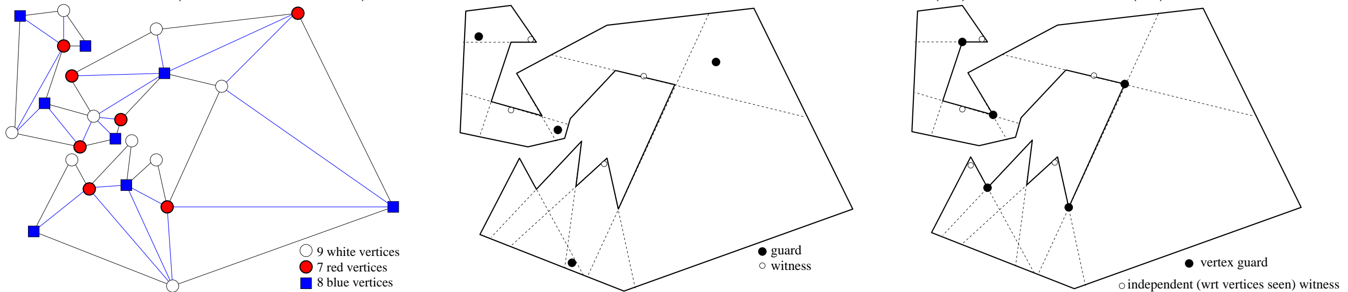
COMPUTATIONAL GEOMETRY
Homework Set # 5 – Solution Notes

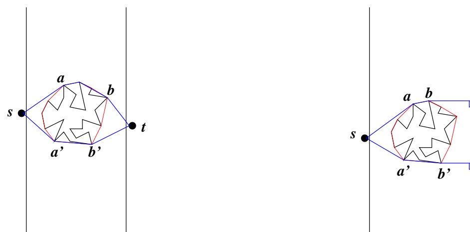
(1). (i). We show a triangulation of the polygon, with a 3-coloring of the vertices. For polygon P we obtain 9 white, 7 red, and 8 blue vertices. (Other triangulations may lead to different counts of each color: some triangulations give smallest color class of size 6 or of size 8.) Thus, Fisk's proof tells us to use 7 guards, one at each red vertex.
(ii). We show four independent witness points (white circles), w_1, w_2, w_3, and w_4 in P below: their visibility regions are disjoint. Thus, no single guard can see both, so P requires at least 4 guards ($g(P) \geq 4$). We also show a set of 4 guards (bold black dots) that suffice to see the entire polygon P; thus, $g(P) \leq 4$. Thus, $g(P) = 4$.
(iii). We show five witness points (white circles) that are independent with respect to the set of vertices seen: no two witness points are seen by a single vertex of P. (Check each witness, mark its visible vertices, and check that these five sets are pairwise disjoint!) Thus, we need at least 5 vertex guards ($g_v(P) \geq 5$). We also show a set of 5 vertex guards (bold black dots) that suffice to see the entire polygon P; thus, $g(P) \leq 5$. Thus, $g(P) = 5$.



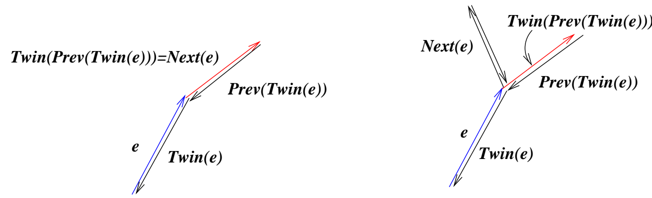
(2). The goal is to find a shortest path from s to t around P. Since P lies inside the river, between two parallel banks (one with s, one with t), local optimality will imply that an optimal path has the following form: Go from s to a tangency point that is a vertex of $CH(P)$, travel along the boundary of P, then exit along a tangency line to go to t. See the figure below. (Of course, if $st \cap P = \emptyset$, then the optimal path is just st.)

To compute an optimal path, we must, in general, compare two options, corresponding to going around P on either of its two sides. First, we compute the convex hull, $CH(P)$; this takes time $O(n)$, since Melkman's algorithm allows us to compute a convex hull of a simple chain/polygon in time $O(n)$. Then, we compute the 2 tangency points, a and a', with respect to s, and the two tangencies (b and b') with respect to t. This can be done in time $O(\log n)$, if we really wanted to, but a simple $O(n)$ scan around the $CH(P)$ suffices, since we already spend linear time anyhow. We then sum up the lengths of the two paths (from s to a, around ∂P to b, then from b to t, and similarly the path from s to t via a' and b'); this takes $O(n)$ time. Thus, overall, we spend $O(n)$ time.

