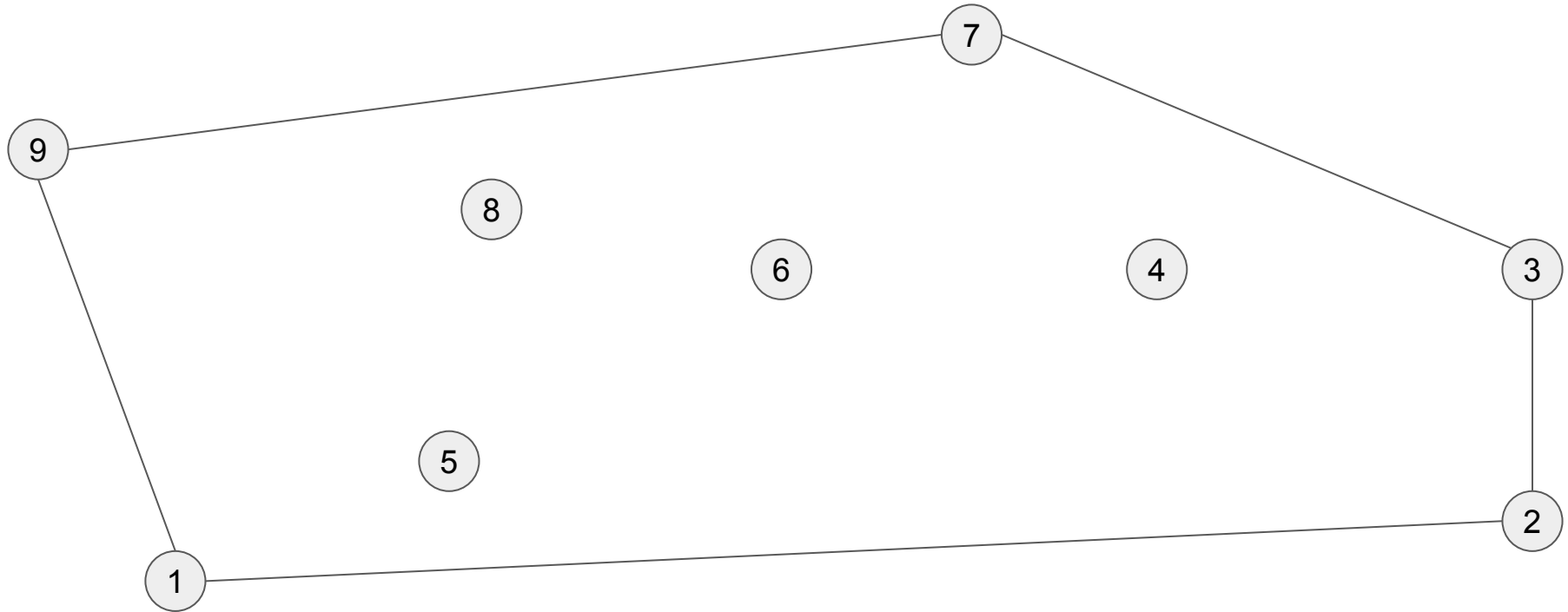
Graham's Scan [12]



Graham's Scan [13]



Graham's Scan [14]



Analysis of Graham's Scan

Calculating one polar angle is $O(1)$. Calculating n of them is $O(n)$.

Sorting n polar angles is $O(n \lg n)$, with any sensible comparison-based sort (including the tie-break logic to discard points with sub-maximal distance).

