Advanced Computational Geometry for Olympiad & Competitive Programming

Your Name Here

May 12, 2025

Preface

This book is designed for aspiring and experienced competitive programmers who wish to master advanced computational geometry. It covers a wide range of topics, from fundamental primitives to sophisticated algorithms and optimization techniques, all with a focus on practical implementation and robust solutions crucial for programming contests.

We aim to provide clear theoretical explanations, illustrative examples, canonical algorithm structures, discussions on precision and common pitfalls, relevant problem patterns, and template-quality C++17 code snippets.

Placeholder for more preface content, acknowledgements, etc.

Introduction

Computational geometry is a cornerstone of algorithm design, frequently appearing in programming olympiads and contests. Problems in this domain often test not only algorithmic knowledge but also the ability to implement these algorithms correctly and robustly, especially when dealing with floating-point arithmetic and degenerate cases.

This book is structured into six main parts:

- **Part I: Foundations & Primitives** lays the groundwork with essential geometric objects, operations, and numerical considerations.
- **Part II: Polygons & Basic Structures** delves into polygons, their properties, and lattice geometry.
- **Part III: Core Geometric Algorithms** covers fundamental algorithms for intersection, distance, and convex hulls.
- **Part IV: Optimisation Techniques with Geometric Flavor** explores powerful DP optimization techniques like CHT, Slope Trick, and geometry-aware DP.
- **Part V: Advanced Algorithms & Data Structures** introduces sweep-line algorithms, spatial data structures, parametric search, and other advanced tools.
- **Part VI: Implementation & Reference** provides practical templates, boilerplate code, and debugging tips.

Each chapter typically follows this structure:

- **Formal Theory**: Definitions, theorems, and mathematical underpinnings.
- **Canonical Algorithms**: Pseudocode, explanations, and complexity analysis.
- **Precision & Implementation Gotchas**: Common errors, numerical stability, and handling degeneracies.
- **Classic and Unusual Use-Cases / Problem Patterns**: Example problems with links and rationale.
- **Template-Quality Code Snippets**: C++17 implementations.
- **Further Reading**: Pointers to authoritative texts and online resources.
- **Open Research Questions or Lesser-Known Tricks**: For deeper exploration.

We hope this book serves as a valuable resource for your competitive programming journey.
Placeholder for more introduction content, how to use the book, target audience, etc.

Contents

Part I

Foundations & Primitives

Chapter 1

Geometry Primitives & Numerical Foundations

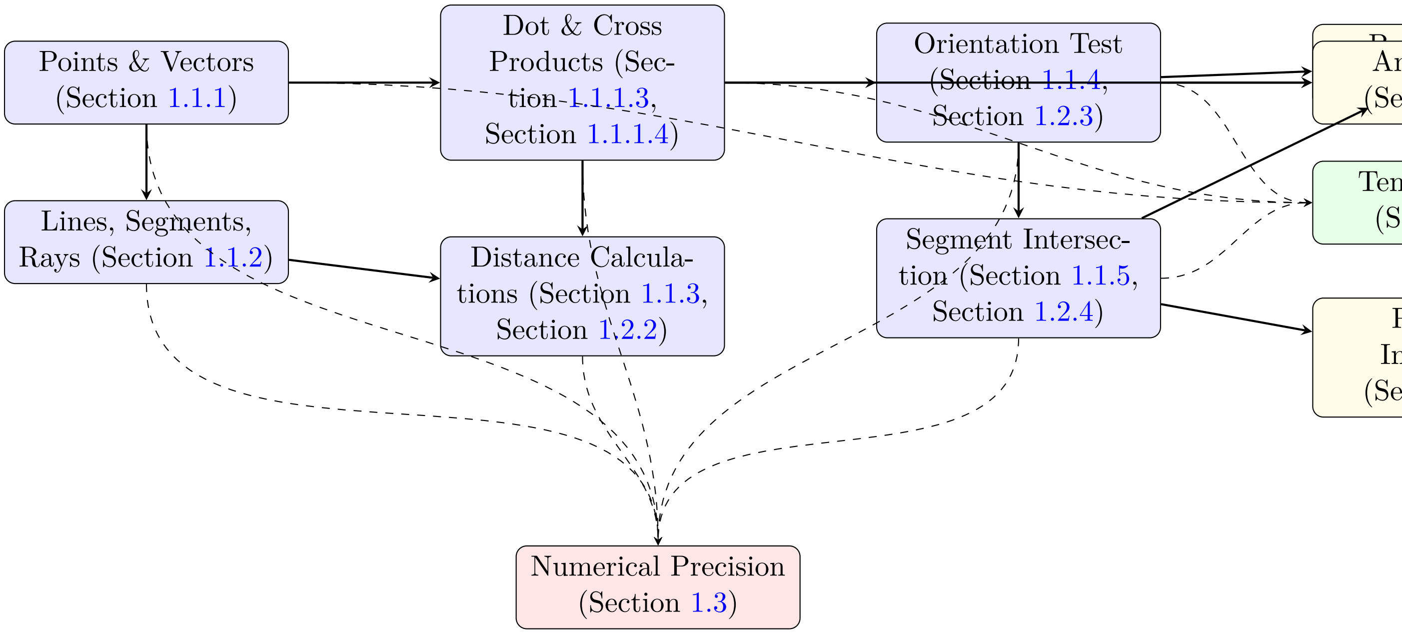
Imagine you're designing the AI for a robot navigating a warehouse filled with obstacles. The robot needs to find the shortest path, identify which items are visible from its current location, and even figure out how to grasp objects. Or picture yourself developing a new strategy game where units need to determine lines of sight, optimal firing angles, and whether their formations will collide. These scenarios, common in robotics, game development, and even geographic information systems (GIS), all rely on a fundamental understanding of how to represent and manipulate geometric objects computationally.

This chapter lays the groundwork for your journey into computational geometry. We'll start with the very basics: points, vectors, lines, and how to perform fundamental operations like calculating distances and determining orientations. Mastering these primitives is crucial because they are the building blocks for almost every advanced algorithm you'll encounter later. Without a solid grasp of these foundations, tackling complex problems like finding convex hulls or detecting line segment intersections becomes incredibly challenging.

Challenge Problem: *The Art Gallery Guardian*

An art gallery is shaped like a simple polygon (a polygon that doesn't self-intersect). A single, stationary security camera needs to be placed at one of the polygon's vertices. From its vertex, the camera can see in all directions (360 degrees). Your task is to determine if there exists a vertex from which the entire interior of the gallery is visible. By the end of this chapter, you won't solve this full problem (it's a classic known as the Art Gallery Problem, often needing more advanced techniques), but you'll have the tools to check if a specific point (like a proposed camera location) can see a specific segment (like a wall of the gallery) without obstruction from other walls. This is a key step! More specifically, you'll be able to determine if two points A and B inside or on the boundary of a polygon are mutually visible (i.e., the segment AB does not intersect the interior of any edge of the polygon it's not part of, unless AB itself is an edge).

Chapter Roadmap


Points & Vectors (Section 1.1.1)
Dot & Cross Products (Section 1.1.1.3, Section 1.1.1.4)
Orientation Test (Section 1.1.4, Section 1.2.3)
Lines, Segments, Rays (Section 1.1.2)
Distance Calculations (Section 1.1.3, Section 1.2.2)
Segment Intersection (Section 1.1.5, Section 1.2.4)
Numerical Precision (Section 1.3)

1.1 Formal Theory: The Language of Shapes

Welcome to the mathematical heart of our geometric toolkit! Before we can write clever algorithms, we need a precise language to talk about shapes and their properties. This section introduces the fundamental "nouns" and "verbs" of 2D computational geometry: points, vectors, lines, and the basic operations that connect them. Don't worry, we'll keep the theory grounded and always link it back to how you'll actually use these concepts in code. Think of this as learning the alphabet before writing poetry.

1.1.1 Points and Vectors: The Atoms of Geometry

Everything in 2D geometry starts with points and vectors. They might seem simple, but understanding them deeply is key.

1.1.1.1 Definitions: What Are They?

Definition 1.1.1 (Point)**.** A **point** in 2D Euclidean space represents a specific location. It is typically defined by a pair of coordinates (x, y) relative to an origin in a Cartesian coordinate system.

> **Intuition:** Think of a point as a pinprick on a map. It has no size or dimension, just a position. In code, you'll usually represent a point as a struct or class with two members, `x` and `y`.

A simple diagram showing a 2D Cartesian grid with the origin O, and a few points like $P(3, 2)$ and $Q(-1, 4)$ plotted and labeled with their coordinates.

Definition 1.1.2 (Vector)**.** A **vector** in 2D Euclidean space represents a quantity possessing both magnitude (length) and direction. It can be visualized as a directed line segment. If $A = (x_A, y_A)$ and $B = (x_B, y_B)$ are two points, the vector $\vec{AB}$ (from A to B) is given by $(x_B - x_A, y_B - y_A)$. A vector can also be defined simply as a pair of components (v_x, v_y) representing displacement from an implicit origin or some starting point.

> **Intuition:** A vector is like an instruction: "Go this far, in this direction." If a point is a destination, a vector is the journey. Crucially, vectors don't have a fixed position; the vector $(1, 1)$ is the same whether it starts at $(0, 0)$ and ends at $(1, 1)$, or starts at $(5, 5)$ and ends at $(6, 6)$. It's all about the displacement. In code, vectors are often represented using the same struct/class as points, since both are pairs of numbers. The meaning (position vs. displacement) comes from context.

Diagram shows two points A and B. An arrow is drawn from A to B, labeled $\vec{AB}$. Below this, the components of $\vec{AB}$ are shown as $(x_B - x_A, y_B - y_A)$. Separately, a vector $\mathbf{v} = (v_x, v_y)$ is shown as an arrow from the origin.

1.1.1.2 Vector Operations: The Algebra of Arrows

Just like numbers, vectors can be added, subtracted, and scaled.

Definition 1.1.3 (Vector Addition, Subtraction, Scalar Multiplication)**.** Let $\mathbf{u} = (u_x, u_y)$ and $\mathbf{v} = (v_x, v_y)$ be two vectors, and k be a scalar (a real number).

- **Addition**: $\mathbf{u} + \mathbf{v} = (u_x + v_x, u_y + v_y)$.
- **Subtraction**: $\mathbf{u} - \mathbf{v} = (u_x - v_x, u_y - v_y)$. This is equivalent to $\mathbf{u} + (-\mathbf{v})$, where $-\mathbf{v} = (-v_x, -v_y)$.
- **Scalar Multiplication**: $k \cdot \mathbf{u} = (k \cdot u_x, k \cdot u_y)$.

Intuition:

- **Addition**: Think "tip-to-tail". To add $\mathbf{u}$ and $\mathbf{v}$, place the tail of $\mathbf{v}$ at the tip of $\mathbf{u}$. The sum is the vector from the tail of $\mathbf{u}$ to the tip of $\mathbf{v}$. (Parallelogram law also works).
- **Subtraction**: $\mathbf{u} - \mathbf{v}$ is the vector that goes from the tip of $\mathbf{v}$ to the tip of $\mathbf{u}$ if they share the same origin. Or, it's $\mathbf{u} + (-\mathbf{v})$.
- **Scalar Multiplication**: $k \cdot \mathbf{u}$ scales the length of $\mathbf{u}$ by $|k|$. If $k > 0$, direction is preserved. If $k < 0$, direction is reversed. If $k = 0$, it becomes the zero vector $(0, 0)$.

A common operation: if P, Q are points, $Q - P$ gives vector $\vec{PQ}$. If P is a point and $\mathbf{v}$ is a vector, $P + \mathbf{v}$ gives a new point.

Panel 1: Vectors $\mathbf{u}$ and $\mathbf{v}$ originating from the same point. $\mathbf{u} + \mathbf{v}$ shown by completing the parallelogram. Panel 2: Vector $\mathbf{u}$ and $\mathbf{v}$. $\mathbf{u} - \mathbf{v}$ shown as $\mathbf{u} + (-\mathbf{v})$. Panel 3: Vector $\mathbf{u}$. Then $2\mathbf{u}$ (longer, same direction) and $-1\mathbf{u}$ (same length, opposite direction) are shown.

1.1.1.3 Dot Product: How Aligned Are They?

Definition 1.1.4 (Dot Product)**.** The **dot product** (or scalar product) of two vectors $\mathbf{a} = (a_x, a_y)$ and $\mathbf{b} = (b_x, b_y)$ is a scalar value defined as:

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y$$

Geometrically, it is also given by:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos(\theta)$$

where $|\mathbf{a}|$ and $|\mathbf{b}|$ are the magnitudes (lengths) of the vectors, and θ is the angle between them $(0 \le \theta \le \pi)$.

Intuition: The dot product tells you how much one vector "goes in the direction of" another.

- If $\mathbf{a} \cdot \mathbf{b} > 0$: The angle θ is acute ($< 90°$). They point in roughly the same direction.
- If $\mathbf{a} \cdot \mathbf{b} < 0$: The angle θ is obtuse ($> 90°$). They point in roughly opposite directions.
- If $\mathbf{a} \cdot \mathbf{b} = 0$: The angle θ is $90°$ (or one/both vectors are zero). They are **orthogonal** (perpendicular). This is a super useful property!

Also, $\mathbf{a} \cdot \mathbf{a} = |\mathbf{a}|^2$. This gives a way to find squared length without `sqrt`.

Panel 1: Vectors $\mathbf{a}, \mathbf{b}$ with acute angle θ. Text: "$\mathbf{a} \cdot \mathbf{b} > 0$". Panel 2: Vectors $\mathbf{a}, \mathbf{b}$ with obtuse angle θ. Text: "$\mathbf{a} \cdot \mathbf{b} < 0$". Panel 3: Vectors $\mathbf{a}, \mathbf{b}$ orthogonal. Text: "$\mathbf{a} \cdot \mathbf{b} = 0$".

Theorem 1.1.1 (Properties of Dot Product)**.** *For any vectors* $\mathbf{u}, \mathbf{v}, \mathbf{w}$ *and scalar* k*:*

1. *Commutative:* $\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u}$
2. *Distributive over addition:* $\mathbf{u} \cdot (\mathbf{v} + \mathbf{w}) = \mathbf{u} \cdot \mathbf{v} + \mathbf{u} \cdot \mathbf{w}$
3. *Bilinear:* $(k\mathbf{u}) \cdot \mathbf{v} = \mathbf{u} \cdot (k\mathbf{v}) = k(\mathbf{u} \cdot \mathbf{v})$
4. $\mathbf{u} \cdot \mathbf{u} = |\mathbf{u}|^2 \geq 0$, *and* $\mathbf{u} \cdot \mathbf{u} = 0 \iff \mathbf{u} = \mathbf{0}$ *(zero vector)*

Mathematical Insight: You can find the angle θ between two non-zero vectors using:

$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}$$

$$\theta = \text{acos}\left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}\right)$$

Be careful with floating point precision when the fraction is very close to 1 or -1. Using `atan2` (see **??**) is often more robust for finding angles if you also have cross product information.

1.1.1.4 Cross Product (2D): Turning and Area

Definition 1.1.5 (2D Cross Product)**.** The **2D cross product** (often called "perp dot product" or "outer product" in 2D context) of two vectors $\mathbf{a} = (a_x, a_y)$ and $\mathbf{b} = (b_x, b_y)$ is a scalar value defined as:

$$\mathbf{a} \times \mathbf{b} = a_x b_y - a_y b_x$$

Geometrically, it is related to the angle θ from $\mathbf{a}$ to $\mathbf{b}$ (measured counter-clockwise):

$$\mathbf{a} \times \mathbf{b} = |\mathbf{a}||\mathbf{b}| \sin(\theta)$$

The value $\mathbf{a} \times \mathbf{b}$ is also the signed area of the parallelogram formed by vectors $\mathbf{a}$ and $\mathbf{b}$ placed at the origin. The area of the triangle formed by origin, $\mathbf{a}$, and $\mathbf{b}$ is $\frac{1}{2}(\mathbf{a} \times \mathbf{b})$.

Intuition: The 2D cross product is incredibly powerful for determining orientation:

- If $\mathbf{a} \times \mathbf{b} > 0$: Vector $\mathbf{b}$ is counter-clockwise (CCW) from vector $\mathbf{a}$ (if they share an origin). Think "left turn" from $\mathbf{a}$ to get to $\mathbf{b}$.
- If $\mathbf{a} \times \mathbf{b} < 0$: Vector $\mathbf{b}$ is clockwise (CW) from vector $\mathbf{a}$. Think "right turn".
- If $\mathbf{a} \times \mathbf{b} = 0$: Vectors $\mathbf{a}$ and $\mathbf{b}$ are **collinear** (point in the same or exactly opposite directions, or one/both are zero).

This is the basis for the crucial "orientation test" (Section 1.1.4). Unlike the 3D cross product which yields a vector, the 2D version (as defined for geometry) gives a scalar. This scalar can be thought of as the z-component of the 3D cross product if $\mathbf{a}$ and $\mathbf{b}$ were in the xy-plane.

Panel 1: Vectors $\mathbf{a}, \mathbf{b}$ from origin O. $\mathbf{b}$ is CCW from $\mathbf{a}$. Text: "$\mathbf{a} \times \mathbf{b} > 0$ (CCW)". Shaded parallelogram. Panel 2: Vectors $\mathbf{a}, \mathbf{b}$ from origin O. $\mathbf{b}$ is CW from $\mathbf{a}$. Text: "$\mathbf{a} \times \mathbf{b} < 0$ (CW)". Shaded parallelogram. Panel 3: Vectors $\mathbf{a}, \mathbf{b}$ from origin O, collinear. Text: "$\mathbf{a} \times \mathbf{b} = 0$".

Theorem 1.1.2 (Properties of 2D Cross Product)**.** *For any 2D vectors* $\mathbf{u}, \mathbf{v}, \mathbf{w}$ *and scalar* k*:*

1. *Anti-commutative:* $\mathbf{u} \times \mathbf{v} = -(\mathbf{v} \times \mathbf{u})$
2. *Distributive over addition:* $\mathbf{u} \times (\mathbf{v} + \mathbf{w}) = \mathbf{u} \times \mathbf{v} + \mathbf{u} \times \mathbf{w}$
3. *Bilinear:* $(k\mathbf{u}) \times \mathbf{v} = \mathbf{u} \times (k\mathbf{v}) = k(\mathbf{u} \times \mathbf{v})$
4. $\mathbf{u} \times \mathbf{u} = 0$

Mathematical Insight: For three points P_1, P_2, P_3, the cross product $(P_2 - P_1) \times (P_3 - P_1)$ gives twice the signed area of triangle $P_1P_2P_3$. Its sign determines if $P_1 \to P_2 \to P_3$ is a CCW turn (positive), CW turn (negative), or if the points are collinear (zero). This is fundamental!

1.1.1.5 Norm (Magnitude) and Norm Squared: Measuring Length

Definition 1.1.6 (Norm and Norm Squared)**.** The **norm** (or magnitude, length) of a vector $\mathbf{v} = (v_x, v_y)$ is denoted $|\mathbf{v}|$ or $||\mathbf{v}||$ and is given by:

$$|\mathbf{v}| = \sqrt{v_x^2 + v_y^2}$$

The **norm squared** (or squared magnitude) is:

$$|\mathbf{v}|^2 = v_x^2 + v_y^2$$

The distance between two points P_1 and P_2 is the norm of the vector $\vec{P_1P_2}$: $d(P_1, P_2) = |\vec{P_1P_2}| = |P_2 - P_1|$.

Intuition: The norm is just what it sounds like: the length of the vector arrow, calculated using the Pythagorean theorem. The norm squared is often used in competitive programming to avoid `sqrt()` calls, which can be slow and introduce floating-point errors. If you only need to compare distances (e.g., "is $d_1 < d_2$?"), you can compare squared distances ("is $d_1^2 < d_2^2$?"), provided distances are non-negative (which they always are).

Tips

Compare Squared Distances: To check if distance A is less than distance B, compare $A^2 < B^2$. This avoids `sqrt` and is safer with integers (prevents float conversion) and faster. Only take the `sqrt` if you need the actual distance value.

1.1.1.6 Vector Projection and Rejection: Decomposing Vectors

Definition 1.1.7 (Vector Projection and Rejection)**.** Let $\mathbf{a}$ and $\mathbf{b}$ be two vectors, with $\mathbf{b} \neq \mathbf{0}$. The **vector projection** of $\mathbf{a}$ onto $\mathbf{b}$ (denoted $\text{proj}_{\mathbf{b}}\mathbf{a}$) is the component of $\mathbf{a}$ that lies in the direction of $\mathbf{b}$.

$$\text{proj}_{\mathbf{b}}\mathbf{a} = \left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{b}|^2}\right)\mathbf{b}$$

The scalar $\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{b}|}$ is the signed length of this projection.

The **vector rejection** of $\mathbf{a}$ from $\mathbf{b}$ (denoted $\text{rej}_{\mathbf{b}}\mathbf{a}$) is the component of $\mathbf{a}$ orthogonal to $\mathbf{b}$.

$$\text{rej}_{\mathbf{b}}\mathbf{a} = \mathbf{a} - \text{proj}_{\mathbf{b}}\mathbf{a}$$

Intuition: Imagine shining a light perpendicularly onto the line containing vector $\mathbf{b}$. The shadow of vector $\mathbf{a}$ on this line is $\text{proj}_{\mathbf{b}}\mathbf{a}$. The rejection is what's "left over" of $\mathbf{a}$ after you subtract its shadow component; it's perpendicular to $\mathbf{b}$. Together, $\text{proj}_{\mathbf{b}}\mathbf{a} + \text{rej}_{\mathbf{b}}\mathbf{a} = \mathbf{a}$. This decomposes $\mathbf{a}$ into two orthogonal parts, one parallel to $\mathbf{b}$ and one perpendicular. This is key for finding the closest point on a line to another point, and thus for point-line distance.

Two vectors $\mathbf{a}$ and $\mathbf{b}$ share an origin. A dashed line extends along $\mathbf{b}$. A perpendicular is dropped from the tip of $\mathbf{a}$ to this line. The vector from the origin to the foot of the perpendicular is $\text{proj}_{\mathbf{b}}\mathbf{a}$. The vector from the foot of the perpendicular to the tip of $\mathbf{a}$ is $\text{rej}_{\mathbf{b}}\mathbf{a}$.

1.1.1.7 Vector Rotation: Spinning Around

Definition 1.1.8 (2D Vector Rotation)**.** To rotate a vector $\mathbf{v} = (x, y)$ counter-clockwise (CCW) by an angle θ around the origin, the new vector $\mathbf{v}' = (x', y')$ is given by:

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

This can be represented by multiplication with a rotation matrix:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Intuition: This is like taking a vector and spinning it around its tail (if tail is at origin) by a certain angle. Special cases are very handy:

- **Rotate 90° CCW**: $(x, y) \to (-y, x)$. (Since $\cos(90°) = 0, \sin(90°) = 1$)
- **Rotate 90° CW**: $(x, y) \to (y, -x)$. (Since $\cos(-90°) = 0, \sin(-90°) = -1$)

These 90-degree rotations are super fast as they only involve swapping coordinates and changing a sign, no trig functions needed! Using complex numbers: If $v = x + iy$, then $v' = v \cdot (\cos\theta + i\sin\theta) = v \cdot e^{i\theta}$.

A vector $\mathbf{v} = (x, y)$ shown. An arc indicates rotation by θ CCW to a new vector $\mathbf{v}' = (x', y')$. Formulas for x', y' are displayed. A small inset shows the special case: $\mathbf{v}$ rotated 90° CCW to $(-y, x)$.

