Algorithm SLOWCONVEXHULL(P)
Input. A set P of points in the plane.
Output. A list $\mathcal{L}$ containing the vertices of $\mathcal{CH}(P)$ in clockwise order.

```
1. $E \leftarrow \emptyset$.
2. for all ordered pairs $(p,q) \in P \times P$ with $p$ not equal to $q$
3.     do $valid \leftarrow$ true
4.        for all points $r \in P$ not equal to $p$ or $q$
5.            do if $r$ lies to the left of the directed line from $p$ to $q$
6.                  then $valid \leftarrow$ false.
7.        if $valid$ then Add the directed edge $\overrightarrow{pq}$ to $E$.
8. From the set $E$ of edges construct a list $\mathcal{L}$ of vertices of $\mathcal{CH}(P)$, 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 $p_1, \dots, p_n$.
2.  Put the points $p_1$ and $p_2$ in a list $\mathcal{L}_{\text{upper}}$, with $p_1$ as the first point.
3.  for $i \leftarrow 3$ to $n$
4.      do Append $p_i$ to $\mathcal{L}_{\text{upper}}$.
5.         while $\mathcal{L}_{\text{upper}}$ contains more than two points and the last three points in $\mathcal{L}_{\text{upper}}$ do
               not make a right turn
6.             do Delete the middle of the last three points from $\mathcal{L}_{\text{upper}}$.
7.  Put the points $p_n$ and $p_{n-1}$ in a list $\mathcal{L}_{\text{lower}}$, with $p_n$ as the first point.
8.  for $i \leftarrow n-2$ downto 1
9.      do Append $p_i$ to $\mathcal{L}_{\text{lower}}$.
10.        while $\mathcal{L}_{\text{lower}}$ contains more than 2 points and the last three points in $\mathcal{L}_{\text{lower}}$ do not
               make a right turn
11.            do Delete the middle of the last three points from $\mathcal{L}_{\text{lower}}$.
12. Remove the first and the last point from $\mathcal{L}_{\text{lower}}$ to avoid duplication of the points where the
    upper and lower hull meet.
13. Append $\mathcal{L}_{\text{lower}}$ to $\mathcal{L}_{\text{upper}}$, and call the resulting list $\mathcal{L}$.
14. return $\mathcal{L}$
```

```
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 $\mathcal{Q}$. Next, insert the segment endpoints into $\mathcal{Q}$; when an
     upper endpoint is inserted, the corresponding segment should be stored with it.
2.   Initialize an empty status structure $\mathcal{T}$.
3.   while $\mathcal{Q}$ is not empty
4.       do Determine the next event point $p$ in $\mathcal{Q}$ and delete it.
5.          HANDLEEVENTPOINT($p$)
```

```
HANDLEEVENTPOINT($p$)
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 $\mathcal{T}$ 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.
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 $\mathcal{T}$.
6.  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 $p$.
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 $\mathcal{T}$.
10.        FINDNEWEVENT($s_l, s_r, p$)
11.   else Let $s'$ be the leftmost segment of $U(p) \cup C(p)$ in $\mathcal{T}$.
12.        Let $s_l$ be the left neighbor of $s'$ in $\mathcal{T}$.
13.        FINDNEWEVENT($s_l, s', p$)
14.        Let $s''$ be the rightmost segment of $U(p) \cup C(p)$ in $\mathcal{T}$.
15.        Let $s_r$ be the right neighbor of $s''$ in $\mathcal{T}$.
16.        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.
```

```
**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 $v_i$.
```

```
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($v_i$)
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.  $helper(e_j) \leftarrow v_i$
```

```
HANDLEREGULARVERTEX($v_i$)
1.  if the interior of $\mathcal{P}$ lies to the right of $v_i$
2.    then if $helper(e_{i-1})$ is a merge vertex
3.           then Insert the diagonal connecting $v_i$ to $helper(e_{i-1})$ in $\mathcal{D}$.
4.         Delete $e_{i-1}$ from $\mathcal{T}$.
5.         Insert $e_i$ in $\mathcal{T}$ and set $helper(e_i)$ to $v_i$.
6.    else Search in $\mathcal{T}$ to find the edge $e_j$ directly left of $v_i$.
7.         if $helper(e_j)$ is a merge vertex
8.           then Insert the diagonal connecting $v_i$ to $helper(e_j)$ in $\mathcal{D}$.
9.         $helper(e_j) \leftarrow v_i$
```

