计算几何研究：

1.Line Segment Intersection

Lemma:

1. Let s_i and s_j be two non-horizontal segments whose interiors intersect in a single point p, and assume there is no third segment passing through p. Then there is an event point above p where s_i and s_j become adjacent and are tested for intersection.

2. Algorithm FINDINTERSECTIONS computes all intersection points and the segments that contain it correctly.

Theorem:

Let S be a set of n line segments in the plane. All intersection points in S, with for each intersection point the segments involved in it, can be reported in $O(n \log n + I \log n)$ time and $O(n)$ space, where I is the number of intersection points.

Pseudo code:

Algorithm FINDINTERSECTIONS(S)

Input. A set S of line segments in the plane.

HANDLEEVENTPOINT(p)

Let $U(p)$ be the set of segments whose upper endpoint is p; these segments are stored with the event point p. (For horizontal segments, the upper endpoint is by definition the left endpoint.)

Find all segments stored in $\mathcal{T}$ that contain p; they are adjacent in $\mathcal{T}$. Let $L(p)$ denote the subset of segments found whose lower endpoint is p, and let $C(p)$ denote the subset of segments found that contain p in their interior.

if $L(p) \cup U(p) \cup C(p)$ contains more than one segment

  then Report p as an intersection, together with $L(p)$, $U(p)$, and $C(p)$.

Delete the segments in $L(p) \cup C(p)$ from $\mathcal{T}$.

Insert the segments in $U(p) \cup C(p)$ into $\mathcal{T}$. The order of the segments in $\mathcal{T}$ should correspond to the order in which they are intersected by a sweep line just below p. If there is a horizontal segment, it comes last among all segments containing p.

(* Deleting and re-inserting the segments of $C(p)$ reverses their order. *)

if $U(p) \cup C(p) = \emptyset$

  then Let s_l and s_r be the left and right neighbors of p in $\mathcal{T}$.

    FINDNEWEVENT(s_l, s_r, p)

  else Let s' be the leftmost segment of $U(p) \cup C(p)$ in $\mathcal{T}$.

    Let s_l be the left neighbor of s' in $\mathcal{T}$.

    FINDNEWEVENT(s_l, s', p)

    Let s'' be the rightmost segment of $U(p) \cup C(p)$ in $\mathcal{T}$.

    Let s_r be the right neighbor of s'' in $\mathcal{T}$.

    FINDNEWEVENT(s'', s_r, p)

FINDNEWEVENT(s_l, s_r, p)

1. **if** s_l and s_r intersect below the sweep line, or on it and to the right of the current event point p, and the intersection is not yet present as an event in $\mathcal{Q}$
2. **then** Insert the intersection point as an event into $\mathcal{Q}$.

Algorithm MAPOVERLAY($\mathcal{S}_1, \mathcal{S}_2$)

Input. Two planar subdivisions $\mathcal{S}_1$ and $\mathcal{S}_2$ stored in doubly-connected edge lists.

Output. The overlay of $\mathcal{S}_1$ and $\mathcal{S}_2$ stored in a doubly-connected edge list $\mathcal{D}$.

1. Copy the doubly-connected edge lists for $\mathcal{S}_1$ and $\mathcal{S}_2$ to a new doubly-connected edge list $\mathcal{D}$.
2. Compute all intersections between edges from $\mathcal{S}_1$ and $\mathcal{S}_2$ with the plane sweep algorithm of Section 2.1. In addition to the actions on $\mathcal{T}$ and $\mathcal{Q}$ required at the event points, do the following:
   - Update $\mathcal{D}$ as explained above if the event involves edges of both $\mathcal{S}_1$ and $\mathcal{S}_2$. (This was explained for the case where an edge of $\mathcal{S}_1$ passes through a vertex of $\mathcal{S}_2$.)
   - Store the half-edge immediately to the left of the event point at the vertex in $\mathcal{D}$ representing it.
3. (* Now $\mathcal{D}$ is the doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1, \mathcal{S}_2)$, except that the information about the faces has not been computed yet. *)
4. Determine the boundary cycles in $\mathcal{O}(\mathcal{S}_1, \mathcal{S}_2)$ by traversing $\mathcal{D}$.
5. Construct the graph $\mathcal{G}$ whose nodes correspond to boundary cycles and whose arcs connect each hole cycle to the cycle to the left of its leftmost vertex, and compute its connected components. (The information to determine the arcs of $\mathcal{G}$ has been computed in line 2, second item.)
6. **for** each connected component in $\mathcal{G}$
7.   **do** Let $\mathcal{C}$ be the unique outer boundary cycle in the component and let f denote the face bounded by the cycle. Create a face record for f, set *OuterComponent*(f) to some half-edge of $\mathcal{C}$, and construct the list *InnerComponents*(f) consisting of pointers to one half-edge in each hole cycle in the component. Let the *IncidentFace*() pointers of all half-edges in the cycles point to the face record of f.
8. Label each face of $\mathcal{O}(\mathcal{S}_1, \mathcal{S}_2)$ with the names of the faces of $\mathcal{S}_1$ and $\mathcal{S}_2$ containing it, as explained above.

2.Polygon Triangulation
Lemma:

Theorem 3.2 (Art Gallery Theorem) *For a simple polygon with n vertices, $\lfloor n/3 \rfloor$ cameras are occasionally necessary and always sufficient to have every point in the polygon visible from at least one of the cameras.*

Theorem 3.3 *Let $\mathcal{P}$ be a simple polygon with n vertices. A set of $\lfloor n/3 \rfloor$ camera positions in $\mathcal{P}$ such that any point inside $\mathcal{P}$ is visible from at least one of the cameras can be computed in $O(n \log n)$ time.*

Lemma 3.4 *A polygon is y-monotone if it has no split vertices or merge vertices.*

Algorithm MAKEMONOTONE($\mathcal{P}$)
Input. A simple polygon $\mathcal{P}$ stored in a doubly-connected edge list $\mathcal{D}$.
Output. A partitioning of $\mathcal{P}$ into monotone subpolygons, stored in $\mathcal{D}$.
1. Construct a priority queue $\mathcal{Q}$ on the vertices of $\mathcal{P}$, using their y-coordinates as priority. If two points have the same y-coordinate, the one with smaller x-coordinate has higher priority.
2. Initialize an empty binary search tree $\mathcal{T}$.
3. **while** $\mathcal{Q}$ is not empty
4.   **do** Remove the vertex v_i with the highest priority from $\mathcal{Q}$.
5.   Call the appropriate procedure to handle the vertex, depending on its type.