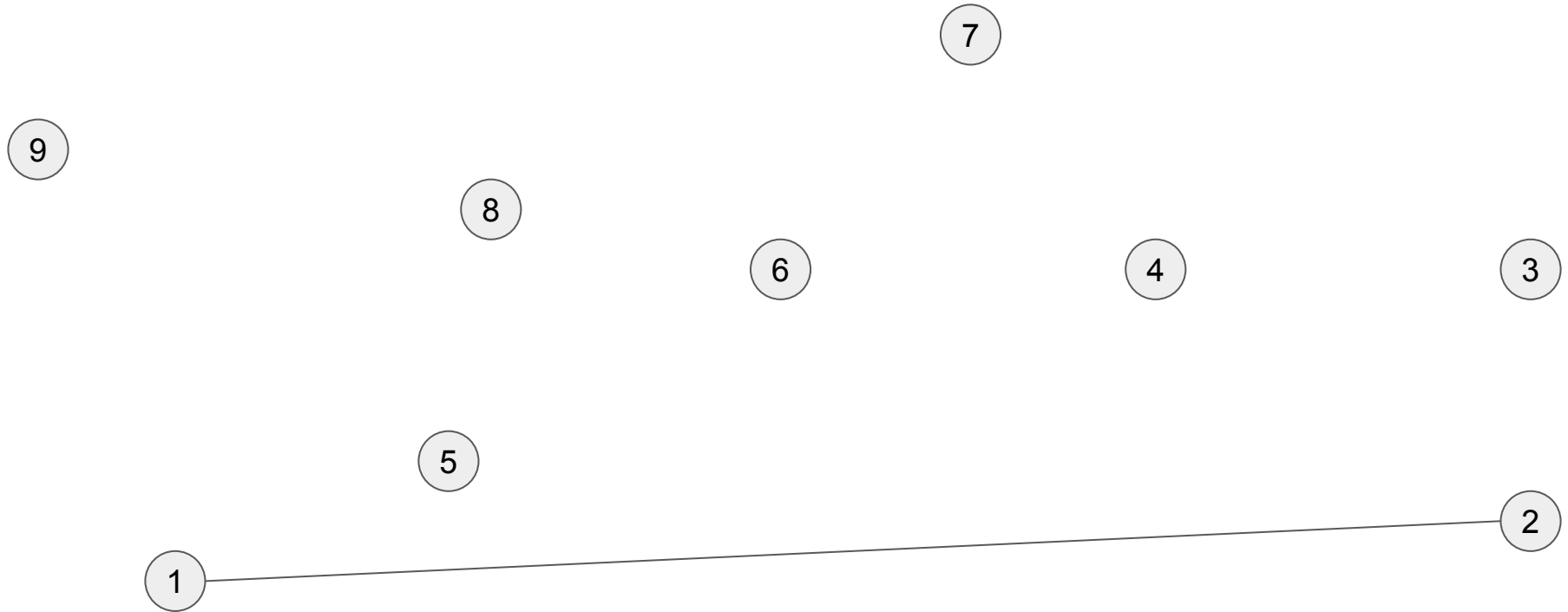
As we walk around the hull, each point is only pushed to the stack at most once and is removed at most once. Every comparison either adds a point to the stack or removes a point from the stack. Hence the walk is $O(n)$.

Graham's Scan costs $O(n \lg n)$, dominated by the sorting step.

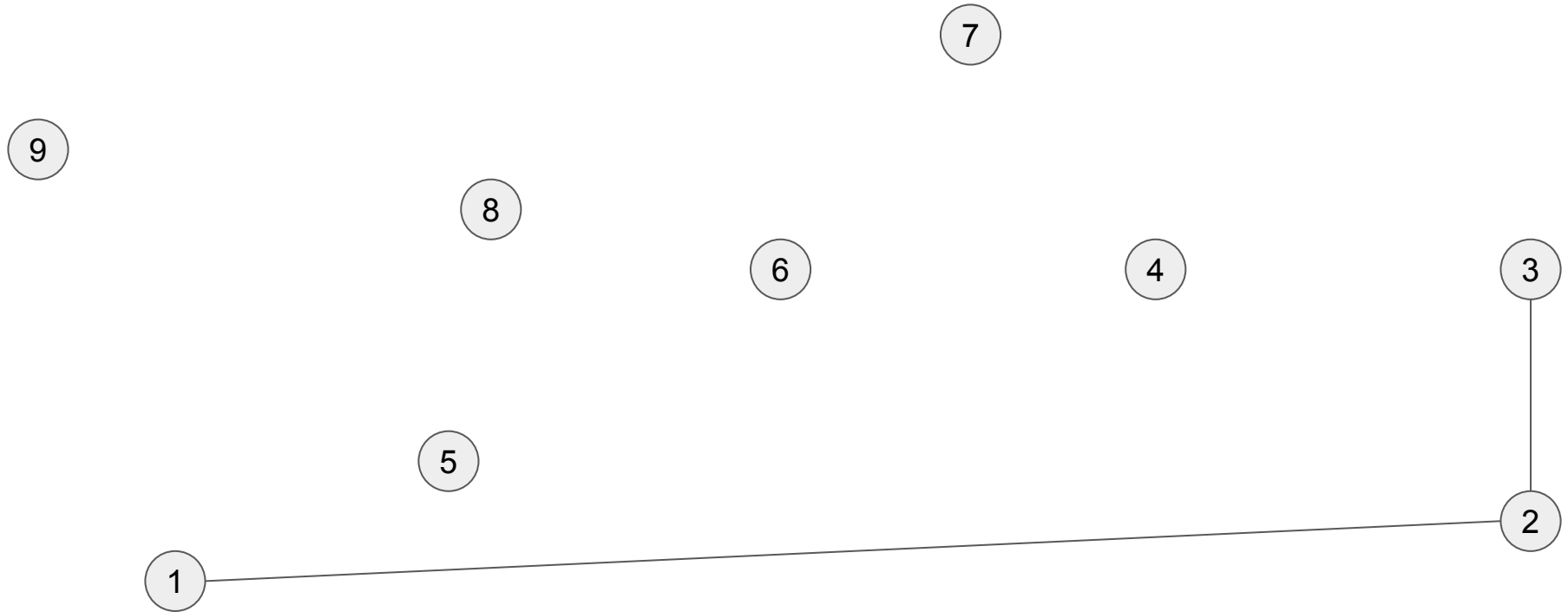
Jarvis's March [1]