Tips

Quick 90-Degree Rotations: Remember $(x, y) \xrightarrow{90°CCW} (-y, x)$ and $(x, y) \xrightarrow{90°CW} (y, -x)$. These are invaluable for quickly finding perpendicular vectors. For example, if you have a line segment represented by vector $\mathbf{d}$, then $\mathbf{d}_{\perp CCW}$ is a normal vector to the line.

1.1.2 Lines, Segments, and Rays: Paths and Boundaries

Points and vectors are fundamental, but we often care about collections of points forming lines, segments, or rays.

1.1.2.1 Representations: Describing Infinite and Finite Paths

There are several common ways to represent these linear objects.

Definition 1.1.9 (Line, Segment, Ray Representations)**.**

- **Two Points**: A line can be uniquely defined by two distinct points P_1, P_2 that lie on it. A segment is defined by its two endpoints P_1, P_2. A ray can be defined by a starting point P_1 and another point P_2 through which it passes.
- **Point and Direction Vector**: A line can be defined by a point P_0 on the line and a non-zero direction vector $\mathbf{d}$ parallel to the line. A ray is defined by its start point P_0 and a direction vector $\mathbf{d}$.

- **Implicit Form (for lines)**: A line in 2D can be represented by the equation $ax+by+c=0$, where a, b, c are constants. The vector (a, b) is a **normal vector** (perpendicular) to the line. This form is unique up to a scaling factor.

Intuition: Choosing a representation depends on the task:

- **Two Points**: Natural for segments. Often how lines/segments are given in problems.
- **Point and Vector**: Great for parametric forms (Section 1.1.2.2) and understanding direction.
- **Implicit Form ($ax + by + c = 0$)**: Useful for checking if a point lies on a line (plug in coordinates), finding distance from a point to a line, and finding intersection of two lines (solve system of equations). If you have two points $P_1(x_1, y_1), P_2(x_2, y_2)$, you can find a, b, c: $a = y_2 - y_1$, $b = x_1 - x_2$, $c = -(ax_1 + by_1)$. (Note: this c is $-(ax_1 + by_1)$, so $ax_1 + by_1 + c = 0$. Alternatively $c = x_2y_1 - x_1y_2$).

Diagram showing: 1. A line L_1 passing through points P_1, P_2. The segment P_1P_2 is highlighted. A ray starting at P_1 and going through P_2 is also shown. 2. A line L_2 defined by point P_0 and direction vector $\mathbf{d}$. 3. A line L_3 with its equation $ax + by + c = 0$. A normal vector $\mathbf{n} = (a, b)$ is shown perpendicular to L_3.

1.1.2.2 Parametric Form: Tracing the Path

Definition 1.1.10 (Parametric Form)**.** Given two distinct points P_1 and P_2, any point $P(t)$ on the line passing through them can be expressed as:

$$P(t) = P_1 + t(P_2 - P_1) = (1 - t)P_1 + tP_2$$

where t is a real-valued parameter.

- **Line**: $t \in (-\infty, \infty)$.
- **Segment** $[P_1, P_2]$: $t \in [0, 1]$. $P(0) = P_1$, $P(1) = P_2$. Values of t between 0 and 1 give points on the segment.
- **Ray starting at P_1 passing through P_2**: $t \in [0, \infty)$.
- **Ray starting at P_2 passing through P_1**: $t \in (-\infty, 1]$ (or equivalently $P_2 + s(P_1 - P_2)$ for $s \in [0, \infty)$).

If using point P_0 and direction vector $\mathbf{d}$: $P(t) = P_0 + t\mathbf{d}$. For a segment from P_0 with length L in direction $\mathbf{d}$ (assuming $\mathbf{d}$ is unit vector): $t \in [0, L]$. (Actually, $P(t) = P_0 + t \cdot \mathbf{d}$ means t scales $\mathbf{d}$. If $\mathbf{d}$ is not unit, $P(1)$ is $P_0 + \mathbf{d}$. For segment of length L using unit vector $\mathbf{u}$, $P(t) = P_0 + t\mathbf{u}$, $t \in [0, L]$. If $\mathbf{d}$ is $P_2 - P_1$, then $t \in [0, 1]$ covers segment P_1P_2.)

Intuition: Think of t as "time".

- For a segment P_1P_2: At $t = 0$, you are at P_1. At $t = 1$, you are at P_2. For $0 < t < 1$, you are somewhere in between. If $t < 0$ or $t > 1$, you are on the line but outside the segment.
- This form is excellent for finding intersection points (solve for t) or checking if a point lies on a segment/ray.

Line through P_1, P_2. Point $P(0) = P_1$, $P(0.5)$ (midpoint), $P(1) = P_2$. Also show $P(-0.5)$ and $P(1.5)$ on the extended line. Labels for segment $(0 \le t \le 1)$ and ray $(t \ge 0)$ parts.

1.1.2.3 Properties: Slope and Intercepts (Mainly for Lines)

Definition 1.1.11 (Slope, Intercepts)**.** For a non-vertical line:

- **Slope (m)**: Measures the steepness. Given two points (x_1, y_1) and (x_2, y_2) on the line with $x_1 \neq x_2$, $m = \frac{y_2 - y_1}{x_2 - x_1}$.
- **Y-intercept (c or b)**: The y-coordinate where the line crosses the y-axis. The line equation can be $y = mx + c$.
- **X-intercept**: The x-coordinate where the line crosses the x-axis.

For a line $ax + by + c = 0$:

- If $b \neq 0$, slope $m = -a/b$. Y-intercept is $-c/b$.
- If $b = 0$ (so $ax + c = 0$, $a \neq 0$): Vertical line $x = -c/a$. Slope is undefined.
- If $a = 0$ (so $by + c = 0$, $b \neq 0$): Horizontal line $y = -c/b$. Slope is 0.

Intuition: Slope-intercept form $y = mx + c$ is familiar but has issues with vertical lines (infinite slope). The $ax + by + c = 0$ form is more general. In competitive programming, we often work with vectors or pairs of points, and calculate slope only if needed, being careful about division by zero for vertical lines. Two lines are parallel if their slopes are equal or both are vertical. They are perpendicular if $m_1 \cdot m_2 = -1$ (unless one is horizontal and other vertical). Using dot product of direction vectors or normal vectors is more robust for checking parallelism/perpendicularity. (Parallel: cross product of direction vectors is 0. Perpendicular: dot product of direction vectors is 0).

Gotcha 1.1.1. **Vertical Lines and Slope**: The concept of slope breaks down for vertical lines ($x_1 = x_2$). Always handle this case separately if your algorithm relies on slope calculation. Using vector directions (e.g., $(0, \Delta y)$ for vertical) or the $ax + by + c = 0$ form avoids this issue.

1.1.3 Distance Formulas: Measuring Separation

Calculating distances between geometric objects is a frequent task.

Definition 1.1.12 (Point-Point Distance)**.** The distance between two points $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ is the norm of the vector $\vec{P_1P_2}$:

$$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The squared distance is $(x_2 - x_1)^2 + (y_2 - y_1)^2$.

Intuition: This is just the Pythagorean theorem. As mentioned in Section 1.1.1.5, prefer comparing squared distances to avoid `sqrt`.

Definition 1.1.13 (Point-Line Distance)**.** The distance from a point $P_0(x_0, y_0)$ to a line L is the length of the perpendicular segment from P_0 to L.

- If line L passes through points A and B (where $A \neq B$):

$$d(P_0, L) = \frac{|\vec{AP_0} \times \vec{AB}|}{|\vec{AB}|} = \frac{|(P_0 - A) \times (B - A)|}{|B - A|}$$

The numerator is the magnitude of the 2D cross product, which is the area of the parallelogram formed by $\vec{AP_0}$ and $\vec{AB}$. Dividing by the base length $|\vec{AB}|$ gives the height.

- If line L is given by $ax + by + c = 0$:

$$d(P_0, L) = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

Intuition: The cross product method is very elegant: area of parallelogram divided by base length equals height. This is often preferred in code with a Point struct. The implicit form formula is also direct. The term $ax_0 + by_0 + c$ tells you something about which side of the line P_0 is on (if normal (a, b) is consistently chosen) and its magnitude is proportional to distance. $\sqrt{a^2 + b^2}$ is the magnitude of the normal vector (a, b).

Line L (passing through A, B). Point P_0 off the line. Perpendicular from P_0 to L meets L at Q. Length P_0Q is the distance. Vectors $\vec{AP_0}$ and $\vec{AB}$ are shown. The parallelogram interpretation is shown with its area being the numerator of the distance formula.

Mathematical Insight: The point-line distance is also the magnitude of the rejection of vector $\vec{AP_0}$ from vector $\vec{AB}$ (i.e., $|\text{rej}_{\vec{AB}}\vec{AP_0}| = |\vec{AP_0} - \text{proj}_{\vec{AB}}\vec{AP_0}|$). The closest point Q on the line AB to P_0 is $A + \text{proj}_{\vec{AB}}\vec{AP_0}$. This is the foot of the perpendicular.

Definition 1.1.14 (Point-Segment Distance)**.** The distance from a point P_0 to a segment $S = [A, B]$ is the shortest distance from P_0 to any point on S. Let L be the infinite line containing segment AB.

1. Calculate $t = \frac{\vec{AP_0} \cdot \vec{AB}}{|\vec{AB}|^2} = \frac{(P_0 - A) \cdot (B - A)}{|B - A|^2}$. This t determines where the projection Q of P_0 onto line L lies relative to segment AB (where $Q = A + t(B - A)$).
2. If $0 \leq t \leq 1$: The projection Q lies on segment AB. The distance $d(P_0, S)$ is the point-line distance $d(P_0, L)$.
3. If $t < 0$: The projection Q lies on line L outside segment AB, on the side of A. The closest point on S to P_0 is A. So $d(P_0, S) = d(P_0, A)$.
4. If $t > 1$: The projection Q lies on line L outside segment AB, on the side of B. The closest point on S to P_0 is B. So $d(P_0, S) = d(P_0, B)$.

If $A = B$ (segment is a point), distance is $d(P_0, A)$. This case implies $|\vec{AB}|^2 = 0$, so t calculation needs care.

Intuition: Imagine "dropping" the point P_0 perpendicularly onto the line containing segment AB. If it lands *on* the segment (projection parameter $t \in [0, 1]$), that's your shortest distance (point-to-line distance). If it lands *off* the segment (e.g., $t < 0$ means it's "before" A, $t > 1$ means it's "after" B), then the closest point on the segment is simply the nearer endpoint (A or B). The dot product helps determine t: $(P_0 - A) \cdot (B - A)$ tells us how much $P_0 - A$ "goes along" $B - A$. Normalizing by $|B - A|^2$ gives the parametric position t.

Segment AB. Case 1: Point P_a. Projection Q_a of P_a onto line AB lies on segment AB. Parameter $t_a \in [0,1]$. Distance is P_aQ_a. Case 2: Point P_b. Projection Q_b lies on line AB but outside segment, past A. Parameter $t_b < 0$. Distance is P_bA. Case 3: Point P_c. Projection Q_c lies on line AB but outside segment, past B. Parameter $t_c > 1$. Distance is P_cB.

Warning

Degenerate Segment: If the segment AB is actually a point (i.e., $A = B$), then $|\vec{AB}|^2 = 0$. The formula for t would involve division by zero. Handle this as a base case: the point-segment distance is simply the distance from P_0 to A (or B).

1.1.4 Orientation Predicate: Which Way Did They Turn?

The orientation predicate is one of the most fundamental tools in computational geometry. It tells us the relative orientation of an ordered triplet of points.

Definition 1.1.15 (Orientation)**.** Given an ordered triplet of points $P_1(x_1, y_1)$, $P_2(x_2, y_2)$, and $P_3(x_3, y_3)$, their **orientation** describes the nature of the turn made when traversing from P_1 to P_2 and then to P_3. It can be:

- **Counter-Clockwise (CCW)** or **Left Turn**: If P_3 is to the left of the directed line $\vec{P_1P_2}$.
- **Clockwise (CW)** or **Right Turn**: If P_3 is to the right of the directed line $\vec{P_1P_2}$.
- **Collinear**: If P_3 lies on the infinite line defined by P_1 and P_2.

The orientation is determined by the sign of the 2D cross product of vectors $\vec{P_1P_2}$ and $\vec{P_1P_3}$:

$$\begin{aligned} \text{Let} \quad \vec{v_1} &= P_2 - P_1 = (x_2 - x_1,\, y_2 - y_1) \\ \text{Let} \quad \vec{v_2} &= P_3 - P_1 = (x_3 - x_1,\, y_3 - y_1) \end{aligned}$$

$$\begin{aligned} \text{The orientation value is:} \quad \text{val} &= \vec{v_1} \times \vec{v_2} \\ &= (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) \end{aligned}$$

- $\text{val} > 0 \implies$ CCW (Left turn from $\vec{P_1P_2}$ to $\vec{P_1P_3}$)
- $\text{val} < 0 \implies$ CW (Right turn from $\vec{P_1P_2}$ to $\vec{P_1P_3}$)
- $\text{val} = 0 \implies$ Collinear (P_1, P_2, P_3 lie on the same line)

(Note: If any two points are identical, e.g. $P_1 = P_2$, the cross product will be 0, correctly indicating collinearity.)

Intuition: Imagine you're an ant walking from P_1 towards P_2. When you are at P_1 and looking at P_2, on which side is P_3? If P_3 is to your left, it's a CCW orientation. If to your right, it's CW. If P_3 is directly in front or behind you (on the line P_1P_2), it's collinear. This value is also twice the signed area of the triangle $\triangle P_1P_2P_3$. A positive area means P_1, P_2, P_3 are listed in CCW order (assuming standard Cartesian coordinates). This is the workhorse for segment intersection, convex hulls, polygon tests, and much more.

- **Panel 1: Counter-Clockwise (CCW) / Left Turn**

  Points P_1, P_2, P_3 form a left turn.
  P_3 is to the **left** of the directed line $\vec{P_1P_2}$.
  Cross product: $(P_2 - P_1) \times (P_3 - P_1) > 0$
  Diagram: Arrow path $P_1 \to P_2 \to P_3$ shows a left turn at P_2.

- **Panel 2: Clockwise (CW) / Right Turn**

  Points P_1, P_2, P_3 form a right turn.
  P_3 is to the **right** of the directed line $\vec{P_1P_2}$.
  Cross product: $(P_2 - P_1) \times (P_3 - P_1) < 0$
  Diagram: Arrow path $P_1 \to P_2 \to P_3$ shows a right turn at P_2.

- **Panel 3: Collinear**

  Points P_1, P_2, P_3 are collinear.
  Cross product: $(P_2 - P_1) \times (P_3 - P_1) = 0$
  Diagram: Vectors $\vec{P_1P_2}$ and $\vec{P_1P_3}$ are aligned.
  Sub-cases: P_3 on segment P_1P_2; P_1 between P_2 and P_3; P_2 between P_1 and P_3; $P_1 = P_3$; etc.

Mathematical Insight: The orientation value

$$(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$$

can also be written as the determinant of a matrix:

$$\text{OrientationValue} = \det \begin{pmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{pmatrix}$$

This is also equivalent to twice the signed area of the triangle $\triangle P_1P_2P_3$. Another common determinant form for this signed area (scaled by 2) is:

$$\text{TwiceSignedArea}(P_1, P_2, P_3) = \det \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$

These formulations are mathematically equivalent.

Tips

Use Integer Arithmetic for Orientation: If point coordinates are integers, the cross product calculation involves only additions, subtractions, and multiplications. The result will be an exact integer, avoiding all floating-point precision issues that can plague geometric algorithms.

Tip: Use `long long` for the calculation if intermediate products ($x_i y_j$) might overflow standard `int`.

For coordinates up to $C_{\max}$:

- $x_2 - x_1$ can be $2C_{\max}$.
- The product $(x_2 - x_1)(y_3 - y_1)$ can be up to $(2C_{\max})^2 = 4C_{\max}^2$.
- The difference can be up to $2 \times ((C_{\max} - (-C_{\max})) \cdot (C_{\max} - (-C_{\max}))) = 8C_{\max}^2$ in magnitude.
- More practically, a single term $(X_A Y_B)$ is $C_{\max}^2$.
- The full expression $X_A Y_B - X_B Y_A$ can be $2C_{\max}^2$.

If $C_{\max} = 10^9$, then $2 \cdot 10^{18}$ fits in `long long`. If $C_{\max} = 10^5$, $2 \cdot 10^{10}$ also fits.

Warning

Collinear is Not Enough Detail: When orientation is 0 (collinear), you often need more information for specific algorithms. For example, for three distinct points P_1, P_2, P_3:

- Does P_3 lie on the segment P_1P_2? (Is $P_1P_3 + P_3P_2 = P_1P_2$ using distances? Or check x, y ranges.)
- Is P_1 between P_2 and P_3? Is P_2 between P_1 and P_3?

These sub-cases are typically handled by checking bounding boxes or dot products on the vectors. See Segment Intersection (Section 1.1.5) and the Point on Segment test (Algorithm 9).

1.1.5 Segment Intersection Predicate: Do They Cross?

Determining if two line segments intersect is another cornerstone predicate, built upon orientation tests.

Definition 1.1.16 (Segment Intersection)**.** Two line segments $S_1 = [P_1, P_2]$ and $S_2 = [P_3, P_4]$ **intersect** if they share at least one common point. The intersection can be:

- **Proper Intersection**: The intersection point is in the interior of both segments (i.e., not an endpoint of either).
- **Improper Intersection**: The intersection point is an endpoint of at least one segment. This includes cases where an endpoint of one segment lies on the other segment, or segments overlap along a line (collinear intersection).

Unless specified, "intersection" usually means any common point (proper or improper).

Intuition: The primary way to check for intersection uses orientation tests: **General Case (Non-Collinear)**: Segments $[P_1, P_2]$ and $[P_3, P_4]$ intersect if and only if P_1 and P_2 lie on opposite sides of the infinite line containing P_3P_4, AND P_3 and P_4 lie on opposite sides of the infinite line containing P_1P_2. "Opposite sides" means:

1. orientation(P_3, P_4, P_1) and orientation(P_3, P_4, P_2) have different non-zero signs. **AND**
2. orientation(P_1, P_2, P_3) and orientation(P_1, P_2, P_4) have different non-zero signs.

This specific condition tests for *proper* intersection.

Handling Endpoint Touching and Collinearity: If one of the orientation tests yields zero (e.g., orientation$(P_3, P_4, P_1) = 0$), it means P_1 is collinear with P_3 and P_4. For an intersection to occur in this case, P_1 must lie *on the segment* $[P_3, P_4]$. This "point on segment" check is crucial for improper intersections. If all four points are collinear (all four orientation tests $o(P_1, P_2, P_3)$, $o(P_1, P_2, P_4)$, $o(P_3, P_4, P_1)$, $o(P_3, P_4, P_2)$ are zero), then the segments intersect if and only if their 1D projections onto an axis overlap. This is typically checked by seeing if an endpoint of one segment lies on the other.

Panel 1:
Two segments $S_1 = [P_1, P_2]$ and $S_2 = [P_3, P_4]$ cross at an interior point.
Orientations $o(P_1, P_2, P_3)$ and $o(P_1, P_2, P_4)$ differ;
$o(P_3, P_4, P_1)$ and $o(P_3, P_4, P_2)$ differ.
Proper Intersection.
Panel 2:
S_1 and S_2 meet at P_3, which is an endpoint of S_2 and lies on S_1 (but not as an endpoint of S_1).
$o(P_1, P_2, P_3) = 0$; P_3 is on segment $[P_1, P_2]$.
Improper Intersection (Endpoint on Segment).
Panel 3:
S_1 and S_2 are collinear and overlap.
All four main orientations are zero.
Improper Intersection (Collinear Overlap).
Panel 4:
S_1 and S_2 do not intersect (e.g., they are parallel and separate, or skew and far apart).
No Intersection.
Panel 5:
S_1 and S_2 are collinear but do not overlap (e.g., P_1-P_2 ... P_3-P_4).
No Intersection (Collinear, Disjoint).

Theorem 1.1.3 (Segment Intersection Criterion using Orientations)**.** *Let $S_1 = [P_1, P_2]$ and $S_2 = [P_3, P_4]$.*

Let

$$o_1 = orientation(P_1, P_2, P_3), \quad o_2 = orientation(P_1, P_2, P_4), \quad o_3 = orientation(P_3, P_4, P_1), \quad o_4 = orientation($$

The segments S_1 and S_2 intersect if and only if:

1. ***General Case (Proper Intersection):***
*$(o_1 \neq 0,\ o_2 \neq 0,\ o_3 \neq 0,\ o_4 \neq 0)$ **AND** $(o_1 \neq o_2)$ **AND** $(o_3 \neq o_4)$.*
This means $o_1 \cdot o_2 < 0$ and $o_3 \cdot o_4 < 0$.

2. ***Special Cases (Improper/Collinear Intersection):***

   - *$o_1 = 0$ and P_3 lies on segment $[P_1, P_2]$ (see Algorithm 9).*

- $o_2 = 0$ *and* P_4 *lies on segment* $[P_1, P_2]$.
- $o_3 = 0$ *and* P_1 *lies on segment* $[P_3, P_4]$.
- $o_4 = 0$ *and* P_2 *lies on segment* $[P_3, P_4]$.

If any of these conditions (General or Special) is met, the segments intersect.

A full algorithm is detailed in Algorithm 10. Note that the conditions in item 2 cover all collinear overlap cases as well as endpoint-touching cases. For instance, if P_1, P_2, P_3, P_4 *are collinear and* S_1, S_2 *overlap, then* $o_1 = o_2 = o_3 = o_4 = 0$. *Then, for example,* P_3 *must lie on* $[P_1, P_2]$ *(if* S_1 *contains* P_3*), satisfying one of the conditions.*

Gotcha 1.1.2. **Distinguishing Intersection Types**:

- **Proper Intersection**: o_1, o_2, o_3, o_4 are all non-zero, AND $o_1 \neq o_2$ AND $o_3 \neq o_4$.
- **Endpoint of one on Interior of other**: e.g., $o_1 = 0$ (so P_3 is on line P_1P_2), P_3 is strictly between P_1, P_2, and o_3, o_4 are non-zero and different.
- **Shared Endpoint**: e.g., $P_1 = P_3$. Then $o_3 = 0$. If P_2, P_4 are on opposite sides of line P_1P_x (where P_x is some other point defining the line with P_1), they touch at P_1.
- **Collinear Overlap**: $o_1 = o_2 = o_3 = o_4 = 0$, and their 1D intervals overlap.

The specific definition of "intersection" needed (any touch, or only proper crossing) varies by problem. The conditions in Theorem 1.1.3 detect *any* intersection.

Insight

A quick pre-check: if the axis-aligned bounding boxes (AABBs) of the two segments do not overlap, they cannot intersect.
The AABB of segment $[(x_a, y_a), (x_b, y_b)]$ is the rectangle defined by $[\min(x_a, x_b), \max(x_a, x_b)]$ and $[\min(y_a, y_b), \max(y_a, y_b)]$.
Two rectangles intersect if their x-intervals overlap and their y-intervals overlap.
This is an $O(1)$ check that can quickly prune many non-intersecting pairs, especially if you are checking many pairs of segments (e.g., in a sweep-line algorithm). However, AABBs overlapping does **not** guarantee segment intersection (e.g., two non-parallel segments whose AABBs overlap but the segments themselves "miss" each other).