HANDLESTARTVERTEX(v_i)
1. Insert e_i in $\mathcal{T}$ and set *helper*(e_i) to v_i.

At the start vertex v_5 in the example figure, for instance, we insert e_5 into the tree $\mathcal{T}$.

HANDLEENDVERTEX(v_i)
1. **if** *helper*(e_{i-1}) is a merge vertex
2.   **then** Insert the diagonal connecting v_i to *helper*(e_{i-1}) in $\mathcal{D}$.
3. Delete e_{i-1} from $\mathcal{T}$.

HANDLESPLITVERTEX(v_i)
1. Search in $\mathcal{T}$ to find the edge e_j directly left of v_i.
2. Insert the diagonal connecting v_i to *helper*(e_j) in $\mathcal{D}$.
3. *helper*$(e_j) \leftarrow v_i$
4. Insert e_i in $\mathcal{T}$ and set *helper*(e_i) to v_i.

HANDLEMERGEVERTEX(v_i)

if *helper*(e_{i-1}) is a merge vertex

  then Insert the diagonal connecting v_i to *helper*(e_{i-1}) in $\mathcal{D}$.

Delete e_{i-1} from $\mathcal{T}$.

Search in $\mathcal{T}$ to find the edge e_j directly left of v_i.

if *helper*(e_j) is a merge vertex

  then Insert the diagonal connecting v_i to *helper*(e_j) in $\mathcal{D}$.

helper$(e_j) \leftarrow v_i$

HANDLEREGULARVERTEX(v_i)

if the interior of $\mathcal{P}$ lies to the right of v_i

  then if *helper*(e_{i-1}) is a merge vertex

    then Insert the diagonal connecting v_i to *helper*(e_{i-1}) in $\mathcal{D}$.

   Delete e_{i-1} from $\mathcal{T}$.

   Insert e_i in $\mathcal{T}$ and set *helper*(e_i) to v_i.

  else Search in $\mathcal{T}$ to find the edge e_j directly left of v_i.

   if *helper*(e_j) is a merge vertex

    then Insert the diagonal connecting v_i to *helper*(e_j) in $\mathcal{D}$.

   helper$(e_j) \leftarrow v_i$

Lemma 3.5 *Algorithm* MAKEMONOTONE *adds a set of non-intersecting diagonals that partitions $\mathcal{P}$ into monotone subpolygons.*

Theorem 3.6 *A simple polygon with n vertices can be partitioned into y-monotone polygons in $O(n \log n)$ time with an algorithm that uses $O(n)$ storage.*

Algorithm TRIANGULATEMONOTONEPOLYGON($\mathcal{P}$)

Input. A strictly y-monotone polygon $\mathcal{P}$ stored in a doubly-connected edge list $\mathcal{D}$.

Output. A triangulation of $\mathcal{P}$ stored in the doubly-connected edge list $\mathcal{D}$.

1. Merge the vertices on the left chain and the vertices on the right chain of $\mathcal{P}$ into one sequence, sorted on decreasing y-coordinate. If two vertices have the same y-coordinate, then the leftmost one comes first. Let $u_1, \ldots, u_n$ denote the sorted sequence.
2. Initialize an empty stack $\mathcal{S}$, and push u_1 and u_2 onto it.
3. **for** $j \leftarrow 3$ **to** $n-1$
4.   **do if** u_j and the vertex on top of $\mathcal{S}$ are on different chains
5.   **then** Pop all vertices from $\mathcal{S}$.
6.   Insert into $\mathcal{D}$ a diagonal from u_j to each popped vertex, except the last one.
7.   Push u_{j-1} and u_j onto $\mathcal{S}$.
8.   **else** Pop one vertex from $\mathcal{S}$.
9.   Pop the other vertices from $\mathcal{S}$ as long as the diagonals from u_j to them are inside $\mathcal{P}$. Insert these diagonals into $\mathcal{D}$. Push the last vertex that has been popped back onto $\mathcal{S}$.
10.   Push u_j onto $\mathcal{S}$.
11. Add diagonals from u_n to all stack vertices except the first and the last one.

Theorem 3.7 *A strictly y-monotone polygon with n vertices can be triangulated in linear time.*

Theorem 3.8 *A simple polygon with n vertices can be triangulated in $O(n \log n)$ time with an algorithm that uses $O(n)$ storage.*

Theorem 3.9 *A planar subdivision with n vertices in total can be triangulated in $O(n \log n)$ time with an algorithm that uses $O(n)$ storage.*

Lemma 4.1 *The polyhedron $\mathcal{P}$ can be removed from its mold by a translation in direction $\vec{d}$ if and only if $\vec{d}$ makes an angle of at least $90°$ with the outward normal of all ordinary facets of $\mathcal{P}$.*

Theorem 4.2 *Let $\mathcal{P}$ be a polyhedron with n facets. In $O(n^2)$ expected time and using $O(n)$ storage it can be decided whether $\mathcal{P}$ is castable. Moreover, if $\mathcal{P}$ is castable, a mold and a valid direction for removing $\mathcal{P}$ from it can be computed in the same amount of time.*

Algorithm INTERSECTHALFPLANES(H)
Input. A set H of n half-planes in the plane.
Output. The convex polygonal region $C := \bigcap_{h \in H} h$.
1. **if** card$(H) = 1$
2. **then** $C \leftarrow$ the unique half-plane $h \in H$
3. **else** Split H into sets H_1 and H_2 of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$.
4. $C_1 \leftarrow$INTERSECTHALFPLANES(H_1)
5. $C_2 \leftarrow$INTERSECTHALFPLANES(H_2)
6. $C \leftarrow$INTERSECTCONVEXREGIONS(C_1, C_2)

Theorem 4.3 *The intersection of two convex polygonal regions in the plane can be computed in $O(n)$ time.*

Corollary 4.4 *The common intersection of a set of n half-planes in the plane can be computed in $O(n \log n)$ time and linear storage.*

Lemma 4.5 *Let $1 \leqslant i \leqslant n$, and let C_i and v_i be defined as above. Then we have*
(i) If $v_{i-1} \in h_i$, then $v_i = v_{i-1}$.
(ii) If $v_{i-1} \notin h_i$, then either $C_i = \emptyset$ or $v_i \in \ell_i$, where ℓ_i is the line bounding h_i.