- Start with the left-most of the bottom-most points, p_1 , which is on the hull.
- Find the point p_2 with the least polar angle relative to a horizontal line through p_1 . p_2 is also on the hull.
- Repeatedly find the point p_{i+1} with the least polar angle relative to the line through p_{i-1} and p_i . p_{i+1} is on the hull. The p_i form the **right chain**.
 - The repetition continues until a top-most point is reached (might not be unique).
- Repeat the previous two bullets to find the **left chain** using greatest polar angles.
- Join the right chain and left chain to get the convex hull.

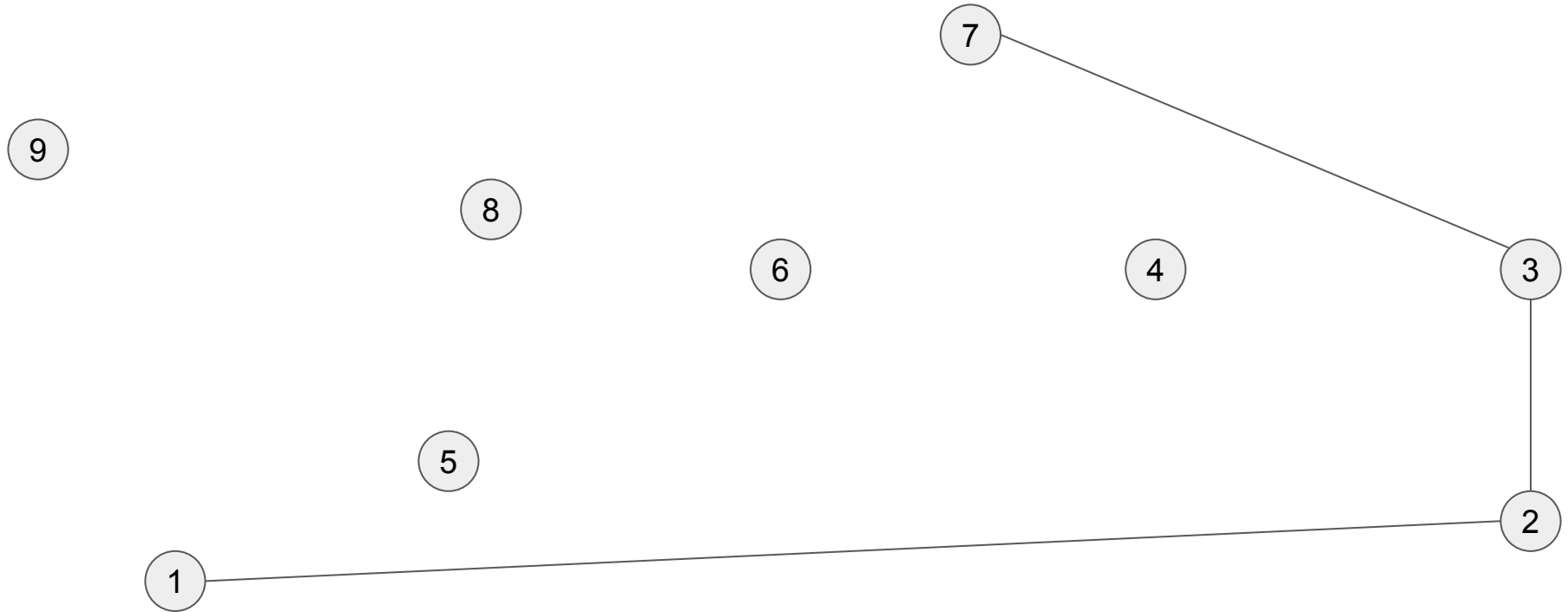
Jarvis's March [2]