1.2 Canonical Algorithms: The Recipes

Now that we've defined our geometric ingredients (Section 1.1), let's look at some standard recipes for combining them. This section provides pseudocode for the most common geometric primitive operations you'll use repeatedly. We'll focus on the logic here; C++ implementations will come later (Section 1.5) or as part of specific examples.

1.2.1 Vector Operations in Pseudocode

These algorithms assume points/vectors are represented as structures with `x` and `y` members (e.g., `P.x`, `P.y`).

Algorithm 1: Dot Product of 2D Vectors $\mathbf{u}, \mathbf{v}$

Input: Vector $\mathbf{u} = (u_x, u_y)$, Vector $\mathbf{v} = (v_x, v_y)$
Output: Scalar dot product $\mathbf{u} \cdot \mathbf{v}$

return $u_x \cdot v_x + u_y \cdot v_y$

Complexity Analysis

$O(1)$ time. Two multiplications and one addition.

Algorithm 2: Cross Product of 2D Vectors $\mathbf{u}, \mathbf{v}$ (z-component)

Input: Vector $\mathbf{u} = (u_x, u_y)$, Vector $\mathbf{v} = (v_x, v_y)$
Output: Scalar cross product $u_x v_y - u_y v_x$

return $u_x \cdot v_y - u_y \cdot v_x$

Complexity Analysis

$O(1)$ time. Two multiplications and one subtraction.

Warning

Ensure the data type for the return value can hold $2 \cdot C_{max}^2$ if coordinates are up to C_{max} (see Section 1.1.4). Use `long long` if inputs are `int`.

Algorithm 3: Norm Squared of a 2D Vector $\mathbf{v}$

Input: Vector $\mathbf{v} = (v_x, v_y)$
Output: Scalar norm squared $|\mathbf{v}|^2$

return $v_x \cdot v_x + v_y \cdot v_y$

Complexity Analysis

$O(1)$ time. Two multiplications and one addition.

Insight

Using `norm_sq` avoids computing the square root, which is slower and can introduce floating point errors if the coordinates are integers. This method is ideal for comparing distances.

Algorithm 4: Rotate 2D Vector $\mathbf{v}$ CCW by Angle θ

Input: Vector $\mathbf{v} = (v_x, v_y)$, Angle θ (in radians)
Output: Rotated vector $\mathbf{v}' = (v'_x, v'_y)$

$c \leftarrow \cos(\theta)$
$s \leftarrow \sin(\theta)$
$v'_x \leftarrow v_x \cdot c - v_y \cdot s$
$v'_y \leftarrow v_x \cdot s + v_y \cdot c$
return (v'_x, v'_y)

Complexity Analysis

$O(1)$ time, but involves trigonometric functions which are computationally more expensive than simple arithmetic.

Implementation Notes: If θ is a special angle like 90° or 180°, use direct coordinate manipulation to avoid `sin`/`cos` and potential precision loss. For 90° CCW: $v'_x \leftarrow -v_y$; $v'_y \leftarrow v_x$. For 90° CW: $v'_x \leftarrow v_y$; $v'_y \leftarrow -v_x$. For 180°: $v'_x \leftarrow -v_x$; $v'_y \leftarrow -v_y$.

Algorithm 5: Project Vector $\mathbf{a}$ onto Non-Zero Vector $\mathbf{b}$

Input: Vector $\mathbf{a}$, Vector $\mathbf{b}$ (non-zero)
Output: Vector projection of $\mathbf{a}$ onto $\mathbf{b}$

if $\mathbf{b}.x = 0$ ***and*** $\mathbf{b}.y = 0$ **then** **return** ***Error*** *or* $\mathbf{0}$ ▷ `Cannot project onto zero vector`

dot_prod ← dot_product($\mathbf{a}, \mathbf{b}$)
norm_sq_b ← norm_sq($\mathbf{b}$)
if *norm_sq_b is very close to 0 (for floats) or is 0 (for integers)* **then**
  return ***Error*** *or* $(0, 0)$ ▷ `Should be caught by initial check if exact 0`
scalar_factor ← dot_prod/norm_sq_b
projected_vector_x ← $\mathbf{b}.x$ · scalar_factor
projected_vector_y ← $\mathbf{b}.y$ · scalar_factor
return (*projected_vector_x, projected_vector_y*)

Complexity Analysis

$O(1)$ time. Involves dot product, norm squared, one division, two multiplications.

Warning

Always check if vector $\mathbf{b}$ is the zero vector before dividing by its norm squared to prevent division by zero. If $\mathbf{b}$ is zero, the projection is often considered undefined or the zero vector depending on context.

1.2.2 Distance Computations in Pseudocode

These algorithms build upon basic vector operations. We usually compute squared distances first and take `sqrt` only if the actual distance is needed.

Algorithm 6: Squared Distance from Point P_0 to Line defined by A, B

Input: Point P_0, Point A on line, Point B on line ($A \neq B$)
Output: Squared distance from P_0 to line AB

vec_AP0 $\leftarrow P_0 - A$
vec_AB $\leftarrow B - A$
norm_sq_AB $\leftarrow$ norm_sq(vec_AB)
if *norm_sq_AB is 0 (or very close for floats)* **then**
  return *norm_sq(vec_AP0)*
▷ A, B `are same point, return dist_sq to A`
cross_prod_val $\leftarrow$ cross_product_2d(vec_AP0, vec_AB)
return (*cross_prod_val* $\cdot$ *cross_prod_val*)/*norm_sq_AB*

Intuition: This comes from $d = \frac{|\vec{AP_0} \times \vec{AB}|}{|\vec{AB}|}$, so $d^2 = \frac{(\vec{AP_0} \times \vec{AB})^2}{|\vec{AB}|^2}$. Using squared distance avoids one `sqrt` call in the numerator and one in the denominator compared to calculating d directly. If you need d, take `sqrt` of the result.

Complexity Analysis

$O(1)$ time. Involves point subtractions, cross product, norm squared, one multiplication, one division.

Gotcha 1.2.1. If A and B are coincident (*norm_sq_AB* $= 0$), the line is undefined. The algorithm returns distance to point A. Problem context usually ensures $A \neq B$.

Algorithm 7: Squared Distance from Point P_0 to Segment AB

Input: Point P_0, Segment endpoint A, Segment endpoint B
Output: Squared distance from P_0 to segment AB

vec_AP0 $\leftarrow P_0 - A$
vec_AB $\leftarrow B - A$
norm_sq_AB $\leftarrow$ norm_sq(vec_AB)
if *norm_sq_AB is 0 (or very close for floats)* **then**
  return *norm_sq(vec_AP0)*
▷ `Segment is a point A, dist is to A`
dot_prod $\leftarrow$ dot_product(vec_AP0, vec_AB)
if *dot_prod* < 0 *(or dot_prod* $< -EPS$ *for floats)* **then**
  ▷ `Closest point on line is outside segment, past A`
  return *norm_sq(vec_AP0)* ▷ `Squared distance to A`
else if *dot_prod* $>$ *norm_sq_AB (or dot_prod* $>$ *norm_sq_AB* $+ EPS$ *for floats)* **then**
  ▷ `Closest point on line is outside segment, past B`
  vec_BP0 $\leftarrow P_0 - B$
  return *norm_sq(vec_BP0)* ▷ `Squared distance to B`
else
  ▷ `Closest point (projection) is on segment. Use point-line distance squared.`
  cross_prod_val $\leftarrow$ cross_product_2d(vec_AP0, vec_AB)
  return (*cross_prod_val* $\cdot$ *cross_prod_val*)/*norm_sq_AB*

Intuition: The conditions $dot_prod < 0$ and $dot_prod > norm_sq_AB$ correspond to the projection parameter $t = \text{dot\_prod}/\text{norm\_sq\_AB}$ being $t < 0$ and $t > 1$ respectively (Section 1.1.3). If $0 \le t \le 1$, the closest point on the segment AB is the projection $Q = A + t\vec{AB}$ of P_0 onto AB; thus, the distance is the perpendicular distance from P_0 to AB.

Complexity Analysis

$O(1)$ time. A few vector operations, dot/cross products, comparisons.

Implementation Notes: To find the actual closest point Q on segment AB to P_0:

- If $dot\_prod < 0$: $Q = A$.
- If $dot\_prod > norm\_sq\_AB$: $Q = B$.
- Else: $t = dot\_prod/norm\_sq\_AB$. $Q = A + t \cdot \vec{AB}$.

This is useful for Item 6.

1.2.3 Orientation Test in Pseudocode

This algorithm implements the orientation predicate defined in Definition 1.1.15.

Algorithm 8: Orientation of Ordered Triplet (P_1, P_2, P_3)

Input: Point $P_1 = (x_1, y_1)$, Point $P_2 = (x_2, y_2)$, Point $P_3 = (x_3, y_3)$
Output: Integer: 1 for CCW (Left), -1 for CW (Right), 0 for Collinear

▷ `Calculates` $(P_2 - P_1) \times (P_3 - P_1)$

val_dx1 ← $P_2.x - P_1.x$
val_dy1 ← $P_2.y - P_1.y$
val_dx2 ← $P_3.x - P_1.x$
val_dy2 ← $P_3.y - P_1.y$
cross_product_val ← val_dx1 · val_dy2 − val_dy1 · val_dx2
if *using floating point numbers and* abs(*cross_product_val*) < *EPSILON* **then**
  return *0* ▷ `Collinear within tolerance`
if *cross_product_val = 0* **then**
  return *0* ▷ `Collinear (exact)`
if *cross_product_val* > 0 **then**
  return *1* ▷ `CCW / Left Turn`
else
  return *-1* ▷ `CW / Right Turn`

Complexity Analysis

$O(1)$ time. Four subtractions, two multiplications, one subtraction.

Tips

Integer Arithmetic is King: As stressed in Section 1.1.4, if coordinates are integers, perform all calculations using `long long` to prevent overflow and maintain exactness. Avoid floats for orientation if at all possible. The `EPSILON` check for floats is notoriously tricky to get right for all geometric configurations.

```cpp
// Assuming PointLL struct with long long x, y from ssec:A.5.1
// and a cross product method: p1.cross(p2) = p1.x*p2.y - p1.y*p2.x
#include <iostream> // For PointLL definition if not separate

// struct PointLL { long long x, y; ... PointLL operator-(PointLL o) const { ... } ... };
// long long cross(PointLL a, PointLL b) { return a.x * b.y - a.y * b.x; }
// long long cross(PointLL O, PointLL A, PointLL B) {
//    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
// }

int orientation(PointLL p1, PointLL p2, PointLL p3) {
    // Equivalent to (p2-p1).cross(p3-p1)
    long long val = (p2.x - p1.x) * (p3.y - p1.y) -
                    (p2.y - p1.y) * (p3.x - p1.x);
    if (val == 0) return 0;  // Collinear
    return (val > 0) ? 1 : -1; // 1 for CCW (Left), -1 for CW (Right)
}
```

1.2.4 Segment Intersection Test in Pseudocode

This algorithm determines if two segments $[P_1, P_2]$ and $[P_3, P_4]$ intersect, covering general, collinear, and endpoint cases, as discussed in Section 1.1.5.

Algorithm 9: Helper: Point on Segment (Collinear Case)

Input: Point P_k, Segment endpoints P_i, P_j. It's **known** that P_i, P_j, P_k are collinear.
Output: Boolean: True if P_k lies on segment P_iP_j (inclusive of endpoints), False otherwise.

▷ `Check if` P_k`'s coordinates are between` P_i`'s and` P_j`'s coordinates along both axes.`

return $(P_k.x \geq \min(P_i.x, P_j.x)$ ***and*** $P_k.x \leq \max(P_i.x, P_j.x)$ ***and***
$P_k.y \geq \min(P_i.y, P_j.y)$ ***and*** $P_k.y \leq \max(P_i.y, P_j.y))$

Intuition: If we already know three points are on the same line, the middle point must have its x-coordinate between the other two x-coordinates (inclusive) AND its y-coordinate between the other two y-coordinates (inclusive). This forms a bounding box check. This function is only called when collinearity is already established by an orientation test returning 0.

Algorithm 10: Segment Intersection Test: $S_1 = [P_1, P_2]$ and $S_2 = [P_3, P_4]$

Input: Endpoints P_1, P_2 of segment S_1. Endpoints P_3, P_4 of segment S_2.
Output: Boolean: True if segments intersect, False otherwise.

o1 $\leftarrow$ orientation(P_1, P_2, P_3)
▷ `Orientation of` $P_1P_2P_3$
o2 $\leftarrow$ orientation(P_1, P_2, P_4)
▷ `Orientation of` $P_1P_2P_4$
o3 $\leftarrow$ orientation(P_3, P_4, P_1)
▷ `Orientation of` $P_3P_4P_1$
o4 $\leftarrow$ orientation(P_3, P_4, P_2)
▷ `Orientation of` $P_3P_4P_2$
▷ `General case: segments cross each other (proper intersection)`
if $o1 \neq 0$ ***and*** $o2 \neq 0$ ***and*** $o3 \neq 0$ ***and*** $o4 \neq 0$ **then**
  if *(*$o1 \neq o2$*)* ***and*** *(*$o3 \neq o4$*)* **then**
    ▷ `Signs differ for both pairs`
    return ***True***
  return ***False*** ▷ `No intersection if all non-zero but signs don't differ correctly`
▷ `Special Cases: Collinear scenarios and endpoint touching`
▷ `If` $o1 = 0$, P_3 `is collinear with` P_1, P_2. `Check if` P_3 `lies on segment` P_1P_2.
if $o1 = 0$ ***and*** *point_on_segment_collinear*(P_3, P_1, P_2) **then**
  return ***True***
▷ `If` $o2 = 0$, P_4 `is collinear with` P_1, P_2. `Check if` P_4 `lies on segment` P_1P_2.
if $o2 = 0$ ***and*** *point_on_segment_collinear*(P_4, P_1, P_2) **then**
  return ***True***
▷ `If` $o3 = 0$, P_1 `is collinear with` P_3, P_4. `Check if` P_1 `lies on segment` P_3P_4.
if $o3 = 0$ ***and*** *point_on_segment_collinear*(P_1, P_3, P_4) **then**
  return ***True***
▷ `If` $o4 = 0$, P_2 `is collinear with` P_3, P_4. `Check if` P_2 `lies on segment` P_3P_4.
if $o4 = 0$ ***and*** *point_on_segment_collinear*(P_2, P_3, P_4) **then**
  return ***True***
return ***False*** ▷ `No intersection based on above conditions`

Complexity Analysis

$O(1)$ time. It involves four orientation tests and up to four point-on-segment tests (if orientations are zero). Each of these sub-operations is $O(1)$.

Insight

The "General case" check in Algorithm 10 (where $o1, o2, o3, o4$ are all non-zero, $o1 \neq o2$, and $o3 \neq o4$) specifically detects **proper intersection**. The subsequent checks for $o_i = 0$ handle **improper intersections** (endpoint of one segment lies on the other, or collinear overlap). The algorithm as a whole detects *any* intersection.

Warning

Degenerate Segments (Points): If a segment is actually a point (e.g., $P_1 = P_2$), the `orientation` function should still behave correctly (e.g., $orientation(P1, P1, P3)$ will be 0 if P_1, P_3 are distinct, indicating collinearity). The $point_on_segment_collinear(Pk, Pi, Pi)$ should correctly check if $P_k = P_i$. If $P_1 = P_2$ and $P_3 = P_4$ (two points), they intersect if $P_1 = P_3$. If $P_1 = P_2$ (segment S_1 is a point), it intersects $S_2 = [P_3, P_4]$ if P_1 lies on segment $[P_3, P_4]$. The algorithm covers this: $o1$ and $o2$ would be based on P_1, P_1, P_3 and P_1, P_1, P_4. These are not well-defined by the standard P_1, P_2, P_k orientation interpretation. A robust implementation should perhaps handle point-segments as a pre-check or ensure orientation function (P_A, P_A, P_B) gives 0. If $P_1 = P_2$, then $o1 = \text{orientation}(P_1, P_1, P_3)$ and $o2 = \text{orientation}(P_1, P_1, P_4)$. Both will be 0. Then $o3 = \text{orientation}(P_3, P_4, P_1)$ and $o4 = \text{orientation}(P_3, P_4, P_1)$, so $o3 = o4$. The general case fails. Then it falls to special cases: Is P_3 on segment P_1P_1? Yes if $P_3 = P_1$. Correct. Is P_1 on segment P_3P_4? Yes if S_1 (a point) is on S_2. Correct. The logic seems to hold.

Listing 1.1: Segment intersection in C++ (using integer PointLL and orientation from **??**)

```cpp
// Assuming PointLL struct and orientation() function are defined
// bool on_segment(PointLL pk, PointLL pi, PointLL pj) from Alg A.2.4

bool on_segment(PointLL pk, PointLL pi, PointLL pj) {
    // Assumes pk, pi, pj are collinear.
    // Check if pk is within the bounding box of pi, pj.
    return (pk.x >= std::min(pi.x, pj.x) && pk.x <= std::max(pi.x, pj.x) &&
            pk.y >= std::min(pi.y, pj.y) && pk.y <= std::max(pi.y, pj.y));
}

bool segments_intersect(PointLL p1, PointLL p2, PointLL p3, PointLL p4) {
    int o1 = orientation(p1, p2, p3);
    int o2 = orientation(p1, p2, p4);
    int o3 = orientation(p3, p4, p1);
    int o4 = orientation(p3, p4, p2);

    // General case
    if (o1 != 0 && o2 != 0 && o3 != 0 && o4 != 0) {
        return (o1 != o2) && (o3 != o4);
    }

    // Special Cases for collinearity / endpoint touching
    if (o1 == 0 && on_segment(p3, p1, p2)) return true;
    if (o2 == 0 && on_segment(p4, p1, p2)) return true;
    if (o3 == 0 && on_segment(p1, p3, p4)) return true;
    if (o4 == 0 && on_segment(p2, p3, p4)) return true;

    return false; // Doesn't fall into any of the above intersection cases
}
```

1.3 Precision & Implementation Gotchas: The Hidden Traps

Computational geometry in contests is notorious for one thing: hidden traps related to numerical precision and edge cases. Understanding these pitfalls is as important as knowing the algorithms themselves. Get this wrong, and your perfectly logical code might fail on seemingly simple test cases!

1.3.1 Floating-Point Arithmetic and Epsilon (`EPS`)

Intuition: Computers cannot represent all real numbers perfectly. Floating-point numbers (`float`, `double`, `long double` in C++) are approximations. This means that calculations that should yield exact results (like `0.1 + 0.2` resulting in `0.3`) might produce something slightly off (e.g., `0.30000000000000004`). This is a fundamental limitation. Think of it like trying to measure a precise length with a ruler that only has markings every millimeter – you can get close, but not perfectly exact for all lengths.

Warning: Direct Equality Comparison for Floats

Never use `==` to compare floating-point numbers for equality! Because `a=(1.0/3.0) * 3.0` might not be exactly `1.0`. Instead, check if their absolute difference is within a small tolerance, called epsilon (`EPS`): `if (std::abs(a - b) < EPS)`

Definition 1.3.1 (Epsilon `EPS`). **Epsilon** (`EPS`) is a small positive constant used as a tolerance for floating-point comparisons. Typical values in competitive programming are 10^{-7} to 10^{-12} (e.g., `const double EPS = 1e-9;`).

- To check if $a \approx b$ (i.e., a is "equal" to b): `std::abs(a - b) < EPS`
- To check if $a \leq b$: `a < b + EPS` or, more robustly, `a - b < EPS`
- To check if $a < b$: `a < b - EPS` or, more robustly, `a - b < -EPS`
- To check if $a \geq b$: `a > b - EPS` or, more robustly, `a - b > -EPS`
- To check if $a > b$: `a > b + EPS` or, more robustly, `a - b > EPS`
- To check if $a \approx 0$: `std::abs(a) < EPS`

The forms like `a - b < EPS` are generally preferred because they are less susceptible to issues when `a` and `b` are very large (this relates to relative vs. absolute error, though for typical contest coordinate ranges, `a < b + EPS` usually works).

Tips

Choosing `EPS`:

- **Too small**: Might fail to identify numbers that *should* be equal but differ due to accumulated precision errors (false negatives for equality).
- **Too large**: Might incorrectly identify distinct numbers as equal (false positives for equality), potentially merging points or lines that should be separate.
- A common choice for `double` in competitive programming is `1e-9`. This often works well when input coordinates are integers and intermediate calculations don't magnify errors too much.
- If problem constraints involve very small differences or require high precision, `1e-12` or even smaller might be needed (use `long double` then). Conversely, if coordinates are small (e.g., up to ± 1000) and operations are simple, `1e-7` might suffice.
- **Problem Specifics**: The required precision can depend on the problem statement (e.g., "output to 6 decimal places" might guide your `EPS`). Sometimes, the problem setters will specify the `EPS` to use or the tolerance for checking answers.
- **Relative `EPS`**: For advanced use, a relative epsilon (`std::abs(a-b) < EPS * std::max(1.0, std::abs(a), std::abs(b))`) is more robust across different magnitudes of numbers, but this is rarely needed in typical contest settings.

Gotcha 1.3.1. **Error Accumulation**: Repeated floating-point operations can cause errors to accumulate. A small error in one step can become a large error after many calculations. For example, rotating a point many times might cause it to drift. This is a strong argument for using integer arithmetic (Section 1.1.4) whenever possible, especially for predicates like orientation.

It's often useful to have a comparison function for doubles:

Listing 1.2: Comparison function for doubles using EPS

```cpp
#include <cmath> // For std::abs
#include <algorithm> // For std::max (if using relative EPS)

const double EPS = 1e-9;

// Returns -1 if a < b, 0 if a == b, 1 if a > b
int dcmp(double a, double b) {
    if (std::abs(a - b) < EPS) {
        return 0; // a is "equal" to b
    }
    if (a < b) {
        return -1; // a is less than b
    }
    return 1; // a is greater than b
}

// Usage examples:
// if (dcmp(x, y) == 0) { /* x is approx equal to y */ }
// if (dcmp(x, y) < 0)  { /* x is approx less than y */ }
// if (dcmp(x, y) <= 0) { /* x is approx less than or equal to y */ }
```

This `dcmp` function encapsulates the epsilon logic and makes comparisons cleaner. For example, `dcmp(val, 0) > 0` means `val` is significantly positive.

Debug Checklist: Floating-Point Arithmetic

- Am I using `==` with floats? (Change to `std::abs(a-b) < EPS` or `dcmp`).
- Is my `EPS` value appropriate for the problem's scale and precision requirements?
- Could errors be accumulating over many operations? (Consider integer arithmetic if possible).
- Am I dividing by a float that could be very close to zero? (Check `std::abs(denominator) < EPS` first).
- Are trigonometric functions (`sin`, `cos`, `tan`, `acos`, `asin`, `atan`) behaving as expected? (`acos(1.000000001)` is NaN!). Ensure arguments are in valid ranges, e.g., for `acos`, clamp argument to `[-1.0, 1.0]`.

1.3.2 Integer Overflow: When Numbers Get Too Big

Intuition: Integer types like `int` and `long long` have a limited range of values they can store. For a signed 32-bit `int`, this is roughly $\pm 2 \cdot 10^9$. For a signed 64-bit `long long`, it's roughly $\pm 9 \cdot 10^{18}$. If a calculation produces a result outside this range, it "wraps around" or overflows, leading to incorrect (and often wildly different) values without any warning from the compiler or runtime!