(b). If our goal is not to get to a specific point t, but to any point on the opposite side of the river, then we modify the tangency/departure points (b and b') to be tangency points with respect to a line orthogonal to the side of the river (e.g., topmost and bottommost points of $CH(P)$, assuming the river runs north/south, as in the figure). The complexity remains $O(n)$.

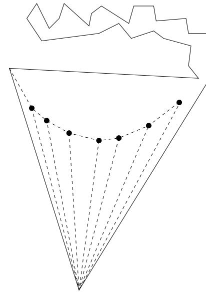


(3). (a) and (d) are always true. (b) is never true (instead, $Next(Prev(\vec{e})) = \vec{e}$). (c) is sometimes true (e.g., if the $Origin(Twin(\vec{e}))$ has degree 2; e.g., left, in the figure below), sometimes false (e.g., right, in the figure below). ((c) is also true if the subdivision consists of a single edge.)



(4). (a). We claim that P can be triangulated in time $O(n \log n)$. This follows from the algorithm given in class to triangulate a PSLG or polygon with holes in time $O(n \log n)$, where n is the total number of vertices. The algorithm first does a sweep (events correspond to vertices, the SLS is a balanced binary tree storing the left-to-right ordering of segments intersected by the horizontal sweep line), processing each event in time $O(\log n)$, yielding the horizontal trapezoidalization of P (with holes) in time $O(n \log n)$. Then, diagonals are added (at most one per trapezoid, joining the vertex defining the "top" to the vertex defining the "bottom"), yielding a partition of P into monotone mountains, each of which is easily triangulated in linear time (e.g., by simple ear clipping).

(b). We claim a lower bound of $\Omega(n \log n)$ from SORTING. Take an input set $\{x_1, \ldots, x_n\}$ of numbers to SORTING. Convert it to a set $\{(x_1, x_1^2), \ldots, (x_n, x_n^2)\}$ of n points in the plane; these will be point holes in a polygon P. We now surround the n point holes with a big triangle (easy to determine how big in time $O(n)$), and then "clip" one corner of the triangle, to add in more vertices of a simple polygon that surrounds the holes, having a total of $2n$ vertices. We now have an instance P of our problem, of size $3n$: a $2n$-gon that has n point holes. Now, there are "forced" diagonals (shown below) that join the point holes in order of x_i, since the points (x_i, x_i^2) are in convex position and each point must join to at least one vertex of P that lies below the supporting tangent line of the parabola $y = x^2$ at the point (x_i, x_i^2). See the figure below. These forced diagonals allow us to read off the sorted list of numbers x_i in linear time, given the DCEL of the triangulation of P. Thus, we have proved an $\Omega(n \log n)$ lower bound for triangulating a polygon having $3n$ vertices, n of which are point holes.



(5). (a). Our goal is to find all k crossing points among the $3n$ sides of the n triangles. This can be done in optimal time $O(k + n \log n)$ for a set of $3n$ segments, using the algorithm of Chazelle-Edelsbrunner (which uses $O(k)$ space) or the algorithm of Balaban (which uses $O(n)$ space). The Bentley-Ottmann sweep algorithm takes longer: $O((k + n) \log n)$.

(b). (i). Note that the depth is 1 if and only if the triangles are pairwise-disjoint. The Bentley-Ottmann sweep algorithm applied to the "detect" problem can be modified to solve this problem in time $O(n \log n)$. The unmodified algorithm will detect if there are any crossings among edges; this is not enough, though, since we also must detect *nesting* among the triangles. We modify the sweep to check for containment: The SLS now maintains the set of *intervals*, in order, that are the intersections of the sweep line with the triangles. The events occur when a vertex is hit; we must check if the hit vertex lies interior to an interval in the SLS.

(ii). The depth of $\mathcal{T}$ is equal to n if and only if the intersection of all n triangles is nonempty; i.e., if and only if the intersection of the $3n$ halfplanes is nonempty. Detecting emptiness of such an intersection (or finding the extreme vertex in a direction c, if nonempty) is exactly what linear programming does. Thus, we can solve in worst-case time $O(n)$, using the LP algorithm of Meggido. (or in expected time $O(n)$ using the randomized incremental algorithm of the textbook that we presented in class)

