Competitive Programming

From Problem 2 Solution in O(1)

Computational Geometry

Point in Polygon

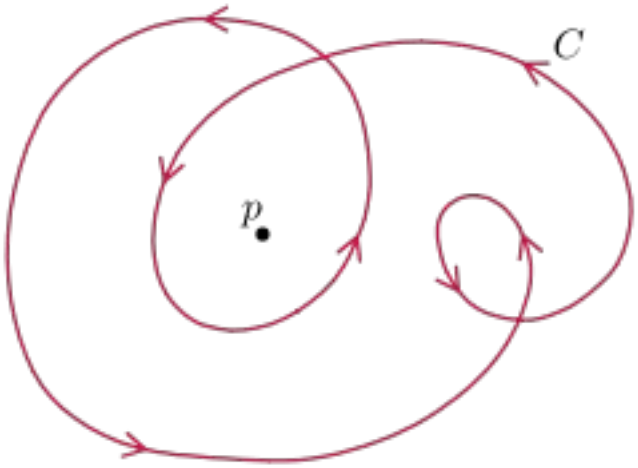
Mostafa Saad Ibrahim
PhD Student @ Simon Fraser University

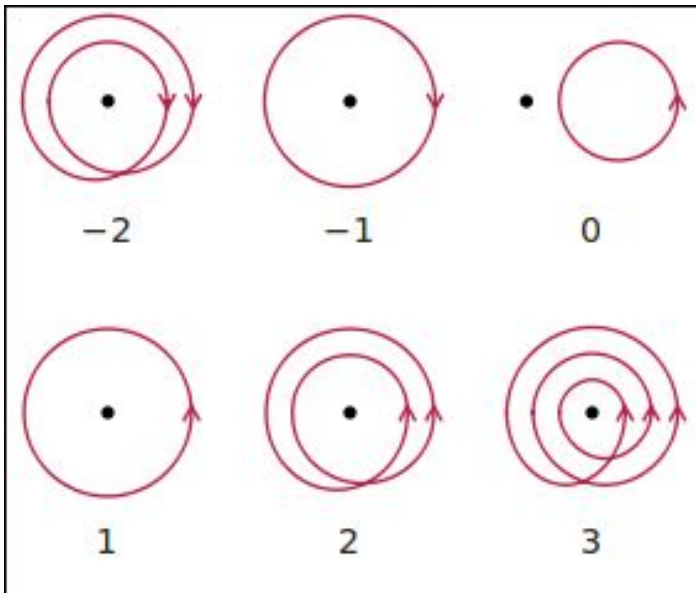
Is point in polygon (pip)?

- Given point and polygon, is it inside it?
  - Lots of Materials: See1, See2, See3, See4
- **Winding number algorithm**
  - One of most efficient and accurate algorithms
  - Originally was based on heavy ***Angles Summation Test***
  - Then a more efficient way is invented
- Others [Not covered]
  - Crossing number algorithm