Warning: Overflow in Geometric Calculations

Geometric calculations, especially those involving products of coordinates, are prime candidates for overflow:

- **Cross Product**: The formula $(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$ involves products of differences. If coordinates x_i, y_i are up to C_{max} (e.g., 10^5), then $x_2 - x_1$ can be $2 \cdot C_{max}$. A term like $(x_2 - x_1)(y_3 - y_1)$ can be up to $(2C_{max}) \cdot (2C_{max}) = 4C_{max}^2$. If $C_{max} = 10^5$, this is $4 \cdot (10^5)^2 = 4 \cdot 10^{10}$. This will overflow a 32-bit `int` (max $\approx 2 \cdot 10^9$) but fits in a 64-bit `long long` (max $\approx 9 \cdot 10^{18}$). The full cross product expression can be up to $2 \cdot ((C_{max} - (-C_{max})) \cdot (C_{max} - (-C_{max}))) = 8C_{max}^2$ in magnitude, although a more direct $X_A Y_B - X_C Y_D$ with coordinates up to C_{max} results in differences up to $2C_{max}^2$.
- **Dot Product**: $x_A x_B + y_A y_B$. Similar issue, each term can be C_{max}^2. Sum can be $2C_{max}^2$.
- **Squared Norm/Distance**: $x^2 + y^2$. If x is C_{max}, x^2 is C_{max}^2. Sum can be $2C_{max}^2$. For example, distance between $(-C_{max}, -C_{max})$ and (C_{max}, C_{max}) is $\sqrt{(2C_{max})^2 + (2C_{max})^2}$, squared distance is $8C_{max}^2$.

Tips

Default to `long long` for Coordinates or Calculations:

- If problem constraints allow coordinates up to $10^4 - 10^5$ or more, it's safest to use `long long` for storing coordinates in your `Point` struct, or at least for intermediate calculations in cross products, dot products, and squared distances.
- Even if coordinates are small enough for `int` (e.g., ± 1000), C_{max}^2 is 10^6, $2C_{max}^2$ is $2 \cdot 10^6$, which fits in `int`. But if $C_{max} \approx 40000$, $C_{max}^2 \approx 1.6 \cdot 10^9$, $2C_{max}^2 \approx 3.2 \cdot 10^9$, which overflows `int`. So, a boundary exists around $C_{max} \approx 30000 - 40000$.
- If intermediate products are cast to `long long` before multiplication, e.g., `(long long)dx1 * dy2`, this helps prevent overflow of the product itself.

Using `long long` consistently for geometric calculations involving products is a good habit in competitive programming. The performance difference is usually negligible.

Mathematical Insight: Maximum coordinate value C_{max}.

- A single coordinate difference, e.g., $dx = x_2 - x_1$: Range $\approx \pm 2C_{max}$.
- Product of two differences, e.g., $dx \cdot dy$: Range $\approx \pm(2C_{max})^2 = \pm 4C_{max}^2$.
- Cross product value: Range $\approx \pm 2 \cdot (4C_{max}^2) = \pm 8C_{max}^2$ is a loose upper bound; more tightly, it's $\approx \pm 2C_{max,diff}^2$ where $C_{max,diff}$ is max coordinate difference. If coordinates are X_i, Y_i, a product X_1Y_2 is C_{max}^2. The difference $X_1Y_2 - X_2Y_1$ could be $2C_{max}^2$.
- A `long long` typically holds up to $\approx 9 \times 10^{18}$. So, $2C_{max}^2 \leq 9 \times 10^{18} \implies C_{max}^2 \leq 4.5 \times 10^{18} \implies C_{max} \leq \sqrt{4.5 \times 10^{18}} \approx 2.1 \times 10^9$. If coordinates themselves can be this large (e.g., problem states 10^9), then `long long` is essential for coordinates. Products would need `__int128` or careful handling. Most contest problems keep coordinates within a range where `long long` suffices for products.

Debug Checklist: Integer Overflow

- What are the maximum possible coordinate values given by problem constraints?
- Am I performing multiplications of coordinates or coordinate differences (e.g., in cross product, dot product, squared distance)?
- Are the intermediate products and final results guaranteed to fit within the chosen integer type (`int` or `long long`)?
- Have I explicitly cast operands to `long long` *before* multiplication if they are stored as `int`s? E.g., `(long long)val_dx1 * val_dy2`
- Test with maximum and minimum coordinate values to check for overflow. E.g., points like $(C_{max}, C_{max}), (C_{max}, -C_{max}), (-C_{max}, -C_{max})$.

1.3.3 `atan2(y,x)`: Uses and Pitfalls

The `atan2(y, x)` function (from `<cmath>` or `<math.h>`) is a very useful tool for finding the angle of a vector (x, y) or point (x, y) relative to the positive x-axis.

Definition 1.3.2 (`atan2(y,x)`). `atan2(y, x)` computes the principal value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the result. It returns an angle in radians, typically in the range $(-\pi, \pi]$ (i.e., $-180° <$ angle $\leq +180°$).

- $(x > 0, y = 0) \implies 0$
- $(x > 0, y > 0) \implies (0, \pi/2)$ (Quadrant I)
- $(x = 0, y > 0) \implies \pi/2$
- $(x < 0, y > 0) \implies (\pi/2, \pi)$ (Quadrant II)
- $(x < 0, y = 0) \implies \pi$
- $(x < 0, y < 0) \implies (-\pi, -\pi/2)$ (Quadrant III)
- $(x = 0, y < 0) \implies -\pi/2$
- $(x > 0, y < 0) \implies (-\pi/2, 0)$ (Quadrant IV)
- $(x = 0, y = 0) \implies 0$ (behavior might vary; usually 0).

Intuition: Why `atan2(y,x)` instead of just `atan(y/x)`? `atan(ratio)` only returns angles in $(-\pi/2, \pi/2)$ (Quadrants I and IV). It loses information about the signs of x and y individually. For example, `atan(1/1)` and `atan(-1/-1)` both compute `atan(1)` which is $\pi/4$, but $(1, 1)$ is in Q1 and $(-1, -1)$ is in Q3. `atan2` correctly distinguishes these, returning $\pi/4$ for $(1, 1)$ and (typically) $-3\pi/4$ or $5\pi/4$ (depending on range, but standard C++ is $(-\pi, \pi]$ so $-3\pi/4$) for $(-1, -1)$. It's essential for angular sorts (Section 1.4.2) and any problem requiring true polar angles.

A Cartesian plane with points in each quadrant: $P_1(2, 2)$ in Q1, angle $\pi/4$. $P_2(-2, 2)$ in Q2, angle $3\pi/4$. $P_3(-2, -2)$ in Q3, angle $-3\pi/4$. $P_4(2, -2)$ in Q4, angle $-\pi/4$. Point on positive x-axis $P_x(2, 0)$, angle 0. Point on negative x-axis $P_{nx}(-2, 0)$, angle π. Point on positive y-axis $P_y(0, 2)$, angle $\pi/2$. Point on negative y-axis $P_{ny}(0, -2)$, angle $-\pi/2$. All angles are labeled, showing the $(-\pi, \pi]$ range.

Warning: Precision with `atan2`

- **Floating-Point Output**: `atan2` returns a `double`. All the usual floating-point precision issues apply (Section 1.3.1). Comparing angles obtained from `atan2` requires `EPS`.
- **Boundary Cases**: Angles close to π and $-\pi$ can be tricky. For example, a point just above the negative x-axis might have an angle slightly less than π (e.g., 3.14159), while a point just below might have an angle slightly more than $-\pi$ (e.g., -3.14159). When sorting, this jump can cause issues if not handled, though standard sort with `EPS` comparisons should work.
- **Zero Vector**: `atan2(0,0)` is often 0. If you need to handle the origin point specially in an angular sort, ensure its behavior is what you expect.

Tips

When to Avoid `atan2`: While `atan2` is powerful, it can be slow and prone to precision issues.

- **Orientation Tests**: Use cross products (Section 1.1.4). They are exact with integers and faster.
- **Angle Comparison**: For sorting points around a pivot, a custom comparator using cross products is often more robust if coordinates are integers (Section 1.4.2). This avoids floats entirely.
- Only use `atan2` when you *absolutely need* the actual angle value (e.g., for physics formulas, specific geometric constructions requiring angles, or when problem asks for an angle).

Gotcha 1.3.2. **Angle Range Normalization**: `atan2` returns angles in $(-\pi, \pi]$. Sometimes you need angles in $[0, 2\pi)$. To convert an angle `ang` from $(-\pi, \pi]$ to $[0, 2\pi)$:

Listing 1.3: Angle Range Conversion

```
if (ang < 0) {
    ang += 2 * PI; // Assuming PI is defined, e.g., std::acos(-1.0)
}
// Now ang is in [0, 2*PI) (or very close for ang near 0 or 2*PI)
```

1.3.4 Collinear Points and Degenerate Cases

Intuition: In geometry, "degenerate" cases are special configurations that might break the general logic of an algorithm. Common examples include:

- Three or more points lying on the same line (collinear points).
- Multiple points at the exact same location (coincident points).
- Segments of zero length (endpoints are coincident).
- Polygons with zero area or self-intersections (if they're supposed to be simple).

These aren't necessarily "errors" but often require specific handling.

Warning: Algorithms Assume General Position

Many geometric algorithms are first described assuming "general position," meaning no three points are collinear, no two points are coincident, etc. In competitive programming, test cases **will** include these degenerate configurations. Your code must handle them robustly!

Insight

Collinearity is a frequent source of bugs.

- **Orientation Test**: When `orientation(P1, P2, P3)` (Algorithm 8) returns 0, points are collinear. What does this imply for your current algorithm?
  - For segment intersection (Algorithm 10), it means you need to check for 1D overlap.
  - For convex hull algorithms (e.g., Graham Scan, *(see **??**)*), you need a tie-breaking rule for collinear points (e.g., keep farthest, keep closest, or discard middle ones depending on variant).
  - For angular sort (Section 1.4.2), collinear points relative to pivot have the same "angle". Tie-break by distance.
- **Coincident Points**: If multiple input points can be at the same location:
  - Does it affect point counts? (e.g., "N distinct points").
  - Can it create zero-length segments or zero-area triangles/polygons?
  - Often, pre-processing to remove duplicates or handle them (e.g., by assigning them a count) can simplify later logic.

Tips

Strategies for Handling Degeneracies:

1. **Explicit Checks**: Add `if` statements to handle collinear/coincident cases separately. This is common for segment intersection.
2. **Robust Primitives**: Ensure your basic functions like orientation and point-on-segment (Algorithm 9) are exact (use integers) and correctly define behavior for coincident points.
3. **Tie-Breaking Rules**: For sorting or selection based on geometric properties (angle, distance), define clear, consistent tie-breaking rules. E.g., if angles are equal, sort by distance. If distances also equal, sort by x-coordinate, then y-coordinate (lexicographical).
4. **Symbolic Perturbation**: An advanced technique (rarely needed in contests unless specified) where coordinates are infinitesimally perturbed to remove degeneracies. This is complex to implement correctly.
5. **Problem Constraints**: Read carefully! Sometimes problems guarantee "no three points are collinear" or "all points are distinct." If not, assume degeneracies exist.

Gotcha 1.3.3. **Zero-Length Segments**: If a segment is defined by P_1, P_2 where $P_1 = P_2$.

- Distance from a point P_0 to segment P_1P_1 is just distance P_0P_1. (Your Algorithm 7 should handle this if `norm_sq_AB` is 0.)
- Intersection of segment P_1P_1 with another segment P_3P_4 occurs if point P_1 lies on segment P_3P_4. (Algorithm 10 should handle this.)
- Cross products involving vector $P_1\vec{P}_1$ (the zero vector) will be zero.

Ensure your primitives don't divide by zero or behave unexpectedly when faced with zero-length vectors derived from such segments.

Debug Checklist: Degenerate Cases

- What happens if three input points A, B, C are collinear?
  - B between A, C?
  - A between B, C?
  - A, B, C are coincident?
  - A, B coincident, C distinct?
- What if an input segment has zero length ($P_1 = P_2$)?
- What if two input segments are collinear and overlap? Collinear and touch at an endpoint? Collinear and disjoint?
- What if an input polygon has collinear vertices? Or repeated vertices?
- Test your code with hand-crafted degenerate inputs. For example, for segment intersection:
  - `[(0,0)-(2,2)]` and `[(1,1)-(3,3)]` (collinear overlap)
  - `[(0,0)-(2,2)]` and `[(2,2)-(3,3)]` (collinear touch at endpoint)
  - `[(0,0)-(2,2)]` and `[(1,0)-(0,1)]` (T-junction, endpoint on segment)
  - `[(0,0)-(0,0)]` and `[(0,0)-(1,1)]` (point segment on segment)

1.4 Classic Use-Cases & Problem Patterns

Okay, theory is great, algorithms are neat, but where does the rubber meet the road? This section is all about seeing our geometric primitives in action! We'll explore common problem patterns that appear in contests and how the tools from Section 1.1 and Section 1.2 are applied to solve them. Recognizing these patterns is a huge step towards becoming a geometry whiz. Each subsection will often be framed around a typical problem type.

1.4.1 Basic Geometric Queries: The Q&A

Many competitive programming problems boil down to answering a series of simple geometric questions. These often serve as subroutines within larger, more complex algorithms.

Point-Line/Segment Relationship

Archetype: Given a point P and a line L (or segment S), determine their spatial relationship. **Common Queries**:

- **Is point P on line L?**
  - Defined by A, B: Check if orientation$(A, B, P) = 0$ (Algorithm 8).
  - Defined by $ax + by + c = 0$: Check if $a \cdot P.x + b \cdot P.y + c = 0$ (use `EPS` for floats, exact for integers).
- **Is point P on segment $S = [A, B]$?**
  - First, check if orientation$(A, B, P) = 0$.
  - If collinear, then check if P lies within the bounding box of A, B using `point_on_segment_collinear(P, A, B)` (Algorithm 9).
- **To which side of directed line $\vec{AB}$ does point P lie?**
  - Directly given by orientation(A, B, P): > 0 is Left (CCW), < 0 is Right (CW).
- **Distance from point P to line L or segment S?**
  - Use formulas from Section 1.1.3 and algorithms from Section 1.2.2 (e.g., Algorithm 6, Algorithm 7). Remember to take $\sqrt{\cdot}$ if actual distance is needed, otherwise use squared distances for comparisons.
- **Closest point on line L (or segment S) to point P?**
  - For line L through A, B: $Q = A + \text{proj}_{\vec{AB}} \vec{AP}$ (see Section 1.1.3).
  - For segment $S = [A, B]$: Find projection parameter t (see Definition 1.1.14). If $t \in [0, 1]$, closest point is $A + t \cdot \vec{AB}$. If $t < 0$, closest point is A. If $t > 1$, closest point is B. (See also Section 1.2.2).

Problem Example 1.4.1 (Problem: "Points and Lines" - Codeforces Gym 100187B (Typical)). *Why*: This type of problem directly tests your understanding and implementation of the fundamental queries listed above. **Problem Description (General Idea)**: You are often given a set of points and a set of lines/segments. You then need to answer queries like "how many points lie on segment S_i?", "which line is closest to point P_j?", or "do segments S_a and S_b intersect?". **Solution Strategy**: Implement robust functions for each required primitive (orientation, point-on-segment, distance calculations, segment intersection). Then, iterate through the queries and apply the appropriate function. Pay close attention to integer overflow (Section 1.3.2) and, if using floats, epsilon comparisons (Section 1.3.1). Collinear and degenerate cases (Section 1.3.4) are usually heavily tested. **URL**: A specific problem like this is common in regional contests or online judge gyms. For example, Codeforces Gym 100187 (NWERC 2012 Practice) Problem B "Point Probe" asks for point-polygon relationship, which builds on these primitives. A simpler variant would just ask point-line/segment type queries. (A general link for practice: `https://codeforces.com/gyms`, search for sets with "geometry" tags).

Insight

Mastering these basic queries is crucial. More complex algorithms, like finding intersections of many segments or computing convex hulls, rely heavily on these primitives performing correctly and efficiently. If your `orientation` function has a bug, everything built on top of it will crumble!

Debug Checklist: Basic Geometric Queries

- **Orientation**: Test with CCW, CW, and various collinear cases (point between, point outside, coincident points). Use integers if possible.
- **Point on Segment**: Ensure it correctly handles endpoints and rejects points that are collinear but outside the segment.
- **Distance Calculations**: Test point-line and point-segment distances. For point-segment, verify all three cases: projection on segment, projection off towards A, projection off towards B. Check degenerate segment (point) case.
- **Floating Point**: If using floats, are `EPS` comparisons consistent? Are you handling `NaN` from `sqrt` of negative (due to precision error) or `acos` of value > 1?
- **Integer Overflow**: Are all intermediate products in `long long` if necessary?

1.4.2 Sorting Points by Angle: Sweeping Around

A common task in computational geometry is to process points in angular order around a central pivot point. This is a key step in algorithms like Graham Scan for convex hulls (*(see* **??***)*), finding tangent lines, or radial sweep algorithms.

Angular Sort

Archetype: Given a pivot point P_0 and a set of other points $\{P_1, P_2, \ldots, P_N\}$, sort these points based on the angle formed by the vector $\vec{P_0P_i}$ with a reference direction (usually the positive x-axis). **Key Challenge**: Doing this robustly and efficiently, especially with integer coordinates.

There are two main approaches:

1.4.2.1 Using `atan2(dy, dx)`

Intuition: The most straightforward way conceptually is to calculate the angle for each vector $\vec{P_0P_i}$ using `atan2(Pi.y - P0.y, Pi.x - P0.x)` (Definition 1.3.2) and then sort the points based on these angles.

Algorithm 11: Angular Sort using `atan2`

Input: Pivot point P_0, list of points $Points = \{P_1, \ldots, P_N\}$
Output: List $Points$ sorted angularly around P_0

1 **foreach** *point* P_i *in* $Points$ **do**
2   $dx \leftarrow P_i.x - P_0.x$
3   $dy \leftarrow P_i.y - P_0.y$
4   $angle_i \leftarrow \text{atan2}(dy, dx)$
5 Sort $Points$ based on $angle_i$
6 **if** *two points* P_i, P_j *have (almost) the same angle* **then**
7   Tie-break by distance to P_0: point closer to P_0 comes first
8   Use squared distance for tie-breaking to avoid `sqrt`: $\text{norm\_sq}(P_i - P_0)$ vs $\text{norm\_sq}(P_j - P_0)$

Listing 1.4: Angular sort with `atan2` C++ comparator

```
\label{code:A.4.2.atan2_sort}
\caption{Angular sort with \texttt{atan2} C++ comparator}
\begin{verbatim}
#include <vector>
#include <algorithm> // For std::sort
#include <cmath>     // For std::atan2, std::sqrt (if dist used)

// Assuming Point struct with double x, y or long long x,y
// For Point<long long>, dx/dy might need to be long double for atan2
// or just cast to double.
struct Point { /* ... members x, y ... */
    double angle_around_pivot;
    // Add original index or other data if needed
};

Point pivot; // Assume this is set

// For norm_sq, if Point has T x,y;
// T distSq(Point p1, Point p2) {
//    T dx = p1.x - p2.x; T dy = p1.y - p2.y; return dx*dx + dy*dy;
// }

bool angular_sort_cmp_atan2(const Point& a, const Point& b) {
    // Angles pre-calculated and stored in Point objects relative to 'pivot'
    // double angle_a = std::atan2(a.y - pivot.y, a.x - pivot.x);
    // double angle_b = std::atan2(b.y - pivot.y, b.x - pivot.x);
    // Assumes angle_around_pivot is already computed.

    if (std::abs(a.angle_around_pivot - b.angle_around_pivot) < EPS) { // EPS from ssec A.3.1
        // Tie-break by distance (closer first)
        // For long long coords, distSq should use long long and return long long
        return distSq(pivot, a) < distSq(pivot, b);
    }
    return a.angle_around_pivot < b.angle_around_pivot;
}

// In main logic:
// pivot = ...;
// std::vector<Point> points_to_sort = ...;
// for (Point& p : points_to_sort) {
//    p.angle_around_pivot = std::atan2(p.y - pivot.y, p.x - pivot.x);
// }
// std::sort(points_to_sort.begin(), points_to_sort.end(), angular_sort_cmp_atan2);
\end{verbatim}
```

Complexity Analysis

Time: $O(N \log N)$ for sorting, with each comparison involving `atan2` (if not precomputed) or float comparison. `atan2` itself is $O(1)$ but more expensive than integer ops. Pre-calculating angles is $O(N)$. Space: $O(N)$ or $O(1)$ depending on whether angles are stored.

Warning

Using `atan2` introduces floating-point numbers.

- **Precision**: Comparisons must use `EPS` (Section 1.3.1).
- **Range** $(-\pi, \pi]$: Be mindful of this range. Standard sort usually handles it correctly. If you need $[0, 2\pi)$, normalize angles (Gotcha 1.3.2).
- **Pivot Coincidence**: If $P_i = P_0$, then $dx = 0, dy = 0$. `atan2(0,0)` is usually 0. This point might need special handling (e.g., place it first or last, or exclude it).

1.4.2.2 Using Cross Product for Comparison

Intuition: Instead of calculating angles, we can compare two points P_A and P_B relative to the pivot P_0 using the orientation test (Definition 1.1.15). If orientation(P_0, P_A, P_B) is CCW (positive cross product of $\vec{P_0P_A}$ and $\vec{P_0P_B}$), then P_A comes before P_B in a CCW angular sort. This method is generally more robust if coordinates are integers, as it avoids floating-point arithmetic entirely.

Algorithm 12: Angular Sort Comparator using Cross Product

Input: Pivot point P_0, two points P_A, P_B to compare
Output: True if P_A comes before P_B angularly around P_0, False otherwise

1 orient_val $\leftarrow$ orientation(P_0, P_A, P_B)
▷ `orientation computes` $(P_A - P_0) \times (P_B - P_0)$
2 **if** $orient\_val = 0$ **then**
▷ `Tie-break by distance: point closer to` P_0 `comes first.`
▷ `(Or farther, if context requires, e.g., Graham Scan upper/lower chain distinction)`
3 **return** $norm\_sq(P_A - P_0) < norm\_sq(P_B - P_0)$
4 **return** $orient\_val > 0$ ▷ `Positive (CCW) means` P_A `is before` P_B

Complexity Analysis

Time: $O(N \log N)$ for sorting. Each comparison is $O(1)$ (an orientation test and possibly a norm squared calculation). This is generally faster than `atan2`-based comparisons if using integers. Space: $O(N)$ or $O(1)$.

Warning

Handling Half-Planes: The simple cross-product comparator works perfectly if all points lie in the same half-plane with respect to P_0 and a horizontal line through P_0 (e.g., all points P_i have $P_i.y \geq P_0.y$, or all $P_i.y \leq P_0.y$). If points span across the horizontal line through P_0 (i.e., some above, some below), this comparator needs adjustment because it only gives relative order for angles within a 180° sweep. For a full 360° sort:

1. Partition points into two groups: those with $P_i.y > P_0.y$ or ($P_i.y == P_0.y$ and $P_i.x > P_0.x$) (upper half-plane including positive x-axis), and the rest (lower half-plane including negative x-axis).

2. Points in the upper half-plane come before points in the lower half-plane.

3. Within each group, use the cross-product comparator (Algorithm 12).

This is essentially what the Graham Scan pivot selection (lowest-then-leftmost) achieves: it places the pivot such that all other points are in a $\leq 180°$ angular range with respect to it and the positive x-axis direction.

