Algorithm StraightSkeletonSimulation(P):

1. Initialize data structures

1.1. Read the polygon P with n vertices (v1, v2, ..., vn) in CCW order.

1.2. For each vertex v:

initialize(){

queue = new PriorityQueue();

// First handle edge events

foreach(**Edge** edge: polygon.edges) {

if (!edge.hasReflexEndpoint()) { // Only if both endpoints are convex

t = **calculateEdgeCollapseTime**(edge);

queue.add(new **EdgeEvent(**t, edge**)**);

}

}

// Then handle potential split events

foreach(**Vertex** vertex: polygon.vertices) {

if (vertex.isReflex()) {

bisector = **calculateBisector**(vertex);

foreach(Edge edge: polygon.edges) {

// Skip adjacent edges and their neighbors for robustness

if (**isNearbyEdge**(edge, vertex)) {

continue;

}

Point intersection = calculateIntersection(vertex, bisector, edge);

if (intersection != null) {

// Time is distance / velocity

double distance = **distance**(vertex.position, intersection);

double t = distance / BISECTOR\_VELOCITY;

queue.add(new **SplitEvent**(t, vertex, edge, intersection));

}

}

}

}

}  
  
double **calculateEdgeCollapseTime**(edge){

# Get the vertices at both ends of the edge

v1 = edge.startVertex

v2 = edge.endVertex

# Get the bisector angles relative to the edge

# Note: edge direction is v2 - v1

edge\_vector = normalize(v2.position - v1.position)

# Calculate angle between edge and first bisector

theta1 = **angleBetween**(edge\_vector, v1.bisector)

# Calculate angle between edge and second bisector

theta2 = **angleBetween**(edge\_vector, v2.bisector)

# Get the edge length

L = distance(v1.position, v2.position)

# Calculate collapse time using formula:

# T = L / (1/sin(θ1) + 1/sin(θ2))

velocity = (1/sin(theta1)) + (1/sin(theta2))

time = L / velocity

return time  
}  
  
double **angleBetween**(vector1, vector2){

# Returns angle between two vectors in radians

dot = **dotProduct**(vector1, vector2)

# Using acos could have numerical stability issues near 0 and π

# atan2 is generally more robust

return atan2(

crossProduct(vector1, vector2),

dotProduct(vector1, vector2)

)  
}  
  
Vector **calculateBisector**(vertex){

# Get vectors for both edges pointing AWAY from vertex

prev\_vector = **normalize**(vertex.position - vertex.prev.position)

next\_vector = **normalize**(vertex.next.position - vertex.position)

# Bisector is the normalized sum of the vectors

# This automatically handles both convex and reflex cases

bisector = normalize(prev\_vector + next\_vector)

return bisector  
}

Vector **normalize**(vector){

length = sqrt(vector.x \* vector.x + vector.y \* vector.y)

return Vector(vector.x / length, vector.y / length)  
}

- Identify edges e\_prev, e\_next.

- Compute the bisector direction b\_v based on e\_prev and e\_next.

1.3. For each edge e:

Angle between two vectors?

# Get vectors for both edges (pointing away from vertex)

prev\_vector = (prev\_x - v\_x, prev\_y - v\_y) # Vector from v to prev vertex

next\_vector = (next\_x - v\_x, next\_y - v\_y) # Vector from v to next vertex

# Use atan2 to get angles in range [-π, π]

prev\_angle = atan2(prev\_vector.y, prev\_vector.x)

next\_angle = atan2(next\_vector.y, next\_vector.x)

# Get interior angle (remember CCW)

**interior\_angle** = (next\_angle - prev\_angle) % (2\*π)

Bisector of that angle?

# Normalize the vectors

prev\_norm = normalize(prev\_vector)

next\_norm = normalize(next\_vector)

# Bisector is sum of normalized vectors

**bisector** = normalize(prev\_norm + next\_norm)

Compute the initial time when it will collapse (if it does before a split).

* θ1 = angle between the edge and the first bisector
* θ2 = angle between the edge and the second bisector

The effective speed at which the endpoints are moving towards each other along the edge direction is:

* v1 = 1/sin(θ1) for the first endpoint
* v2 = 1/sin(θ2) for the second endpoint

The total speed of collapse is   
v = v1 + v2  
Collapse Time:  
T*collapse time* = L *edge length* / v

T *collapse time* = L *edge length* / (1/sin(θ1) + 1/sin(θ2))

- Insert an EdgeEvent into the priority queue with that time.

2. While the event queue is not empty:

2.1. Extract the event E with the smallest time T.

2.2. If E is an EdgeEvent(e):

- If edge e is still valid (not already collapsed or split):

- Perform collapse of e at time T:

\* Let v\_left and v\_right be the vertices adjacent to e in the polygon.

\* Mark e as collapsed (or remove from polygon structure).

\* Merge edges around e:

- Let the polygon edges incident to e be e\_left and e\_right.

- Their endpoints now become adjacent.

\* Record arcs in the skeleton:

- Each vertex that traveled to collapse e traces out an arc on its bisector.

- Create a 'node' in the skeleton if 3 arcs meet.

\* Update data structures:

- Remove e from the polygon.

- Possibly update e\_left and e\_right to reflect new adjacency.

- Recompute times for any newly formed edge or changed edges.

- Insert new or updated EdgeEvents into the priority queue.

\* Handle any vertex that might have triggered a SplitEvent simultaneously.

2.3. Else if E is a SplitEvent(v, e'):

- If v is still reflex and the split edge e' is valid:

- Perform the split at time T:

\* The vertex v meets edge e' at point X.

\* Record the arc from v’s old position to X in the skeleton.

\* Split the polygon into two sub-polygons P1 and P2 at X.

\* For each sub-polygon:

- Re-initialize or update edges/vertices as in Step 1.2.

- Compute and insert new EdgeEvents (and potential SplitEvents) into the event queue.

- Else: Ignore/skip if geometry no longer matches the event conditions.

3. Output the skeleton structure

3.1. Once all edges vanish (or no events remain), the polygon is fully collapsed.

3.2. The recorded arcs and nodes form the \*\*straight skeleton\*\* S(P).

4. Return S(P)