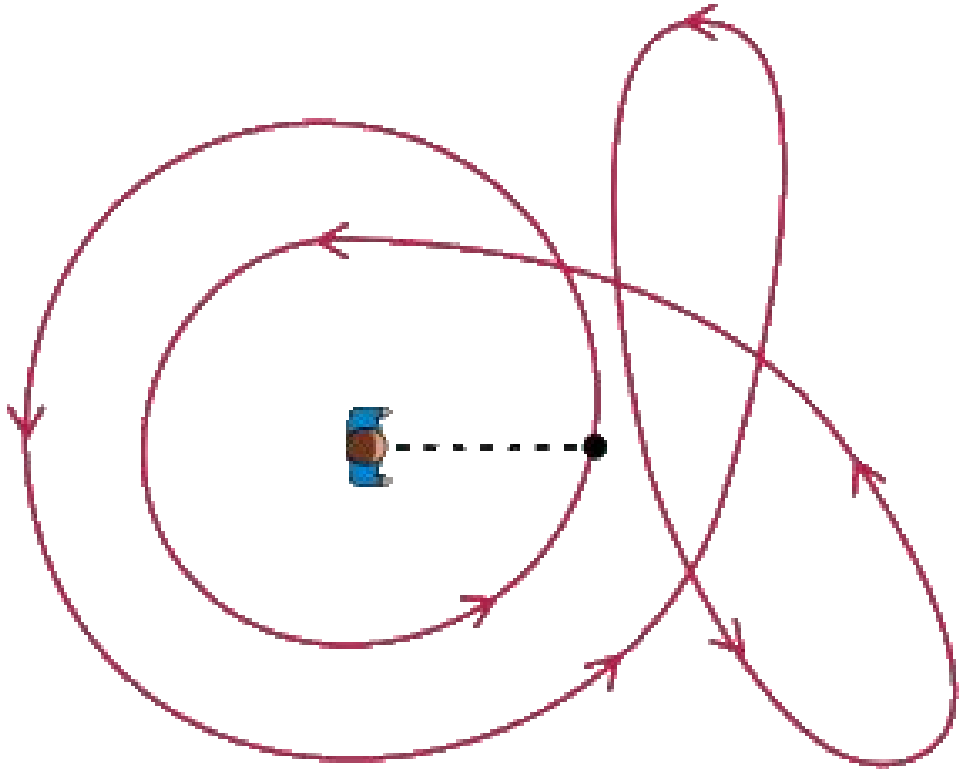
The winding number

- The total number of times that **curve/polygon** winds/travels around the point
