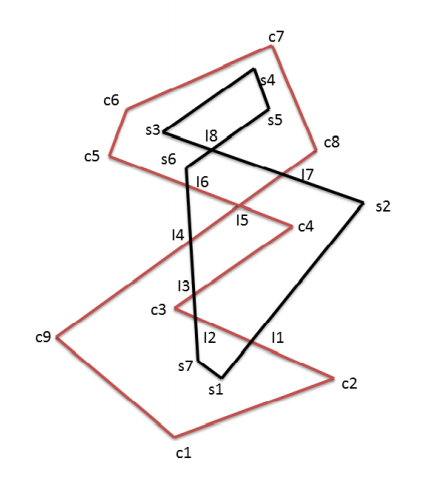
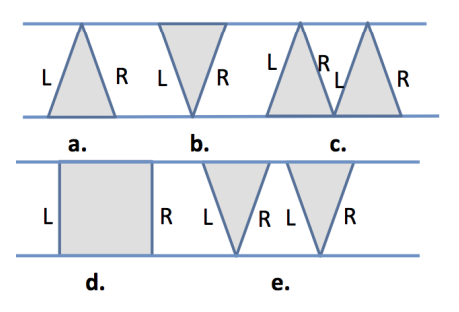
Vatti’s Polygon Clipping Algorithm

Terminology

* **Bound**
  + comprises of edges starting at a **local minima** and ending at a **local maxima**
  + edges on left bound = **left edges**
  + edges on right bound = **right edges**
  + Vertices call also be labeled as minima, maxima, left and right
* **contributing edges/vertices**
  + make the output polygon
* **scanline/ sweepline**
  + drawn through each vertex such that there are no vertices in between two such lines
  + area between two successive scanlines form a **scanbeam**



* **Lines must not be horizontal.** 
  + They can be offset a tiny amount to avoid this
* Divide and Conquer
  + Compute scanbeams separately
  + Sub-problems can vary in size
* **Partial polygons**
  + formed by intersecting the m scanlines with B and likewise with O



(Possible partial polygons)

* **Virtual vertices**
  + additional vertices produced by intersecting scanlines with input polygons
  + stored along with individual partial polygons as pivots which can be used later in the merging phase

## Lemma 1: Labeling Locally

The polygon edges in a scanbeam can be labeled as left or right without looking at the overall geometry and the labels alternate one another.

## Lemma 2: Independent Identification of Contributing Vertices

A vertex can be classified as contributing or not independently in a scanbeam.

In a scanbeam, a vertex can either be a start/end vertex or a new intersection. As such, two cases are possible:

### Case 1: Intersection vertex

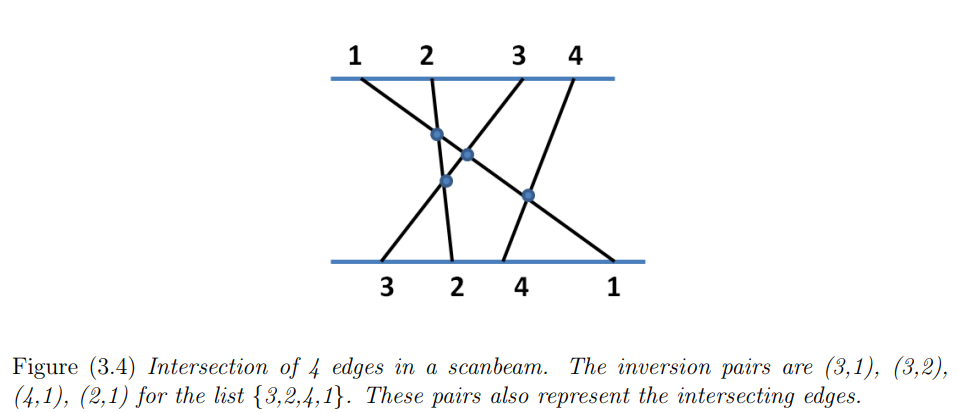
The vertices lying in between the scanlines are intersecting points which can be either i) intersections between the edges from the same polygon or ii) from different polygons. In case ii), the vertices are contributing irrespective of the clipping operation involved. Case i) depends on the clipping operation involved and it can be resolved using point-in-partial-polygon test using only the edges in a scanbeam.

### Case 2: Segment start or end vertex

A start/end vertex may lie either on the top or bottom scanline. In both cases, one can decide whether this vertex is contributing or not depending on its position with respect to other vertices on the scanline using point-inpartial-polygon test.

### Point-in-partial-polygon test

In a sequential point-in-polygon testing, one pass through all the edges of a polygon is required in order to determine if the point is contributing or not. But, when scanbeams are already created, this test can be performed faster since only the edges in a given scanbeam needs to be considered, other edges can be safely ignored. Assuming that the edges in a scanbeam denoted by E are sorted (by x-coordinate of edge’s intersection with scanline), in order to find if a vertex from subject polygon is contributing or not, the number of edges from clip polygon to its left in E are counted. If the count is odd then the edge is contributing otherwise non-contributing. This parity test can be expressed as a standard prefix sum problem.



## Lemma 3: Contributing Vertices in O(logn) Time

A logarithmic time algorithm can be developed to find whether a vertex on a scanline is contributing or not.

Let us consider a set E of n edges in a scanbeam which is sorted on the xcoordinate of intersection of E with a given scanline. For this ordered set, let L = [l1, l2, .., ln] represent labels for the corresponding edges in E and a binary associative operator +. Assign 0 to the label of the edges belonging to B and 1 to those belonging to O. A prefix sum for L is denoted by P [l1,(l1 +l2), ..,(l1 +l2 +..+ln)], where Pi is represented as (l1 +l2 +..+li). A vertex v of an edge Ei ∈ B lying on a scanline, is contributing if and only if Pi is odd. Now, repeat the prefix sum by reversing the 0/1 label for the edges of B and O to determine the contributing vertices for polygon O. This all-prefix-sum even-odd parity test can find all the contributing vertices in a scanbeam by invoking it twice (for lower and upper scanline). This algorithm requires sorting and prefix sum computation. With n processors, both operations can be done in O(logn) time.

#### Finding Intersections using Inversions

If the edges span a bounded region, 29 number of edge intersections can be found out within the region simply by knowing the order in which the edges intersect the boundary of the region [61]. For example, as shown in Figure 3.4, the order of edges L intersecting the lower scanline is {3, 2, 4, 1} and the number of inversions in L is equal to the number of edge intersections in the scanbeam. In the worst case, there can be O(n 2 ) inversions in an unsorted list but by extending mergesort, inversions in a scanbeam can be counted in O(nlogn) time sequentially. If we add the inversions for all the scanbeams, the number of intersections between two arbitrary polygons can be found out. To allocate processors in an output-sensitive manner, we need to first find the number of intersections and then allocate that number of additional processors to the pairs of intersecting segments.

## Lemma 4

The number of intersections in a scanbeam can be computed in O(logn) time using O(n) processors.

We show how parallel mergesort can be extended to find the intersecting pairs of edges by counting the number of inversions first and reporting them subsequently. Cole’s mergesort [62] is a pipelined algorithm which works at several levels of the tree at once and overlaps the merging process at different nodes. Let us consider left sublist Al = {A1, .., Amid} and right sublist Ar = {Amid+1, .., An} of edge indices present in two children of an internal node of the binary merge tree whose leaf nodes contain the edge indices. The inversions in the leaf nodes have to be identified and counted while sorting Al and Ar. Let the number of inversions found in Al and Ar be Invl and Invr respectively. To illustrate merging, let us consider that in a given timestep while merging the two sublists, we are at index i in list Al and at index j in list Ar. Since, Al and Ar are already sorted, if at any step, Al [i] > Ar[j], then {Al [i+1], .., Al [mid]} will also be greater than Ar[j]. As such Invm for an element at j consists of set {(i, j),(i + 1, j), ..,(mid, j)} containing |mid − i + 1| inversions. Invl and Invr are added to the number of inversions found during the merging (Invm) of sorted list Al and Ar for each internal node of the binary tree. With this modification, the total number of inversions (Invroot) is available in the root node of the merging tree when mergesort algorithm terminates.

In case of Cole’s algorithm, the left (Al) and right sublists (Ar) are dynamically growing and getting merged together according to the timestep as shown in Table 3.1. At first timestep, every fourth element from Al and Ar is compared. In the second timestep, similarly, every second element is compared. In the last timestep, remaining elements are compared. At each timestep, inversions are marked by utilizing the cross-rank of the elements computed by the original Cole’s algorithm in left and right sublists. Please refer to [62] for details on Cole’s algorithm.