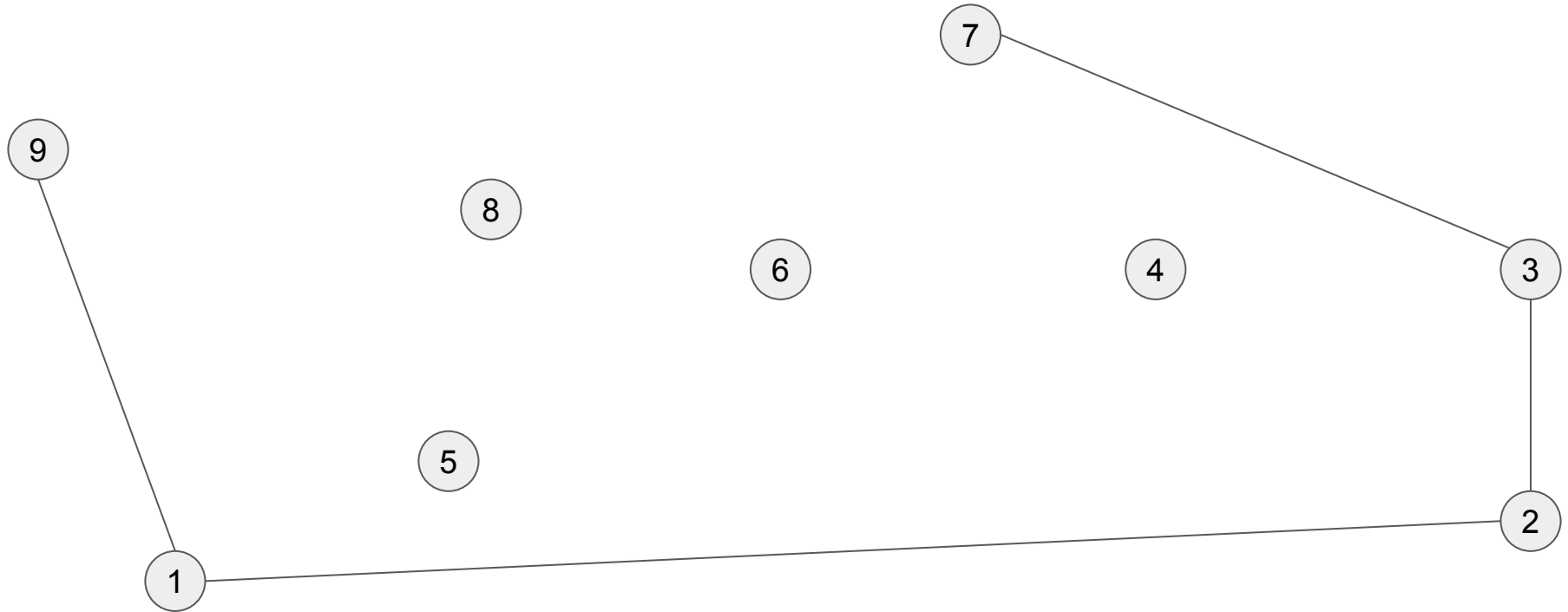
Jarvis's March [3]



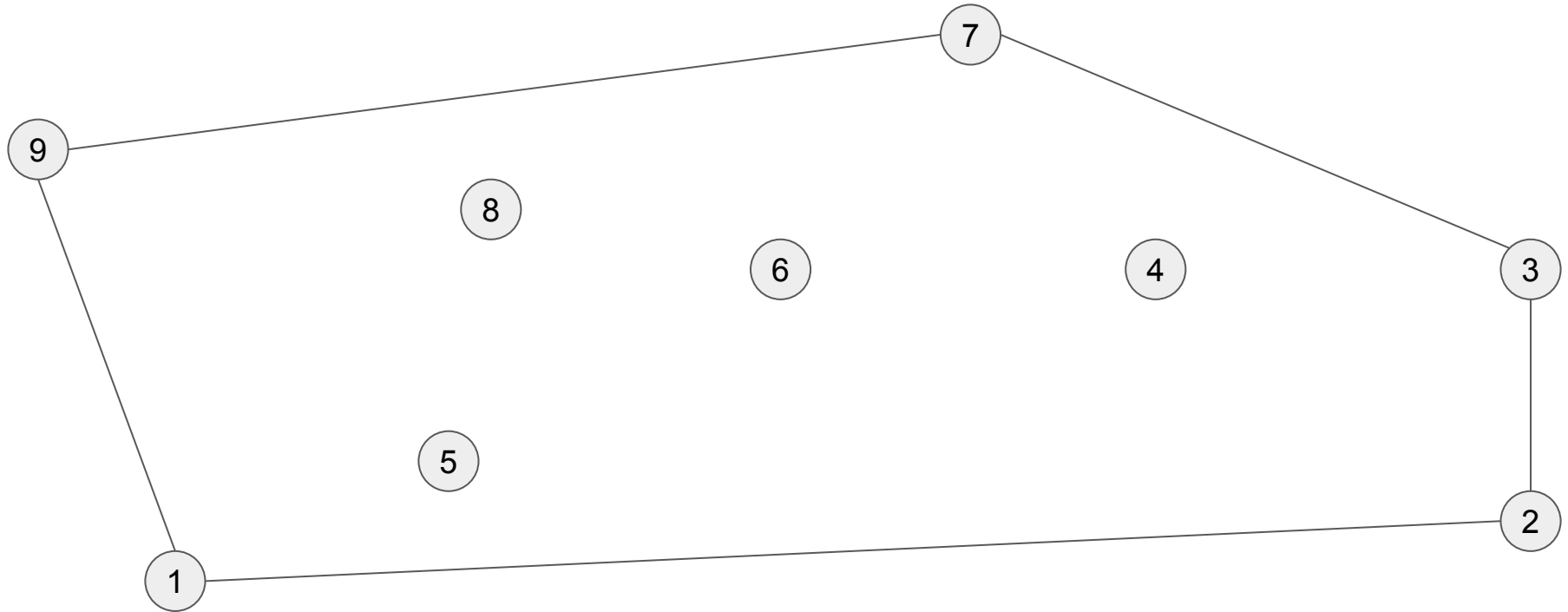
Jarvis's March [4]