Gotcha 1.4.1. **Collinear Tie-Breaking is Crucial**: When orientation$(P_0, P_A, P_B) = 0$, P_A and P_B are collinear with P_0.

- For general angular sort, typically the point closer to P_0 comes first.
- For Graham Scan (*(see* **??***)*), if building a hull, you might want the *farthest* point first among collinear points, or discard all but the farthest to avoid including interior points on the hull edge. If multiple points are coincident with the pivot, they might be handled first or last.

Ensure your tie-breaking logic matches the specific requirements of your application. Using squared norm for distance comparison is standard (Section 1.1.1.5).

Problem Example 1.4.2 (Problem: "Polygon" - TopCoder SRM 144 Div1 Easy)**.** *Why*: This problem (or similar ones like constructing a simple polygon from points) requires sorting points angularly around a chosen pivot (often the lowest-then-leftmost point, as in Graham Scan). **Problem Description (General Idea)**: Given a set of points, determine if they can form a simple polygon, or construct one. A key step is to sort them angularly. **Solution Strategy**: 1. Pick a pivot point P_0. A common choice is the point with the smallest y-coordinate, breaking ties with the smallest x-coordinate. This ensures all other points lie in an angular range of $[0, \pi]$ relative to a horizontal line through P_0. 2. Sort all other points P_i using a comparator based on the cross product $(P_i - P_0) \times (P_j - P_0)$ as in Algorithm 12. Handle collinear points by distance (e.g., farther point first for constructing a "tight" boundary for some hull variants, or closer point first for a general sort). **URL**: TopCoder SRM 144 Problem "Polygon" statement: https://community.topcoder.com/stat?c=problem_statement&pm=1666&rd=4472

Implementation Notes: Robust 360-degree Cross-Product Sort Comparator: To sort points A and B around a pivot P_0 across the full 360° range without `atan2`:

Listing 1.5: Full 360 degree angular sort comparator (C++)

```cpp
// Assuming PointLL struct, pivot P0, orientation(), distSq() defined
// PointLL P0; // The pivot point, global or captured by lambda

// Helper to determine half-plane (conceptual)
// 0 for on pivot, 1 for on positive x-axis from pivot or upper half-plane,
// 2 for on negative x-axis from pivot or lower half-plane
int get_half_plane(PointLL P) {
```

```
    if (P.x == P0.x && P.y == P0.y) return 0; // Coincident with pivot
    if (P.y > P0.y || (P.y == P0.y && P.x > P0.x)) {
        return 1; // Upper half-plane or positive x-axis
    }
    return 2; // Lower half-plane or negative x-axis
}

bool robust_angular_cmp(PointLL A, PointLL B) {
    int hp_A = get_half_plane(A);
    int hp_B = get_half_plane(B);

    if (hp_A != hp_B) {
        return hp_A < hp_B; // Points in "earlier" half-planes come first
    }

    // If A or B is P0 itself, P0 comes first (or handle as per problem)
    if (hp_A == 0) return true; // A is P0, B is not (or also P0, then equal)
    if (hp_B == 0) return false; // B is P0, A is not

    // Both points are in the same half-plane (and not P0)
    long long orient_val = orientation(P0, A, B);
    if (orient_val == 0) { // Collinear
        return distSq(P0, A) < distSq(P0, B); // Closer first
    }
    return orient_val > 0; // CCW means A comes before B
}

// Usage:
// P0 = find_lowest_leftmost_point(points); // (Or any chosen pivot)
// std::sort(other_points.begin(), other_points.end(), robust_angular_cmp);
```

This comparator correctly orders points across the full circle. The `get_half_plane` function partitions points. Points on the pivot are handled first. Then points in the upper half-plane (including positive x-axis) come before points in the lower half-plane (including negative x-axis). Within each half-plane, the cross product determines order.

1.4.3 Checking Path Self-Intersection: Untangling Spaghetti

A common problem is to determine if a path, defined by a sequence of connected line segments, crosses itself. This is crucial for validating if a sequence of vertices forms a "simple" polygon boundary.

Path/Polygon Self-Intersection

Archetype: Given a path $P_0, P_1, \ldots, P_{N-1}$ (forming segments $[P_0, P_1], [P_1, P_2], \ldots, [P_{N-2}, P_{N-1}]$), determine if any two non-adjacent segments intersect. **Key Tool**: Segment Intersection Test (Section 1.1.5, Algorithm 10).

Definition 1.4.1 (Path Self-Intersection)**.** A path $P_0, \ldots, P_{N-1}$ **self-intersects** if there exist two non-adjacent segments $[P_i, P_{i+1}]$ and $[P_j, P_{j+1}]$ that intersect. "Non-adjacent" means the segments do not share an endpoint due to path ordering. That is, $i \neq j$, $i \neq j+1$, and $i+1 \neq j$. A common practical condition is $|i-j| > 1$. Adjacent segments like $[P_i, P_{i+1}]$ and $[P_{i+1}, P_{i+2}]$ share P_{i+1} by definition. This is typically not considered a "problematic" self-intersection unless stated otherwise (e.g., if P_i, P_{i+1}, P_{i+2} are collinear and the segments improperly overlap, creating a "spike" or retracing).

Intuition: Imagine drawing the path segment by segment. If your pen crosses a line it has already drawn (and not just at the point where you finished the previous segment), then it's a self-intersection. The brute-force approach is to simply check every possible pair of non-adjacent segments.

Algorithm 13: Brute-Force Path Self-Intersection Check

Input: Path: a list of N points $P_0, P_1, \ldots, P_{N-1}$
Output: Boolean: True if path self-intersects, False otherwise

if $N < 4$ **then**
  return ***False***
▷ Need at least 2 non-adjacent segments; path $P_0P_1P_2P_3$ has segments (P_0P_1, P_2P_3)
for $i \leftarrow 0$ ***to*** $N-2$ **do**
  ▷ Iterate through first segment $S_i = [P_i, P_{i+1}]$
  for $j \leftarrow i+2$ ***to*** $N-2$ **do**
    ▷ Iterate through second segment $S_j = [P_j, P_{j+1}]$
    ▷ Ensure S_j is non-adjacent to S_i: $j \neq i+1$. Here $j \geq i+2$ ensures this.
    if *segments_intersect*$(P_i, P_{i+1}, P_j, P_{j+1})$ **then**
      ▷ Use Algorithm 10
      ▷ Need to define what "intersect" means. Usually proper intersection.
      return ***True*** ▷ Self-intersection found
return ***False*** ▷ No self-intersections found

Complexity Analysis

Time: There are $O(N^2)$ pairs of segments. Each segment intersection test is $O(1)$ (using Algorithm 10). Total time complexity: $O(N^2)$. Space: $O(N)$ for storing points, or $O(1)$ if points are processed on the fly.

Gotcha 1.4.2. **Definition of "Intersection" Matters**: For a path to be "simple" (not self-intersecting), usually we care about **proper intersections** between non-adjacent segments. A proper intersection is when segments cross at a point interior to both.

- If segment S_i and S_j (non-adjacent) share an endpoint (e.g., $P_i = P_j$ or $P_i = P_{j+1}$ etc.), this means the path revisits a vertex. This is a form of self-intersection. The general `segments_intersect` from Algorithm 10 will detect this.
- If an endpoint of S_i lies on the interior of S_j (or vice-versa), this is also a self-intersection.
- If segments S_i and S_j are collinear and overlap in their interiors, this is a self-intersection.

The Algorithm 10 typically detects any shared point. For polygon simplicity, any intersection between non-adjacent edges is usually disallowed. Sometimes, problems might only care about "crossings" where segments pass through each other, not just "touching". The standard segment intersection test handles all touches. If you need to distinguish, you'd analyze the orientations $o1, o2, o3, o4$ more deeply (see Gotcha 1.1.2). For path self-intersection, usually any touch of non-adjacent segments (not at a shared path vertex that connects them) is bad. The condition $|i-j| > 1$ for segments $[P_i, P_{i+1}]$ and $[P_j, P_{j+1}]$ correctly identifies non-adjacent segments. $S_i = [P_i, P_{i+1}]$, $S_{i+1} = [P_{i+1}, P_{i+2}]$. These are adjacent. The loop structure `for j from i+2` ... ensures P_j is at least P_{i+2}, so segment $[P_j, P_{j+1}]$ starts at least at P_{i+2}. So it cannot share P_{i+1} with $[P_i, P_{i+1}]$. It also cannot share P_i.

Problem Example 1.4.3 (Problem: "Segments" - Timus Online Judge 1401)**.** *Why*: This problem directly asks to count intersections among a given set of general segments. Checking path self-intersection is a special case where segments are connected end-to-end. **Problem Description**: Given N line segments (not necessarily forming a path), count the total number of intersection points among them. **Solution Strategy for Path Variant**: If the problem were to check self-intersection of a polygonal chain $P_0, \ldots, P_{N-1}$: 1. Iterate through all pairs of segments $([P_i, P_{i+1}], [P_j, P_{j+1}])$ such that $j > i + 1$. (This ensures non-adjacency). 2. For each pair, use the robust segment intersection test (Algorithm 10). 3. If any such pair intersects, the path self-intersects. The Timus problem is more general (any pair of segments, not just non-adjacent from a path) and asks for count, typically requiring a sweep-line algorithm for $O(N \log N + K \log N)$ where K is number of intersections. For just detecting *if* a path self-intersects, the $O(N^2)$ approach is often sufficient if N is small (e.g., $N \leq 2000 - 5000$). **URL**: `https://acm.timus.ru/problem.aspx?space=1&num=1401` (This Timus problem itself is harder, usually solved with sweep line for N up to 10^5. The $O(N^2)$ self-intersection check is for smaller N).

Insight

For a large number of segments ($N > 5000$ or so), an $O(N^2)$ check for intersections is too slow. More advanced algorithms like the Bentley-Ottmann sweep-line algorithm (*(see Chapter X on Sweep Line)*) can find all K intersections among N segments in $O((N + K) \log N)$ time or detect if any intersection exists in $O(N \log N)$ time.

1.5 Template-Quality Code Snippets

Talk is cheap, show me the code! This section provides well-commented, reusable C++ code snippets for the fundamental geometric structures and operations discussed in this chapter. These are designed to be copy-paste-adapt friendly for your contest template. We'll prioritize clarity, correctness (especially with integer types), and typical competitive programming style.

1.5.1 Point / Vector Struct in C++

A robust `Point` (or `Vector`) struct is the cornerstone of any geometry library. Here's a template-based C++17 version.

Listing 1.6: "Point/Vector Struct in C++17"

```cpp
#include <cmath>     // For std::sqrt, std::atan2 (for PointD methods)
#include <iostream>  // For operator<< (optional)
#include <algorithm> // For std::min, std::max (for on_segment if included here)

const double PI = std::acos(-1.0); // For angle conversions if needed
const double EPS_DEFAULT = 1e-9;   // Default epsilon for PointD

template <typename T>
struct Point {
    T x, y;

    // Constructors
    Point(T _x = 0, T _y = 0) : x(_x), y(_y) {}

    // Vector operations
    Point operator+(const Point& other) const { return Point(x + other.x, y + other.y); }
    Point operator-(const Point& other) const { return Point(x - other.x, y - other.y); }
    Point operator*(T scalar) const { return Point(x * scalar, y * scalar); }
    // Note: Scalar division might need care for integer T (truncation)
    // Point operator/(T scalar) const { return Point(x / scalar, y / scalar); } // Add if needed

    // Dot and Cross products
    T dot(const Point& other) const { return x * other.x + y * other.y; }
    T cross(const Point& other) const { return x * other.y - y * other.x; }
    // Cross product for three points P, A, B (as P->A x P->B)
    // static T cross(const Point& P, const Point& A, const Point& B) {
    //    return (A - P).cross(B - P);
    // }


    // Magnitude and distance
    T norm_sq() const { return x * x + y * y; } // Squared norm (magnitude squared)

    // Methods that might return double even if T is integer
    double norm() const { return std::sqrt(static_cast<double>(norm_sq())); }
    double angle() const { return std::atan2(static_cast<double>(y), static_cast<double>(x)); } // Angle w.r.t positive x-axis

    // Rotation (typically returns Point<double> or needs careful handling for Point<int>)
    // Rotates CCW by 'a' radians around origin.
    Point<double> rotate(double angle_rad) const {
        double s = std::sin(angle_rad);
        double c = std::cos(angle_rad);
        double new_x = static_cast<double>(x) * c - static_cast<double>(y) * s;
        double new_y = static_cast<double>(x) * s + static_cast<double>(y) * c;
        return Point<double>(new_x, new_y);
    }
    // Specific 90-degree rotation (preserves T if T is integer)
    Point rotate90_ccw() const { return Point(-y, x); }
    Point rotate90_cw() const { return Point(y, -x); }

    // Comparison operators
    bool operator<(const Point& other) const { // Lexicographical comparison
        if (x != other.x) return x < other.x;
        return y < other.y;
    }
    bool operator==(const Point& other) const {
        if constexpr (std::is_floating_point_v<T>) {
```

```cpp
            return std::abs(x - other.x) < EPS_DEFAULT && std::abs(y - other.y) < EPS_DEFAULT;
        } else {
            return x == other.x && y == other.y;
        }
    }
    bool operator!=(const Point& other) const { return !(*this == other); }

    // Optional: Input/Output
    friend std::istream& operator>>(std::istream& is, Point& p) {
        is >> p.x >> p.y;
        return is;
    }
    friend std::ostream& operator<<(std::ostream& os, const Point& p) {
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }
};

using PointLL = Point<long long>; // Common choice for integer coordinates
using PointD = Point<double>;   // For problems requiring floating point precision

// Example: Orientation function using the Point struct methods
template <typename T>
T orientation_val(const Point<T>& p1, const Point<T>& p2, const Point<T>& p3) {
    return (p2 - p1).cross(p3 - p1);
}

template <typename T>
int orientation_sign(const Point<T>& p1, const Point<T>& p2, const Point<T>& p3) {
    T val = orientation_val(p1, p2, p3);
    if constexpr (std::is_floating_point_v<T>) {
        if (std::abs(val) < EPS_DEFAULT) return 0; // Collinear
    } else {
        if (val == 0) return 0; // Collinear
    }
    return (val > 0) ? 1 : -1; // 1 for CCW, -1 for CW
}
```

Implementation Notes: Design Choices and Considerations:

- **Templated Type `T`**: Using a template parameter `T` allows this struct to work with `int`, `long long`, `double`, or `long double`. This is flexible. For competitive programming, you'll typically typedef `Point<long long>` as `PointLL` (or just `Pt`) and `Point<double>` as `PointD`.
- **Default Constructor**: `Point(T _x = 0, T _y = 0)` initializes to origin by default.
- **Vector Operations**: `+`, `-`, `*` (scalar) are natural. Operator overloading makes code cleaner (e.g., `vec_AB = B - A;`). Be cautious with `/` (scalar) for integer types due to truncation; it might be better to omit or make it return `Point<double>`.
- **Dot and Cross Products**: Essential. `cross` as defined is $P.x \cdot O.y - P.y \cdot O.x$, which is $\vec{OP} \times \vec{OO'}$ if O is origin and O' is `other`. If P and Q are vectors, P.`cross`(Q) is $P_x Q_y - P_y Q_x$. The static `cross(P,A,B)` version or using `(A-P).cross(B-P)` is often for orientation.
- **Norms**: `norm_sq()` is crucial for exact distance comparisons with integers. `norm()` returning `double` is convenient for when actual length is needed.
- **Rotation**: General `rotate(angle_rad)` almost always involves `double`s and trig functions. Specific `rotate90_ccw()` and `rotate90_cw()` are exact for integers and very fast.
- **Comparison `operator<`**: Lexicographical sort is standard for sorting points or using them in `std::map`/`std::set`.

- **Equality `operator==`**: This is tricky for floats.
  Using `if constexpr (std::is_floating_point_v<T>)`
  (C++17) allows different logic for float vs. integer types within the same template. For floats, comparison uses `EPS_DEFAULT`.
- **Input/Output**: `operator»` and `operator«` are convenient for debugging and I/O.
- **`long long` Default for Contests**: For integer coordinates, `PointLL = Point<long long>` is generally the safest default to avoid overflow in cross/dot products (see Section 1.3.2), unless problem constraints guarantee small coordinates.
- **`double` Variant for Precision Needs**: `PointD = Point<double>` when calculations inherently require floats (e.g., non-trivial angles, intersections that don't fall on integer coords). The `EPS_DEFAULT` should be chosen carefully (see Section 1.3.1).

Tips

Best Practices for Usage:

- **Clarity**: Name instances clearly, e.g., `Point pivot; Vector dir;`.
- **Operations**: Prefer `(B-A).cross(C-A)` for orientation of A, B, C rather than global functions if methods are available.
- **Templating vs. Separate Structs**: While templating is powerful, some prefer two distinct structs (`PointInt`, `PointDouble`) to avoid `if constexpr` and to be more explicit about types. This is a style choice.
- **Keep it Minimal**: Only add methods you frequently use. A bloated struct can be harder to manage. Common additions: `dist_sq(other)`, `dist(other)`.

Key Definitions:

- **Point**: A location (x, y) in 2D space. (Definition 1.1.1)
- **Vector**: Direction and magnitude, e.g., $\vec{AB} = B - A$. (Definition 1.1.2)
- **Dot Product**: $\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y = |\vec{a}||\vec{b}| \cos\theta$. Measures alignment. Zero if orthogonal. (Definition 1.1.4)
- **Cross Product (2D)**: $\vec{a} \times \vec{b} = a_x b_y - a_y b_x = |\vec{a}||\vec{b}| \sin\theta$. Signed area of parallelogram. Zero if collinear. Sign gives orientation. (Definition 1.1.5)
- **Orientation Test**: Uses cross product $(P_2 - P_1) \times (P_3 - P_1)$ to determine if P_3 is Left (CCW), Right (CW), or Collinear w.r.t. directed line P_1P_2. (Definition 1.1.15)
- **Segment Intersection**: Two segments share a common point. Checked using orientations and handling collinear cases. (Definition 1.1.16)

Core Algorithms:

- Point-Line Distance: $|\text{cross}(\vec{AP}, \vec{AB})|/|\vec{AB}|$. (Definition 1.1.13, **??**)
- Point-Segment Distance: Check projection; if outside segment, use endpoint distance. (Definition 1.1.14, **??**)
- Angular Sort: Use `atan2(dy, dx)` (careful with range/precision) or cross-product comparator (robust with integers). (Section 1.4.2)

- Segment Intersection Test: Combines orientation tests for general and collinear cases. (Algorithm 10)

Critical Gotchas:

- **Floating Point Precision**: Use `EPS` for comparisons. Avoid equality checks. `1e-9` is a common `EPS`. (Section 1.3.1)
- **Integer Overflow**: Cross products and squared norms can exceed `int` range. Use `long long`. (Section 1.3.2)
- **Collinear Cases**: Often edge cases in orientation, intersection, angular sort. Handle explicitly. (Section 1.1.4, Gotcha 1.4.1)
- **`atan2(y,x)` Range**: $(-\pi, \pi]$. Be careful when comparing angles near $\pm\pi$. (Section 1.4.2.1, Section 1.3.3)

When to Use What:

- **Integers + Cross Product**: For exact orientation, collinearity, and intersection tests if coordinates are integers. Most robust. (Section 1.1.4)
- **Floats + `atan2`**: For direct angle calculation, but be wary of precision. (Section 1.4.2.1)
- **Squared Norms**: For distance comparisons to avoid `sqrt`. (Section 1.1.1.5)

1. Implement a `Point` struct for `long long` coordinates with methods for addition, subtraction, dot product, cross product, and squared norm. Test it thoroughly with various inputs, including edge cases like zero vectors and coincident points. (Section 1.5.1)

2. Write a function `orientation(P, Q, R)` that returns -1 (CW), 0 (Collinear), or 1 (CCW) for three points P, Q, R using only integer arithmetic. Ensure it correctly handles potential overflows by using `long long` for intermediate cross product calculations. (Definition 1.1.15, Algorithm 8)

3. Implement a function `segment_intersect(P1, P2, P3, P4)` that correctly handles general cases, collinear overlaps, and endpoint touching. Test with edge cases like T-junctions, overlapping collinear segments, and segments that are single points. (Section 1.1.5, Algorithm 10)

4. Given a point P_0 and a list of other points $P_1, \ldots, P_N$, sort these points angularly around P_0. Implement this using both `atan2` and a cross-product based comparator. Discuss the pros and cons of each method for competitive programming, especially concerning precision, speed, and handling of points in different quadrants relative to P_0. (Section 1.4.2)

5. What is the maximum possible value of the z-component of the cross product $(P_2 - P_1) \times (P_3 - P_1)$ if the coordinates of P_1, P_2, P_3 are integers between -10^5 and 10^5? What data type is needed to store this result without overflow? (Hint: $(x_2 - x_1)$ can be $2 \cdot 10^5$). (Section 1.3.2)

6. Describe how you would modify the point-segment distance algorithm (**??**) to return not just the distance, but also the coordinates of the closest point on the segment to the query point P_0.

7. Consider three distinct collinear points A, B, C. How can you use only dot products (and no `sqrt` or division if possible) involving vectors formed by these points to determine if B lies strictly between A and C? (Hint: consider vectors $\vec{BA}$ and $\vec{BC}$).

8. (Challenge) Given a simple polygon (a list of vertices in order) and two points A and B (which can be inside, outside, or on the boundary). Determine if the segment AB intersects any edge of the polygon. What further checks are needed if A and B are inside and you want to know if they are mutually visible (i.e., AB doesn't cross any edge)? Consider cases where A or B are vertices of the polygon, or AB is collinear with an edge.

1.6 Further Reading & Resources

This chapter has laid the foundation. If you're eager to explore these concepts more deeply or see alternative explanations, here are some excellent resources:

1.6.1 Computational Geometry: Algorithms and Applications by de Berg et al. (Chapter 1)

Further Reading 1.6.1 (*Computational Geometry: Algorithms and Applications*, 3rd Edition, by M. de Berg, O. Cheong, M. van Kreveld, M. Overmars (Springer, 2008) [1]). Chapter 1, "Geometric Primitives," of this classic textbook (often just called "de Berg") provides a formal and rigorous introduction to the topics covered in our chapter. It's a standard academic reference. **Key Takeaways Relevant to Our Chapter A**:

- **Precise Definitions**: Establishes clear mathematical definitions for points, vectors, lines, segments, and their representations (Sections 1.1, 1.2).
- **Orientation Test**: Discusses the `Orientation` (or `CCW`) test in detail, including its derivation from determinants and its importance as a primitive operation (Section 1.3). This aligns with our Section 1.1.4.
- **Segment Intersection**: Provides a careful treatment of line segment intersection, including handling of degenerate cases (Section 1.3). This corresponds to our Section 1.1.5.
- **Numerical Issues**: Briefly touches upon the robustness issues with floating-point arithmetic in geometric computations, setting the stage for why exact arithmetic or careful handling is needed.

While more theoretical than a competitive programming tutorial, de Berg offers excellent explanations of the "why" behind these primitives and their geometric significance. It's a great place to solidify your understanding if you find the mathematical aspects intriguing. The rest of the book covers many advanced topics we'll encounter later.

