Algorithm \textsc{SlowConvexHull}(P)

\textit{Input}. A set \( P \) of points in the plane.
\textit{Output}. A list \( \mathcal{L} \) containing the vertices of \( \mathcal{C} \mathcal{H}(P) \) in clockwise order.

1. \( E \leftarrow \emptyset \).
2. \textbf{for} all ordered pairs \( (p,q) \in P \times P \) with \( p \) not equal to \( q \)
3. \quad \textbf{do} \hspace{1em} \text{valid} \leftarrow \text{true}
4. \quad \textbf{for} all points \( r \in P \) not equal to \( p \) or \( q \)
5. \quad \quad \textbf{do} \hspace{1em} \text{if} \hspace{1em} r \text{ lies to the left of the directed line from } p \text{ to } q
6. \quad \quad \quad \textbf{then} \hspace{1em} \text{valid} \leftarrow \text{false}.
7. \quad \quad \textbf{if} \hspace{1em} \text{valid} \hspace{1em} \textbf{then} \hspace{1em} \text{Add the directed edge } \overrightarrow{pq} \text{ to } E.
8. From the set \( E \) of edges construct a list \( \mathcal{L} \) of vertices of \( \mathcal{C} \mathcal{H}(P) \), sorted in clockwise order.
**Algorithm** CONVEXHULL\((P)\)

*Input.* A set \(P\) of points in the plane.

*Output.* A list containing the vertices of \(\mathcal{CH}(P)\) in clockwise order.

1. Sort the points by \(x\)-coordinate, resulting in a sequence \(p_1, \ldots, p_n\).
2. Put the points \(p_1\) and \(p_2\) in a list \(L_{\text{upper}}\), with \(p_1\) as the first point.
3. **for** \(i \leftarrow 3 \text{ to } n\)
4. **do** Append \(p_i\) to \(L_{\text{upper}}\).
5. **while** \(L_{\text{upper}}\) contains more than two points **and** the last three points in \(L_{\text{upper}}\) do not make a right turn **do** Delete the middle of the last three points from \(L_{\text{upper}}\).
6. Put the points \(p_n\) and \(p_{n-1}\) in a list \(L_{\text{lower}}\), with \(p_n\) as the first point.
7. **for** \(i \leftarrow n-2 \text{ downto } 1\)
8. **do** Append \(p_i\) to \(L_{\text{lower}}\).
9. **while** \(L_{\text{lower}}\) contains more than 2 points **and** the last three points in \(L_{\text{lower}}\) do not make a right turn **do** Delete the middle of the last three points from \(L_{\text{lower}}\).
10. Remove the first and the last point from \(L_{\text{lower}}\) to avoid duplication of the points where the upper and lower hull meet.
11. Append \(L_{\text{lower}}\) to \(L_{\text{upper}}\), and call the resulting list \(L\).
12. **return** \(L\)
**Algorithm** FINDINTERSECTIONS($S$)

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

*Output.* The set of intersection points among the segments in $S$, with for each intersection point the segments that contain it.

1. Initialize an empty event queue $\Omega$. Next, insert the segment endpoints into $\Omega$; when an upper endpoint is inserted, the corresponding segment should be stored with it.
2. Initialize an empty status structure $\mathcal{T}$.
3. while $\Omega$ is not empty
4. do Determine the next event point $p$ in $\Omega$ and delete it.
5. HANDLEEVENTPOINT($p$)
HANDLE_EVENT_POINT(p)
1. 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.)
2. Find all segments stored in $T$ that contain $p$; they are adjacent in $T$. 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.
3. if $L(p) \cup U(p) \cup C(p)$ contains more than one segment
4. then Report $p$ as an intersection, together with $L(p)$, $U(p)$, and $C(p)$.
5. Delete the segments in $L(p) \cup C(p)$ from $T$.
6. Insert the segments in $U(p) \cup C(p)$ into $T$. The order of the segments in $T$ 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 $p$.
7. (* Deleting and re-inserting the segments of $C(p)$ reverses their order. *)
8. if $U(p) \cup C(p) = \emptyset$
9. then Let $s_l$ and $s_r$ be the left and right neighbors of $p$ in $T$.
10. FIND_NEWEVENT($s_l, s_r, p$)
11. else Let $s'$ be the leftmost segment of $U(p) \cup C(p)$ in $T$.
12. Let $s_l$ be the left neighbor of $s'$ in $T$.
13. FIND_NEWEVENT($s_l, s', p$)
14. Let $s''$ be the rightmost segment of $U(p) \cup C(p)$ in $T$.
15. Let $s_r$ be the right neighbor of $s''$ in $T$.
16. FIND_NEWEVENT($s'', s_r, p$)
\textbf{FINDNEWEVENT}(s_l, s_r, p)

1. \textbf{if} \hspace{1mm} s_l \text{ and } s_r \text{ intersect below the sweep line, or on it and to the right of the current event point } p, \text{ and the intersection is not yet present as an event in } \mathcal{Q} \\
2. \textbf{then} \hspace{1mm} \text{Insert the intersection point as an event into } \mathcal{Q}.
Algorithm MAPOverlay($S_1, S_2$)

Input. Two planar subdivisions $S_1$ and $S_2$ stored in doubly-connected edge lists.

Output. The overlay of $S_1$ and $S_2$ stored in a doubly-connected edge list $D$.

1. Copy the doubly-connected edge lists for $S_1$ and $S_2$ to a new doubly-connected edge list $D$.

2. Compute all intersections between edges from $S_1$ and $S_2$ with the plane sweep algorithm of Section 2.1. In addition to the actions on $T$ and $Q$ required at the event points, do the following:
   - Update $D$ as explained above if the event involves edges of both $S_1$ and $S_2$. (This was explained for the case where an edge of $S_1$ passes through a vertex of $S_2$.)
   - Store the half-edge immediately to the left of the event point at the vertex in $D$ representing it.

3. (* Now $D$ is the doubly-connected edge list for $O(S_1, S_2)$, except that the information about the faces has not been computed yet. *)

4. Determine the boundary cycles in $O(S_1, S_2)$ by traversing $D$.

5. Construct the graph $G$ 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 $G$ has been computed in line 2, second item.)