Jarvis's March [5]



Jarvis's March [6]



Analysis of Jarvis's March

Calculating one polar angle is $O(1)$. Calculating n of them is $O(n)$.

Finding the minimum of n numbers is $O(n)$.

Repeating that h times is $O(n h)$.

The right/left chain allows us to exploit the cross product trick for comparisons because the polar angles, θ , we handle are always in the range $-\pi/2 \leq \theta < \pi/2$.

Revision Guide / Summary of Algorithms 2 [1]

- Graphs
 - Representing the edge set with adjacency lists and adjacency matrices
 - Terminology
- Graph colouring problems: vertex, edge, face colouring
- Breadth-first search
 - With the concept of 'depth' to solve vertex colouring
 - Subgraph induced by the predecessors: breadth first tree
- Depth-first search
 - Discovery time and finish time for each vertex
 - Topological sort
- Edge classification: tree edge, back edge, forward edge, cross edge

Revision Guide / Summary of Algorithms 2 [2]

- Strongly connected components
 - Two DFSs and the transpose graph
- Shortest path problems:
 - Single-source shortest paths
 - Single-destination shortest paths
 - Single-pair shortest path
 - All-pairs shortest paths
- Complications caused by negative edges, negative cycles, zero-weight cycles
- Bellman-Ford
 - Introduced the concept of edge relaxation
 - Special case for directed acyclic graphs with lower costs

Revision Guide / Summary of Algorithms 2 [3]

- Optimal substructure led to Dijkstra's algorithm
 - Unable to handle negative edge weights
 - Proof of correctness using the convergence lemma
- Matrix multiplication methods for all-pairs shortest paths
 - Mapping domain-specific problems to other theory, to pull in speed-ups from other research
 - Repeated squaring
 - Floyd-Warshall
- Johnson's algorithm
 - Introduced the concept of reweighting

Revision Guide / Summary of Algorithms 2 [4]

- Flow networks
 - Capacity
 - Max-Flow Min-Cut Theorem
 - Ford-Fulkerson (Edmunds-Karp as optimisation)
 - Augmenting paths, flow cancellation
- Bipartite matchings
 - Maximum bipartite matchings (Hopcroft-Karp as an optimisation)
 - Maximum and maximal matchings
- Minimum spanning trees
 - Safe edge theorem
 - Kruskal's algorithm
 - Prim's algorithm

Revision Guide / Summary of Algorithms 2 [5]

- Amortised analysis
 - Aggregate method
 - Accounting method
 - Potential method
- Mergeable Priority Queues
 - Binomial Heaps
 - Fibonacci Heaps, golden ratio, peculiar property giving them their name
- Disjoint set representations
 - Path compression and union-by-rank

Revision Guide / Summary of Algorithms 2 [6]

- Geometric algorithms
 - Simple, planar and closed polygons
 - Defining the inside and outside
 - Winding numbers
 - Line segment intersection problems
 - Cross-product tricks for numerical stability and performance
- Convex Hulls
 - Graham's scan
 - Jarvis's March
 - ... and I tantalised you with the “Search and Prune” asymptotically optimal method!

Thank you for listening!

*I hope you enjoyed the course.
Please fill in the lecture feedback forms.
Good luck in the exams!*