1.6.2 CP-Algorithms: Basic Geometry

Further Reading 1.6.2 (CP-Algorithms: Basic Geometry (`https://cp-algorithms.com/geometry/basic-geomet` [2]). This section of CP-Algorithms is a fantastic, concise resource tailored specifically for competitive programmers. It covers many of the same primitives we've discussed, often with direct C++ implementations. **Specific Topics Covered Aligning with Our Chapter A**:

- **Point/Vector Structures**: Shows typical C++ structs for points and vectors, including common operations like addition, subtraction, dot/cross products (Section 1.1.1, Section 1.5.1).
- **Dot and Cross Product**: Explains their geometric meaning (angle, area, orientation) and formulas (Section 1.1.1.3, Section 1.1.1.4).
- **Orientation Test**: Provides logic for the CCW test using cross products (Section 1.1.4).
- **Distance Formulas**: Covers point-point, point-line, and point-segment distances (Section 1.1.3).
- **Segment Intersection**: Explains the orientation-based approach to check for segment intersection (Section 1.1.5).
- **Code Examples**: Provides C++ snippets for many of these, which can be a good reference alongside our template code.

- **Common Pitfalls**: Often discusses issues like floating point precision and integer overflow, reinforcing the lessons from Section 1.3.

CP-Algorithms is highly recommended for its practical focus and clear explanations. It's a go-to site for many competitive programmers.

1.7 Lesser-Known Tricks & Advanced Tidbits

Beyond the standard textbook material, there are always interesting nuances, clever optimizations, or more robust ways to handle tricky situations in computational geometry. This section briefly touches on a couple of such areas related to our foundational topics. These might not be "open research" in the academic sense, but they represent deeper dives or alternative perspectives.

1.7.1 Robustly Templating Geometry for Integers vs. Floats

Open Question/Trick 1.7.1 (Consistent Template Handling of Integer vs. Floating-Point Geometry). As seen in our `Point<T>` struct (listing 1.6), designing C++ templates for geometric structures that seamlessly and correctly handle both integer types (`int`, `long long`) and floating-point types (`double`, `long double`) presents several challenges. The goal is to write generic code that behaves intuitively and correctly for both, minimizing boilerplate or error-prone specializations.

Key Challenges:

- **EPS Comparisons**: Floating-point equality needs `EPS` (definition 1.3.1); integers use direct `==`. How do you make `operator==` or functions like `is_zero(value)` generic? `if constexpr (std::is_floating_point_v<T>)` (C++17) is the modern solution for this, allowing compile-time dispatch to different code paths.
- **Division**: Integer division truncates (e.g., `Point<int>(5,5) / 2` becomes `(2,2)`). Floating-point division is fractional. If `Point::operator/(T scalar)` is defined, its behavior differs. Sometimes, integer division is desired; other times, a promotion to float is implicitly expected.
- **Return Types of Mixed Operations**: Functions like `norm()` (calculating $\sqrt{x^2 + y^2}$) almost always return a `double`, even if input `Point<T>` has `T=long long`, because `sqrt` operates on and returns doubles. This means `Point<long long> p; auto len = p.norm();` makes `len` a `double`. This is usually fine but needs awareness. In contrast, `norm_sq()` can safely return `T` (or a wider integer type like `long long` if `T` is `int` to prevent overflow of $x^2 + y^2$).
- **Functions Involving Transcendental Operations**: Operations like `rotate(angle_rad)` (listing 1.6) or `angle()` (using `atan2`) inherently involve `doubles` for angles and trigonometric functions. If you have `Point<long long> p;`, should `p.rotate(theta)` return `Point<double>` (as in our example, which is common) or attempt to round back to `Point<long long>` (losing precision and generally a bad idea)?
- **Type Promotion**: When combining `Point<int>` with `Point<double>`, or a `Point<int>` with a `double` scalar, what should the resulting type be? C++ default promotion rules apply, but consistent library design might want more explicit control.

Potential Design Patterns and Compromises:

- **`if constexpr`** (C++17 and later): This is the cleanest way to handle many differences, as it allows for compile-time selection of code paths based on type traits like `std::is_floating_point_v<T>` or `std::is_integral_v<T>`. Used in listing 1.6 for `operator==`.
- **Policy-Based Design**: Define `policy` classes that encapsulate behaviors differing by type (e.g., an `EqualityPolicy` with different implementations for int/float). The `Point` struct could then be templated on the coordinate type `T` and also a `MathPolicy<T>`. This offers high customization but increases complexity.

- **Tag Dispatching**: Use overloaded functions that take an extra `tag` argument (e.g., `struct float_tag {}; struct int_tag {};`) to manually dispatch to different implementations. `foo(point, int_tag{})` vs `foo(point, float_tag{})`. Can be verbose.

- **CRTP (Curiously Recurring Template Pattern)**: Can be used to add type-specific functionality from a base template. Less common for simple Point structs.

- **Separate Structs**: Simply define `PointInt`, `PointLL`, `PointDouble` as distinct, non-templated structs. This is the simplest approach, avoids template complexities, but means some code duplication if many operations are identical in logic. Conversions between them must be explicit. This is often a pragmatic choice in timed contests if full template generality isn't crucial.

- **Careful Method Naming and Return Types**: Be explicit. For example, `norm_sq()` returns `T` (or `long long`), while `norm()` returns `double`. `rotate_exact_90()` returns `Point<T>`, while `rotate_angle(double ang)` returns `Point<double>`.

Achieving a perfectly consistent, intuitive, and efficient templated geometry library that handles all integer/float nuances elegantly remains a design art. For competitive programming, a well-thought-out `Point<T>` with `if constexpr` for critical differences, and careful use of `PointLL` and `PointD` typedefs, strikes a good balance.

1.7.2 Radian-less Angle Comparison: The Integer Way

Open Question/Trick 1.7.2 (Radian-less Angle Comparison). As discussed in angular sort (Section 1.4.2), comparing angles or sorting points angularly without explicit angle calculation (i.e., avoiding `atan2`) can significantly improve robustness (by staying with integers) and sometimes speed. The challenge is to create a comparator `bool compare_angles(Point P0, Point A, Point B)` that returns true if vector $\vec{P_0A}$ comes before vector $\vec{P_0B}$ in a standard counter-clockwise sweep from the positive x-axis direction, handling all quadrants and collinear cases correctly using only integer arithmetic (assuming integer coordinates).

Core Idea: Partition points by half-planes relative to P_0, then use cross-product within half-planes.

Detailed Logic for a Robust Comparator Function `less_angle(P0, A, B)`:
Assume P_0 is the pivot. We want to determine if angle $\angle XP_0A < \angle XP_0B$, where X is a point far along the positive X-axis from P_0.
Let $v_A = A - P_0$ and $v_B = B - P_0$. We are comparing vectors v_A and v_B.

1. **Handle Coincidence with Pivot**: If v_A is the zero vector (i.e., $A = P_0$), it usually comes first (or last, depending on convention). If v_B is zero vector and v_A is not, v_B comes first. If both are zero, they are equal.

```
    // bool is_A_zero = (A.x == P0.x && A.y == P0.y);
    // bool is_B_zero = (B.x == P0.x && B.y == P0.y);
    // if (is_A_zero && is_B_zero) return false; // Equal, A not strictly less
    // if (is_A_zero) return true;  // A is P0, B is not, P0 comes first
    // if (is_B_zero) return false; // B is P0, A is not, A does not come first

```

2. **Determine Half-Planes/Quadrants**: A common way is to check which side of the y-axis (passing through P_0) the points A and B lie, and their y-coordinates relative to P_0. This helps establish a coarse ordering.
   A point P (relative to P_0) is in the "upper visual field" if $P.y > P_0.y$, or if $P.y = P_0.y$ and $P.x > P_0.x$. Otherwise, it's in the "lower visual field" (assuming it's not P_0 itself).
   Let's define a `quadrant_group(Point P, Point P0)` function:

   - Returns 0: $P = P_0$.

- Returns 1: P is on the positive x-axis relative to P_0 ($P.y = P_0.y, P.x > P_0.x$) or in upper half-plane ($P.y > P_0.y$). This covers angles in $[0, \pi)$.
- Returns 2: P is on the negative x-axis relative to P_0 ($P.y = P_0.y, P.x < P_0.x$) or in lower half-plane ($P.y < P_0.y$). This covers angles in $[\pi, 2\pi)$.

If `quadrant_group(A, P0) < quadrant_group(B, P0)`, then A comes before B.

```
    // int quad_A = get_quadrant_group(A, P0); // Using a helper like from Impl A.4.2
    // int quad_B = get_quadrant_group(B, P0);
    // if (quad_A != quad_B) {
    //     return quad_A < quad_B;
    // }

```

The `get_half_plane` function from **??** is an example of this partitioning.

3. **Cross Product for Same Half-Plane/Region**: If A and B are in the same half-plane (e.g., both `quadrant_group` returned 1, or both returned 2), use the orientation test orientation(P_0, A, B).
The value is $(A - P_0) \times (B - P_0)$.

- If orientation$(P_0, A, B) > 0$: A is CCW from P_0 to B. So A comes before B. Return `true`.
- If orientation$(P_0, A, B) < 0$: A is CW from P_0 to B. So B comes before A. Return `false`.
- If orientation$(P_0, A, B) = 0$: P_0, A, B are collinear.

```
    // long long orient_val = orientation_val(P0, A, B); // from Point struct/helper
    // if (orient_val == 0) { ... handle collinear ... }
    // return orient_val > 0; // If in upper half, CCW means A is smaller angle
                              // If in lower half, need to be careful.
                              // The half-plane logic above simplifies this:
                              // if quad_A == quad_B, then orient_val > 0 means A < B.

```

4. **Collinear Case Tie-Breaking**: If P_0, A, B are collinear, the one closer to P_0 is typically considered to have the "smaller" angle if they are in the same direction from P_0. If they are in opposite directions, the half-plane check should have already differentiated them.
Use squared distance: $\|A - P_0\|^2$ vs $\|B - P_0\|^2$.
If A, B are in the same direction from P_0, point closer to P_0 comes first.

```
    // if (orient_val == 0) { // Collinear
    //     return distSq(P0, A) < distSq(P0, B); // Closer first
    // }

```

The comparator from Section 1.8 provides a concrete C++ implementation of this logic. It correctly handles all cases for a full 360° sort around P_0, assuming P_0 is the reference and angles are measured CCW from the direction ($P_0 \to P_0 + (1, 0)$).

Considerations for robustness:

- **Pivot Point Handling**: If A or B is P_0 itself. Usually P_0 comes before any other point.
- **Integer Overflow**: All cross products and squared norms must use `long long` if intermediate coordinate products can exceed `int` limits.
- **Consistency**: The definition of `half-plane` and tie-breaking rules must be consistent. The `get_half_plane` logic in Section 1.8 implicitly defines angles starting from the positive x-axis direction from P_0 and sweeping CCW.

This type of comparator is a fundamental building block for algorithms like Graham Scan convex hull (*(see* **??***)*) where selecting an appropriate pivot (e.g., lowest then leftmost point) simplifies the angular sort to mostly occur within a 180° range, making the cross-product comparisons directly applicable without complex half-plane logic.

1.8 Full 360 Cross Product Sort

Part II

Polygons & Lattice Geometry

Chapter 2

Polygons and Lattice Geometry

2.1 Core Concepts

2.1.1 Polygon Fundamentals

Definition 2.1.1 (Simple Polygon)**.** A polygon is **simple** if its edges don't intersect except at vertices. Think of a rubber band stretched around pins - it naturally forms a simple polygon.

Definition 2.1.2 (Convex Polygon)**.** A polygon is **convex** if any line segment between two points inside the polygon lies entirely inside. Alternatively: all interior angles are less than 180°.

2.1.2 The Shoelace Formula

Theorem 2.1.1 (Shoelace Formula)**.** *For a simple polygon with vertices* $(x_1, y_1), ..., (x_n, y_n)$ *listed in order:*

$$\textit{Area} = \frac{1}{2}\left|\sum_{i=1}^{n}(x_i y_{i+1} - x_{i+1} y_i)\right|$$

where indices wrap around: $(x_{n+1}, y_{n+1}) = (x_1, y_1)$*.*

Intuition: Imagine lacing a shoe - you create triangular sections as you cross the laces. The formula sums signed areas of triangles formed by each edge with the origin.

2.1.3 Lattice Points and Pick's Theorem

Definition 2.1.3 (Lattice Point)**.** A point (x, y) where both x and y are integers. On a grid, these are the intersection points.

Theorem 2.1.2 (Pick's Theorem)**.** *For a simple polygon with integer vertices:*

$$\textit{Area} = I + \frac{B}{2} - 1$$

where I *= interior lattice points,* B *= boundary lattice points.*

2.2 Algorithms Implementation

2.2.1 Computing Polygon Area

Algorithm 14: Shoelace Algorithm

Input: *vertices*
Output: Area of the polygon

$n \leftarrow vertices.size()$
$area \leftarrow 0$
for $i \leftarrow 0$ **to** $n - 1$ **do**
  $j \leftarrow (i + 1) \bmod n$
  $area \leftarrow area + vertices[i].x \times vertices[j].y$
  $area \leftarrow area - vertices[j].x \times vertices[i].y$
return $|area|/2$

Complexity Analysis

- **Time:** $O(n)$ where n is the number of vertices
- **Space:** $O(1)$ additional space
- **When to use:** Always for simple polygons; works for both convex and non-convex

Implementation Notes:

Tips

- **Overflow:** For large coordinates, intermediate products can overflow. Use `long long` or `__int128`.
- **Sign:** The raw result's sign indicates orientation (CW/CCW). Take absolute value for area.
- **Precision:** For integer coordinates, the result before division by 2 is always even.

2.2.2 Point-in-Polygon Testing

Algorithm 15: Ray Casting Algorithm

Input: *point*, *polygon*
Output: True if inside, False otherwise

$crossings \leftarrow 0$
$n \leftarrow polygon.size()$
for $i \leftarrow 0$ **to** $n - 1$ **do**
  $j \leftarrow (i + 1) \bmod n$
  if *ray from point to* $+\infty$ *crosses edge* (i, j) **then**
    $crossings \leftarrow crossings + 1$
return $crossings \bmod 2 = 1$

Warning

Edge cases require careful handling:

- Ray passing through a vertex
- Ray overlapping with a horizontal edge
- Point exactly on polygon boundary

2.2.3 Counting Lattice Points

Algorithm 16: Boundary Points on a Segment

Input: p_1, p_2

$dx \leftarrow |p_2.x - p_1.x|$

$dy \leftarrow |p_2.y - p_1.y|$

return $\gcd(dx, dy) + 1$

Insight

The number of lattice points on a segment is related to the GCD of the coordinate differences. This connects geometry to number theory!

2.3 Code Templates

2.3.1 C++ Implementation

Listing 2.1: C++ Implementation of Polygon Area Calculation

```cpp

struct Point {
    long long x, y;
    Point(long long x = 0, long long y = 0) : x(x), y(y) {}
};

// Polygon area using shoelace formula
long long polygonArea2(const vector<Point>& poly) {
    int n = poly.size();
    long long area = 0;
    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        area += poly[i].x * poly[j].y;
        area -= poly[j].x * poly[i].y;
    }
    return abs(area);  // Returns 2 * area
}
// Count boundary points on a segment
long long boundaryPoints(Point p1, Point p2) {
    return __gcd(abs(p2.x - p1.x), abs(p2.y - p1.y));
}

// Pick's theorem: find interior points
long long interiorPoints(const vector<Point>& poly) {
    long long area2 = polygonArea2(poly);
    long long boundary = 0;
    int n = poly.size();

    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        boundary += boundaryPoints(poly[i], poly[j]);
    }

    return (area2 - boundary + 2) / 2;
}
```

2.4 Problem Patterns

2.4.1 Direct Application

Direct Application

- **Recognition:** Problem explicitly asks for polygon area or lattice point count
- **Approach:** Apply formulas directly
- **Common Variations:** Area of union/intersection of polygons

CSES - Polygon Area

Statement: Given a polygon with n vertices, calculate its area.

Key Insight: Direct application of shoelace formula

Solution Sketch: Implement the formula, watch for overflow

2.4.2 Hidden Geometry

- **Recognition:** Problem about grids/lattices that doesn't mention geometry
- **Approach:** Model as polygons, apply Pick's theorem
- **Common Variations:** Counting valid positions, grid path problems

2.5 Gotchas & Debugging

2.5.1 Common Issues and Debug Checklist

Gotcha 2.5.1. **Issue:** Integer overflow in area calculation **Symptom:** Wrong answers for large coordinates **Fix:** Use `long long` or `__int128` **Prevention:** Always check max coordinate range; $(10^6)^2 > \text{int}$

Gotcha 2.5.2. **Issue:** Wrong vertex ordering (CW vs CCW) **Symptom:** Negative area or wrong orientation tests **Fix:** Take absolute value of area; be consistent with ordering **Prevention:** Document your convention; test with known examples

Debug Checklist: Polygons

- Verify polygon is simple (no self-intersections)
- Check edge cases: collinear points, duplicate vertices
- Test with triangles and squares to verify area calculation
- For Pick's theorem: manually count points on small examples

2.6 Practice Problems

2.6.1 Problem Set

1. **Easy** - *CSES* - Polygon Area [Shoelace formula]

2. **Easy** - *CSES* - Point in Polygon [Ray casting or winding number]
3. **Medium** - *USACO* - Fence [Pick's theorem, lattice points]
4. **Medium** - *AtCoder* - Lattice Points [Pick's theorem with modifications]
5. **Hard** - *CodeForces* - Polygon Union [Sweep line + Pick's theorem]

2.7 Deep Dive

2.7.1 Why Pick's Theorem Works

Pick's theorem reveals a deep connection between continuous and discrete geometry. The proof uses:

- Triangulation of polygons
- Euler's formula for planar graphs
- Inclusion-exclusion principle

2.7.2 Generalizations

- 3D Pick's theorem (much more complex)
- Pick's theorem for polygons with holes
- Connections to generating functions

2.8 Quick Reference

2.8.1 Chapter Summary Card

Polygon & Lattice Geometry Reference
Key Formulas:

- Shoelace: Area $= \frac{1}{2}|\sum_{i=1}^{n}(x_i y_{i+1} - x_{i+1} y_i)|$
- Pick's: Area $= I + \frac{B}{2} - 1$
- Boundary points on segment: $\gcd(|x_2 - x_1|, |y_2 - y_1|) + 1$

	Polygon area	$O(n)$
Complexity:	Point-in-polygon	$O(n)$
	Point-in-convex-polygon	$O(\log n)$

Common Pitfalls:

- Integer overflow with large coordinates
- Edge cases in point-in-polygon
- Vertex ordering assumptions

2.9 End-Chapter Exercises

Part III

Core Geometric Algorithms

Chapter 3

Convex Hull & Post-Hull Algorithms

Imagine you're designing a game where characters navigate a complex level filled with obstacles. You need to determine the "visible" area for each character, taking into account the walls and other obstructions. A fundamental tool in solving this and similar problems is the convex hull.

In this chapter, we delve into the fascinating world of convex hulls. They're much more than a geometric curiosity; they serve as essential building blocks for solving a wide variety of computational geometry problems that appear regularly in competitive programming. We will start by exploring the definition and properties of convex hulls, focusing on how to efficiently compute them.

Why does this matter? Consider problems like finding the shortest path that avoids obstacles or identifying the furthest pair of points in a set. Convex hulls provide elegant and efficient solutions to these types of challenges. The key lies in recognizing how the convex hull simplifies a complex problem by reducing the amount of data we need to consider.

- **Real-World Connection:** Imagine a sensor network deployed to monitor an area. The convex hull of the sensors represents the smallest region they cover. If a target moves outside this region, it means the sensors have "lost" it.

- **Challenge Problem:** Given a set of points representing the locations of buildings in a city, design an algorithm to find the smallest rectangular plot of land that can enclose *all* the buildings, oriented in any direction. We will solve this using a clever combination of convex hulls and the rotating calipers technique! This problem combines the concepts of a convex hull with optimization, providing a challenging application of the techniques we will cover.

By the end of this chapter, you'll have a solid understanding of what a convex hull is, how to compute it efficiently, and, most importantly, how to apply it to solve a wide range of geometric problems. Get ready to unleash the power of convex hulls!

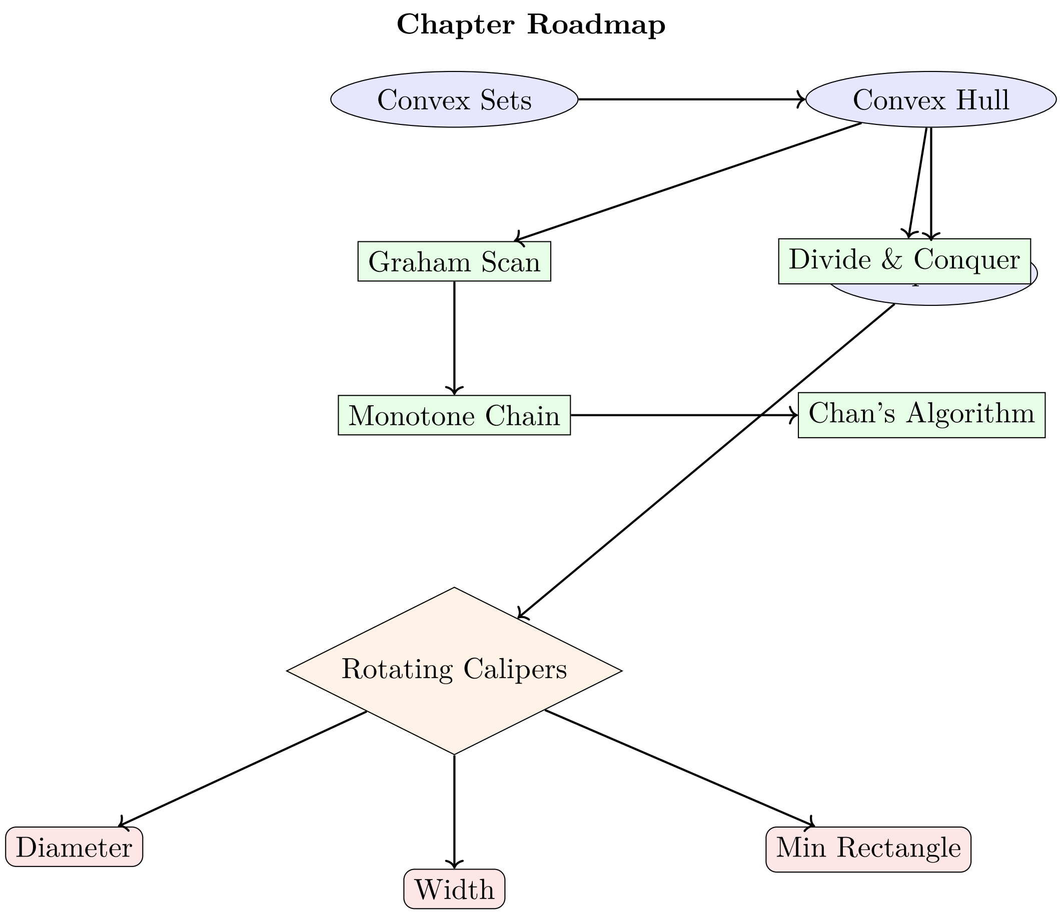


Figure 3.1: From convex hull theory to practical applications

3.1 Formal Theory

In this section, we'll establish the fundamental definitions and properties of convex sets and convex hulls. This will serve as a foundation for understanding the algorithms and applications we'll explore later.

3.1.1 Convex Set, Convex Combination

We begin with the core definitions.

Definition 3.1.1 (Convex Set)**.** A set S in the Euclidean plane is called **convex** if for any two points $p, q \in S$, the entire line segment connecting p and q also lies in S.

Intuition: Imagine a shape. If you can draw a straight line between any two points inside the shape, and that line stays *completely* within the shape, then the shape is convex. Think of it like this: a convex set has no "dents" or "caves." It's like a perfectly smooth surface.

Let's see some examples and non-examples.

Draw four diagrams. 1. A circle, labeled "Convex" 2. A triangle, labeled "Convex" 3. A rectangle, labeled "Convex" 4. A star shape with inward corners, labeled "Non-Convex"

Next, let's define the concept of convex combination.

Definition 3.1.2 (Convex Combination)**.** A **convex combination** of points $p_1, p_2, \ldots, p_n$ is a linear combination $\sum_{i=1}^{n} \lambda_i p_i$ where:

- Each coefficient $\lambda_i \geq 0$ (non-negative).
- The sum of all coefficients equals 1: $\sum_{i=1}^{n} \lambda_i = 1$.

Intuition: A convex combination represents a weighted average of points. Each point contributes to the 'average' based on its weight (λ_i). These weights must be non-negative, meaning no point can "cancel out" another. Furthermore, because the weights add up to 1, the combination will always fall *inside* or *on the boundary of* the shape formed by the points.

Draw a diagram with 3 points, p_1, p_2, p_3 that form a triangle. 1. Show a point that is a convex combination of the three points, and label the point. The diagram should illustrate how it is a weighted average. 2. Show the same points, p_1, p_2, p_3, and a point outside the triangle that is *not* a convex combination of the three points. Highlight the weights of this "combination", and demonstrate why they do not add up to 1.

Here are a couple of crucial cases:
* If $\lambda_1 = 1$ and all other $\lambda_i = 0$, the convex combination equals p_1 (a vertex). * If $\lambda_1 = \lambda_2 = 0.5$ and all other $\lambda_i = 0$, the convex combination equals the midpoint of p_1 and p_2.
A fundamental property ties convex sets and convex combinations together:

Theorem 3.1.1. *A set is convex if and only if it contains all convex combinations of its points.*

Mathematical Insight: This theorem provides an alternative way to define convexity. If we can show that all convex combinations of points within a set also lie within that set, we've proven that the set is convex. This is extremely useful in proofs and helps to solidify the intuition around what "convex" truly means.

3.1.2 Convex Hull Definition

Now, let's define the central concept of this chapter.

Definition 3.1.3 (Convex Hull)**.** The **convex hull** of a set of points P can be defined in several equivalent ways:

1. The smallest convex set containing all points in P.
2. The intersection of all convex sets containing P.
3. The intersection of all half-planes containing P.
4. The set of all convex combinations of points in P.

Intuition: Think of the convex hull as the "skin" that wraps tightly around a set of points. It's the smallest convex shape that can enclose all the given points. Imagine the elastic band stretched around nails (as we saw in the chapter introduction). The shape the band forms is the convex hull.

For practical algorithmic purposes, Definitions 1 and 3 are particularly helpful.

Draw four different diagrams representing the same set of points. 1. In the first, illustrate the smallest convex set containing all the points (Definition 1). 2. In the second, show the intersection of multiple half-planes, whose borders are lines connecting pairs of points from the original set. Highlight the intersection, as it forms the convex hull (Definition 3). 3. In the third, show the convex hull again and label all the vertices as coming from the original point set. 4. In the fourth, show a convex hull and a point that is not in the original set.

In competitive programming, we primarily work with finite sets of points. The convex hull for a finite set of points will always be a convex polygon. The vertices of this polygon will be a subset of the original points.

Insight

The concept of half-planes is crucial. Each edge of the convex hull corresponds to a half-plane whose boundary line contains that edge, with all other points lying inside the half-plane.

3.1.3 Properties of Convex Hulls

(Vertices are input points, edges connect input points)

Convex hulls have several key properties that are used extensively in algorithms. Understanding these properties is crucial.

1. **Vertex property:** The vertices of the convex hull are a subset of the original points in P. No "new" points are created.
2. **Edge property:** Each edge of the convex hull connects two points from the original set P.

3. **Extremal property:** For any direction, the point in P that is extreme in that direction (farthest in that direction) is a vertex of the convex hull.

4. **Supporting line property:** For every edge of the convex hull, all points in P lie on or to one side of the line containing that edge.

5. **Minimal representation:** The convex hull is the minimal convex polygon (in terms of number of vertices or edges) that contains all points in P.

6. **Invariance under affine transformations:** If you apply an affine transformation (translation, rotation, scaling, shearing) to all points in P, the convex hull of the transformed points is the transformation of the convex hull of P.

7. **Monotonicity:** If $A \subseteq B$ are two point sets, then $\text{ConvexHull}(A) \subseteq \text{ConvexHull}(B)$.

8. **Computational complexity:** The convex hull of n points in the plane can be computed in $O(n \log n)$ time in the worst case, and this is optimal in the comparison model.

9. **Output size:** The convex hull of n points in the plane can have at most n vertices, and in the worst case (when all points are on the hull), it has exactly n vertices.

Insight

The extremal property is often used to efficiently find points on the convex hull by searching in different directions. The supporting line property is fundamental to the rotating calipers technique *(see* **??***)*.

Mathematical Insight: The computational complexity result tells us that we can't generally hope to compute a convex hull faster than $O(n \log n)$. This result is extremely important as you start to analyze the time and space complexities of your algorithms.

These properties, when combined, provide a powerful toolkit for solving geometric problems by providing a minimal representation that captures the essential structure of the original point set.

3.2 Canonical Algorithms

Warning

When implementing convex hull algorithms, be extremely careful with the orientation test (whether three points make a left or right turn). Even small errors in this calculation can result in incorrect hulls. Always use a robust implementation of the orientation test that handles collinear points properly.

3.2.1 Graham Scan

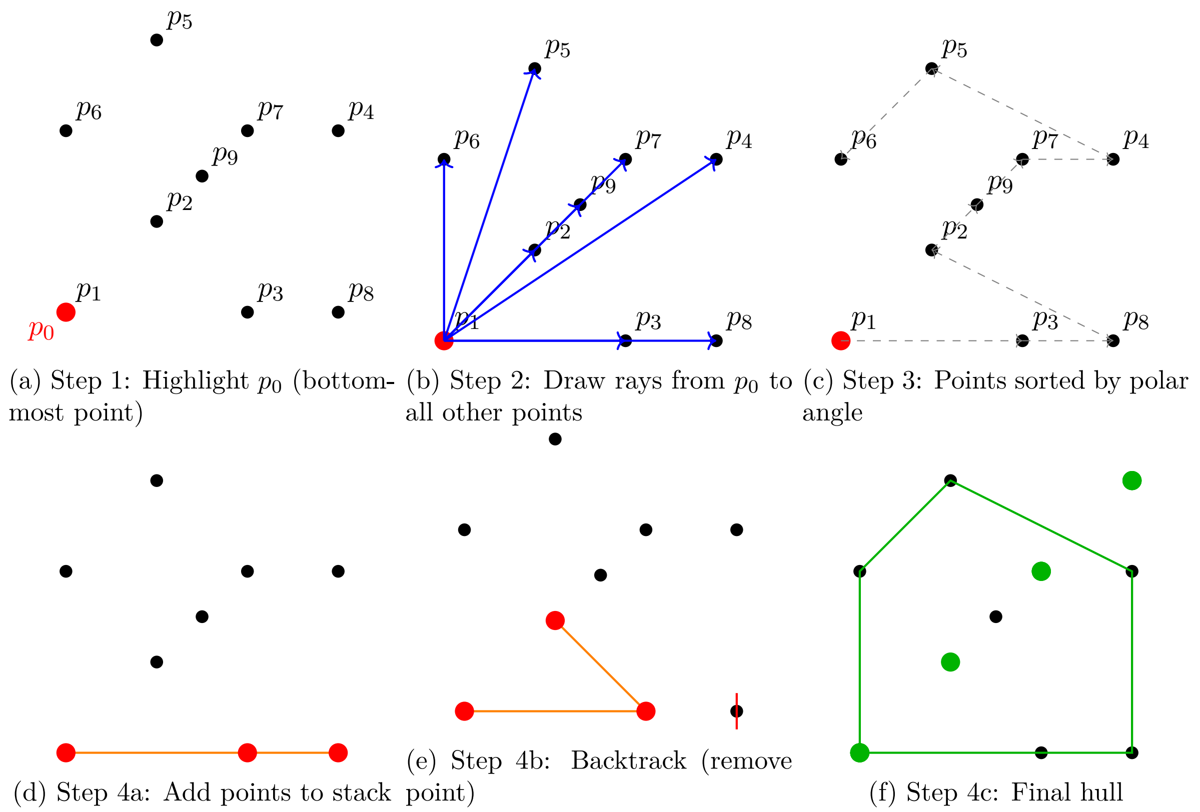
Graham Scan is one of the most well-known algorithms for computing the convex hull of a set of points in the plane. The convex hull is the smallest convex polygon that contains all the given points, and finding it is a fundamental problem in computational geometry with applications in pattern recognition, image processing, and more. Graham Scan is notable for its efficiency and elegance: it sorts the points by polar angle and then constructs the hull in a single pass, making it both intuitive and practical for a wide range of inputs.

Algorithm 17: Graham Scan

Input: A set P of n points in the plane
Output: The convex hull of P as a sequence of vertices in counterclockwise order

Find the point p_0 with the lowest y-coordinate (leftmost in case of ties)
Sort the remaining points by polar angle around p_0
Initialize stack S with p_0 and the first sorted point
for *each remaining point p_i in sorted order* **do**
  while *S contains at least 2 points AND the last 3 points in S don't make a left turn* **do**
    Pop the middle of the last 3 points from S
  end
  Push p_i onto S
end
return S

Intuition: Graham Scan builds the convex hull in a single counterclockwise sweep. The key insight is that as we process points in order of increasing polar angle, we can maintain a sequence of points that form the convex hull of all points processed so far. When we add a new point, we "backtrack" along our current hull, removing any points that would create a "dent" with this new point.



(a) Step 1: Highlight p_0 (bottom-most point)

(b) Step 2: Draw rays from p_0 to all other points

(c) Step 3: Points sorted by polar angle

(d) Step 4a: Add points to stack

(e) Step 4b: Backtrack (remove point)

(f) Step 4c: Final hull

Figure 3.2: Step-by-step illustration of Graham Scan

Tips

- For numerical stability, use the cross product directly for the left turn test rather than computing actual angles.
- Handle collinear points by sorting them by distance from p_0 when they have the same polar angle.
- Be careful with the initial sorting—proper comparison of angles is crucial for correctness.

Listing 3.1: Graham Scan

```
vector<Point> graham_scan(vector<Point> points) {
    int n = points.size();
    if (n <= 3) return points;

    // Find bottom-most point (and leftmost if tied)
    int bottom = 0;
    for (int i = 1; i < n; i++) {
        if (points[i].y < points[bottom].y ||
           (points[i].y == points[bottom].y && points[i].x < points[bottom].x)) {
            bottom = i;
        }
    }
    swap(points[0], points[bottom]);
    Point pivot = points[0];

    // Sort by polar angle with respect to pivot
    sort(points.begin() + 1, points.end(), [&pivot](const Point& a, const Point& b) {
        int64_t cross = (a - pivot) ^ (b - pivot);
```

```
        if (cross == 0) {
            // Collinear points: sort by distance
            return (a - pivot).norm2() < (b - pivot).norm2();
        }
        return cross > 0;
    });

    // Build hull using stack
    vector<Point> hull;
    hull.push_back(points[0]);
    hull.push_back(points[1]);

    for (int i = 2; i < n; i++) {
        while (hull.size() >= 2) {
            Point top = hull.back();
            Point second = hull[hull.size() - 2];
            if (((top - second) ^ (points[i] - second)) <= 0) {
                hull.pop_back();
            } else {
                break;
            }
        }
        hull.push_back(points[i]);
    }

    return hull;
}
```

Complexity Analysis

$O(n \log n)$ time. Sorting dominates.

Invariant: Stack maintains points forming a left-turning chain

3.2.2 Monotone Chain (Andrew's Algorithm)

Algorithm 18: Monotone Chain (Andrew's Algorithm)

Input: A set P of n points in the plane
Output: The convex hull of P as a sequence of vertices in counterclockwise order

Sort points by x-coordinate (breaking ties by y-coordinate)
Initialize empty lower and upper hulls
Build lower hull:
for *each point p from left to right* **do**
  while *lower hull contains at least 2 points AND the last 3 points don't make a left turn* **do**
    Remove the middle of the last 3 points from the lower hull
  end
  Add p to the lower hull
end
Build upper hull:
for *each point p from right to left* **do**
  while *upper hull contains at least 2 points AND the last 3 points don't make a left turn* **do**
    Remove the middle of the last 3 points from the upper hull
  end
  Add p to the upper hull
end
Remove the duplicate endpoints from both hulls
return *concatenated lower and upper hulls*

Intuition: The Monotone Chain algorithm constructs the convex hull of a set of points in two sweeps: first, it builds the lower hull from left to right, then the upper hull from right to left. Unlike the Graham Scan, which sorts points by polar angle and sweeps from the lowest point, Monotone Chain only requires sorting by x-coordinate (breaking ties by y-coordinate). This makes the approach conceptually simpler and easier to implement, as it avoids angle calculations and instead uses straightforward coordinate comparisons.

Insight

A key advantage of the Monotone Chain algorithm is its numerical stability. Sorting points by x-coordinate is generally more robust and less error-prone than sorting by polar angle, especially when dealing with floating-point coordinates. As a result, the implementation is cleaner and less susceptible to subtle bugs caused by floating-point inaccuracies.

Listing 3.2: Monotone Chain

```
vector<Point> monotone_chain(vector<Point> points) {
    int n = points.size();
    if (n <= 3) return points;

    sort(points.begin(), points.end());

    // Build lower hull
    vector<Point> lower;
    for (int i = 0; i < n; i++) {
        while (lower.size() >= 2) {
            int sz = lower.size();
            if (((lower[sz-1] - lower[sz-2]) ^ (points[i] - lower[sz-2])) <= 0) {
                lower.pop_back();
            } else {
```

```
                break;
            }
        }
        lower.push_back(points[i]);
    }

    // Build upper hull
    vector<Point> upper;
    for (int i = n - 1; i >= 0; i--) {
        while (upper.size() >= 2) {
            int sz = upper.size();
            if (((upper[sz-1] - upper[sz-2]) ^ (points[i] - upper[sz-2])) <= 0) {
                upper.pop_back();
            } else {
                break;
            }
        }
        upper.push_back(points[i]);
    }

    // Remove last point of each half because it's repeated
    lower.pop_back();
    upper.pop_back();

    // Combine hulls
    lower.insert(lower.end(), upper.begin(), upper.end());
    return lower;
}
```

Complexity Analysis

The overall time complexity is $O(n \log n)$, dominated by the initial sorting step. The construction of the hull itself is linear in the number of points.

3.2.3 Chan's Algorithm

Algorithm 19: Chan's Algorithm

Input: A set P of n points in the plane
Output: The convex hull of P

```
for m = 3, 6, 12, ..., n do
    Partition P into groups of size ≤ m
    Compute the convex hull of each group using a standard algorithm
    /* Attempt to compute the full hull using the group hulls */
    Initialize h with the leftmost point
    for i = 1 to m do
        /* Find the tangent from the last hull point to each group hull */
        Find the point p across all groups that gives the most counter-clockwise turn
          from the last hull point
        if p is the starting point then
            return hull h
            /* Complete hull found */
        end
        Add p to h
    end
    /* Failed to find the complete hull with this value of m */
end
```

Insight

Chan's algorithm is a breakthrough in convex hull computation, achieving an optimal output-sensitive time complexity of $O(n \log h)$, where n is the number of input points and h is the number of points on the convex hull. The core idea is to iteratively "guess" the hull size and combine the strengths of divide-and-conquer and gift-wrapping approaches. By carefully balancing these phases, the algorithm efficiently adapts to the actual output size.

Mathematical Insight: The $O(n \log h)$ complexity arises as follows: For each guess $m \geq h$, the algorithm spends $O(n \log m)$ time computing mini-hulls (using a method like Graham's scan) and $O(m \cdot n/m) = O(n)$ time in the gift-wrapping phase to merge them. Since m doubles each iteration, the total work across all guesses is $O(n \log h)$. This makes Chan's algorithm the first to match the lower bound for output-sensitive convex hull algorithms.

Listing 3.3: Chan's Algorithm

```cpp
vector<Point> chan_algorithm(vector<Point> points) {
    int n = points.size();
    if (n <= 3) return points;

    // Try increasing values of m
    for (int m = 3; m <= n; m *= 2) {
        vector<Point> hull = chan_step(points, m);
        if (!hull.empty()) return hull;
    }
    return {};
}

vector<Point> chan_step(vector<Point>& points, int m) {
    int n = points.size();
    vector<vector<Point>> groups;
```

```

    // Partition into groups of size m
    for (int i = 0; i < n; i += m) {
        vector<Point> group;
        for (int j = i; j < min(i + m, n); j++) {
            group.push_back(points[j]);
        }
        // Compute convex hull of each group
        groups.push_back(monotone_chain(group));
    }

    // Gift wrapping on the groups
    Point start = *min_element(points.begin(), points.end(),
        [](const Point& a, const Point& b) {
            return a.y < b.y || (a.y == b.y && a.x < b.x);
        });

    vector<Point> hull;
    hull.push_back(start);

    Point current = start;
    for (int step = 0; step < m; step++) {
        Point next = groups[0][0];

        // Find the next hull point across all groups
        for (const auto& group : groups) {
            Point candidate = find_tangent(current, group);
            if (hull.size() == 1 ||
                ((candidate - current) ^ (next - current)) > 0) {
                next = candidate;
            }
        }

        if (next == start) return hull;  // Completed the hull
        hull.push_back(next);
        current = next;
    }

    return {};  // Hull has more than m points
}

Point find_tangent(const Point& p, const vector<Point>& hull) {
    int n = hull.size();
    if (n == 1) return hull[0];
    if (n == 2) {
        // Return the more "right" tangent
        return ((hull[0] - p) ^ (hull[1] - p)) < 0 ? hull[0] : hull[1];
    }

    auto is_right_turn = [&](int i, int j) {
        // True if p is to the right of the directed edge hull[i] -> hull[j]
        return ((hull[j] - hull[i]) ^ (p - hull[i])) < 0;
    };

    int low = 0, high = n;
    while (low < high) {
        int mid = (low + high) / 2;
        int prev = (mid - 1 + n) % n;
        int next = (mid + 1) % n;

        bool mid_right = is_right_turn(mid, next);
        bool prev_right = is_right_turn(prev, mid);

        if (!mid_right && prev_right) {
            // Found the tangent
            return hull[mid];
        }
        // If both are right turns, move right
        if (mid_right) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return hull[low % n];
}
```

Complexity Analysis

Time complexity: $O(n \log h)$, where n is the number of input points and h is the number of points on the convex hull.

Note: This is the optimal output-sensitive algorithm for the convex hull problem.

3.2.4 Divide-and-Conquer Convex Hull

Algorithm 20: Divide-and-Conquer Convex Hull

Input: A set P of n points in the plane
Output: The convex hull of P

if $n \leq 3$ **then**
  return *the trivial convex hull of* P
end
Sort points by x-coordinate
Divide P into left half P_L and right half P_R
Recursively compute the convex hull H_L of P_L
Recursively compute the convex hull H_R of P_R
Merge H_L and H_R to obtain the convex hull of P

Intuition: The divide-and-conquer approach recursively splits the problem into smaller subproblems, solves them independently, and then combines the results. The key challenge is the merge step, where we need to find the upper and lower "bridges" connecting the two sub-hulls.

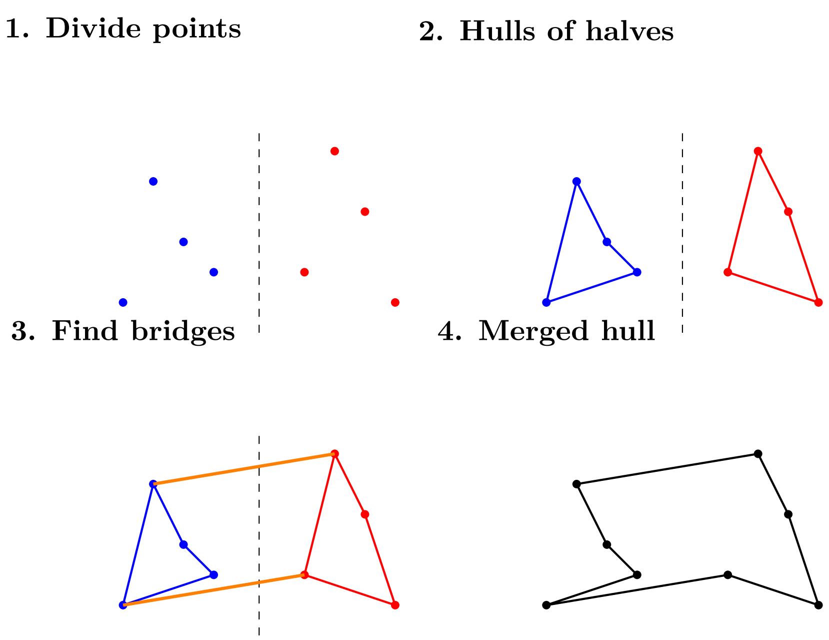
Listing 3.4: Divide-and-Conquer Convex Hull

```
vector<Point> divide_and_conquer_hull(vector<Point> points) {
    int n = points.size();
    if (n <= 5) {
        // Base case: use Graham scan or brute force for small sets
        return graham_scan(points);
    }

    // Sort points by x-coordinate
    sort(points.begin(), points.end());

    // Divide
    int mid = n / 2;
    vector<Point> left_points(points.begin(), points.begin() + mid);
    vector<Point> right_points(points.begin() + mid, points.end());

    // Conquer
    vector<Point> left_hull = divide_and_conquer_hull(left_points);
    vector<Point> right_hull = divide_and_conquer_hull(right_points);

    // Merge - find upper and lower tangent
    return merge_hulls(left_hull, right_hull);
}

vector<Point> merge_hulls(const vector<Point>& left_hull, const vector<Point>& right_hull) {
    int nl = left_hull.size(), nr = right_hull.size();
    if (nl == 0) return right_hull;
    if (nr == 0) return left_hull;

    // Find rightmost point of left hull
    int right_l = 0;
    for (int i = 1; i < nl; i++) {
        if (left_hull[i].x > left_hull[right_l].x) right_l = i;
    }

    // Find leftmost point of right hull
    int left_r = 0;
    for (int i = 1; i < nr; i++) {
        if (right_hull[i].x < right_hull[left_r].x) left_r = i;
    }

    // Find upper tangent
    int upper_l = right_l, upper_r = left_r;
    bool done = false;
    while (!done) {
        done = true;
        while (orientation(right_hull[upper_r], left_hull[upper_l],
                           left_hull[(upper_l+1)%nl]) >= 0)
```

```
            upper_l = (upper_l + 1) % nl;

        while (orientation(left_hull[upper_l], right_hull[upper_r],
                          right_hull[(upper_r-1+nr)%nr]) <= 0) {
            upper_r = (upper_r - 1 + nr) % nr;
            done = false;
        }
    }

    // Find lower tangent
    int lower_l = right_l, lower_r = left_r;
    done = false;
    while (!done) {
        done = true;
        while (orientation(left_hull[lower_l], right_hull[lower_r],
                          right_hull[(lower_r+1)%nr]) >= 0)
            lower_r = (lower_r + 1) % nr;

        while (orientation(right_hull[lower_r], left_hull[lower_l],
                          left_hull[(lower_l-1+nl)%nl]) <= 0) {
            lower_l = (lower_l - 1 + nl) % nl;
            done = false;
        }
    }

    // Construct the merged hull
    vector<Point> merged_hull;
    int i = upper_l;
    do {
        merged_hull.push_back(left_hull[i]);
        i = (i + 1) % nl;
    } while (i != lower_l);

    i = lower_r;
    do {
        merged_hull.push_back(right_hull[i]);
        i = (i + 1) % nr;
    } while (i != upper_r);

    return merged_hull;
}
```

Complexity Analysis

$O(n \log n)$ time, which is optimal in the comparison model.