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 $u_1, \dots, u_n$ 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.
```

```
**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($C_1, C_2$)
```

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 $A[rndindex]$.
```

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.

```
1.  Determine whether there is a direction vector $\vec{d}$ such that $\vec{d} \cdot \vec{c} > 0$ and $\vec{d} \cdot \vec{\eta}(h) \geqslant 0$ 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 and quit.
7.  Let $h_1, h_2 \in H$ be certificates proving that $(H, \vec{c})$ is bounded and has a unique lexicographi-
    cally smallest solution.
8.  Let $v_2$ be the intersection of $\ell_1$ and $\ell_2$.
9.  Let $h_3, h_4, \dots, h_n$ be a random permutation of the remaining half-planes in $H$.
10. for $i \leftarrow 3$ 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 $\ell_i$ that maximizes $f_{\vec{c}}(p)$, subject to the constraints in
                  $H_{i-1}$.
14.               if $p$ does not exist
15.                 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.                      Report that the linear program is infeasible, with $h_i, h_j, h_k$ as certifi-
                         cates, and quit.
17. return $v_n$
```

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 $g_1, g_2, \ldots, g_d$.
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 $f_{\vec{c}}(p)$, 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 certifi-
                          cates, 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($\{p_1, \ldots, p_{i-1}\}, p_i$)
7.  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 $D_j \leftarrow$ 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 $k \leftarrow 1$ to $n$
3.      do if $p_k \in D_{k-1}$
4.            then $D_k \leftarrow D_{k-1}$
5.            else  $D_k \leftarrow$the disc with $q_1$, $q_2$, and $p_k$ on its boundary
6.  return $D_n$
```

```
Algorithm ParanoidMaximum($A$)
1. if $\mathrm{card}(A) = 1$
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 $x \leqslant x'$
6.            then return $x'$
7.            else Now we suspect that $x$ is the maximum, but to be absolutely sure, we
                   compare $x$ with all $\mathrm{card}(A) - 1$ other elements of $A$.
8.                 return $x$
```

```
FINDSPLITNODE($\mathcal{T}, x, x'$)
Input. A tree $\mathcal{T}$ and two values $x$ and $x'$ with $x \leqslant x'$.
Output. The node $\nu$ where the paths to $x$ and $x'$ split, or the leaf where both paths end.
1.  $\nu \leftarrow root(\mathcal{T})$
2.  while $\nu$ is not a leaf and ($x' \leqslant x_\nu$ or $x > x_\nu$)
3.     do if $x' \leqslant x_\nu$
4.           then $\nu \leftarrow lc(\nu)$
5.           else $\nu \leftarrow rc(\nu)$
6.  return $\nu$
```

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.   $\nu_{\text{split}} \leftarrow$ FINDSPLITNODE($\mathcal{T}$, $x$, $x'$)
2.   if $\nu_{\text{split}}$ is a leaf
3.     then Check if the point stored at $\nu_{\text{split}}$ must be reported.
4.     else (* Follow the path to $x$ and report the points in subtrees right of the path. *)
5.          $\nu \leftarrow lc(\nu_{\text{split}})$
6.          while $\nu$ is not a leaf
7.            do if $x \leqslant x_\nu$
8.                 then REPORTSUBTREE($rc(\nu)$)
9.                      $\nu \leftarrow lc(\nu)$
10.                else $\nu \leftarrow rc(\nu)$
11.         Check if the point stored at the leaf $\nu$ 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 $\ell$ through the median $x$-coordinate
                     of the points in P. Let $P_1$ be the set of points to the left of $\ell$ or on $\ell$, and let
                     $P_2$ be the set of points to the right of $\ell$.
5.              else Split P into two subsets with a horizontal line $\ell$ through the median $y$-
                     coordinate of the points in P. Let $P_1$ be the set of points below $\ell$ or on $\ell$,
                     and let $P_2$ be the set of points above $\ell$.
6.           $v_{\text{left}} \leftarrow$ BuildKdTree($P_1$, $depth+1$)
7.           $v_{\text{right}} \leftarrow$ BuildKdTree($P_2$, $depth+1$)
8.           Create a node $v$ storing $\ell$, make $v_{\text{left}}$ the left child of $v$, and make $v_{\text{right}}$ the right
             child of $v$.
9.           return $v$
```