6. for each connected component in $G$

7. do Let $C$ 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 $C$, 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 $O(S_1, S_2)$ with the names of the faces of $S_1$ and $S_2$ containing it, as explained above.
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 \(\Omega\) 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** \(\Omega\) is not empty
4. **do** Remove the vertex \(v_i\) with the highest priority from \(\Omega\).
5. **Call** the appropriate procedure to handle the vertex, depending on its type.
\textbf{HANDLESTARTVERTEX}(v_i)

1. Insert $e_i$ in $I$ and set $\text{helper}(e_i)$ to $v_i$. 
HANDLE\textsc{EndVertex}(v_i)
1. \textbf{if} helper(e_{i-1}) \textbf{is a merge vertex}
2. \textbf{then} Insert the diagonal connecting v_i to helper(e_{i-1}) in D.
3. Delete e_{i-1} from \mathcal{T}.
HANDLE_SPLIT_VERTEX(v_i)
1. Search in T to find the edge e_j directly left of v_i.
2. Insert the diagonal connecting v_i to helper(e_j) in D.
3. \( \text{helper}(e_j) \leftarrow v_i \)
4. Insert \( e_i \) in T and set \( \text{helper}(e_i) \) to \( v_i \).
HANDLEMERGEVERTEX($v_i$)
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 $\mathcal{T}$.
4. Search in $\mathcal{T}$ to find the edge $e_j$ directly left of $v_i$.
5. if $helper(e_j)$ is a merge vertex
6. then Insert the diagonal connecting $v_i$ to $helper(e_j)$ in $\mathcal{D}$.
7. $helper(e_j) \leftarrow v_i$
HANDLE\textsc{RegularVertex}(v_i)
1. if the interior of $\mathcal{P}$ lies to the right of $v_i$
2. then if $\text{helper}(e_{i-1})$ is a merge vertex
3. \hspace{1em} then Insert the diagonal connecting $v_i$ to $\text{helper}(e_{i-1})$ in $\mathcal{D}$.
4. \hspace{1em} Delete $e_{i-1}$ from $\mathcal{T}$.
5. \hspace{1em} Insert $e_i$ in $\mathcal{T}$ and set $\text{helper}(e_i)$ to $v_i$.
6. else Search in $\mathcal{T}$ to find the edge $e_j$ directly left of $v_i$.
7. \hspace{1em} if $\text{helper}(e_j)$ is a merge vertex
8. \hspace{1em} then Insert the diagonal connecting $v_i$ to $\text{helper}(e_j)$ in $\mathcal{D}$.
9. \hspace{1em} $\text{helper}(e_j) \leftarrow v_i$
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 \(S\), 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 \(S\) are on different chains
5. then Pop all vertices from \(S\).
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 \(S\).
8. else Pop one vertex from \(S\).
9. Pop the other vertices from \(S\) 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 \(S\).
10. Push \(u_j\) onto \(S\).
11. Add diagonals from \(u_n\) to all stack vertices except the first and the last one.
**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** $\text{card}(H) = 1$
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. $C_1 \leftarrow \text{INTERSECTHALFPLANES}(H_1)$
5. $C_2 \leftarrow \text{INTERSECTHALFPLANES}(H_2)$
6. $C \leftarrow \text{INTERSECTCONVEXREGIONS}(C_1, C_2)$
Algorithm RANDOMPERMUTATION(A)

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($k$)
Algorithm 2D\textsc{RandomizedLP}(H, \vec{c})

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

\textbf{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.

1. Determine whether there is a direction vector \(\vec{d}\) such that \(\vec{d} \cdot \vec{c} > 0\) and \(\vec{d} \cdot \vec{\eta}(h) \geq 0\) for all \(h \in H\).
2. \textbf{if} \(\vec{d}\) exists
3. \hspace{1em} \textbf{then} compute \(H'\) and determine whether \(H'\) is feasible.
4. \hspace{1em} \textbf{if} \(H'\) is feasible
5. \hspace{2em} \textbf{then} Report a ray proving that \((H, \vec{c})\) is unbounded and quit.
6. \hspace{2em} \textbf{else} Report that \((H, \vec{c})\) is infeasible and quit.
7. Let \(h_1, h_2 \in H\) be certificates proving that \((H, \vec{c})\) is bounded and has a unique lexicographically smallest solution.
8. Let \(\ell_1, \ell_2\) be the intersection of \(\ell_1\) and \(\ell_2\).
9. Let \(h_3, h_4, \ldots, h_n\) be a random permutation of the remaining half-planes in \(H\).
10. \textbf{for} \(i \leftarrow 3\) to \(n\)
11. \hspace{1em} \textbf{do} if \(v_{i-1} \in h_i\)
12. \hspace{2em} \textbf{then} \(v_i \leftarrow v_{i-1}\)
13. \hspace{2em} \textbf{else} \(v_i \leftarrow \text{the point on } \ell_i \text{ that maximizes } f_{\vec{c}}(p), \text{ subject to the constraints in } H_{i-1}\).
14. \hspace{2em} \textbf{if} \(p\) does not exist
15. \hspace{3em} \textbf{then} Let \(h_j, h_k\) (with \(j, k < i\)) be the certificates (possibly \(h_j = h_k\)) with \(h_j \cap h_k \cap \ell_i = \emptyset\).
16. \hspace{3em} Report that the linear program is infeasible, with \(h_i, h_j, h_k\) as certificates, and quit.
17. \textbf{return} \(v_n\)
Algorithm $\text{RANDOMIZEDLP}(H, \overrightarrow{c})$

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

Output. If $(H, \overrightarrow{c})$ 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 $f_{\overrightarrow{c}}(p)$ is reported.

1. Determine whether a direction vector $\overrightarrow{d}$ exists such that $\overrightarrow{d} \cdot \overrightarrow{c} > 0$ and $\overrightarrow{d} \cdot \overrightarrow{\eta}(h) \geq 0$ for all $h \in H$.
2. if $\overrightarrow{d}$ exists
3. then compute $H'$ and determine whether $H'$ is feasible.
4. if $H'$ is feasible
5. then Report a ray proving that $(H, \overrightarrow{c})$ is unbounded and quit.
6. else Report that $(H, \overrightarrow{c})$ is infeasible, provide certificates, and quit.
7. Let $h_1, h_2, \ldots, h_d$ be certificates proving that $(H, \overrightarrow{c})$ is bounded.
8. Let $v_d$ be the intersection of $g_1, g_2, \ldots, g_d$.
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 $f_{\overrightarrow{c}}(p)$, 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 $v_n$
Algorithm MINIDISC(P)

Input. A set P of n points in the plane.
Output. The smallest enclosing disc for P.
1. Compute a random permutation $p_1, \ldots, p_n$ of P.
2. Let $D_2$ be the smallest enclosing disc for $\{p_1, p_2\}$.
3. for $i \leftarrow 3$ to $n$
   4. do if $p_i \in D_{i-1}$
   5. then $D_i \leftarrow D_{i-1}$
   6. else $D_i \leftarrow$ MINIDISCWITHPOINT($\{p_1, \ldots, p_{i-1}\}, p_i$)
4. return $D_n$
**MINIDISCWITHPOINT(P,q)**

*Input.* A set $P$ of $n$ points in the plane, and a point $q$ such that there exists an enclosing disc for $P$ with $q$ on its boundary.

*Output.* The smallest enclosing disc for $P$ with $q$ on its boundary.

1. Compute a random permutation $p_1, \ldots, p_n$ of $P$.
2. Let $D_1$ be the smallest disc with $q$ and $p_1$ on its boundary.
3. for $j \leftarrow 2$ to $n$
4. do if $p_j \in D_{j-1}$
5. then $D_j \leftarrow D_{j-1}$
6. else $D_j \leftarrow \text{MINIDISCWITH2POINTS}([p_1, \ldots, p_{j-1}], p_j, q)$
7. return $D_n$
MINIDISCWITH2POINTS($P, q_1, q_2$)

*Input.* A set $P$ of $n$ points in the plane, and two points $q_1$ and $q_2$ such that there exists an enclosing disc for $P$ with $q_1$ and $q_2$ on its boundary.