Implementation Notes: The merge step requires finding the two tangent lines between the left and right hulls. This can be done in linear time by walking along the hulls, similar to the merge step in mergesort. Pay careful attention to the handling of the upper and lower bridges to ensure the final hull is correctly constructed.

3.2.5 Rotating Calipers Technique & Applications

Listing 3.5: Rotating Calipers

```
int64_t convex_polygon_diameter2(const vector<Point>& hull) {
    int n = hull.size();
    if (n <= 1) return 0;
    if (n == 2) return (hull[1] - hull[0]).norm2();

    int64_t max_dist2 = 0;
    int j = 1;

    for (int i = 0; i < n; i++) {
        // Advance j to find the furthest point from edge (i, i+1)
        while (true) {
            int next_j = (j + 1) % n;
            Point edge = hull[(i + 1) % n] - hull[i];
            Point to_j = hull[j] - hull[i];
            Point to_next_j = hull[next_j] - hull[i];

            // Check if next_j is further from the edge than j
            if ((edge ^ to_next_j) >= (edge ^ to_j)) {
                j = next_j;
            } else {
                break;
            }
        }

        // Update maximum distance
        max_dist2 = max(max_dist2, (hull[i] - hull[j]).norm2());
        max_dist2 = max(max_dist2, (hull[(i + 1) % n] - hull[j]).norm2());
    }

    return max_dist2;
}
```

Complexity Analysis

$O(n)$ time.

Invariant: j advances monotonically around the polygon

Insight

The rotating calipers technique is remarkably versatile. Beyond finding the diameter and width of a convex polygon, it can be used to:

- Find all antipodal pairs of vertices in linear time
- Compute the minimum-area enclosing rectangle
- Find the maximum-area/perimeter inscribed triangle/quadrilateral
- Determine the minimum-area/perimeter enclosing triangle
- Compute the minimum-width annulus containing all points

All these problems can be solved in $O(n)$ time after computing the convex hull, making rotating calipers one of the most powerful techniques in computational geometry.

Definition 3.2.1 (Polygon Diameter)**.** The diameter of a convex polygon is the maximum distance between any two points on the polygon. For a convex polygon, this maximum always occurs between two vertices.

Algorithm 21: Finding Polygon Diameter Using Rotating Calipers

Input: A convex polygon P with vertices in counterclockwise order

Output: The squared diameter of P

1. Find the indices i_{min} and i_{max} of the leftmost and rightmost vertices
2. Initialize $j = i_{max}$
3. Initialize $maxDist^2 = 0$
4. **for** *each vertex i from i_{min} to $i_{min} + n$* **do**
5.   **while** *area of triangle formed by vertices i, $i+1$, and $j+1$ > area of triangle formed by vertices i, $i+1$, and j* **do**
6. $j = (j+1) \bmod n$
7.   **end**
8.   Update $maxDist^2 = \max(maxDist^2, dist^2(vertices[i], vertices[j]))$
9.   Update $maxDist^2 = \max(maxDist^2, dist^2(vertices[i+1], vertices[j]))$
10. **end**
11. **return** $maxDist^2$

Insight

The key insight of this algorithm is that the diameter of a convex polygon must occur between two vertices that form an antipodal pair—vertices that admit parallel supporting lines. The rotating calipers technique efficiently finds all such pairs in linear time.

3.2.5.1 Finding Polygon Width

Listing 3.6: Finding Polygon Width

```cpp
double convex_polygon_width(const vector<Point>& hull) {
    int n = hull.size();
    if (n <= 2) return 0;

    double min_width = 1e18;
    int j = 1;

    for (int i = 0; i < n; i++) {
        Point edge = hull[(i + 1) % n] - hull[i];
        double edge_len = edge.norm();

        // Find the vertex furthest from edge (i, i+1)
        while (true) {
            int next_j = (j + 1) % n;
            double dist_j = abs((hull[j] - hull[i]) ^ edge) / edge_len;
            double dist_next_j = abs((hull[next_j] - hull[i]) ^ edge) / edge_len;

            if (dist_next_j > dist_j) {
                j = next_j;
            } else {
                break;
            }
        }

        // Width is the distance from vertex j to edge (i, i+1)
        double width = abs((hull[j] - hull[i]) ^ edge) / edge_len;
        min_width = min(min_width, width);
    }

    return min_width;
}
```

Complexity Analysis

$O(n)$ time.

Given a convex polygon, find the triangle of maximum area (or perimeter) whose vertices are vertices of the polygon.

Intuition: For any fixed base (an edge of the polygon), the area of a triangle increases as the third vertex moves farther from the base. Using rotating calipers, we can efficiently find the farthest vertex for each possible base.

Algorithm 22: Maximum Area Triangle in Convex Polygon

Input: A convex polygon P with vertices in counterclockwise order
Output: The maximum area triangle formed by three vertices of P

```
Initialize maxArea = 0, bestTriangle = ∅
for each edge (v_i, v_{i+1}) in P do
    Initialize j to be the vertex farthest from the line through v_i and v_{i+1}
    for each vertex v_k from v_i to v_{i+1} do
        while area of triangle (v_k, v_{k+1}, v_{j+1}) > area of triangle (v_k, v_{k+1}, v_j) do
            j = (j + 1) mod n
        end
        area = area of triangle (v_k, v_{k+1}, v_j)
        if area > maxArea then
            maxArea = area
            bestTriangle = (v_k, v_{k+1}, v_j)
        end
    end
end
return bestTriangle
```

Implementation Notes: For the maximum perimeter triangle, the approach is similar, but we use the perimeter calculation instead of area. Interestingly, the maximum area and maximum perimeter triangles are not necessarily the same.

Complexity Analysis

Both algorithms run in $O(n)$ time where n is the number of vertices in the polygon.

Given two convex polygons P and Q, compute their intersection efficiently.

Intuition: We can compute the intersection of two convex polygons by using a linear-time algorithm that exploits their convexity. The algorithm walks around both polygons simultaneously, always advancing along the polygon that currently "lags behind" the other.

Algorithm 23: Convex Polygon Intersection

```
Input: Two convex polygons P and Q with vertices in counterclockwise order
Output: The intersection polygon R
if P or Q is empty then
    return empty polygon
end
if P is contained in Q then
    return P
end
if Q is contained in P then
    return Q
end
Initialize result polygon R = ∅
Find a vertex of P inside Q (or vice versa)
Set starting position at this vertex
repeat
    Add current position to R
    Compute the next intersection of the boundaries of P and Q
    Move to this intersection
until we return to the starting position
return R
```

Implementation Notes: The actual implementation is more complex, as we need to carefully handle various edge cases:

- Finding the initial vertex that lies in both polygons
- Detecting when the polygons don't intersect
- Handling the case where one polygon is entirely inside the other
- Dealing with edges that overlap exactly

Complexity Analysis

The algorithm runs in $O(n+m)$ time, where n and m are the numbers of vertices in the two polygons.

Given a convex polygon, find the minimum-area (or minimum-perimeter) rectangle that encloses it.

Insight

A key insight is that the minimum-area enclosing rectangle must have at least one side flush with an edge of the convex polygon. This allows us to use rotating calipers to consider only $O(n)$ potential rectangles.

Algorithm 24: Minimum Area Enclosing Rectangle

Input: A convex polygon P with vertices in counterclockwise order
Output: The minimum-area rectangle enclosing P

Initialize four support lines (left, right, top, bottom)
Initialize $minArea = \infty$, $bestRect = \emptyset$
for *each edge* e *of* P **do**
    Update the support lines to be parallel/perpendicular to e and touching P
    Compute the area of the resulting rectangle
    if $area < minArea$ **then**
        $minArea = area$
        $bestRect =$ current rectangle
    end
    Determine which support line should advance next
    Advance that support line to the next vertex
end
return $bestRect$

Create a sequence of diagrams showing: 1. A convex polygon with a rectangle enclosing it 2. The rectangle rotating to align with different edges of the polygon 3. The support lines advancing during rotation 4. The final minimum-area rectangle

Implementation Notes: When implementing this algorithm, it's important to:

- Correctly determine which support line to advance at each step
- Handle parallel edges in the polygon
- Account for numerical precision issues when computing areas

Complexity Analysis

The algorithm runs in $O(n)$ time where n is the number of vertices in the polygon.

Recognizing When to Apply Convex Hull
A problem likely requires convex hull when:

- It involves finding the extreme points of a set in a plane
- It asks for the smallest enclosing shape of a set of points
- There's a need to discard "interior" points that don't contribute to a solution
- The problem mentions finding the "boundary" or "outline" of a point set
- There's a geometric optimization problem involving a point set (finding maximum distance, minimum enclosing area, etc.)
- The problem involves line of sight, visibility, or covering

Convex Hull + Rotating Calipers Pattern This powerful combination is especially useful when:

- Finding the diameter (maximum distance between any two points)
- Finding the width (minimum distance between parallel supporting lines)
- Computing the minimum-area or minimum-perimeter enclosing rectangle
- Finding all antipodal pairs of vertices (pairs that admit parallel supporting lines)

When to Choose Each Algorithm

- **Graham Scan**: General-purpose, easy to implement, good when the entire hull is needed
- **Monotone Chain**: More numerically stable, suitable when points can be sorted by x-coordinate
- **Chan's Algorithm**: When the output size might be much smaller than the input size
- **Divide-and-Conquer**: When parallelization is available or for educational purposes
- **Rotating Calipers**: When you need to solve a post-hull optimization problem

3.3 Precision Gotchas

Warning

Convex hull algorithms are particularly sensitive to numerical precision issues due to their geometric nature. Common issues include:

- Incorrectly determining if three points are collinear
- Errors in orientation tests leading to non-convex "hulls"
- Sorting issues when points have similar polar angles

Implementation Notes: For competitive programming, consider these approaches to mitigate precision issues:

- Use integer coordinates with cross products rather than angles
- Add a small epsilon when comparing floating-point values
- Implement robust predicates for critical geometric operations

3.3.1 convex_hull_monotone_chain

Maecenas non massa. Vestibulum pharetra nulla at lorem. Duis quis quam id lacus dapibus interdum. Nulla lorem. Donec ut ante quis dolor bibendum condimentum. Etiam egestas tortor vitae lacus. Praesent cursus. Mauris bibendum pede at elit. Morbi et felis a lectus interdum facilisis. Sed suscipit gravida turpis. Nulla at lectus. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Praesent nonummy luctus nibh. Proin turpis nunc, congue eu, egestas ut, fringilla at, tellus. In hac habitasse platea dictumst.

3.3.2 rotating_calipers_diameter

Vivamus eu tellus sed tellus consequat suscipit. Nam orci orci, malesuada id, gravida nec, ultricies vitae, erat. Donec risus turpis, luctus sit amet, interdum quis, porta sed, ipsum. Suspendisse condimentum, tortor at egestas posuere, neque metus tempor orci, et tincidunt urna nunc a purus. Sed facilisis blandit tellus. Nunc risus sem, suscipit nec, eleifend quis, cursus quis, libero. Curabitur et dolor. Sed vitae sem. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Maecenas ante. Duis ullamcorper enim. Donec tristique enim eu leo. Nullam molestie elit eu dolor. Nullam bibendum, turpis vitae tristique gravida, quam sapien tempor lectus, quis pretium tellus purus ac quam. Nulla facilisi.

3.3.3 rotating_calipers_min_enclosing_rectangle

Duis aliquet dui in est. Donec eget est. Nunc lectus odio, varius at, fermentum in, accumsan non, enim. Aliquam erat volutpat. Proin sit amet nulla ut eros consectetuer cursus. Phasellus dapibus aliquam justo. Nunc laoreet. Donec consequat placerat magna. Duis pretium tincidunt justo. Sed sollicitudin vestibulum quam. Nam quis ligula. Vivamus at metus. Etiam imperdiet imperdiet pede. Aenean turpis. Fusce augue velit, scelerisque sollicitudin, dictum vitae, tempor et, pede. Donec wisi sapien, feugiat in, fermentum ut, sollicitudin adipiscing, metus.

When your convex hull algorithm isn't working correctly, check these common issues:

Debug Checklist: Convex Hull Algorithms

- **Orientation test**: Verify your cross product calculation is correct and handles numerical precision issues.
- **Sorting**: For Graham Scan, check if points are sorted correctly by polar angle.
- **Initial point selection**: Verify that your algorithm correctly identifies the point to start from (lowest y-coordinate for Graham Scan).
- **Collinear points**: Test your algorithm on sets of three or more collinear points.
- **Edge cases**: Test with inputs of size 0, 1, 2, and 3.
- **Stack operations**: For stack-based algorithms, ensure push/pop operations maintain the correct hull.
- **Visualization**: Draw the intermediate steps to identify where your algorithm deviates from the expected behavior.
- **Orientation consistency**: Make sure you're consistently checking for clockwise or counterclockwise orientation throughout your code.
- **Degeneracy handling**: Check that your code correctly handles degenerate cases like all points being collinear.

3.3.4 Algorithms by Sedgewick & Wayne

Cras dapibus, augue quis scelerisque ultricies, felis dolor placerat sem, id porta velit odio eu elit. Aenean interdum nibh sed wisi. Praesent sollicitudin vulputate dui. Praesent iaculis viverra augue. Quisque in libero. Aenean gravida lorem vitae sem ullamcorper cursus. Nunc adipiscing rutrum ante. Nunc ipsum massa, faucibus sit amet, viverra vel, elementum semper, orci. Cras eros sem, vulputate et, tincidunt id, ultrices eget, magna. Nulla varius ornare odio. Donec accumsan mauris sit amet augue. Sed ligula lacus, laoreet non, aliquam sit amet, iaculis tempor, lorem. Suspendisse eros. Nam porta, leo sed congue tempor, felis est ultrices eros, id mattis velit felis non metus. Curabitur vitae elit non mauris varius pretium. Aenean lacus sem, tincidunt ut, consequat quis, porta vitae, turpis. Nullam laoreet fermentum urna. Proin iaculis lectus.

3.3.5 TopCoder SRM 2 Convex Hull Problem Tutorial

Sed mattis, erat sit amet gravida malesuada, elit augue egestas diam, tempus scelerisque nunc nisl vitae libero. Sed consequat feugiat massa. Nunc porta, eros in eleifend varius, erat leo rutrum dui, non convallis lectus orci ut nibh. Sed lorem massa, nonummy quis, egestas id, condimentum at, nisl. Maecenas at nibh. Aliquam et augue at nunc pellentesque ullamcorper. Duis nisl nibh, laoreet suscipit, convallis ut, rutrum id, enim. Phasellus odio. Nulla nulla elit, molestie non, scelerisque at, vestibulum eu, nulla. Ut odio nisl, facilisis id, mollis et, scelerisque nec, enim. Aenean sem leo, pellentesque sit amet, scelerisque sit amet, vehicula pellentesque, sapien.

3.3.6 Fully Dynamic Convex Hull Maintenance

Sed consequat tellus et tortor. Ut tempor laoreet quam. Nullam id wisi a libero tristique semper. Nullam nisl massa, rutrum ut, egestas semper, mollis id, leo. Nulla ac massa eu risus blandit mattis. Mauris ut nunc. In hac habitasse platea dictumst. Aliquam eget tortor. Quisque dapibus pede in erat. Nunc enim. In dui nulla, commodo at, consectetuer nec, malesuada nec, elit. Aliquam ornare tellus eu urna. Sed nec metus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

3.3.7 Quickhull Algorithm

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.

1. Implement a function that determines if a point lies inside a convex polygon in $O(\log n)$ time using binary search. Test it on various cases including points on the boundary.

2. Modify the Graham Scan algorithm to handle collinear points by including only the extreme points on each line segment.

3. Implement Chan's algorithm and compare its performance against Graham Scan on various inputs, particularly when the output hull size is much smaller than the input size.

4. Using the rotating calipers technique, implement an algorithm that finds the smallest circle containing a set of points. Hint: The circle must be defined by either two or three points on the convex hull.

5. Given a convex polygon, find the triangle with maximum perimeter whose vertices are vertices of the polygon. Is it always formed by three consecutive vertices? Prove or provide a counterexample.

6. Implement an algorithm that dynamically maintains the convex hull as points are added one by one. What is the time complexity for adding m points to an existing hull with n vertices?

7. Design an algorithm that finds the largest empty circle among a set of points (i.e., a circle that contains no points and has its center inside the convex hull).

8. The width of a convex polygon is the minimum distance between any two parallel supporting lines. Prove that this minimum is attained when one of the lines contains an edge of the polygon.

9. For a set of n points, the convex layers (or onion peeling) are defined by repeatedly computing the convex hull and removing its vertices. Implement an algorithm to compute all convex layers of a point set. What is the worst-case time complexity?

10. Research and implement Fortune's algorithm for computing the Voronoi diagram of a set of points. How can convex hulls be used in this context?

Key Definitions:

- **Convex Set**: A set where for any two points, the line segment connecting them lies entirely in the set (Section 3.1.1)

- **Convex Combination**: A weighted average of points with non-negative weights summing to 1 (Section 3.1.1)

- **Convex Hull**: Smallest convex set containing all given points (Section 3.1.2)

- **Antipodal Pair**: Pair of points on a convex polygon admitting parallel supporting lines (Section 3.2.5.1)
- **Supporting Line**: A line touching the convex hull with all points of the hull on one side (Section 3.1.3)

Core Algorithms:

- **Graham Scan**: Sort by polar angle, build hull using a stack. $O(n \log n)$ (Section 3.2.1)
- **Monotone Chain**: Sort by x-coordinate, build upper/lower hulls. $O(n \log n)$ (Section 3.2.2)
- **Chan's Algorithm**: Optimally combines divide-and-conquer with gift wrapping. $O(n \log h)$ (Section 3.2.3)
- **Divide-and-Conquer**: Split, recurse, merge with tangent lines. $O(n \log n)$ (Section 3.2.4)
- **Rotating Calipers**: Simulate rotating parallel lines around hull. $O(n)$ (Section 3.2.5)

Critical Gotchas:

- Handle collinear points correctly (include only extremes)
- Check orientation test for numerical stability
- Special cases: fewer than 3 points, all collinear points
- Ensure consistent clockwise/counterclockwise orientation
- For floating-point coordinates, use appropriate epsilon values

When to Use What:

- **Graham Scan**: Standard approach, widely applicable
- **Monotone Chain**: When numerical stability is crucial
- **Chan's Algorithm**: When output size h « input size n
- **Rotating Calipers**: For post-hull optimization (diameter, width, min-area rectangle)
- **Divide-and-Conquer**: When parallelization is needed

Bibliography

[1] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars. *Computational Geometry: Algorithms and Applications (3rd ed.)*. Springer-Verlag, 2008.

[2] CP-Algorithms. *Basic Geometry*. Retrieved from https://cp-algorithms.com/geometry/basic-geometry.html

[3] J. E. Goodman, J. O'Rourke (Eds.). *Handbook of Discrete and Computational Geometry (3rd ed.)* (Chapter on Polygons). CRC Press, 2017.

[4] TopCoder Tutorial. *Geometry Concepts Part 2 - Polygons, Area, Centroid, Pick's Theorem*. (URL placeholder: Search TopCoder tutorials, e.g., https://community.topcoder.com/tc?module=Static&d1=tutorials&d2=geometry2)

[5] C. Ericson. *Real-Time Collision Detection* (Chapters on intersection tests, SAT). Morgan Kaufmann, 2005.

[6] Nifty Game Dev. *Separating Axis Theorem (SAT) Explained*. (URL placeholder: https://www.niftygamedevelopment.com/separating-axis-theorem-sat-explained/ or similar visual explanation)

[7] R. Sedgewick, K. Wayne. *Algorithms (4th ed.)* (Section on Convex Hulls). Addison-Wesley Professional, 2011.

[8] TopCoder SRM Convex Hull Problem Tutorial (e.g., SRM 202). (URL placeholder: Search TopCoder SRM editorials or tutorials on Convex Hulls)

[9] KTH Algorithm Competition Template Library (KACTL). *LineContainer.h*. Retrieved from https://github.com/kth-competitive-programming/kactl/blob/main/content/data-structures/LineContainer.h

[10] G. Šinkevičius. *Convex Hull Trick* (PEGWiki or CMU CS). Retrieved from (e.g., https://www.cs.cmu.edu/~ginsca/convex_hull_trick.txt or common PEGWiki mirrors)

[11] maspypy. (Relevant blog post on Slope Trick). (URL placeholder: Search maspypy's blog, e.g., https://maspypy.github.io/)

[12] olphe. (Relevant blog post on Slope Trick). (URL placeholder: Search olphe's blog, e.g., https://olphe1.github.io/)

[13] USACO Guide. *Slope Trick*. Retrieved from https://usaco.guide/adv/slope

[14] M. L. Fredman. (1979). "The Gaps Between Consecutive Primes and Related Problems." (More appropriate reference for QI/Knuth opt: Yao, F. F. (1980). Efficient dynamic programming using quadrangle inequalities. STOC '80.)

[15] D. E. Knuth. (1971). "Optimum binary search trees." *Acta Informatica*.

[16] CP-Algorithms. *Knuth's Optimization.* Retrieved from https://cp-algorithms.com/dynamic_programming/knuth-optimization.html

[17] M. de Berg, et al. *Computational Geometry: Algorithms and Applications (3rd ed.)* (Chapter 2: Line Segment Intersection, Chapter 10: More Geometric Data Structures). (See Item 1)

[18] Stanford CS166 Notes. *Sweep Algorithms.* (URL placeholder: Search for Stanford CS166 lecture notes on sweep-line algorithms)

[19] H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, 2006.

[20] KTH Algorithm Competition Template Library (KACTL). (e.g., KDTree.h). (URL placeholder: https://github.com/kth-competitive-programming/kactl/tree/main/content/data-structures)

[21] N. Megiddo. (Various papers on Parametric Search, e.g., "Applying parallel computation algorithms in the design of serial algorithms" (1983)). (URL placeholder: Search for Nimrod Megiddo's papers on parametric search)

[22] TopCoder SRM Editorials. (Problems solvable by parametric search). (General reference, specific SRM editorials would be better if known)

[23] Blog posts on "Alien Trick" (e.g., by scott_wu, pashka). (URL placeholder: Search for "Alien Trick competitive programming blog")

[24] M. de Berg, et al. *Computational Geometry: Algorithms and Applications (3rd ed.)* (Chapters on Arrangements, Duality, Voronoi, Delaunay). (See Item 1)

[25] J. O'Rourke. (1998). *Computational Geometry in C (2nd ed.).* Cambridge University Press.