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.

```
1.  if ν is a leaf
2.    then Report the point stored at ν if it lies in R.
3.    else if region(lc(ν)) is fully contained in R
4.           then REPORTSUBTREE(lc(ν))
5.           else if region(lc(ν)) intersects R
6.                  then SEARCHKDTREE(lc(ν), R)
7.         if region(rc(ν)) is fully contained in R
8.           then REPORTSUBTREE(rc(ν))
9.           else if region(rc(ν)) intersects R
10.                 then SEARCHKDTREE(rc(ν), R)
```

```
Algorithm BUILD2DRANGETREE($P$)
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 $\mathcal{T}_{\text{assoc}}$ on the set $P_y$ of y-
    coordinates of the points in $P$. Store at the leaves of $\mathcal{T}_{\text{assoc}}$ 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 $\nu$ storing this point, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of $\nu$.
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.         $\nu_{\text{left}} \leftarrow$ BUILD2DRANGETREE($P_{\text{left}}$)
6.         $\nu_{\text{right}} \leftarrow$ BUILD2DRANGETREE($P_{\text{right}}$)
7.         Create a node $\nu$ storing $x_{\text{mid}}$, make $\nu_{\text{left}}$ the left child of $\nu$, make $\nu_{\text{right}}$ the right
           child of $\nu$, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of $\nu$.
8.  return $\nu$
```

```
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.
1.  $\nu_{\text{split}} \leftarrow$FindSplitNode($\mathcal{T}, x, x'$)
2.  if $\nu_{\text{split}}$ is a leaf
3.    then Check if the point stored at $\nu_{\text{split}}$ must be reported.
4.    else (* Follow the path to $x$ and call 1DRangeQuery on the subtrees right of the
           path. *)
5.         $\nu \leftarrow lc(\nu_{\text{split}})$
6.         while $\nu$ is not a leaf
7.            do if $x \leqslant x_\nu$
8.                  then 1DRangeQuery($\mathcal{T}_{\text{assoc}}(rc(\nu)), [y : y']$)
9.                       $\nu \leftarrow lc(\nu)$
10.                 else $\nu \leftarrow rc(\nu)$
11.        Check if the point stored at $\nu$ must be reported.
12.        Similarly, follow the path from $rc(\nu_{\text{split}})$ 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 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.
1.  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.
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 $\mathcal{T}$ properly intersected by $s_i$.
5.         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 $s_i$.
6.         Remove the leaves for $\Delta_0, \Delta_1, \ldots, \Delta_k$ from $\mathcal{D}$, and create leaves for the new trape-
           zoids. 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 $s_i$.
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.  while $q$ lies to the right of $rightp(\Delta_j)$
5.     do if $rightp(\Delta_j)$ lies above $s_i$
6.           then Let $\Delta_{j+1}$ be the lower right neighbor of $\Delta_j$.
7.           else  Let $\Delta_{j+1}$ be the upper right neighbor of $\Delta_j$.
8.         $j \leftarrow j+1$
9.  return $\Delta_0, \Delta_1, \ldots, \Delta_j$
```

```
Algorithm VORONOIDIAGRAM($P$)
Input. A set $P := \{p_1, \dots, p_n\}$ 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($\gamma$), 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($\gamma$)
1.  Delete the leaf $\gamma$ that represents the disappearing arc $\alpha$ 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 $\alpha$ from $\mathcal{Q}$; these can be found using the pointers from the predecessor and the successor of $\gamma$ in $\mathcal{T}$. (The circle event where $\alpha$ 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 $\alpha$ 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.
```

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 Vor($S$) of $S$ inside a sufficiently large bounding box.
2.  Locate the cells of Vor($P$) that contain $q_{\text{start}}$ and $q_{\text{end}}$.
3.  Determine the point $p_{\text{start}}$ on Vor($S$) by moving $q_{\text{start}}$ away from the nearest line segment in
    $S$. Similarly, determine the point $p_{\text{end}}$ on Vor($S$) by moving $q_{\text{end}}$ away from the nearest line
    segment in $S$. Add $p_{\text{start}}$ and $p_{\text{end}}$ as vertices to 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 $p_{\text{start}}$ to $p_{\text{end}}$ in $\mathcal{G}$. If so, report
    the line segment from $q_{\text{start}}$ to $p_{\text{start}}$, the path in $\mathcal{G}$ from $p_{\text{start}}$ to $p_{\text{end}}$, and the line segment
    from $p_{\text{end}}$ to $q_{\text{end}}$ as the path. Otherwise, report that no path exists.
```

Algorithm CONSTRUCTARRANGEMENT(L)
Input. A set L of n lines in the plane.
Output. The doubly-connected edge list for the subdivision induced by $\mathcal{B}(L)$ and the part of $\mathcal{A}(L)$ inside $\mathcal{B}(L)$, where $\mathcal{B}(L)$ is a bounding box containing all vertices of $\mathcal{A}(L)$ in its interior.

```
1.  Compute a bounding box $\mathcal{B}(L)$ that contains all vertices of $\mathcal{A}(L)$ in its interior.
2.  Construct the doubly-connected edge list for the subdivision induced by $\mathcal{B}(L)$.
3.  for $i \leftarrow 1$ to $n$
4.      do Find the edge $e$ on $\mathcal{B}(L)$ that contains the leftmost intersection point of $\ell_i$ and $\mathcal{A}_i$.
5.         $f \leftarrow$ the bounded face incident to $e$
6.         while $f$ is not the unbounded face, that is, the face outside $\mathcal{B}(L)$
7.             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 $\overline{p_i p_j}$
2.      do (* Flip $\overline{p_i p_j}$ *)
3.          Let $p_i p_j p_k$ and $p_i p_j p_l$ be the two triangles adjacent to $\overline{p_i p_j}$.
4.          Remove $\overline{p_i p_j}$ from $\mathcal{T}$, and add $\overline{p_k p_l}$ 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 $p_0 p_{-1} p_{-2}$.
3.  Initialize $\mathcal{T}$ as the triangulation consisting of the single triangle $p_0 p_{-1} p_{-2}$.
4.  Compute a random permutation $p_1, p_2, \ldots, p_n$ of $P \setminus \{p_0\}$.
5.  for $r \leftarrow 1$ to $n$
6.      do (* Insert $p_r$ into $\mathcal{T}$: *)
7.         Find a triangle $p_i p_j p_k \in \mathcal{T}$ containing $p_r$.
8.         if $p_r$ lies in the interior of the triangle $p_i p_j p_k$
9.           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.
10.               LEGALIZEEDGE($p_r, \overline{p_i p_j}, \mathcal{T}$)
11.               LEGALIZEEDGE($p_r, \overline{p_j p_k}, \mathcal{T}$)
12.               LEGALIZEEDGE($p_r, \overline{p_k p_i}, \mathcal{T}$)
13.          else (* $p_r$ lies on an edge of $p_i p_j p_k$, say the edge $\overline{p_i p_j}$ *)
14.               Add edges from $p_r$ to $p_k$ and to the third vertex $p_l$ of the other triangle that
                  is incident to $\overline{p_i p_j}$, thereby splitting the two triangles incident to $\overline{p_i p_j}$ into
                  four triangles.
15.               LEGALIZEEDGE($p_r, \overline{p_i p_l}, \mathcal{T}$)
16.               LEGALIZEEDGE($p_r, \overline{p_l p_j}, \mathcal{T}$)
17.               LEGALIZEEDGE($p_r, \overline{p_j p_k}, \mathcal{T}$)
18.               LEGALIZEEDGE($p_r, \overline{p_k p_i}, \mathcal{T}$)
19. Discard $p_{-1}$ and $p_{-2}$ with all their incident edges from $\mathcal{T}$.
20. return $\mathcal{T}$
```

```
LEGALIZEEDGE($p_r, \overline{p_i p_j}, \mathcal{T}$)
1.  (* 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. *)
2.  if $\overline{p_i p_j}$ is illegal
3.     then Let $p_i p_j p_k$ be the triangle adjacent to $p_r p_i p_j$ along $\overline{p_i p_j}$.
4.          (* Flip $\overline{p_i p_j}$: *) Replace $\overline{p_i p_j}$ with $\overline{p_r p_k}$.
5.          LEGALIZEEDGE($p_r, \overline{p_i p_k}, \mathcal{T}$)
6.          LEGALIZEEDGE($p_r, \overline{p_k p_j}, \mathcal{T}$)
```

```
Algorithm ConstructIntervalTree($I$)
Input. A set $I$ of intervals on the real line.
Output. The root of an interval tree for $I$.
1.  if $I = \emptyset$
2.     then return an empty leaf
3.     else Create a node $\nu$. Compute $x_{\text{mid}}$, the median of the set of interval endpoints, and
            store $x_{\text{mid}}$ with $\nu$.
4.          Compute $I_{\text{mid}}$ and construct two sorted lists for $I_{\text{mid}}$: a list $\mathcal{L}_{\text{left}}(\nu)$ sorted on left
            endpoint and a list $\mathcal{L}_{\text{right}}(\nu)$ sorted on right endpoint. Store these two lists at $\nu$.
5.          $lc(\nu) \leftarrow$ ConstructIntervalTree($I_{\text{left}}$)
6.          $rc(\nu) \leftarrow$ ConstructIntervalTree($I_{\text{right}}$)
7.          return $\nu$
```

```
Algorithm QueryIntervalTree($\nu, q_x$)
Input. The root $\nu$ of an interval tree and a query point $q_x$.
Output. All intervals that contain $q_x$.
1.  if $\nu$ is not a leaf
2.    then if $q_x < x_{\text{mid}}(\nu)$
3.           then Walk along the list $\mathcal{L}_{\text{left}}(\nu)$, starting at the interval with the leftmost end-
                  point, reporting all the intervals that contain $q_x$. Stop as soon as an interval
                  does not contain $q_x$.
4.                QueryIntervalTree($lc(\nu), q_x$)
5.           else Walk along the list $\mathcal{L}_{\text{right}}(\nu)$, 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.                QueryIntervalTree($rc(\nu), q_x$)
```

```
REPORTINSUBTREE($\nu, q_x$)
Input. The root $\nu$ of a subtree of a priority search tree and a value $q_x$.
Output. All points in the subtree with $x$-coordinate at most $q_x$.
1.  if $\nu$ is not a leaf and $(p(\nu))_x \leqslant q_x$
2.     then Report $p(\nu)$.
3.          REPORTINSUBTREE($lc(\nu), q_x$)
4.          REPORTINSUBTREE($rc(\nu), q_x$)
```

```
Algorithm 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 $\nu_{\text{split}}$ be the node where the two search paths split.
2.  for each node $\nu$ on the search path of $q_y$ or $q'_y$
3.      do if $p(\nu) \in (-\infty : q_x] \times [q_y : q'_y]$ then report $p(\nu)$.
4.  for each node $\nu$ on the path of $q_y$ in the left subtree of $\nu_{\text{split}}$
5.      do if the search path goes left at $\nu$
6.          then ReportInSubtree($rc(\nu), q_x$)
7.  for each node $\nu$ on the path of $q'_y$ in the right subtree of $\nu_{\text{split}}$
8.      do if the search path goes right at $\nu$
9.          then ReportInSubtree($lc(\nu), q_x$)
```

Algorithm QUERYSEGMENTTREE(ν, 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 $I(\nu)$.
2.  if $\nu$ is not a leaf
3.     then if $q_x \in \mathrm{Int}(lc(\nu))$
4.             then QUERYSEGMENTTREE($lc(\nu), q_x$)
5.             else QUERYSEGMENTTREE($rc(\nu), q_x$)
```

Algorithm InsertSegmentTree($\nu, [x:x']$)
Input. The root of a (subtree of a) segment tree and an interval.
Output. The interval will be stored in the subtree.

```
1.  if Int($\nu$) $\subseteq [x:x']$
2.    then store $[x:x']$ at $\nu$
3.    else  if Int($lc(\nu)$) $\cap [x:x'] \neq \emptyset$
4.            then InsertSegmentTree($lc(\nu), [x:x']$)
5.          if Int($rc(\nu)$) $\cap [x:x'] \neq \emptyset$
6.            then InsertSegmentTree($rc(\nu), [x:x']$)
```

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

```
1.  Find four points $p_1, p_2, p_3, p_4$ in $P$ that form a tetrahedron.
2.  $\mathcal{C} \leftarrow \mathcal{CH}(\{p_1, p_2, p_3, p_4\})$
3.  Compute a random permutation $p_5, p_6, \dots, p_n$ 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 PaintersAlgorithm($\mathcal{T}, p_{\text{view}}$)
1.  Let $\nu$ be the root of $\mathcal{T}$.
2.  if $\nu$ is a leaf
3.    then Scan-convert the object fragments in $S(\nu)$.
4.    else if $p_{\text{view}} \in h_\nu^+$
5.           then PaintersAlgorithm($\mathcal{T}^-, p_{\text{view}}$)
6.                Scan-convert the object fragments in $S(\nu)$.
7.                PaintersAlgorithm($\mathcal{T}^+, p_{\text{view}}$)
8.           else if $p_{\text{view}} \in h_\nu^-$
9.                  then PaintersAlgorithm($\mathcal{T}^+, p_{\text{view}}$)
10.                      Scan-convert the object fragments in $S(\nu)$.
11.                      PaintersAlgorithm($\mathcal{T}^-, p_{\text{view}}$)
12.                 else (* $p_{\text{view}} \in h_\nu$ *)
13.                      PaintersAlgorithm($\mathcal{T}^+, p_{\text{view}}$)
14.                      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 card($S$) $\leqslant 1$
2.      then Create a tree $\mathcal{T}$ consisting of a single leaf node, where the set $S$ is stored explicitly.
3.           return $\mathcal{T}$
4.      else (* Use $\ell(s_1)$ as the splitting line. *)
5.           $S^+ \leftarrow \{s \cap \ell(s_1)^+ : s \in S\}$;     $\mathcal{T}^+ \leftarrow$ 2DBSP($S^+$)
6.           $S^- \leftarrow \{s \cap \ell(s_1)^- : s \in S\}$;     $\mathcal{T}^- \leftarrow$ 2DBSP($S^-$)
7.           Create a BSP tree $\mathcal{T}$ with root node $\nu$, left subtree $\mathcal{T}^-$, right subtree $\mathcal{T}^+$, and with
             $S(\nu) = \{s \in S : s \subset \ell(s_1)\}$.
8.           return $\mathcal{T}$
```

```
Algorithm 2dRandomBsp($S$)
1.  Generate a random permutation $S' = s_1, \ldots, s_n$ of the set $S$.
2.  $\mathcal{T} \leftarrow$ 2dBsp($S'$)
3.  return $\mathcal{T}$
```

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

```
Algorithm 3DRANDOMBSP2($S$)
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 $i \leftarrow 1$ 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(σ, G, k)
Input. A region σ, a set G of guards in the interior of σ, and an integer $k \geqslant 1$.
Output. A BSP tree $\mathcal{T}$ such that each leaf region contains at most k guards.

```
1.  if card$(G) \leqslant k$
2.    then Create a BSP tree $\mathcal{T}$ 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 ex-
                     plained above.
5.              else Determine the splitting lines $\ell_v(\sigma)$ and $\ell_h(\sigma)$ for a quadtree split, as ex-
                     plained above.
6.           Create a BSP tree $\mathcal{T}$ with three internal nodes; the root of $\mathcal{T}$ 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 $\mathcal{T}$ by a BSP tree $\mathcal{T}_\mu$ computed recursively on the region
             corresponding to $\mu$ and the guards inside that region.
8.  return $\mathcal{T}$
```

```
Algorithm LowDensityBSP2d($S$)
Input. A set $S$ of $n$ objects in the plane.
Output. A BSP tree $\mathcal{T}$ for $S$.
1.  Let $G(S)$ be the set of $4n$ bounding-box vertices of the objects in $S$.
2.  $k \leftarrow 1$; $done \leftarrow$ false; $U \leftarrow$ a bounding square of $S$
3.  while not $done$
4.      do $k \leftarrow 2k$; $\mathcal{T} \leftarrow$Phase1$(U, G(S), k)$; $done \leftarrow$ true
5.          for each leaf $\mu$ of $\mathcal{T}$
6.              do Compute the set $S(\mu)$ of object fragments in the region of $\mu$.
7.                  if card$(S(\mu)) > 5k$ then $done \leftarrow$ false
8.   for each leaf $\mu$ of $\mathcal{T}$
9.       do Compute a BSP tree $\mathcal{T}_\mu$ for $S(\mu)$ and replace $\mu$ by $\mathcal{T}_\mu$.
10.  return $\mathcal{T}$
```

```
Algorithm COMPUTEFREESPACE($S$)
Input. A set $S$ of disjoint polygons.
Output. A trapezoidal map of $\mathcal{C}_{\text{free}}(\mathcal{R}, S)$ for a point robot $\mathcal{R}$.
1.  Let $E$ be the set of edges of the polygons in $S$.
2.  Compute the trapezoidal map $\mathcal{T}(E)$ with algorithm 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 COMPUTEPATH($\mathcal{T}(\mathcal{C}_{\text{free}}), \mathcal{G}_{\text{road}}, p_{\text{start}}, p_{\text{goal}}$)
Input. The trapezoidal map $\mathcal{T}(\mathcal{C}_{\text{free}})$ of the free space, the road map $\mathcal{G}_{\text{road}}$, a start position p_{start}, and goal position p_{goal}.
Output. A path from p_{start} to p_{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 $angle(v_i v_{i+1}) < angle(w_j w_{j+1})$
6.        then $i \leftarrow (i+1)$
7.        else if $angle(v_i v_{i+1}) > angle(w_j w_{j+1})$
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($\mathcal{CP}_1, \ldots, \mathcal{CP}_n$)
Input. A collection of $\mathcal{C}$-obstacles.
Output. The forbidden space $\mathcal{C}_{\text{forb}} = \bigcup_{i=1}^{n} \mathcal{CP}_i$.
1.  if $n = 1$
2.     then return $\mathcal{CP}_1$
3.     else  $\mathcal{C}^1_{\text{forb}} \leftarrow$ForbiddenSpace($\mathcal{P}_1, \ldots, \mathcal{P}_{\lceil n/2 \rceil}$)
4.           $\mathcal{C}^2_{\text{forb}} \leftarrow$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 NORTHNEIGHBOR($\nu, \mathcal{T}$)
Input. A node ν in a quadtree $\mathcal{T}$.
Output. The deepest node ν' whose depth is at most the depth of ν such that $\sigma(\nu')$ is a north-neighbor of $\sigma(\nu)$, and **nil** if there is no such node.

```
1.  if $\nu = root(\mathcal{T})$ then return nil
2.  if $\nu =$ SW-child of $parent(\nu)$ then return NW-child of $parent(\nu)$
3.  if $\nu =$ SE-child of $parent(\nu)$ then return NE-child of $parent(\nu)$
4.  $\mu \leftarrow$ NORTHNEIGHBOR($parent(\nu), \mathcal{T}$)
5.  if $\mu =$ nil or $\mu$ is a leaf
6.    then return $\mu$
7.    else if $\nu =$ NW-child of $parent(\nu)$
8.           then return SW-child of $\mu$
9.           else return SE-child of $\mu$
```

```
Algorithm 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.  while $\mathcal{L}$ is not empty
3.     do Remove a leaf $\mu$ from $\mathcal{L}$.
4.        if $\sigma(\mu)$ has to be split
5.           then Make $\mu$ into an internal node with four children, which are leaves that
                  correspond to the four quadrants of $\sigma(\mu)$. If $\mu$ stores a point, then store the
                  point in the correct new leaf instead.
6.                Insert the four new leaves into $\mathcal{L}$.
7.                Check if $\sigma(\mu)$ had neighbors that now need to be split and, if so, insert them
                  into $\mathcal{L}$.
8.  return $\mathcal{T}$
```

```
Algorithm GenerateMesh($S$)
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 $\mathcal{M}$ that is conforming, respects the input, consists of well-shaped
   triangles, and is non-uniform.
1.  Construct a quadtree $\mathcal{T}$ 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.  $\mathcal{T} \leftarrow$ BalanceQuadTree($\mathcal{T}$)
3.  Construct the doubly-connected edge list for the quadtree subdivision $\mathcal{M}$ corresponding to
    $\mathcal{T}$.
4.  for each face $\sigma$ of $\mathcal{M}$
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 $\mathcal{M}$.
7.            else if $\sigma$ has only vertices at its corners
8.                   then Add a diagonal of $\sigma$ as an edge to $\mathcal{M}$.
9.                   else Add a Steiner point in the center of $\sigma$, connect it to all vertices on
                          the boundary of $\sigma$, and change $\mathcal{M}$ accordingly.
10. return $\mathcal{M}$
```

```
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.  $\mathcal{G}_{\text{vis}} \leftarrow$ VisibilityGraph($S \cup \{p_{\text{start}}, p_{\text{goal}}\}$)
2.  Assign each arc $(v, w)$ in $\mathcal{G}_{\text{vis}}$ 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 $\mathcal{G}_{\text{vis}}$.
```

Algorithm VISIBILITYGRAPH(S)
Input. A set S of disjoint polygonal obstacles.
Output. The visibility graph $\mathcal{G}_{\text{vis}}(S)$.

```
1.  Initialize a graph $\mathcal{G} = (V, E)$ 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 $W \leftarrow$ VISIBLEVERTICES($v, S$)
4.         For every vertex $w \in W$, add the arc $(v, w)$ to $E$.
5.  return $\mathcal{G}$
```

```
Algorithm VisibleVertices($p, S$)
Input. A set $S$ of polygonal obstacles and a point $p$ that does not lie in the interior of any obstacle.
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 $w_1, \dots, w_n$ 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 $\mathcal{T}$ in the order
      in which they are intersected by $\rho$.
3.  $W \leftarrow \emptyset$
4.  for $i \leftarrow 1$ to $n$
5.      do if Visible($w_i$) then Add $w_i$ to $W$.
6.          Insert into $\mathcal{T}$ the obstacle edges incident to $w_i$ that lie on the clockwise side of the
              half-line from $p$ to $w_i$.
7.          Delete from $\mathcal{T}$ the obstacle edges incident to $w_i$ that lie on the counterclockwise side
              of the half-line from $p$ to $w_i$.
8.  return $W$
```

```
VISIBLE($w_i$)
1.  if $\overline{pw_i}$ 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 $\overline{pw_i}$
4.           then Search in $\mathcal{T}$ for the edge $e$ in the leftmost leaf.
5.                if $e$ exists and $\overline{pw_i}$ 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 $\mathcal{T}$ for an edge $e$ that intersects $\overline{w_{i-1}w_i}$.
11.                      if $e$ exists
12.                        then return false
13.                        else return true
```

Algorithm 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 $\nu$ of the root of $\mathcal{T}$
5.               do if $t(\nu) \subset h$
6.                    then $\Upsilon \leftarrow \Upsilon \cup \{\nu\}$
7.                    else if $t(\nu) \cap h \neq \emptyset$
8.                            then $\Upsilon \leftarrow \Upsilon \cup$ SELECTINHALFPLANE($h, \mathcal{T}_\nu$)
9.  return $\Upsilon$
```

Algorithm SELECTINTSEGMENTS($\ell, \mathcal{T}$)
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 $\mathcal{T}$ 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 $\mathcal{T}$
5.              do if $t(\nu) \subset \ell^+$
6.                    then $\Upsilon \leftarrow \Upsilon \cup$ SELECTINHALFPLANE($\ell^-, \mathcal{T}_\nu^{\text{assoc}}$)
7.                    else if $t(\nu) \cap \ell \neq \emptyset$
8.                            then $\Upsilon \leftarrow \Upsilon \cup$ SELECTINTSEGMENTS($\ell, \mathcal{T}_\nu$)
9.  return $\Upsilon$
```

```
**Algorithm** SELECTBELOWPOINT($q, \mathcal{T}$)
*Input.* A query point $q$ 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.  $\Upsilon \leftarrow \emptyset$
2.  if $\mathcal{T}$ consists of a single leaf $\mu$
3.    then if the line stored at $\mu$ lies below $q$ then $\Upsilon \leftarrow \{\mu\}$
4.    else for each child $\nu$ of the root of $\mathcal{T}$
5.              do Check if $q$ lies in $t(\nu)$.
6.           Let $\nu_q$ be the child such that $q \in t(\nu_q)$.
7.           $\Upsilon \leftarrow \{\nu_q\} \cup$ SELECTBELOWPOINT($q, \mathcal{T}_{\nu_q}$)
8.  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 $\nu$ of the root of $\mathcal{T}$
5.                do Check if $q_1$ lies in $t(\nu)$.
6.            Let $\nu_{q_1}$ be the child such that $q_1 \in t(\nu_{q_1})$.
7.            $\Upsilon_1 \leftarrow$ SELECTBELOWPOINT($q_2, \mathcal{T}^{\text{assoc}}_{\nu_{q_1}}$)
8.            $\Upsilon_2 \leftarrow$ SELECTBELOWPAIR($q_1, q_2, \mathcal{T}_{\nu_{q_1}}$)
9.            $\Upsilon \leftarrow \Upsilon_1 \cup \Upsilon_2$
10. return $\Upsilon$
```