*Output.* The smallest enclosing disc for $P$ with $q_1$ and $q_2$ on its boundary.

1. Let $D_0$ be the smallest disc with $q_1$ and $q_2$ on its boundary.
2.  
   for $k ← 1$ to $n$
3.      do if $p_k ∈ D_{k−1}$
4.         then $D_k ← D_{k−1}$
5.      else $D_k ←$ the disc with $q_1, q_2$, and $p_k$ on its boundary
6.  return $D_n$
Algorithm PARANOIDMAXIMUM(A)
1. if card(A) = 1
2. then return the unique element $x \in A$
3. else Pick a random element $x$ from $A$.
4. $x' \leftarrow$ PARANOIDMAXIMUM($A \setminus \{x\}$)
5. if $x \leq x'$
6. then return $x'$
7. else Now we suspect that $x$ is the maximum, but to be absolutely sure, we compare $x$ with all card($A$) − 1 other elements of $A$.
8. return $x$
FIND_SPLITNODE(\(T, x, x'\))

*Input.* A tree \(T\) and two values \(x\) and \(x'\) with \(x \leq x'\).

*Output.* The node \(v\) where the paths to \(x\) and \(x'\) split, or the leaf where both paths end.

1. \(v \leftarrow \text{root}(T)\)
2. while \(v\) is not a leaf and \((x' \leq x_v \text{ or } x > x_v)\)
3. do if \(x' \leq x_v\)
4. then \(v \leftarrow \text{lc}(v)\)
5. else \(v \leftarrow \text{rc}(v)\)
6. return \(v\)
**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.

1. \(v_{\text{split}} \leftarrow \text{FindSplitNode}(\mathcal{T}, x, x')\)
2. if \(v_{\text{split}}\) is a leaf
3. then Check if the point stored at \(v_{\text{split}}\) must be reported.
4. else (* Follow the path to \(x\) and report the points in subtrees right of the path. *)
5. \(v \leftarrow lc(v_{\text{split}})\)
6. while \(v\) is not a leaf
7. do if \(x \leq x_v\)
8. then \(\text{ReportSubtree}(rc(v))\)
9. \(v \leftarrow lc(v)\)
10. else \(v \leftarrow rc(v)\)
11. Check if the point stored at the leaf \(v\) must be reported.
12. 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.
Algorithm BUILDKDTREE(P, depth)

Input. A set of points P and the current depth depth.
Output. The root of a kd-tree storing P.
1. if P contains only one point
2. then return a leaf storing this point
3. else if depth is even
4. then Split P into two subsets with a vertical line ℓ through the median x-coordinate of the points in P. Let P₁ be the set of points to the left of ℓ or on ℓ, and let P₂ be the set of points to the right of ℓ.
5. else Split P into two subsets with a horizontal line ℓ through the median y-coordinate of the points in P. Let P₁ be the set of points below ℓ or on ℓ, and let P₂ be the set of points above ℓ.
6. ν_left ← BUILDKDTREE(P₁, depth + 1)
7. ν_right ← BUILDKDTREE(P₂, depth + 1)
8. Create a node ν storing ℓ, make ν_left the left child of ν, and make ν_right the right child of ν.
9. return ν
Algorithm \( \text{SEARCHKdTree}(v, R) \)

Input. The root of (a subtree of) a \( \text{kd-tree} \), and a range \( R \).

Output. All points at leaves below \( v \) that lie in the range.

1. if \( v \) is a leaf
2. then Report the point stored at \( v \) if it lies in \( R \).
3. else if \( \text{region}(l(v)) \) is fully contained in \( R \)
   4. then \( \text{REPORTSUBTREE}(l(v)) \)
5. else if \( \text{region}(l(v)) \) intersects \( R \)
   6. then \( \text{SEARCHKdTREE}(l(v), R) \)
7. if \( \text{region}(r(v)) \) is fully contained in \( R \)
   8. then \( \text{REPORTSUBTREE}(r(v)) \)
9. else if \( \text{region}(r(v)) \) intersects \( R \)
   10. then \( \text{SEARCHKdTREE}(r(v), R) \)
Algorithm BUILD2DRANGE TREE(P)

Input. A set $P$ of points in the plane.
Output. The root of a 2-dimensional range tree.

1. Construct the associated structure: Build a binary search tree $T_{\text{assoc}}$ on the set $P_y$ of $y$-coordinates of the points in $P$. Store at the leaves of $T_{\text{assoc}}$ not just the $y$-coordinate of the points in $P_y$, but the points themselves.

2. If $P$ contains only one point
   3. then Create a leaf $v$ storing this point, and make $T_{\text{assoc}}$ the associated structure of $v$.
   4. else Split $P$ into two subsets; one subset $P_{\text{left}}$ contains the points with $x$-coordinate less than or equal to $x_{\text{mid}}$, the median $x$-coordinate, and the other subset $P_{\text{right}}$ contains the points with $x$-coordinate larger than $x_{\text{mid}}$.
   5. $v_{\text{left}} \leftarrow \text{BUILD2DRANGE TREE}(P_{\text{left}})$
   6. $v_{\text{right}} \leftarrow \text{BUILD2DRANGE TREE}(P_{\text{right}})$
   7. Create a node $v$ storing $x_{\text{mid}}$, make $v_{\text{left}}$ the left child of $v$, make $v_{\text{right}}$ the right child of $v$, and make $T_{\text{assoc}}$ the associated structure of $v$.
   8. return $v$
Algorithm 2DRangeQuery(T, [x : x′] × [y : y′])

Input. A 2-dimensional range tree T and a range [x : x′] × [y : y′].

Output. All points in T that lie in the range.

1. \( v_{split} \leftarrow \text{FindSplitNode}(T, x, x') \)
2. if \( v_{split} \) is a leaf
3. then Check if the point stored at \( v_{split} \) must be reported.
4. else (* Follow the path to \( x \) and call 1DRangeQuery on the subtrees right of the path. *)
5. \( v \leftarrow \text{lc}(v_{split}) \)
6. while \( v \) is not a leaf
7. do if \( x \leq x_v \)
8. then 1DRangeQuery(\( T_{assoc}(rc(v)) \), [y : y′])
9. \( v \leftarrow \text{lc}(v) \)
10. else \( v \leftarrow rc(v) \)
11. Check if the point stored at \( v \) must be reported.
12. Similarly, follow the path from \( rc(v_{split}) \) to \( x' \), call 1DRangeQuery with the range \([y : y']\) on the associated structures of subtrees left of the path, and check if the point stored at the leaf where the path ends must be reported.
**Algorithm** TRAPEZOIDAL-MAP($S$)

*Input.* A set $S$ of $n$ non-crossing line segments.

*Output.* The trapezoidal map $T(S)$ and a search structure $D$ for $T(S)$ in a bounding box.

1. Determine a bounding box $R$ that contains all segments of $S$, and initialize the trapezoidal map structure $T$ and search structure $D$ for it.
2. Compute a random permutation $s_1, s_2, \ldots, s_n$ of the elements of $S$.
3. **for** $i \leftarrow 1$ **to** $n$
   4. **do** Find the set $\Delta_0, \Delta_1, \ldots, \Delta_k$ of trapezoids in $T$ properly intersected by $s_i$.
   5. Remove $\Delta_0, \Delta_1, \ldots, \Delta_k$ from $T$ and replace them by the new trapezoids that appear because of the insertion of $s_i$.
   6. Remove the leaves for $\Delta_0, \Delta_1, \ldots, \Delta_k$ from $D$, 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 FOLLOW\textsc{SEGMENT}(\mathcal{T}, \mathcal{D}, s_i)

\textbf{Input.} A trapezoidal map \mathcal{T}, a search structure \mathcal{D} for \mathcal{T}, and a new segment \( s_i \).

\textbf{Output.} The sequence \( \Delta_0, \ldots, \Delta_k \) of trapezoids intersected by \( s_i \).

1. Let \( p \) and \( q \) be the left and right endpoint of \( s_i \).
2. Search with \( p \) in the search structure \( \mathcal{D} \) to find \( \Delta_0 \).
3. \( j \leftarrow 0; \)
4. \hspace{1em} \textbf{while} \( q \) lies to the right of \( \text{right}_p(\Delta_j) \)
5. \hspace{2em} \textbf{do if} \( \text{right}_p(\Delta_j) \) lies above \( s_i \)
6. \hspace{3em} \textbf{then} Let \( \Delta_{j+1} \) be the lower right neighbor of \( \Delta_j \).
7. \hspace{3em} \textbf{else} Let \( \Delta_{j+1} \) be the upper right neighbor of \( \Delta_j \).
8. \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1...
Algorithm \textsc{VoronoiDiagram}(P)

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

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

1. Initialize the event queue \( Q \) with all site events, initialize an empty status structure \( \mathcal{T} \) and an empty doubly-connected edge list \( \mathcal{D} \).
2. \textbf{while} \( Q \) is not empty
3. \hspace{1em} \textbf{do} Remove the event with largest \( y \)-coordinate from \( Q \).
4. \hspace{2em} \textbf{if} the event is a site event, occurring at site \( p_i \)
5. \hspace{3em} \textbf{then} \textsc{HandleSiteEvent}(p_i)
6. \hspace{2em} \textbf{else} \textsc{HandleCircleEvent}(\( \gamma \)), where \( \gamma \) 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 $\alpha$ vertically above $p_i$. If the leaf representing $\alpha$ 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 $\alpha$ 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 $\alpha$. 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 $T$. Update the tuples representing the breakpoints at the internal nodes. Perform rebalancing operations on $T$ if necessary. Delete all circle events involving $α$ from $Ω$; these can be found using the pointers from the predecessor and the successor of $γ$ in $T$. (The circle event where $α$ is the middle arc is currently being handled, and has already been deleted from $Ω$.)

2. Add the center of the circle causing the event as a vertex record to the doubly-connected edge list $D$ 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 $Ω$, and set pointers between the new circle event in $Ω$ and the corresponding leaf of $T$. Do the same for the triple where the former right neighbor is the middle arc.
Algorithm RETRACTION($S, q_{start}, q_{end}, r$)

**Input.** A set $S := \{s_1, \ldots, s_n\}$ 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 $G$ be the graph corresponding to the vertices and edges of the Voronoi diagram. Remove all edges from $G$ 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 $G$. If so, report the line segment from $q_{start}$ to $p_{start}$, the path in $G$ 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.
**Algorithm** \textsc{ConstructArrangement}(L)

\textit{Input.} A set \( L \) of \( n \) lines in the plane.

\textit{Output.} The doubly-connected edge list for the subdivision induced by \( B(L) \) and the part of \( A(L) \) inside \( B(L) \), where \( B(L) \) is a bounding box containing all vertices of \( A(L) \) in its interior.

1. Compute a bounding box \( B(L) \) that contains all vertices of \( A(L) \) in its interior.
2. Construct the doubly-connected edge list for the subdivision induced by \( B(L) \).
3. \textbf{for} \( i \leftarrow 1 \) \textbf{to} \( n \)
4. \hspace{5mm} \textbf{do} Find the edge \( e \) on \( B(L) \) that contains the leftmost intersection point of \( \ell_i \) and \( A_i \).
5. \hspace{5mm} \hspace{5mm} \( f \leftarrow \) the bounded face incident to \( e \)
6. \hspace{5mm} \hspace{5mm} \textbf{while} \( f \) is not the unbounded face, that is, the face outside \( B(L) \)
7. \hspace{5mm} \hspace{5mm} \hspace{5mm} \textbf{do} Split \( f \), and set \( f \) to be the next intersected face.
**Algorithm** LEGALTRIANGULATION($\mathcal{T}$)

*Input.* Some triangulation $\mathcal{T}$ of a point set $P$.

*Output.* A legal triangulation of $P$.

1. **while** $\mathcal{T}$ contains an illegal edge $p_ip_j$
2. **do** (∗ Flip $p_ip_j$ ∗)
3. Let $p_ip_jp_k$ and $p_ip_jp_l$ be the two triangles adjacent to $p_ip_j$.
4. Remove $p_ip_j$ from $\mathcal{T}$, and add $p_kp_l$ instead.
5. **return** $\mathcal{T}$
Algorithm DELAUNAYTRIANGULATION($P$)  

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

1. Let $p_0$ be the lexicographically highest point of $P$, that is, the rightmost among the points with largest $y$-coordinate.  
2. 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 $p_0p_{-1}p_{-2}$.  
3. Initialize $\mathcal{T}$ as the triangulation consisting of the single triangle $p_0p_{-1}p_{-2}$.  
4. Compute a random permutation $p_1, p_2, \ldots, p_n$ of $P \setminus \{p_0\}$.  
5. for $r ← 1$ to $n$  
6. do (* Insert $p_r$ into $\mathcal{T}$: *)  
7. Find a triangle $p_ip_jp_k \in \mathcal{T}$ containing $p_r$.  
8. if $p_r$ lies in the interior of the triangle $p_ip_jp_k$  
9. then Add edges from $p_r$ to the three vertices of $p_ip_jp_k$, thereby splitting $p_ip_jp_k$ into three triangles.  
10. LEGALIZEEDGE($p_r, p_ip_j, \mathcal{T}$)  
11. LEGALIZEEDGE($p_r, p_jp_k, \mathcal{T}$)  
12. LEGALIZEEDGE($p_r, p_kp_i, \mathcal{T}$)  
13. else (* $p_r$ lies on an edge of $p_ip_jp_k$, say the edge $p_ip_j$ *)  
14. Add edges from $p_r$ to $p_k$ and to the third vertex $p_l$ of the other triangle that is incident to $p_ip_j$, thereby splitting the two triangles incident to $p_ip_j$ into four triangles.  
15. LEGALIZEEDGE($p_r, p_ip_l, \mathcal{T}$)  
16. LEGALIZEEDGE($p_r, p_jp_k, \mathcal{T}$)  
17. LEGALIZEEDGE($p_r, p_kp_l, \mathcal{T}$)  
18. LEGALIZEEDGE($p_r, p_lp_i, \mathcal{T}$)  
19. Discard $p_{-1}$ and $p_{-2}$ with all their incident edges from $\mathcal{T}$.  
20. return $\mathcal{T}$
LEGALIZEEDGE($p_r, \overline{p_ip_j}, \mathcal{T}$)
1. (* The point being inserted is $p_r$, and $\overline{p_ip_j}$ is the edge of $\mathcal{T}$ that may need to be flipped. *)
2. if $\overline{p_ip_j}$ is illegal
3. then Let $p_ip_jp_k$ be the triangle adjacent to $p_ip_j$ along $\overline{p_ip_j}$.
4. (* Flip $\overline{p_ip_j}$: *) Replace $\overline{p_ip_j}$ with $\overline{p_ip_k}$.
5. LEGALIZEEDGE($p_r, \overline{p_ip_k}, \mathcal{T}$)
6. LEGALIZEEDGE($p_r, \overline{p_kp_j}, \mathcal{T}$)

37
Algorithm CONSTRUCTINTERVALTREE(I)

Input. A set I of intervals on the real line.

Output. The root of an interval tree for I.

1. if I = ∅
2. then return an empty leaf
3. else Create a node ν. Compute $x_{\text{mid}}$, the median of the set of interval endpoints, and store $x_{\text{mid}}$ with ν.
4. Compute $I_{\text{mid}}$ and construct two sorted lists for $I_{\text{mid}}$: a list $L_{\text{left}}(\nu)$ sorted on left endpoint and a list $L_{\text{right}}(\nu)$ sorted on right endpoint. Store these two lists at ν.
5. $lc(\nu) \leftarrow$ CONSTRUCTINTERVALTREE($I_{\text{left}}$)
6. $rc(\nu) \leftarrow$ CONSTRUCTINTERVALTREE($I_{\text{right}}$)
7. return ν
Algorithm \textsc{QueryIntervalTree}(v, q_x)

\textit{Input.} The root $v$ of an interval tree and a query point $q_x$.

\textit{Output.} All intervals that contain $q_x$.

1. \textbf{if} $v$ is not a leaf
2. \textbf{then if} $q_x < x_{\text{mid}}(v)$
3. \textbf{then} Walk along the list $L_{\text{left}}(v)$, starting at the interval with the leftmost endpoint, reporting all the intervals that contain $q_x$. Stop as soon as an interval does not contain $q_x$.
4. \textbf{else} \textsc{QueryIntervalTree}(lc(v), q_x)
5. \textbf{else} Walk along the list $L_{\text{right}}(v)$, starting at the interval with the rightmost endpoint, reporting all the intervals that contain $q_x$. Stop as soon as an interval does not contain $q_x$.
6. \textsc{QueryIntervalTree}(rc(v), q_x)
\textsc{ReportInSubtree}(v, q_x) \\
\textit{Input.} The root $v$ of a subtree of a priority search tree and a value $q_x$. \\
\textit{Output.} All points in the subtree with $x$-coordinate at most $q_x$. \\
1. \textbf{if} $v$ is not a leaf and $(p(v))_x \leq q_x$ \\
2. \textbf{then} Report $p(v)$. \\
3. \textbf{ReportInSubtree}(lc(v), q_x) \\
4. \textbf{ReportInSubtree}(rc(v), q_x)
Algorithm $\text{QUERYPrioSearchTree}(\mathcal{T}, (-\infty : q_x] \times [q_y : q_y'])$  

**Input.** A priority search tree and a range, unbounded to the left.  

**Output.** All points lying in the range.  

1. Search with $q_y$ and $q_y'$ in $\mathcal{T}$. Let $v_{\text{split}}$ be the node where the two search paths split.  
2. for each node $v$ on the search path of $q_y$ or $q_y'$  
3. do if $p(v) \in (-\infty : q_x] \times [q_y : q_y']$ then report $p(v)$.  
4. for each node $v$ on the path of $q_y$ in the left subtree of $v_{\text{split}}$  
5. do if the search path goes left at $v$  
6. then REPORTINSUBTREE($rc(v), q_x$)  
7. for each node $v$ on the path of $q_y'$ in the right subtree of $v_{\text{split}}$  
8. do if the search path goes right at $v$  
9. then REPORTINSUBTREE($lc(v), q_x$)
Algorithm $\text{QUERYSEGMENTTREE}(v, q_x)$

*Input.* The root of a (subtree of a) segment tree and a query point $q_x$.

*Output.* All intervals in the tree containing $q_x$.

1. Report all the intervals in $I(v)$.
2. if $v$ is not a leaf
3. then if $q_x \in \text{Int}(lc(v))$
4. then $\text{QUERYSEGMENTTREE}(lc(v), q_x)$
5. else $\text{QUERYSEGMENTTREE}(rc(v), q_x)$
**Algorithm** \textsc{InsertSegmentTree}(v, [x : x'])

*Input.* The root of a (subtree of a) segment tree and an interval.

*Output.* The interval will be stored in the subtree.

1. if \( \text{Int}(v) \subseteq [x : x'] \)
2. then store \([x : x']\) at \(v\)
3. else if \( \text{Int}(\text{lc}(v)) \cap [x : x'] \neq \emptyset \)
4. then \textsc{InsertSegmentTree}(\text{lc}(v), [x : x'])
5. if \( \text{Int}(\text{rc}(v)) \cap [x : x'] \neq \emptyset \)
6. then \textsc{InsertSegmentTree}(\text{rc}(v), [x : x'])
Algorithm **CONVEXHULL**(P)

*Input.* A set P of n points in three-space.

*Output.* The convex hull \(\mathcal{C}(P)\) of P.

1. Find four points \(p_1, p_2, p_3, p_4\) in P that form a tetrahedron.
2. \(\mathcal{C} \leftarrow \mathcal{C}(\{p_1, p_2, p_3, p_4\})\)
3. Compute a random permutation \(p_5, p_6, \ldots, p_n\) of the remaining points.
4. Initialize the conflict graph \(\mathcal{G}\) with all visible pairs \((p_t, f)\), where \(f\) is a facet of \(\mathcal{C}\) and \(t > 4\).
5. for \(r \leftarrow 5\) to \(n\)
6. do (* Insert \(p_r\) into \(\mathcal{C}\): *)
7. if \(F_{\text{conflict}}(p_r)\) is not empty (* that is, \(p_r\) lies outside \(\mathcal{C}\) *)
8. then Delete all facets in \(F_{\text{conflict}}(p_r)\) from \(\mathcal{C}\).
9. 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.
10. for all \(e \in \mathcal{L}\)
11. do Connect \(e\) to \(p_r\) by creating a triangular facet \(f\).
12. if \(f\) is coplanar with its neighbor facet \(f'\) along \(e\)
13. then Merge \(f\) and \(f'\) into one facet, whose conflict list is the same as that of \(f'\).
14. else (* Determine conflicts for \(f\): *)
15. Create a node for \(f\) in \(\mathcal{G}\).
16. Let \(f_1\) and \(f_2\) be the facets incident to \(e\) in the old convex hull.
17. \(P(e) \leftarrow P_{\text{conflict}}(f_1) \cup P_{\text{conflict}}(f_2)\)
18. for all points \(p \in P(e)\)
19. do If \(f\) is visible from \(p\), add \((p, f)\) to \(\mathcal{G}\).
20. 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.
21. return \(\mathcal{C}\)
Algorithm \textsc{PaintersAlgorithm}(\mathcal{T}, p_{\text{view}})

1. Let $v$ be the root of $\mathcal{T}$.
2. \textbf{if} $v$ is a leaf \textbf{then} Scan-convert the object fragments in $S(v)$.
3. \textbf{else if} $p_{\text{view}} \in h_v^+$ \textbf{then} \textsc{PaintersAlgorithm}(\mathcal{T}^-, p_{\text{view}})
4. \hspace{1em} \text{Scan-convert the object fragments in } S(v).
5. \hspace{1em} \textsc{PaintersAlgorithm}(\mathcal{T}^+, p_{\text{view}})
6. \textbf{else if} $p_{\text{view}} \in h_v^-$ \textbf{then} \textsc{PaintersAlgorithm}(\mathcal{T}^+, p_{\text{view}})
7. \hspace{1em} \text{Scan-convert the object fragments in } S(v).
8. \hspace{1em} \textsc{PaintersAlgorithm}(\mathcal{T}^-, p_{\text{view}})
9. \textbf{else} (* p_{\text{view}} \in h_v *)
10. \hspace{1em} \textsc{PaintersAlgorithm}(\mathcal{T}^+, p_{\text{view}})
11. \hspace{1em} \textsc{PaintersAlgorithm}(\mathcal{T}^-, p_{\text{view}})
12. \hspace{1em} \textsc{PaintersAlgorithm}(\mathcal{T}^-, p_{\text{view}})
Algorithm 2DBSP(S)

Input. A set \( S = \{s_1, s_2, \ldots, s_n\} \) of segments.

Output. A BSP tree for \( S \).

1. if \( \text{card}(S) \leq 1 \) then Create a tree \( T \) consisting of a single leaf node, where the set \( S \) is stored explicitly.
2. return \( T \)
3. else (* Use \( \ell(s_1) \) as the splitting line. *)
4. \( S^+ \leftarrow \{s \cap \ell(s_1)^+ : s \in S\} \); \( \mathcal{T}^+ \leftarrow 2\text{DBSP}(S^+) \)
5. \( S^- \leftarrow \{s \cap \ell(s_1)^- : s \in S\} \); \( \mathcal{T}^- \leftarrow 2\text{DBSP}(S^-) \)
6. Create a BSP tree \( \mathcal{T} \) with root node \( \nu \), left subtree \( \mathcal{T}^- \), right subtree \( \mathcal{T}^+ \), and with \( S(\nu) = \{s \in S : s \subset \ell(s_1)\} \).
7. return \( \mathcal{T} \)
**Algorithm** 2DRANDOMBsp($S$)

1. Generate a random permutation $S' = s_1, \ldots, s_n$ of the set $S$.
2. $T \leftarrow 2$DBsp($S'$)
3. return $T$
Algorithm 3DBSP(S)

Input. A set $S = \{t_1, t_2, \ldots, t_n\}$ of triangles in $\mathbb{R}^3$.

Output. A BSP tree for $S$.

1. if $\text{card}(S) \leq 1$
2. then Create a tree $T$ consisting of a single leaf node, where the set $S$ is stored explicitly.
3. return $T$
4. else (* Use $h(t_1)$ as the splitting plane. *)
5. $S^+ \leftarrow \{t \cap h(t_1)^+ : t \in S\}$; $T^+ \leftarrow 3\text{DBSP}(S^+)$
6. $S^- \leftarrow \{t \cap h(t_1)^- : t \in S\}$; $T^- \leftarrow 3\text{DBSP}(S^-)$
7. Create a BSP tree $T$ with root node $\nu$, left subtree $T^-$, right subtree $T^+$, and with $S(\nu) = \{t \in S : t \subset h(t_1)\}$.
8. return $T$
Algorithm 3D RANDOM BSP2(S)

Input. A set $S = \{t_1, t_2, \ldots, t_n\}$ of triangles in $\mathbb{R}^3$.

Output. A BSP tree for $S$.

1. Generate a random permutation $t_1, \ldots, t_n$ of the set $S$.
2. for $i ← 1$ to $n$
3.   do Use $h(t_i)$ to split every cell where the split is useful.
4.     Make all possible free splits.
Algorithm PHASE1$(\sigma, G, k)$

*Input.* A region $\sigma$, a set $G$ of guards in the interior of $\sigma$, and an integer $k \geq 1$.

*Output.* A BSP tree $T$ such that each leaf region contains at most $k$ guards.

1. if $\text{card}(G) \leq k$
2. then Create a BSP tree $T$ consisting of a single leaf node.
3. else if exactly one quadrant of $\sigma$ contains more than $k$ guards in its interior
4. then Determine the splitting lines $\ell_v(\sigma)$ and $\ell_h(\sigma)$ for a shrinking step, as explained above.
5. else Determine the splitting lines $\ell_v(\sigma)$ and $\ell_h(\sigma)$ for a quadtree split, as explained above.
6. Create a BSP tree $T$ with three internal nodes; the root of $T$ stores $\ell_v(\sigma)$ as its splitting line, and both children of the root store $\ell_h(\sigma)$ as their splitting line.
7. Replace each leaf $\mu$ of $T$ by a BSP tree $T_\mu$ computed recursively on the region corresponding to $\mu$ and the guards inside that region.
8. return $T$
Algorithm LOWDENSITYBSP2D(S)
Input. A set $S$ of $n$ objects in the plane.
Output. A BSP tree $T$ for $S$.
1. Let $G(S)$ be the set of $4n$ bounding-box vertices of the objects in $S$.
2. $k ← 1$; $done ← false$; $U ←$ a bounding square of $S$
3. while not done
4. do $k ← 2k$; $T ←$ PHASE1($U, G(S), k$); $done ← true$
5. for each leaf $µ$ of $T$
6. do Compute the set $S(µ)$ of object fragments in the region of $µ$.
7. if $\text{card}(S(µ)) > 5k$ then $done ← false$
8. for each leaf $µ$ of $T$
9. do Compute a BSP tree $T_µ$ for $S(µ)$ and replace $µ$ by $T_µ$.
10. return $T$
**Algorithm** \textsc{ComputeFreeSpace}(S)

*Input.* A set $S$ of disjoint polygons.

*Output.* A trapezoidal map of $c_{\text{free}}(R,S)$ for a point robot $R$.

1. Let $E$ be the set of edges of the polygons in $S$.
2. Compute the trapezoidal map $\mathcal{T}(E)$ with algorithm \textsc{TrapezoidalMap} described in Chapter 6.
3. Remove the trapezoids that lie inside one of the polygons from $\mathcal{T}(E)$ and return the resulting subdivision.
Algorithm \text{COMPUTE}\text{PATH}(\mathcal{T}(C_{\text{free}}), \mathcal{G}_{\text{road}}, p_{\text{start}}, p_{\text{goal}})

Input. The trapezoidal map $\mathcal{T}(C_{\text{free}})$ of the free space, the road map $\mathcal{G}_{\text{road}}$, a start position $p_{\text{start}}$, and goal position $p_{\text{goal}}$.

Output. A path from $p_{\text{start}}$ to $p_{\text{goal}}$ if it exists. If a path does not exist, this fact is reported.

1. Find the trapezoid $\Delta_{\text{start}}$ containing $p_{\text{start}}$ and the trapezoid $\Delta_{\text{goal}}$ containing $p_{\text{goal}}$.
2. if $\Delta_{\text{start}}$ or $\Delta_{\text{goal}}$ does not exist
3. then Report that the start or goal position is in the forbidden space.
4. else Let $v_{\text{start}}$ be the node of $\mathcal{G}_{\text{road}}$ in the center of $\Delta_{\text{start}}$.
5. Let $v_{\text{goal}}$ be the node of $\mathcal{G}_{\text{road}}$ in the center of $\Delta_{\text{goal}}$.
6. Compute a path in $\mathcal{G}_{\text{road}}$ from $v_{\text{start}}$ to $v_{\text{goal}}$ using breadth-first search.
7. if there is no such path
8. then Report that there is no path from $p_{\text{start}}$ to $p_{\text{goal}}$.
9. else Report the path consisting of a straight-line motion from $p_{\text{start}}$ to $v_{\text{start}}$, the path found in $\mathcal{G}_{\text{road}}$, and a straight-line motion from $v_{\text{goal}}$ to $p_{\text{goal}}$. 
**Algorithm MinkowskiSum**($\mathcal{P}, \mathcal{R}$)

*Input.* A convex polygon $\mathcal{P}$ with vertices $v_1, \ldots, v_n$, and a convex polygon $\mathcal{R}$ with vertices $w_1, \ldots, w_m$. The lists of vertices are assumed to be in counterclockwise order, with $v_1$ and $w_1$ being the vertices with smallest $y$-coordinate (and smallest $x$-coordinate in case of ties).

*Output.* The Minkowski sum $\mathcal{P} \oplus \mathcal{R}$.

1. $i \leftarrow 1$; $j \leftarrow 1$
2. $v_{n+1} \leftarrow v_1$; $v_{n+2} \leftarrow v_2$; $w_{m+1} \leftarrow w_1$; $w_{m+2} \leftarrow w_2$
3. **repeat**
4. Add $v_i + w_j$ as a vertex to $\mathcal{P} \oplus \mathcal{R}$.
5. **if** $\text{angle}(v_iv_{i+1}) < \text{angle}(w_jw_{j+1})$
6. **then** $i \leftarrow (i + 1)$
7. **else if** $\text{angle}(v_iv_{i+1}) > \text{angle}(w_jw_{j+1})$
8. **then** $j \leftarrow (j + 1)$
9. **else** $i \leftarrow (i + 1)$; $j \leftarrow (j + 1)$
10. **until** $i = n + 1$ and $j = m + 1$
**Algorithm** FORBIDDENSPACE(\(\mathcal{C}P_1, \ldots, \mathcal{C}P_n\))

**Input.** A collection of \(\mathcal{C}\)-obstacles.

**Output.** The forbidden space \(\mathcal{C}_{\text{forb}} = \bigcup_{i=1}^{n} \mathcal{C}P_i\).

1. if \(n = 1\)
2. then return \(\mathcal{C}P_1\)
3. else \(\mathcal{C}^1_{\text{forb}} \leftarrow \text{FORBIDDENSPACE}(\mathcal{P}_1, \ldots, \mathcal{P}_{\lceil n/2 \rceil})\)
4. \(\mathcal{C}^2_{\text{forb}} \leftarrow \text{FORBIDDENSPACE}(\mathcal{P}_{\lceil n/2 \rceil + 1}, \ldots, \mathcal{P}_n)\)
5. Compute \(\mathcal{C}_{\text{forb}} = \mathcal{C}^1_{\text{forb}} \cup \mathcal{C}^2_{\text{forb}}\)
6. return \(\mathcal{C}_{\text{forb}}\)
Algorithm \textsc{NorthNeighbor}(v, T)

\textit{Input.} A node \(v\) in a quadtree \(T\).

\textit{Output.} The deepest node \(v'\) whose depth is at most the depth of \(v\) such that \(\sigma(v')\) is a north-neighbor of \(\sigma(v)\), and \texttt{nil}\ if there is no such node.

1. \textbf{if} \(v = \text{root}(T)\) \textbf{then return nil}
2. \textbf{if} \(v = \text{SW-child of parent}(v)\) \textbf{then return NW-child of parent}(v)
3. \textbf{if} \(v = \text{SE-child of parent}(v)\) \textbf{then return NE-child of parent}(v)
4. \(\mu \leftarrow \textsc{NorthNeighbor}(\text{parent}(v), T)\)
5. \textbf{if} \(\mu = \texttt{nil}\) or \(\mu\) is a leaf \textbf{then return}\ \(\mu\)
6. \textbf{else if} \(v = \text{NW-child of parent}(v)\)
7. \textbf{then return SW-child of } \mu
8. \textbf{else return SE-child of } \mu
**Algorithm** \textsc{BalanceQuadTree}($\mathcal{T}$)  

*Input.* A quadtree $\mathcal{T}$.  

*Output.* A balanced version of $\mathcal{T}$.  

1. Insert all the leaves of $\mathcal{T}$ into a linear list $\mathcal{L}$.  
2. \textbf{while} $\mathcal{L}$ is not empty \textbf{do}  
3.  
4. \textbf{if} $\sigma(\mu)$ has to be split \textbf{then}  
5.  
6. \textbf{if} $\sigma(\mu)$ has a neighbor that now needs to be split \textbf{then}  
7.  
8. \textbf{return} $\mathcal{T}$
Algorithm GENERATE_MESH(S)
Input. A set $S$ of components inside the square $[0 : U] \times [0 : U]$ with the properties stated at the beginning of this section.
Output. A triangular mesh $M$ that is conforming, respects the input, consists of well-shaped triangles, and is non-uniform.

1. Construct a quadtree $T$ on the set $S$ inside the square $[0 : U] \times [0 : U]$ with the following stopping criterion: a square is split as long as it is larger than unit size and its closure intersects the boundary of some component.
2. $T \leftarrow$ BALANCE_QUAD_TREE($T$)
3. Construct the doubly-connected edge list for the quadtree subdivision $M$ corresponding to $T$.
4. for each face $\sigma$ of $M$
5. do if the interior of $\sigma$ is intersected by an edge of a component
6. then Add the intersection (which is a diagonal) as an edge to $M$.
7. else if $\sigma$ has only vertices at its corners
8. then Add a diagonal of $\sigma$ as an edge to $M$.
9. else Add a Steiner point in the center of $\sigma$, connect it to all vertices on the boundary of $\sigma$, and change $M$ accordingly.
10. return $M$
Algorithm SHORTESTPATH(\(S, p_{\text{start}}, p_{\text{goal}}\))

Input. A set \(S\) of disjoint polygonal obstacles, and two points \(p_{\text{start}}\) and \(p_{\text{goal}}\) in the free space.

Output. The shortest collision-free path connecting \(p_{\text{start}}\) and \(p_{\text{goal}}\).

1. \(G_{\text{vis}} \leftarrow \text{VISIBILITYGRAPH}(S \cup \{p_{\text{start}}, p_{\text{goal}}\})\)
2. Assign each arc \((v, w)\) in \(G_{\text{vis}}\) a weight, which is the Euclidean length of the segment \(\overline{vw}\).
3. Use Dijkstra’s algorithm to compute a shortest path between \(p_{\text{start}}\) and \(p_{\text{goal}}\) in \(G_{\text{vis}}\).
Algorithm $\text{VISIBILITY\GRAPH}(S)$

*Input.* A set $S$ of disjoint polygonal obstacles.

*Output.* The visibility graph $G_{\text{vis}}(S)$.

1. Initialize a graph $G = (V, E)$ where $V$ is the set of all vertices of the polygons in $S$ and $E = \emptyset$.
2. for all vertices $v \in V$
3. do $W \leftarrow \text{VISIBLE\VERTICES}(v, S)$
4. For every vertex $w \in W$, add the arc $(v, w)$ to $E$.
5. return $G$
Algorithm \textsc{VisibleVertices}(p, S)

\textit{Input.} A set $S$ of polygonal obstacles and a point $p$ that does not lie in the interior of any obstacle.

\textit{Output.} The set of all obstacle vertices visible from $p$.

1. Sort the obstacle vertices according to the clockwise angle that the half-line from $p$ to each vertex makes with the positive $x$-axis. In case of ties, vertices closer to $p$ should come before vertices farther from $p$. Let $w_1, \ldots, w_n$ be the sorted list.

2. Let $\rho$ be the half-line parallel to the positive $x$-axis starting at $p$. Find the obstacle edges that are properly intersected by $\rho$, and store them in a balanced search tree $T$ in the order in which they are intersected by $\rho$.

3. $W \leftarrow \emptyset$

4. \textbf{for} $i \leftarrow 1 \text{ to } n$

5. \hspace{1em} \textbf{do if} \textsc{Visible}(w_i) \textbf{then} Add $w_i$ to $W$.

6. \hspace{1em} Insert into $T$ the obstacle edges incident to $w_i$ that lie on the clockwise side of the half-line from $p$ to $w_i$.

7. \hspace{1em} Delete from $T$ the obstacle edges incident to $w_i$ that lie on the counterclockwise side of the half-line from $p$ to $w_i$.

8. \textbf{return} $W$
VISIBLE($w_i$)
1. if $p_w_i$ intersects the interior of the obstacle of which $w_i$ is a vertex, locally at $w_i$
2. then return false
3. else if $i = 1$ or $w_{i-1}$ is not on the segment $p_w_i$
4. then Search in $T$ for the edge $e$ in the leftmost leaf.
5. if $e$ exists and $p_w_i$ intersects $e$
6. then return false
7. else return true
8. else if $w_{i-1}$ is not visible
9. then return false
10. else Search in $T$ for an edge $e$ that intersects $w_{i-1}w_i$.
11. if $e$ exists
12. then return false
13. else return true
**Algorithm** \( \text{SELECTINHALFPLANE}(h, \mathcal{T}) \)

*Input.* A query half-plane \( h \) and a partition tree or subtree of it.

*Output.* A set of canonical nodes for all points in the tree that lie in \( h \).

1. \( \Upsilon \leftarrow \emptyset \)
2. if \( \mathcal{T} \) consists of a single leaf \( \mu \)
3. then if the point stored at \( \mu \) lies in \( h \) then \( \Upsilon \leftarrow \{ \mu \} \)
4. else for each child \( v \) of the root of \( \mathcal{T} \)
5. do if \( t(v) \subseteq h \)
6. then \( \Upsilon \leftarrow \Upsilon \cup \{ v \} \)
7. else if \( t(v) \cap h \neq \emptyset \)
8. then \( \Upsilon \leftarrow \Upsilon \cup \text{SELECTINHALFPLANE}(h, \mathcal{T}_v) \)
9. return \( \Upsilon \)
Algorithm SELECTINSEGMENTS(ℓ, T)

Input. A query line ℓ and a partition tree or subtree of it.
Output. A set of canonical nodes for all segments in the tree that are intersected by ℓ.

1. \( \Upsilon \leftarrow \emptyset \)
2. if \( T \) consists of a single leaf \( \mu \)
3. then if the segment stored at \( \mu \) intersects \( \ell \) then \( \Upsilon \leftarrow \{ \mu \} \)
4. else for each child \( \nu \) of the root of \( T \)
5. do if \( t(\nu) \subset \ell^+ \)
6. then \( \Upsilon \leftarrow \Upsilon \cup \text{SELECTINHALFPLANE}(\ell^-, T^{assoc}_\nu) \)
7. else if \( t(\nu) \cap \ell \neq \emptyset \)
8. then \( \Upsilon \leftarrow \Upsilon \cup \text{SELECTINSEGMENTS}(\ell, T_\nu) \)
9. return \( \Upsilon \)
Algorithm \textsc{SelectBelowPoint}(q, \mathcal{T})

\textit{Input.} A query point \( q \) and a cutting tree or subtree of it.

\textit{Output.} A set of canonical nodes for all lines in the tree that lie below \( q \).

1. \( \Upsilon \leftarrow \emptyset \)
2. \textbf{if} \( \mathcal{T} \) consists of a single leaf \( \mu \)
3. \quad \textbf{then if} the line stored at \( \mu \) lies below \( q \) \textbf{then} \( \Upsilon \leftarrow \{ \mu \} \)
4. \quad \textbf{else} \textbf{for} each child \( \nu \) of the root of \( \mathcal{T} \)
5. \quad \quad \textbf{do} Check if \( q \) lies in \( t(\nu) \).
6. \quad \quad \textbf{Let} \( v_q \) be the child such that \( q \in t(v_q) \).
7. \quad \quad \Upsilon \leftarrow \{ v_q \} \cup \textsc{SelectBelowPoint}(q, \mathcal{T}_{v_q})
8. \textbf{return} \( \Upsilon \)
Algorithm `SELECTBELOWPAIR(q_1, q_2, \mathcal{T})`

*Input.* Two query points \( q_1 \) and \( q_2 \) and a cutting tree or subtree of it.

*Output.* A set of canonical nodes for all lines in the tree that lie below \( q_1 \) and \( q_2 \).

1. \( \Upsilon \leftarrow \emptyset \)
2. if \( \mathcal{T} \) consists of a single leaf \( \mu \)
3. then if the line stored at \( \mu \) lies below \( q_1 \) and \( q_2 \) then \( \Upsilon \leftarrow \{ \mu \} \)
4. else for each child \( v \) of the root of \( \mathcal{T} \)
5. do Check if \( q_1 \) lies in \( t(v) \).
6. Let \( v_{q_1} \) be the child such that \( q_1 \in t(v_{q_1}) \).
7. \( \Upsilon_1 \leftarrow SELECTBELOWPOINT(q_2, \mathcal{T}_{assoc}^{v_{q_1}}) \)
8. \( \Upsilon_2 \leftarrow SELECTBELOWPAIR(q_1, q_2, \mathcal{T}_{v_{q_1}}) \)
9. \( \Upsilon \leftarrow \Upsilon_1 \cup \Upsilon_2 \)
10. return \( \Upsilon \)