  - counterclockwise motion +1
  - clockwise motion -1





The winding number



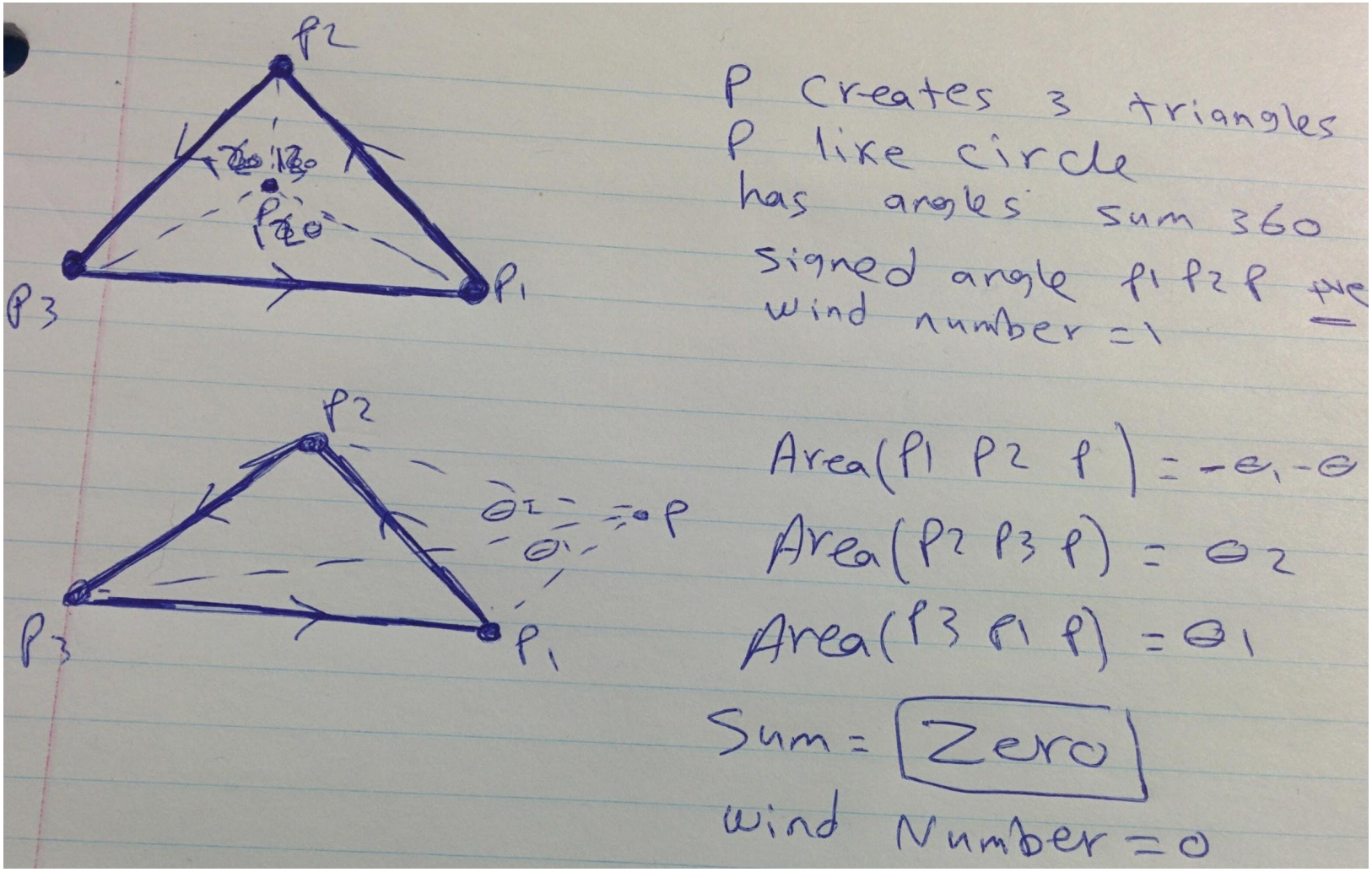
Src: https://en.wikipedia.org/wiki/Winding_number