Proof. (i) Let $v_{i-1} \in h_i$. Because $C_i = C_{i-1} \cap h_i$ and $v_{i-1} \in C_{i-1}$ this means that $v_{i-1} \in C_i$. Furthermore, the optimal point in C_i cannot be better than the optimal point in C_{i-1}, since $C_i \subseteq C_{i-1}$. Hence, v_{i-1} is the optimal vertex in C_i as well.

(ii) Let $v_{i-1} \notin h_i$. Suppose for a contradiction that C_i is not empty and that v_i does not lie on ℓ_i. Consider the line segment $\overline{v_{i-1}v_i}$. We have $v_{i-1} \in C_{i-1}$ and, since $C_i \subset C_{i-1}$, also $v_i \in C_{i-1}$. Together with the convexity of C_{i-1}, this implies that the segment $v_{i-1}v_i$ is contained in C_{i-1}. Since v_{i-1} is the optimal point in C_{i-1} and the objective function $f_{\vec{c}}$ is linear, it follows that $f_{\vec{c}}(p)$ increases monotonically along $\overline{v_{i-1}v_i}$ as p moves from v_i to v_{i-1}. Now consider the intersection point q of $\overline{v_{i-1}v_i}$ and ℓ_i. This intersection point exists, because $v_{i-1} \notin h_i$ and $v_i \in C_i$. Since $\overline{v_{i-1}v_i}$ is contained in C_{i-1}, the point q must be in C_i. But the value of the objective function increases along $\overline{v_{i-1}v_i}$, so $f_{\vec{c}}(q) > f_{\vec{c}}(v_i)$. This contradicts the definition of v_i. $\square$

Lemma 4.6 *A 1-dimensional linear program can be solved in linear time. Hence, if case (ii) of Lemma 4.5 arises, then we can compute the new optimal vertex v_i, or decide that the linear program is infeasible, in $O(i)$ time.*

Algorithm 2DRANDOMIZEDLP($H, \vec{c}$)

Input. A linear program $(H, \vec{c})$, where H is a set of n half-planes and $\vec{c} \in \mathbb{R}^2$.

Output. If $(H, \vec{c})$ is unbounded, a ray is reported. If it is infeasible, then two or three certificate half-planes are reported. Otherwise, the lexicographically smallest point p that maximizes $f_{\vec{c}}(p)$ is reported.

Determine whether there is a direction vector $\vec{d}$ such that $\vec{d} \cdot \vec{c} > 0$ and $\vec{d} \cdot \vec{\eta}(h) \geqslant 0$ for all $h \in H$.
if $\vec{d}$ exists
    then compute H' and determine whether H' is feasible.
        if H' is feasible
            then Report a ray proving that $(H, \vec{c})$ is unbounded and quit.
            else Report that $(H, \vec{c})$ is infeasible and quit.
Let $h_1, h_2 \in H$ be certificates proving that $(H, \vec{c})$ is bounded and has a unique lexicographically smallest solution.
Let v_2 be the intersection of ℓ_1 and ℓ_2.
Let $h_3, h_4, \dots, h_n$ be a random permutation of the remaining half-planes in H.
for $i \leftarrow 3$ **to** n
    do if $v_{i-1} \in h_i$
        then $v_i \leftarrow v_{i-1}$
        else $v_i \leftarrow$ the point p on ℓ_i that maximizes $f_{\vec{c}}(p)$, subject to the constraints in H_{i-1}.
            if p does not exist
                then Let h_j, h_k (with $j, k < i$) be the certificates (possibly $h_j = h_k$) with $h_j \cap h_k \cap \ell_i = \emptyset$.
                    Report that the linear program is infeasible, with h_i, h_j, h_k as certificates, and quit.
return v_n

Lemma 4.7 *Algorithm* 2DBOUNDEDLP *computes the solution to a bounded linear program with n constraints and two variables in $O(n^2)$ time and linear storage.*

Algorithm 2DRANDOMIZEDBOUNDEDLP($H, \vec{c}, m_1, m_2$)

Input. A linear program $(H \cup \{m_1, m_2\}, \vec{c})$, where H is a set of n half-planes, $\vec{c} \in \mathbb{R}^2$, and m_1, m_2 bound the solution.

Output. If $(H \cup \{m_1, m_2\}, \vec{c})$ is infeasible, then this fact is reported. Otherwise, the lexicographically smallest point p that maximizes $f_{\vec{c}}(p)$ is reported.

Let v_0 be the corner of C_0.
Compute a *random* permutation $h_1, \dots, h_n$ of the half-planes by calling RANDOMPERMUTATION($H[1 \cdots n]$).
for $i \leftarrow 1$ **to** n
    do if $v_{i-1} \in h_i$
        then $v_i \leftarrow v_{i-1}$
        else $v_i \leftarrow$ the point p on ℓ_i that maximizes $f_{\vec{c}}(p)$, subject to the constraints in H_{i-1}.
            if p does not exist
                then Report that the linear program is infeasible and quit.
return v_n

Algorithm RANDOMPERMUTATION(A)

Input. An array $A[1\cdots n]$.

Output. The array $A[1\cdots n]$ with the same elements, but rearranged into a random permutation.

1. **for** $k \leftarrow n$ **downto** 2
2.   **do** *rndindex* $\leftarrow$ RANDOM(k)
3.   Exchange $A[k]$ and $A[\mathit{rndindex}]$.

Lemma 4.8 *The 2-dimensional linear programming problem with n constraints can be solved in $O(n)$ randomized expected time using worst-case linear storage.*

Lemma 4.9 *A linear program $(H,\vec{c})$ is unbounded if and only if there is a vector $\vec{d}$ with $\vec{d}\cdot\vec{c}>0$ such that $\vec{d}\cdot\vec{\eta}(h)\geqslant 0$ for all $h\in H$ and the linear program $(H',\vec{c})$ is feasible, where $H'=\{h\in H:\vec{\eta}(h)\cdot\vec{d}=0\}$.*

Algorithm RANDOMIZEDLP($H,\vec{c}$)

Input. A linear program $(H,\vec{c})$, where H is a set of n half-spaces in $\mathbb{R}^d$ and $\vec{c}\in\mathbb{R}^d$.

Output. If $(H,\vec{c})$ is unbounded, a ray is reported. If it is infeasible, then at most $d+1$ certificate half-planes are reported. Otherwise, the lexicographically smallest point p that maximizes $f_{\vec{c}}(p)$ is reported.

1. Determine whether a direction vector $\vec{d}$ exists such that $\vec{d}\cdot\vec{c}>0$ and $\vec{d}\cdot\vec{\eta}(h)\geqslant 0$ for all $h\in H$.
2. **if** $\vec{d}$ exists
3.   **then** compute H' and determine whether H' is feasible.
4.   **if** H' is feasible
5.   **then** Report a ray proving that $(H,\vec{c})$ is unbounded and quit.
6.   **else** Report that $(H,\vec{c})$ is infeasible, provide certificates, and quit.
7. Let $h_1,h_2,\ldots,h_d$ be certificates proving that $(H,\vec{c})$ is bounded.
8. Let v_d be the intersection of $g_1,g_2,\cdots,g_d$.
9. Compute a random permutation $h_{d+1},\ldots,h_n$ of the remaining half-spaces in H.
10. **for** $i\leftarrow d+1$ **to** n
11.   **do if** $v_{i-1}\in h_i$
12.   **then** $v_i\leftarrow v_{i-1}$
13.   **else** $v_i\leftarrow$the point p on g_i that maximizes $f_{\vec{c}}(p)$, subject to the constraints $\{h_1,\ldots,h_{i-1}\}$
14.   **if** p does not exist
15.   **then** Let H^* be the at most d certificates for the infeasibility of the $(d-1)$-dimensional program.
16.   Report that the linear program is infeasible, with $H^*\cup h_i$ as certificates, and quit.
17. **return** v_n

Lemma 4.11 *Let $1\leqslant i\leqslant n$, and let C_i and v_i be defined as above. Then we have*

(i) *If $v_{i-1}\in h_i$, then $v_i=v_{i-1}$.*

(ii) *If $v_{i-1}\notin h_i$, then either $C_i=\emptyset$ or $v_i\in g_i$, where g_i is the hyperplane that bounds h_i.*

4.Orthogonal Range Searching

FINDSPLITNODE($\mathcal{T}, x, x'$)

Input. A tree $\mathcal{T}$ and two values x and x' with $x \leqslant x'$.

Output. The node ν where the paths to x and x' split, or the leaf where both paths end.

$\nu \leftarrow root(\mathcal{T})$
while ν is not a leaf **and** ($x' \leqslant x_\nu$ **or** $x > x_\nu$)
    do if $x' \leqslant x_\nu$
        then $\nu \leftarrow lc(\nu)$
        else $\nu \leftarrow rc(\nu)$
return ν

Algorithm 1DRANGEQUERY($\mathcal{T}, [x : x']$)

Input. A binary search tree $\mathcal{T}$ and a range $[x : x']$.

Output. All points stored in $\mathcal{T}$ that lie in the range.

$\nu_{\text{split}} \leftarrow$ FINDSPLITNODE($\mathcal{T}, x, x'$)
if ν_{split} is a leaf
    then Check if the point stored at ν_{split} must be reported.
    else (* Follow the path to x and report the points in subtrees right of the path. *)
        $\nu \leftarrow lc(\nu_{\text{split}})$
        while ν is not a leaf
            do if $x \leqslant x_\nu$
                then REPORTSUBTREE($rc(\nu)$)
                    $\nu \leftarrow lc(\nu)$
                else $\nu \leftarrow rc(\nu)$
        Check if the point stored at the leaf ν must be reported.
        Similarly, follow the path to x', report the points in subtrees left of the path, and check if the point stored at the leaf where the path ends must be reported.

Lemma 5.1 *Algorithm* 1DRANGEQUERY *reports exactly those points that lie in the query range.*

Theorem 5.2 *Let P be a set of n points in 1-dimensional space. The set P can be stored in a balanced binary search tree, which uses $O(n)$ storage and has $O(n \log n)$ construction time, such that the points in a query range can be reported in time $O(k + \log n)$, where k is the number of reported points.*

Algorithm BUILDKDTREE(P, *depth*)
Input. A set of points P and the current depth *depth*.
Output. The root of a kd-tree storing P.
if P contains only one point
    then return a leaf storing this point
    else if *depth* is even
        then Split P into two subsets with a vertical line ℓ through the median x-coordinate of the points in P. Let P_1 be the set of points to the left of ℓ or on ℓ, and let P_2 be the set of points to the right of ℓ.

Algorithm SEARCHKDTREE(ν, R)
Input. The root of (a subtree of) a kd-tree, and a range R.
Output. All points at leaves below ν that lie in the range.
if ν is a leaf
    then Report the point stored at ν if it lies in R.
    else if *region*$(lc(\nu))$ is fully contained in R
        then REPORTSUBTREE($lc(\nu)$)
        else if *region*$(lc(\nu))$ intersects R
            then SEARCHKDTREE($lc(\nu), R$)
      if *region*$(rc(\nu))$ is fully contained in R
        then REPORTSUBTREE($rc(\nu)$)
        else if *region*$(rc(\nu))$ intersects R
            then SEARCHKDTREE($rc(\nu), R$)

Lemma 5.4 *A query with an axis-parallel rectangle in a kd-tree storing n points can be performed in $O(\sqrt{n}+k)$ time, where k is the number of reported points.*

Algorithm 2DRANGEQUERY($\mathcal{T}, [x:x'] \times [y:y']$)
Input. A 2-dimensional range tree $\mathcal{T}$ and a range $[x:x'] \times [y:y']$.
Output. All points in $\mathcal{T}$ that lie in the range.

```
v_split ← FINDSPLITNODE(T, x, x')
if v_split is a leaf
  then Check if the point stored at v_split must be reported.
  else (* Follow the path to x and call 1DRANGEQUERY on the subtrees
          right of the path. *)
       v ← lc(v_split)
       while v is not a leaf
         do if x ≤ x_v
              then 1DRANGEQUERY(T_assoc(rc(v)), [y : y'])
                   v ← lc(v)
              else v ← rc(v)
       Check if the point stored at v must be reported.
       Similarly, follow the path from rc(v_split) to x', call 1DRANGE-
       QUERY with the range [y : y'] on the associated structures of sub-
       trees left of the path, and check if the point stored at the leaf where
       the path ends must be reported.
```

Lemma 5.7 *A query with an axis-parallel rectangle in a range tree storing n points takes $O(\log^2 n+k)$ time, where k is the number of reported points.*

Theorem 5.8 *Let P be a set of n points in the plane. A range tree for P uses $O(n\log n)$ storage and can be constructed in $O(n\log n)$ time. By querying this range tree one can report the points in P that lie in a rectangular query range in $O(\log^2 n+k)$ time, where k is the number of reported points.*

Theorem 5.11 *Let P be a set of n points in d-dimensional space, with $d \geqslant 2$. A layered range tree for P uses $O(n\log^{d-1} n)$ storage and it can be constructed in $O(n\log^{d-1} n)$ time. With this range tree one can report the points in P that lie in a rectangular query range in $O(\log^{d-1} n+k)$ time, where k is the number of reported points.*

5.Point Location

Lemma 6.1 *Each face in a trapezoidal map of a set S of line segments in general position has one or two vertical sides and exactly two non-vertical sides.*

Algorithm TRAPEZOIDALMAP(S)
Input. A set S of n non-crossing line segments.
Output. The trapezoidal map $\mathcal{T}(S)$ and a search structure $\mathcal{D}$ for $\mathcal{T}(S)$ in a bounding box.

Determine a bounding box R that contains all segments of S, and initialize the trapezoidal map structure $\mathcal{T}$ and search structure $\mathcal{D}$ for it.
Compute a random permutation $s_1, s_2, \ldots, s_n$ of the elements of S.
for $i \leftarrow 1$ **to** n
  do Find the set $\Delta_0, \Delta_1, \ldots, \Delta_k$ of trapezoids in $\mathcal{T}$ properly intersected by s_i.
    Remove $\Delta_0, \Delta_1, \ldots, \Delta_k$ from $\mathcal{T}$ and replace them by the new trapezoids that appear because of the insertion of s_i.
    Remove the leaves for $\Delta_0, \Delta_1, \ldots, \Delta_k$ from $\mathcal{D}$, and create leaves for the new trapezoids. Link the new leaves to the existing inner nodes by adding some new inner nodes, as explained below.

Algorithm FOLLOWSEGMENT($\mathcal{T}, \mathcal{D}, s_i$)
Input. A trapezoidal map $\mathcal{T}$, a search structure $\mathcal{D}$ for $\mathcal{T}$, and a new segment s_i.
Output. The sequence $\Delta_0, \ldots, \Delta_k$ of trapezoids intersected by s_i.

Let p and q be the left and right endpoint of s_i.
Search with p in the search structure $\mathcal{D}$ to find Δ_0.
$j \leftarrow 0$;
while q lies to the right of $rightp(\Delta_j)$
  do if $rightp(\Delta_j)$ lies above s_i
      then Let Δ_{j+1} be the lower right neighbor of Δ_j.
      else Let Δ_{j+1} be the upper right neighbor of Δ_j.
    $j \leftarrow j+1$
return $\Delta_0, \Delta_1, \ldots, \Delta_j$

Theorem 6.3 *Algorithm* TRAPEZOIDALMAP *computes the trapezoidal map $\mathcal{T}(S)$ of a set S of n line segments in general position and a search structure $\mathcal{D}$ for $\mathcal{T}(S)$ in $O(n \log n)$ expected time. The expected size of the search structure is $O(n)$ and for any query point q the expected query time is $O(\log n)$.*

Corollary 6.4 *Let $\mathcal{S}$ be a planar subdivision with n edges. In $O(n \log n)$ expected time one can construct a data structure that uses $O(n)$ expected storage, such that for any query point q, the expected time for a point location query is $O(\log n)$.*

Theorem 6.5 *Algorithm* TRAPEZOIDALMAP *computes the trapezoidal map* $\mathcal{T}(S)$ *of a set* S *of* n *non-crossing line segments and a search structure* $\mathcal{D}$ *for*

$\mathcal{T}(S)$ *in* $O(n \log n)$ *expected time. The expected size of the search structure is* $O(n)$ *and for any query point* q *the expected query time is* $O(\log n)$.

6.Voronoi Diagrams

Theorem 7.2 *Let* P *be a set of* n *point sites in the plane. If all the sites are collinear then* $\mathrm{Vor}(P)$ *consists of* $n-1$ *parallel lines. Otherwise,* $\mathrm{Vor}(P)$ *is*

Theorem 7.3 *For* $n \geqslant 3$*, the number of vertices in the Voronoi diagram of a set of* n *point sites in the plane is at most* $2n-5$ *and the number of edges is at most* $3n-6$.

Theorem 7.4 *For the Voronoi diagram* $\mathrm{Vor}(P)$ *of a set of points* P *the following holds:*

(i) *A point* q *is a vertex of* $\mathrm{Vor}(P)$ *if and only if its largest empty circle* $C_P(q)$ *contains three or more sites on its boundary.*

(ii) *The bisector between sites* p_i *and* p_j *defines an edge of* $\mathrm{Vor}(P)$ *if and only if there is a point* q *on the bisector such that* $C_P(q)$ *contains both* p_i *and* p_j *on its boundary but no other site.*

Lemma 7.6 *The only way in which a new arc can appear on the beach line is through a site event.*

Lemma 7.7 *The only way in which an existing arc can disappear from the beach line is through a circle event.*

Lemma 7.8 *Every Voronoi vertex is detected by means of a circle event.*

Algorithm VORONOIDIAGRAM(P)

Input. A set $P := \{p_1, \ldots, p_n\}$ of point sites in the plane.

Output. The Voronoi diagram Vor(P) given inside a bounding box in a doubly-connected edge list $\mathcal{D}$.

1. Initialize the event queue $\mathcal{Q}$ with all site events, initialize an empty status structure $\mathcal{T}$ and an empty doubly-connected edge list $\mathcal{D}$.
2. **while** $\mathcal{Q}$ is not empty
3. **do** Remove the event with largest y-coordinate from $\mathcal{Q}$.
4. **if** the event is a site event, occurring at site p_i
5. **then** HANDLESITEEVENT(p_i)
6. **else** HANDLECIRCLEEVENT(γ), where γ is the leaf of $\mathcal{T}$ representing the arc that will disappear
7. The internal nodes still present in $\mathcal{T}$ correspond to the half-infinite edges of the Voronoi diagram. Compute a bounding box that contains all vertices of the Voronoi diagram in its interior, and attach the half-infinite edges to the bounding box by updating the doubly-connected edge list appropriately.
8. Traverse the half-edges of the doubly-connected edge list to add the cell records and the pointers to and from them.

HANDLESITEEVENT(p_i)

1. If $\mathcal{T}$ is empty, insert p_i into it (so that $\mathcal{T}$ consists of a single leaf storing p_i) and return. Otherwise, continue with steps 2– 5.
2. Search in $\mathcal{T}$ for the arc α vertically above p_i. If the leaf representing α has a pointer to a circle event in $\mathcal{Q}$, then this circle event is a false alarm and it must be deleted from $\mathcal{Q}$.
3. Replace the leaf of $\mathcal{T}$ that represents α with a subtree having three leaves. The middle leaf stores the new site p_i and the other two leaves store the site p_j that was originally stored with α. Store the tuples $\langle p_j, p_i \rangle$ and $\langle p_i, p_j \rangle$ representing the new breakpoints at the two new internal nodes. Perform rebalancing operations on $\mathcal{T}$ if necessary.
4. Create new half-edge records in the Voronoi diagram structure for the edge separating $\mathcal{V}(p_i)$ and $\mathcal{V}(p_j)$, which will be traced out by the two new breakpoints.
5. Check the triple of consecutive arcs where the new arc for p_i is the left arc to see if the breakpoints converge. If so, insert the circle event into $\mathcal{Q}$ and add pointers between the node in $\mathcal{T}$ and the node in $\mathcal{Q}$. Do the same for the triple where the new arc is the right arc.

HANDLECIRCLEEVENT(γ)

1. Delete the leaf γ that represents the disappearing arc α from $\mathcal{T}$. Update the tuples representing the breakpoints at the internal nodes. Perform rebalancing operations on $\mathcal{T}$ if necessary. Delete all circle events involving α from $\mathcal{Q}$; these can be found using the pointers from the predecessor and the successor of γ in $\mathcal{T}$. (The circle event where α is the middle arc is currently being handled, and has already been deleted from $\mathcal{Q}$.)
2. Add the center of the circle causing the event as a vertex record to the doubly-connected edge list $\mathcal{D}$ storing the Voronoi diagram under construction. Create two half-edge records corresponding to the new breakpoint of the beach line. Set the pointers between them appropriately. Attach the three new records to the half-edge records that end at the vertex.
3. Check the new triple of consecutive arcs that has the former left neighbor of α as its middle arc to see if the two breakpoints of the triple converge. If so, insert the corresponding circle event into $\mathcal{Q}$. and set pointers between the new circle event in $\mathcal{Q}$ and the corresponding leaf of $\mathcal{T}$. Do the same for the triple where the former right neighbor is the middle arc.

Lemma 7.9 *The algorithm runs in $O(n \log n)$ time and it uses $O(n)$ storage.*

Theorem 7.10 *The Voronoi diagram of a set of n point sites in the plane can be computed with a sweep line algorithm in $O(n \log n)$ time using $O(n)$ storage.*

Theorem 7.11 *The Voronoi diagram of a set of n disjoint line segment sites can be computed in $O(n \log n)$ time using $O(n)$ storage.*

Algorithm RETRACTION($S, q_{\text{start}}, q_{\text{end}}, r$)

Input. A set $S := \{s_1, \dots, s_n\}$ of disjoint line segments in the plane, and two discs D_{start} and D_{end} centered at q_{start} and q_{end} with radius r. The two disc positions do not intersect any line segment of S.

Output. A path that connects q_{start} to q_{end} such that no disc of radius r with its center on the path intersects any line segment of S. If no such path exists, this is reported.

1. Compute the Voronoi diagram $\text{Vor}(S)$ of S inside a sufficiently large bounding box.
2. Locate the cells of $\text{Vor}(P)$ that contain q_{start} and q_{end}.
3. Determine the point p_{start} on $\text{Vor}(S)$ by moving q_{start} away from the nearest line segment in S. Similarly, determine the point p_{end} on $\text{Vor}(S)$ by moving q_{end} away from the nearest line segment in S. Add p_{start} and p_{end} as vertices to $\text{Vor}(S)$, splitting the arcs on which they lie into two.
4. Let $\mathcal{G}$ be the graph corresponding to the vertices and edges of the Voronoi diagram. Remove all edges from $\mathcal{G}$ for which the smallest distance to the nearest sites is smaller than or equal to r.
5. Determine with depth-first search whether a path exists from p_{start} to p_{end} in $\mathcal{G}$. If so, report the line segment from q_{start} to p_{start}, the path in $\mathcal{G}$ from p_{start} to p_{end}, and the line segment from p_{end} to q_{end} as the path. Otherwise, report that no path exists.

Theorem 7.12 *Given n disjoint line segment obstacles and a disc-shaped robot, the existence of a collision-free path between two positions of the robot can be determined in $O(n \log n)$ time using $O(n)$ storage.*

Theorem 7.14 *Given a set of n points in the plane, its farthest-point Voronoi diagram can be computed in $O(n \log n)$ expected time using $O(n)$ storage.*

8.Delaunay Triangulations

Lemma 9.4 *Let edge $\overline{p_ip_j}$ be incident to triangles $p_ip_jp_k$ and $p_ip_jp_l$, and let C be the circle through p_i, p_j, and p_k. The edge $\overline{p_ip_j}$ is illegal if and only if the point p_l lies in the interior of C. Furthermore, if the points p_i, p_j, p_k, p_l form a convex quadrilateral and do not lie on a common circle, then exactly one of $\overline{p_ip_j}$ and $\overline{p_kp_l}$ is an illegal edge.*

Theorem 9.1 *Let P be a set of n points in the plane, not all collinear, and let k denote the number of points in P that lie on the boundary of the convex hull of P. Then any triangulation of P has $2n-2-k$ triangles and $3n-3-k$ edges.*

Theorem 9.2 *Let C be a circle, ℓ a line intersecting C in points a and b, and p, q, r, and s points lying on the same side of ℓ. Suppose that p and q lie on C, that r lies inside C, and that s lies outside C. Then*

Algorithm LEGALTRIANGULATION($\mathcal{T}$)
Input. Some triangulation $\mathcal{T}$ of a point set P.
Output. A legal triangulation of P.

while $\mathcal{T}$ contains an illegal edge $\overline{p_ip_j}$
  do (* Flip $\overline{p_ip_j}$ *)
    Let $p_ip_jp_k$ and $p_ip_jp_l$ be the two triangles adjacent to $\overline{p_ip_j}$.
    Remove $\overline{p_ip_j}$ from $\mathcal{T}$, and add $\overline{p_kp_l}$ instead.
return $\mathcal{T}$

Theorem 9.5 *The Delaunay graph of a planar point set is a plane graph.*

Theorem 9.6 *Let P be a set of points in the plane.*

(i) Three points $p_i, p_j, p_k \in P$ are vertices of the same face of the Delaunay graph of P if and only if the circle through p_i, p_j, p_k contains no point of P in its interior.

(ii) Two points $p_i, p_j \in P$ form an edge of the Delaunay graph of P if and only if there is a closed disc C that contains p_i and p_j on its boundary and does not contain any other point of P.

Theorem 9.6 readily implies the following characterization of Delaunay triangulations.

Theorem 9.7 *Let P be a set of points in the plane, and let $\mathcal{T}$ be a triangulation of P. Then $\mathcal{T}$ is a Delaunay triangulation of P if and only if the circumcircle of any triangle of $\mathcal{T}$ does not contain a point of P in its interior.*

Since we argued before that a triangulation is good for the purpose of height interpolation if its angle-vector is as large as possible, our next step should be to look at the angle-vector of Delaunay triangulations. We do this by a slight detour through legal triangulations.

Theorem 9.8 *Let P be a set of points in the plane. A triangulation $\mathcal{T}$ of P is legal if and only if $\mathcal{T}$ is a Delaunay triangulation of P.*

Theorem 9.9 *Let P be a set of points in the plane. Any angle-optimal triangulation of P is a Delaunay triangulation of P. Furthermore, any Delaunay triangulation of P maximizes the minimum angle over all triangulations of P.*

Algorithm DELAUNAYTRIANGULATION(P)

Input. A set P of $n+1$ points in the plane.

Output. A Delaunay triangulation of P.

Let p_0 be the lexicographically highest point of P, that is, the rightmost among the points with largest y-coordinate.
Let p_{-1} and p_{-2} be two points in $\mathbb{R}^2$ sufficiently far away and such that P is contained in the triangle $p_0 p_{-1} p_{-2}$.
Initialize $\mathcal{T}$ as the triangulation consisting of the single triangle $p_0 p_{-1} p_{-2}$.
Compute a random permutation $p_1, p_2, \dots, p_n$ of $P \setminus \{p_0\}$.
for $r \leftarrow 1$ **to** n
    do (* Insert p_r into $\mathcal{T}$: *)
        Find a triangle $p_i p_j p_k \in \mathcal{T}$ containing p_r.
        if p_r lies in the interior of the triangle $p_i p_j p_k$
            then Add edges from p_r to the three vertices of $p_i p_j p_k$, thereby splitting $p_i p_j p_k$ into three triangles.
                    LEGALIZEEDGE($p_r, \overline{p_i p_j}, \mathcal{T}$)
                    LEGALIZEEDGE($p_r, \overline{p_j p_k}, \mathcal{T}$)
                    LEGALIZEEDGE($p_r, \overline{p_k p_i}, \mathcal{T}$)
            else (* p_r lies on an edge of $p_i p_j p_k$, say the edge $\overline{p_i p_j}$ *)
                    Add edges from p_r to p_k and to the third vertex p_l of the other triangle that is incident to $\overline{p_i p_j}$, thereby splitting the two triangles incident to $\overline{p_i p_j}$ into four triangles.
                    LEGALIZEEDGE($p_r, \overline{p_i p_l}, \mathcal{T}$)
                    LEGALIZEEDGE($p_r, \overline{p_l p_j}, \mathcal{T}$)
                    LEGALIZEEDGE($p_r, \overline{p_j p_k}, \mathcal{T}$)
                    LEGALIZEEDGE($p_r, \overline{p_k p_i}, \mathcal{T}$)
Discard p_{-1} and p_{-2} with all their incident edges from $\mathcal{T}$.
return $\mathcal{T}$

LEGALIZEEDGE($p_r, \overline{p_i p_j}, \mathcal{T}$)

(* The point being inserted is p_r, and $\overline{p_i p_j}$ is the edge of $\mathcal{T}$ that may need to be flipped. *)
if $\overline{p_i p_j}$ is illegal
    then Let $p_i p_j p_k$ be the triangle adjacent to $p_r p_i p_j$ along $\overline{p_i p_j}$.
            (* Flip $\overline{p_i p_j}$: *) Replace $\overline{p_i p_j}$ with $\overline{p_r p_k}$.
            LEGALIZEEDGE($p_r, \overline{p_i p_k}, \mathcal{T}$)
            LEGALIZEEDGE($p_r, \overline{p_k p_j}, \mathcal{T}$)

Lemma 9.10 *Every new edge created in* DELAUNAYTRIANGULATION *or in* LEGALIZEEDGE *during the insertion of p_r is an edge of the Delaunay graph of* $\{p_{-2}, p_{-1}, p_0, \ldots, p_r\}$.

Lemma 9.11 *The expected number of triangles created by algorithm* DELAUNAYTRIANGULATION *is at most* $9n+1$.

Theorem 9.12 *The Delaunay triangulation of a set P of n points in the plane can be computed in $O(n \log n)$ expected time, using $O(n)$ expected storage.*

Lemma 9.13 *If P is a point set in general position, then*

$$\sum_{\Delta} \operatorname{card}(K(\Delta)) = O(n \log n),$$

where the summation is over all Delaunay triangles Δ created by the algorithm.

9.Convex Hulls

Theorem 11.1 *Let $\mathcal{P}$ be a convex polytope with n vertices. The number of edges of $\mathcal{P}$ is at most $3n-6$, and the number of facets of $\mathcal{P}$ is at most $2n-4$.*

Corollary 11.2 *The complexity of the convex hull of a set of n points in three-dimensional space is $O(n)$.*

Algorithm SLOWCONVEXHULL(P)
Input. A set P of points in the plane.
Output. A list L containing the vertices of CH(P) in clockwise order. 1. $E \leftarrow 0/$.
2. for all ordered pairs $(p,q) \in P \times P$ with p not equal to q
3. do *valid* $\leftarrow$ true

4. forallpoints$r \in P$notequalto $p$or$q$
5. do if r lies to the left of the directed line from p to q

6. then *valid* ← false.
7. if *valid* then Add the directed edge *pq* to *E*.
8. From the set *E* of edges construct a list L of vertices of CH(P), sorted in clockwise order.

Algorithm CONVEXHULL(P)
Input. A set P of n points in three-space.
Output. The convex hull $\mathcal{CH}(P)$ of P.

Find four points p_1, p_2, p_3, p_4 in P that form a tetrahedron.
$\mathcal{C} \leftarrow \mathcal{CH}(\{p_1, p_2, p_3, p_4\})$
Compute a random permutation $p_5, p_6, \ldots, p_n$ of the remaining points.
Initialize the conflict graph $\mathcal{G}$ with all visible pairs (p_t, f), where f is a facet of $\mathcal{C}$ and $t > 4$.
for $r \leftarrow 5$ **to** n
  do (* Insert p_r into $\mathcal{C}$: *)
    if $F_{\text{conflict}}(p_r)$ is not empty (* that is, p_r lies outside $\mathcal{C}$ *)
      then Delete all facets in $F_{\text{conflict}}(p_r)$ from $\mathcal{C}$.
        Walk along the boundary of the visible region of p_r (which consists exactly of the facets in $F_{\text{conflict}}(p_r)$) and create a list $\mathcal{L}$ of horizon edges in order.
        for all $e \in \mathcal{L}$
          do Connect e to p_r by creating a triangular facet f.
            if f is coplanar with its neighbor facet f' along e
              then Merge f and f' into one facet, whose conflict list is the same as that of f'.
              else (* Determine conflicts for f: *)
                Create a node for f in $\mathcal{G}$.
                Let f_1 and f_2 be the facets incident to e in the old convex hull.
                $P(e) \leftarrow P_{\text{conflict}}(f_1) \cup P_{\text{conflict}}(f_2)$
                for all points $p \in P(e)$
                  do If f is visible from p, add (p, f) to $\mathcal{G}$.
        Delete the node corresponding to p_r and the nodes corresponding to the facets in $F_{\text{conflict}}(p_r)$ from $\mathcal{G}$, together with their incident arcs.
return $\mathcal{C}$

Lemma 11.4 *Algorithm* CONVEXHULL *computes the convex hull of a set P of n points in $\mathbb{R}^3$ in $O(n \log n)$ expected time, where the expectation is with respect to the random permutation used by the algorithm.*

Lemma 11.5 *A flap $\Delta = (p,q,s,t)$ is in $\mathcal{T}(S)$ if and only if $\overline{pq}$, $\overline{ps}$, and $\overline{qt}$ are edges of the convex hull $\mathcal{CH}((S)$, there is a facet f_1 incident to $\overline{pq}$ and $\overline{ps}$, and a different facet f_2 incident to $\overline{pq}$ and $\overline{qt}$. Furthermore, if one of the facets f_1 or f_2 is visible from a point $x \in P$ then $x \in K(\Delta)$.*

Lemma 11.6 *The expected value of $\sum_e \text{card}(P(e))$, where the summation is over all horizon edges that appear at some stage of the algorithm, is $O(n \log n)$.*

Theorem 11.7 *The convex hull of a set of n points in $\mathbb{R}^3$ can be computed in $O(n \log n)$ randomized expected time.*

Theorem 11.8 *Let P be a set of points in 3-dimensional space, all lying in the plane $z = 0$. Let H be the set of planes $h(p)$, for $p \in P$, defined as above. Then the projection of $\mathcal{UE}(H)$ on the plane $z = 0$ is the Voronoi diagram of P.*