(iii). [6 points] Describe (briefly) how the *convex hull* of the k points of intersection can be computed in time $O(k + n \log n)$. (iii).First, we determine the k crossing points in time $O(k + n \log n)$, as in part (a). Now, on each of the $3n$ edges, we look at the crossing points (if any) along that edge and we keep only the *extreme* crossing points, closest to each of the two endpoints of the edge. We can discard all other crossing points (since, by definition of convexity, they lie within the convex hull of the kept crossing points). Thus, we now face a problem of computing the convex hull of at most $3n \cdot 2 = 6n$ points. By Chan's algorithm, this can be done in time $O(n \log h)$, which is dominated by $O(n \log n)$ (so we may as well use Graham scan or any other $O(n \log n)$ convex hull algorithm). In

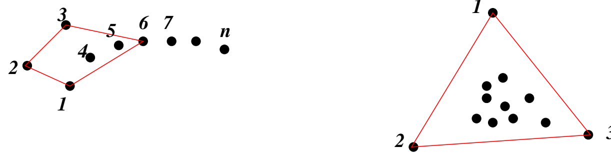
total the convex hull is computed in time $O(k + n \log n)$.

(6). (a). We proved that the expected running time was $O(n \log n)$.

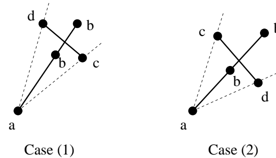
(b). The worst-case running time is $O(n^2)$.

(c). A bad example that achieves time $\Theta(n^2)$ is shown below (left). Explanation: At each insertion of a new point p_i in the "bad" order shown, each of the remaining points $p_{i+1}, \ldots, p_n$ must have its conflict edge updated, costing us $\Theta(n-i)$. Since $\sum_i \Theta(n-i) = \Theta(n^2)$, we get the claimed behavior.

(d). The best-case running time is $O(n)$, which occurs, e.g., in the example below (right): The initial convex hull is just $\Delta p_1 p_2 p_3$, and each point p_i $(i > 3)$ is checked and found to be inside this triangle (in time $O(n)$), so the algorithm stops immediately.



(7). (a). We split into two cases: If d is (strictly) to the left of $\overrightarrow{ac}$, then we get the picture shown in (1) below, and we test if b lies in the convex cone determined by a, c, d (i.e., if b is left of $\overrightarrow{ac}$ and d is left of $\overrightarrow{ab}$). Otherwise (c is to the left of $\overrightarrow{ad}$), we get the picture shown in (2) below, and we test if b lies in the convex cone determined by a, d, c (i.e., if b is left of $\overrightarrow{ad}$ and c is left of $\overrightarrow{ab}$). The code below does exactly this. Note that if Collinear(a, c, d) (or Collinear(a, b, c), or Collinear(a, b, d)) RaySegIntersectProp returns FALSE. However, if Collinear(b, c, d) and segment cd crosses $\overrightarrow{ab}$ at b, it correctly returns TRUE.




```
/*---------------------------------------------------------------------
Returns TRUE iff the ray (a,b) properly intersects the segment (c,d).
---------------------------------------------------------------------*/
bool   RaySegIntersectProp( tPointi a, tPointi b, tPointi c, tPointi d )
{
  if  ( Left( a, c, d ) )                        /* Case (1)  */
    {
    if  ( Left( a, c, b ) && Left( a, b, d ) )
      {
      return  TRUE;
      }
    }
  else if ( Left( a, d, b ) && Left( a, b, c ) )    /* Case (2) */
    {
    return  TRUE;
    }
  return  FALSE;
}
```

(b). We split into two cases: If a is (strictly) to the left of $\overrightarrow{cd}$, then we get the picture shown in (1) above, and we test if b lies to the left of $\overrightarrow{ac}$ *and* $\overrightarrow{ab} \times \overrightarrow{cd} > 0$ (i.e., Left$(0, b-a, d-c)$); if both are true, we return true, since the rays must meet. Otherwise (a is right of $\overrightarrow{cd}$), we get the picture shown in (2) above, and we test if c lies to the left of $\overrightarrow{ab}$ *and* $\overrightarrow{cd} \times \overrightarrow{ab} > 0$ (i.e., Left$(0, d-c, b-a)$); if both are true, we return true. Otherwise, we return false.

```
/*---------------------------------------------------------------------
Returns TRUE iff the ray (a,b), with endpoint a, pointing in direction of b,
properly intersects the ray (c,d), with endpoint c, pointing in direction of d.
---------------------------------------------------------------------*/
bool   RayRayIntersectProp( tPointi a, tPointi b, tPointi c, tPointi d )
{
  if  ( Left( a, c, d ) )                        /* Case (1)  */
    {
    if  ( Left( a, c, b ) && Left( 0, b-a, d-c ) )
      {
```

```
      return  TRUE;
      }
    }
  else if ( Left( a, b, c ) && Left( 0, d-c, b-a ) )    /* Case (2) */
    {
    return  TRUE;
    }
  return  FALSE;
}
```

(8). (a). Sort the points S from left to right (increasing x): let them be $p_1, p_2, \ldots, p_n$ in left-to-right order. This step takes $O(n \log n)$ time.

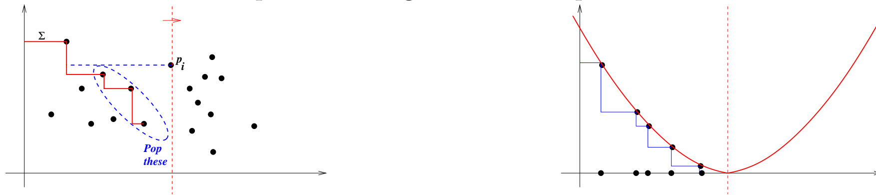
Now, step through the points, in a "Graham scan" like way, maintaining in a stack, Σ, the "staircase" of nondominated points seen so far. The stack Σ stores points in order of decreasing y. When we come to a new point, p_i, we pop the stack until the top of the stack is a point that is above (in y) the point p_i. (Potentially, we pop all elements off of the stack!) We then push p_i onto the stack. Continue.

Claim: The resulting stack stores the non-dominated points in x-order (also in decreasing y-order). We also claim that the "scan" takes $O(n)$ time, since each operation can be charged off to a point that is popped.

(b). The total time is $O(n \log n)$.

(c). We prove a lower bound of $\Omega(n \log n)$ from SORTING. (**NOTE: It is not enough to say "Oh, well my algorithm uses sorting as a first step, so the lower bound is $\Omega(n \log n)$, since sorting has such a lower bound.** Just because **YOU** used sorting in **YOUR** algorithm does not mean that it is essential – think of linear-time convex hull (Melkman) of a simple polygon: we could have sorted the points and done Graham scan....)

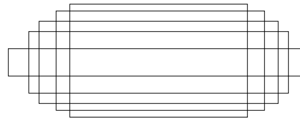
Take an input to SORTING, $\{x_1, \ldots, x_n\}$, and create a set of points, $S = \{(x_i, (x_i - M)^2), i = 1, \ldots, n\}$, where $M = 1 + \max_i x_i$ is easily found in time $O(n)$. This lifts the points onto a parabola in the first quadrant, and all of the points S are non-dominated. The output of our algorithm will report them in sorted order by x-coordinate.



(9). (a). Sweep the rectangles with a horizontal line. The events occur when the line passes through the top/bottom of a rectangle. The SLS maintains a left-to-right sorted list of the sides of rectangles currently stabbed by the sweep line. When the top of a rectangle is encountered, we do a binary search to locate its left side and a binary search to locate its right side in the SLS, and insert them in the SLS; this costs $O(\log n)$. By subtracting the indices obtained in the location, we have a count on how many vertical sides are crossed; we add this count to our running total count. When the bottom of a rectangle is encountered, we remove its left/right sides from the SLS. In total, we have $2n$ events, at a cost of $O(\log n)$ per event, for a total of $O(n \log n)$.

(Note that we avoid the costly reporting version, which is $O(k + n \log n)$, where k could be quadratic in n.)

(b). The maximum possible value of k (as a function of n) is $4\binom{n}{2} = 2n(n-1)$, since each pair of rectangles can intersect in 4 crossing points.



(10). The first 7 rows are shown below.

Event	Event Queue, Q	Sweep Status, $\mathcal{L}$
-	$a_1, a_4, a_5, a_3, a_2, b_3, b_4, b_2, b_5, b_1$	()
a_1	$a_4, a_5, a_3, a_2, b_3, b_4, b_2, b_5, b_1$	(1)
a_4	$a_5, a_3, a_2, x_{14}, b_3, b_4, b_2, b_5, b_1$	(1,4)
a_5	$a_3, x_{45}, a_2, b_3, b_4, b_2, b_5, b_1$	(1,5,4)
a_3	$x_{45}, a_2, b_3, b_4, b_2, b_5, b_1$	(1,5,4,3)
x_{45}	$a_2, x_{14}, b_3, b_4, b_2, b_5, b_1$	(1,4,5,3)
a_2	$x_{25}, x_{14}, b_3, b_4, b_2, b_5, b_1$	(1,4,5,2,3)