The winding number

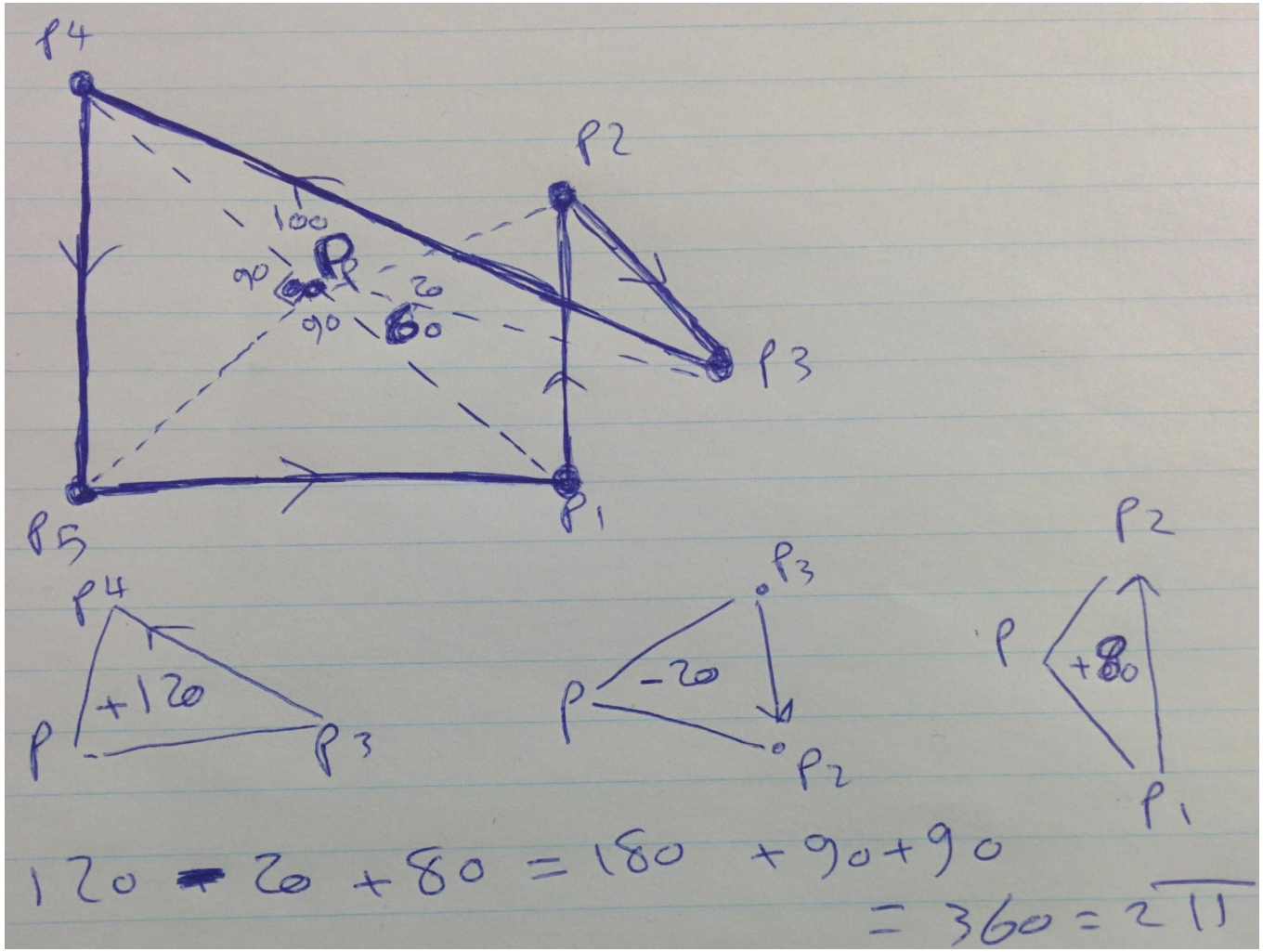
- Similar to iterating on edges to count rounds, one can iterate to sum the signed angles of point p with the polygon edges
  - Think in **signed angles** similar to computing Polygon area
  - E.g. areas cancellations similar to cancelling angles
- The summation will be either
  - 0 if point outside polygon
  - Multiple of 2PI if inside polygon

$$\mathbf{wn}(P,\mathbf{C}) = \frac{1}{2\pi}\sum_{i=0}^{n-1}\theta_i$$

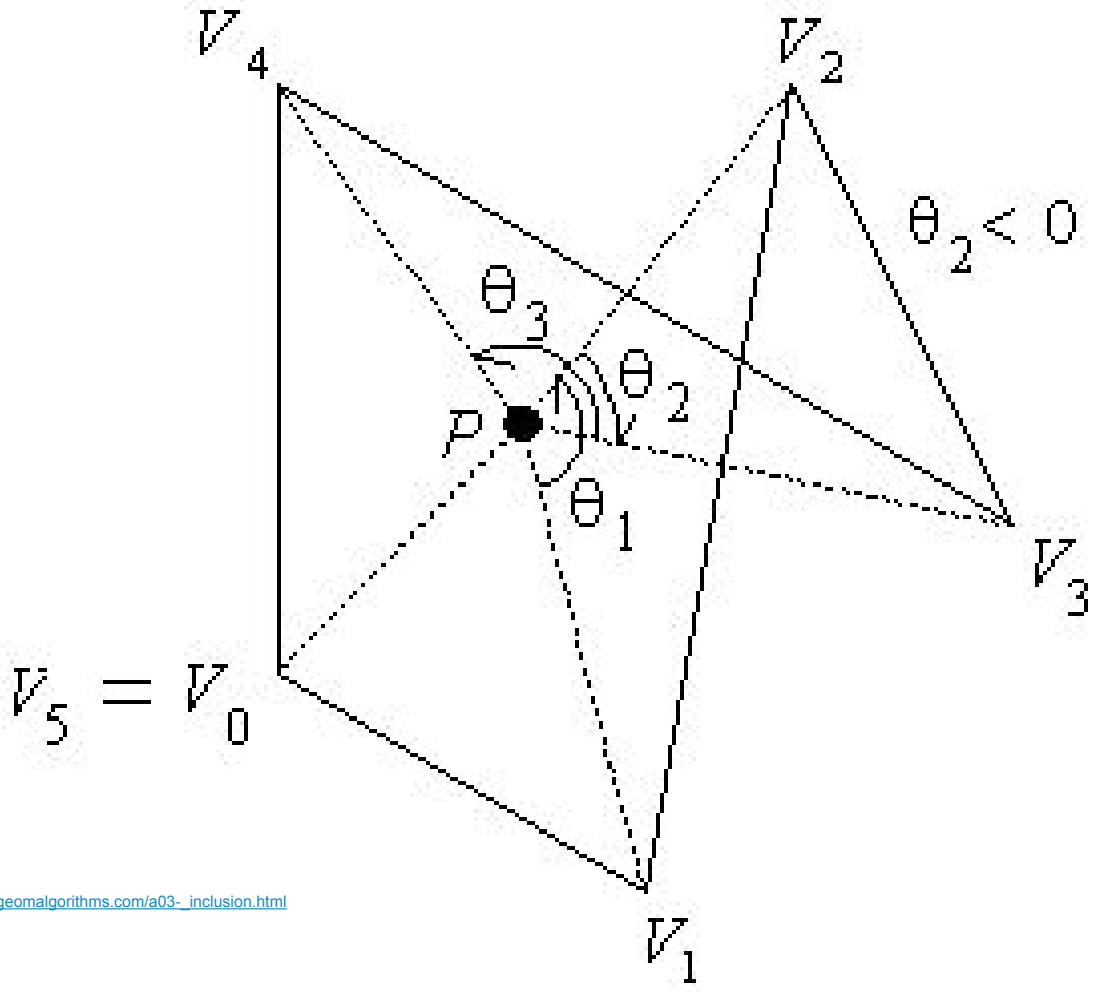
The winding number



The winding number



The winding number



Src: http://geomalgorithms.com/a03-_inclusion.html

The winding number: angle

- Computing Min Angle between 2 vectors?
  - Dot product theta is simply that

```
double fixAngle(double A) {
  return A > 1 ? 1 : (A < -1 ? -1 : A);
}

// return min angle: aOb / bOa
// dp(v1, v2) = |v1|*|v2|*cos(theta)
double angleO(point a, point O, point b) {
  point v1(a - O), v2(b - O);
  return acos( fixAngle ( dp(v1, v2) / length(v1) / length(v2) ) );
}
```

- Or directly compute signed angle
  - It uses 2 atan calls

The winding number: algorithm

```
// Easy idea, multiple precision concerns
bool isInsidePoly(vector<point> &p, point pt) {
  double anglesSum = 0;

  p.push_back(p[0]);
  for (int i = 0; i < sz(p) - 1; i++) {
    if (ccw(p[i], p[i + 1], pt) == 0) { // is pt on segment pi-pj?
      p.pop_back();
      return true;
    }
    anglesSum += angle0(p[i], pt, p[i + 1]) *
        ccw(pt, p[i], p[i + 1]); // angle pt-pi-pj is ccw?
  }
  p.pop_back();
  // Answer is either 0 (outside) or 2PI (inside)
  // return fabs(fabs(anglesSum) - 2 * PI) < EPS;
  return fabs(anglesSum) > PI;  // let's avoid EPS comparison :)
}
```

Note on the angles based style

- Easy idea, need careful implementation
  - In dp style, if did not fix angle before acos ⇒ nan
  - Be careful: 2 ccw calls with different points arrangement
  - Mistake: anglesSum shouldn't be initialized with 2PI
- Slow and precision concerns
  - Our code has much angles +/-
  - A very small EPS can causes WA (e.g. 1e-15 vs use **1e-8**)
  - But, Comparing against PI resolves EPS problem
- Summary
  - If coded correctly, the actual problem is being **slow**
  - Good lesson for how easy geom idea can be code killer

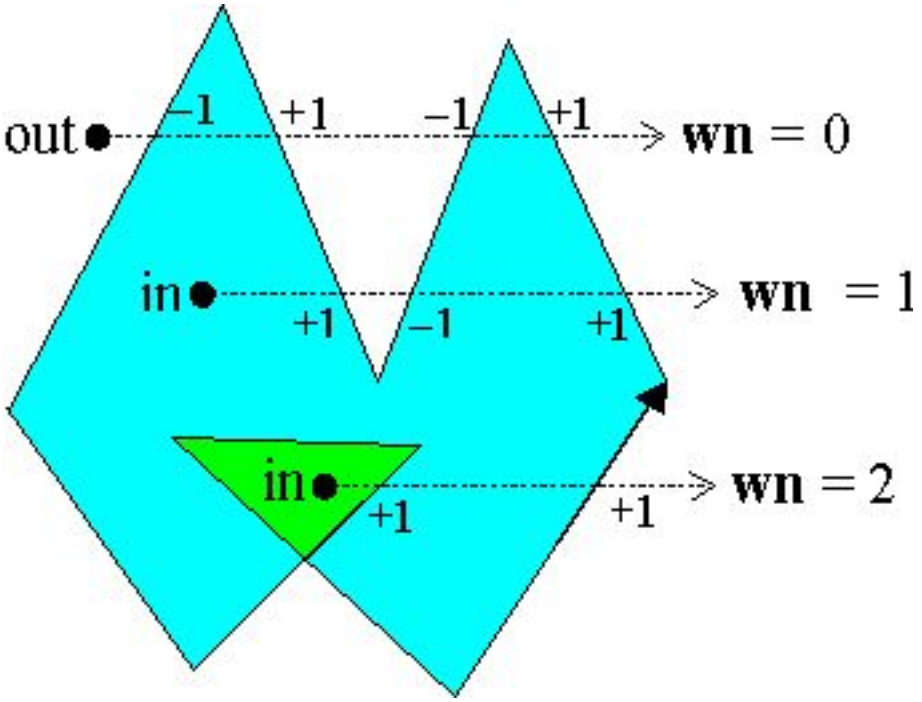
Equivalent Approach

- Draw horizontal ray from p to +ve x-axis
- Count signed edges. 0 means outside
- Special case
  - the horizontal line

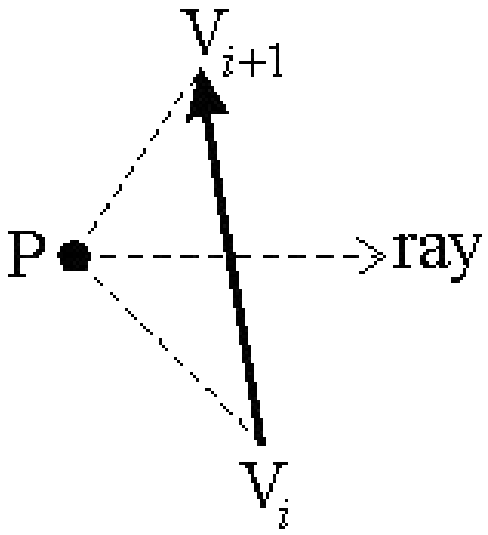
    with horizontal edge

    so test on edge

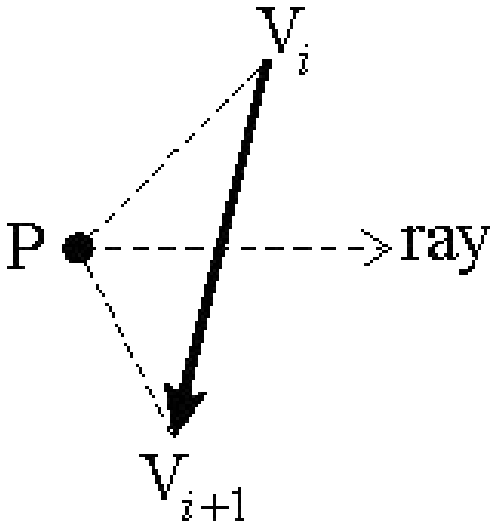
    independently



How to check up/down ward?



upward edge



downward edge

- Using P.Y verse v1.Y can help know if it upward or downward
- Make sure using the V2.Y to make sure P is between the edge 2 points
- Avoid Horizontal edges
- Test separately if point on segment

The winding number: algorithm

```
int isInsidePoly(vector<point> p, point p0) {
  int wn = 0;  // the winding number counter

  for (int i = 0; i < sz(p); i++) {
    point cur = p[i], nxt = p[(i + 1) % sz(p)];
    if (isPointOnSegment(cur, nxt, p0))
      return true;
    if (cur.Y <= p0.Y) {     // Upward edge
      if (nxt.Y > p0.Y && cp(nxt-cur, p0-cur) > EPS)
        ++wn;
    } else {                 // Downward edge
      if (nxt.Y <= p0.Y && cp(nxt-cur, p0-cur) < -EPS)
        --wn;
    }
  }
  return wn != 0;
}
```

Optimizations for PIP test

- One easy optimization in case of too many such tests is to create a bounding box around the polygon
  - The tightest rectangle around polygon
  - If it is inside the box => call the test
  - If it is outside the box => false
  - Similar to Bbox, one can use Circle
- For convex polygon, we can do it in O(logn)
  - See note about Preparata algorithm. Also Read here

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً