COMP4045 Computational Geometry Assignment 2

Enoch Lau
SID 200415765

10 May 2007

Contents

Note about notation: In algorithms, ▷ will be used to denote a line containing comments.

1 Question 1: d-dimensional Kd-Trees

1.1 Part a)

1.1.1 Algorithm

We present BUILD-KD-TREE in Algorithm 1 that builds a d-dimensional kd-tree given a set of points P. We assume that no two points share the same coordinate; if this is not the case, it is easily remedied by the use of a induced lexicographic ordering using a composite-number space[1]. Let $p_{(i)}$ be the value of the i-th dimension of a point p.

Algorithm 1 BUILD-KD-TREE(P, *depth*)

Input: A set of d-dimensional points P and the current depth *depth*
Output: The root of a kd-tree storing P

1: **if** P contains one point **then**
2: **return** a leaf storing this point
3: **else**
4: ▷ *Calculate the dimension to use for the split*
5: $i \leftarrow (depth + 1) \mod d$
6: ▷ *Divide the points into two halves*
7: $l \leftarrow \lfloor n/2 \rfloor$-th smallest value of the i-th dimension of points in P
8: $P_1 \leftarrow \{p \in P : p_{(i)} \leq l\}$
9: $P_2 \leftarrow \{p \in P : p_{(i)} > l\}$
10: ▷ *Construct node*
11: $v_l \leftarrow$ BUILD-KD-TREE(P_1, $depth + 1$)
12: $v_r \leftarrow$ BUILD-KD-TREE(P_2, $depth + 1$)
13: Create a node v storing the dimension i and the value at which the split was performed l, and make v_l the left child of v and v_r the right child of v
14: **return** v
15: **end if**

The depth parameter is zero at the first call, and it is used to determine the dimension that we use to split up the points; for the root, for example, the first dimension is used. For a split using dimension i, the points are roughly split in two by a hyperplane perpendicular to the x_i-axis. Otherwise, the algorithm proceeds as per the two-dimensional case. The algorithm will terminate; because of the choice of the value l, the median of two values is the lower one.

1.1.2 Storage Complexity

Theorem 1. *The kd-tree created by Algorithm 1 uses linear storage.*

Proof. We assume that the number of dimensions d is a constant. Each leaf in the kd-tree stores a distinct point of P, and thus there are n leaves. The tree is a binary tree, and so, there are $O(n)$ nodes. Each leaf and internal node consumes $O(1)$ storage, so in total, the amount of storage required is $O(n)$. □

[1] Section 5.5 of *Computational Geometry: Algorithms and Applications* (2nd ed.), by M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf

1.1.3 Time Complexity

Theorem 2. *Algorithm 1 runs in $O(n \log n)$ time.*

Proof. We again assume that the number of dimensions d is a constant. Let $T(n)$ be the time for constructing the kd-tree. The base case executes in $O(1)$ time. For the recursive case, we incur the following costs:

- Calculating the dimension i: $O(1)$
- Finding the split value l: $O(n)$[2]
- Dividing P into P_1 and P_2: $O(n)$
- Recursive calls: $2T(\lceil n/2 \rceil)$
- Constructing return value: $O(1)$

Adding it all up, this gives us the recurrence

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n) + 2T(\lceil n/2 \rceil) & \text{if } n > 1 \end{cases}$$

which is solved by $T(n) = O(n \log n)$, as required. □

1.2 Part b)

1.2.1 Algorithm

The query algorithm for the d-dimensional kd-tree is similar to the query algorithm for the two-dimensional case, with the only change being to recognise the additional dimensions in certain statements; the algorithm SEARCH-KD-TREE from the textbook is presented in reproduced in Algorithm 2. Starting from the root, one of three things can happen to either the left or right child of a node:

- If the child's region is completed surrounded by the query region, report all points stored at the leaves underneath it.
- If the child's region intersects partially with the query region, recursively perform a query on that child.
- If the child's region does not intersect at all with the query region, go no further.

We denote the region represented by a node v by $region(v)$, the left child by $lc(v)$ and the right child by $rc(v)$. The subroutine REPORT-SUBTREE simply traverses a subtree and reports all points at the leaves.

[2] See, for example, SELECT in section 9.3 of *Introduction to Algorithms* (2nd ed.), by T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein.

Algorithm 2 SEARCH-KD-TREE(v, R)

Input: The root v of a d-dimensional kd-tree, and a d-dimensional range R
Output: All points at leaves below v that lie in R

```
1: if v is a leaf then
2:    Report the point stored at v if it lies in R
3: else
4:    if region(lc(v)) is fully contained in R then
5:       REPORT-SUBTREE(lc(v))
6:    else if region(lc(v)) intersects R then
7:       SEARCH-KD-TREE(lc(v)), R
8:    end if
9:    if region(rc(v)) is fully contained in R then
10:      REPORT-SUBTREE(rc(v))
11:   else if region(rc(v)) intersects R then
12:      SEARCH-KD-TREE(rc(v)), R
13:   end if
14: end if
```

When checking whether *region*($lc(v)$) is fully contained in R or whether *region*($rc(v)$) is fully contained in R, this is done by checking that the range of values covered by *region*($lc(v)$) or *region*($rc(v)$) for each dimension i is bounded by the range of values covered by the i-th dimension in R. The region for each node can be computed during the pre-processing stage, or this can be maintained by combining the split values in the nodes visited. When checking whether *region*($lc(v)$) intersects R, this is done by checking if the lower bound for the i-th dimension in R is less than or equal to the split value l. Similarly, when checking whether *region*($rc(v)$) intersects R, this is done by checking if the upper bound for the i-th dimension in R is greater than the split value l.

1.2.2 Time Complexity

Theorem 3. *A query with an axis-parallel d-dimensional range in a kd-tree storing n points can be performed in $O(n^{1-1/d} + k)$, where k is the number of reported points.*

Proof. As in the two-dimensional case, the time taken to traverse a subtree and report the points in a subtree is linear in the number of reported points in that subtree, and thus, the total time taken executing REPORT-SUBTREE is $O(k)$, where k is the number of reported points.

We now consider the recursive case, where a node v intersects, but is not fully contained within, the query range. That is, the boundary of R intersects *region*(v). We bound the number of such nodes by examining the number of regions intersected by any hyperplane. Let us consider hyperplanes perpendicular to the x_i-axis for a dimension i. Let $Q(n)$ be the number of intersected regions in a kd-tree storing n points whose root contains a split on dimension i. The base case for $n = 1$ is clearly $O(1)$. For the recursive case, we need to go down a further depth of d before we reach another region that is split along dimension i again. On the both the left and right hand sides of the split, there are 2^{d-1} such regions (and in total, there are 2^d regions on either side before we get to another region with a split in dimension i). If our hyperplane goes through the regions on the left hand side, then this adds 2^{d-1} regions to our count, and then we can go and consider splits in those

2^{d-1} regions recursively; the same applies for the right. Hence we have the following recurrence:

$$Q(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2^{d-1} + 2^{d-1} Q\left(\frac{n}{2^d}\right) & \text{if } n > 1 \end{cases}$$

Using the Master Theorem[3], the recurrence can be solved as

$$Q(n) = O\left(n^{\log_{2^d} 2^{d-1}}\right) = O\left(n^{1-1/d}\right)$$

since

$$\log_{2^d} 2^{d-1} = \frac{\log 2^{d-1}}{\log 2^d} = \frac{d-1}{d}$$

This argument can be applied to all dimensions, and thus to all "sides" of the d-dimensional query range. Because we need to recurse through $O(n^{1-1/d})$ intersecting nodes, and use $O(k)$ time reporting the points in the range, we have a total time of $O(n^{1-1/d} + k)$. □

1.3 Part c)

Theorem 4. *If d is no longer taken to be a constant, then the amount of storage is $O(dn)$, the construction time is $O(n \log n)$ and the query time is $O(d^2 n^{1-1/d} + k)$.*

Proof. For the storage, the change is that the each of the n leaves now requires $O(d)$ storage. We still have $O(n)$ internal nodes of size $O(1)$. In total, we require $O(dn + n) = O(dn)$ storage.

None of the operations in the construction algorithm depend on the number of dimensions, and thus the construction time remains at $O(n \log n)$. This is because, although we split on d dimensions, we only ever consider one dimension for a particular node.

For the query time, the number of intersections by one hyperplane is $O(n^{1-1/d})$. However, there are $O(d)$ faces to a d-dimensional region. Furthermore, in each call of SEARCH-KD-TREE, we need to check whether the left or the right child's region is fully contained within R; this test is proportional to the number of dimensions. This gives us $O(d^2 n^{1-1/d} + k)$. (We assume that we can report nodes by, say, providing a list of pointers, the size of which is independent of the number of dimensions. If we are printing it out to screen, then the reporting cost would change and we would get $O(d^2 n^{1-1/d} + dk)$. □

[3] Section 4.3 of *Introduction to Algorithms* (2nd ed.), by T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein.

2 Question 2: Lower Bound for Computing Voronoi Diagrams

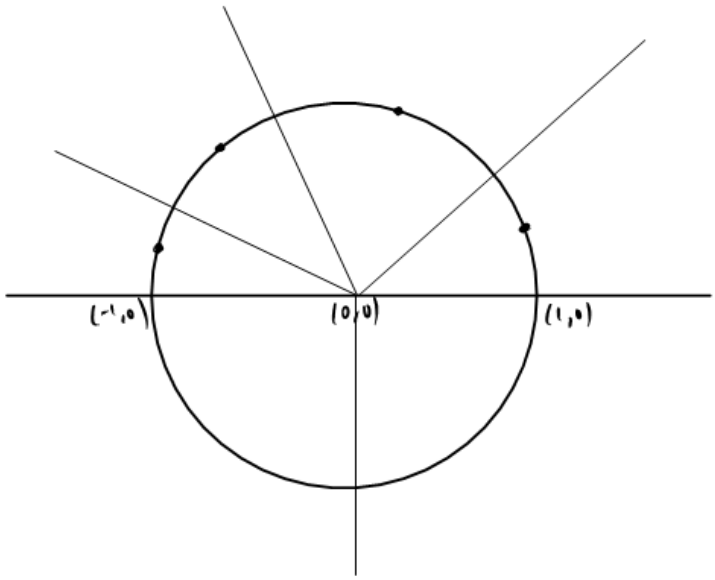
We provide a lower bound for computing Voronoi diagrams by reducing the sorting problem to the problem of computing Voronoi diagrams. The reduction VORONOI-SORT is displayed in Algorithm 3. Without loss of generality, the algorithm sorts a list of real numbers.

Algorithm 3 VORONOI-SORT(X)

Input: A list of real numbers to be sorted $X = \{x_1, \ldots, x_n\}$
Output: A list containing the items from X in sorted order

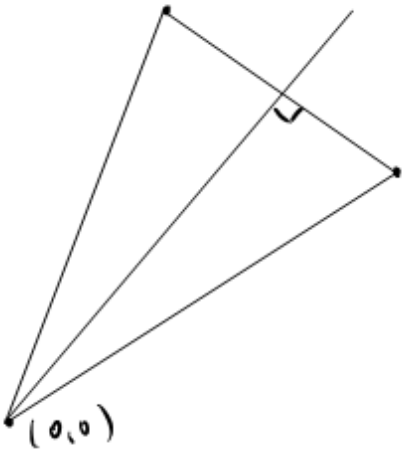
▷ *Map the values to the upper half of the unit circle*
$min \leftarrow$ smallest value in X
$max \leftarrow$ largest value in X
$\Theta \leftarrow \left\{\frac{x-min}{max-min} \cdot \pi : x \in X\right\}$
$P \leftarrow \{(\cos\theta, \sin\theta) : \theta \in \Theta\}$ (if we allow duplicate values, only include a site once, but remember the multiplicity)
▷ *Protect against degenerate case of 2 or less unique sites*
if $|P| \leq 2$ **then**
    Partition the list by brute force, because we know there are 2 or less unique values
    return the sorted list
end if
▷ *Compute the Voronoi diagram for set of sites P*
$V \leftarrow$ VORONOI-DIAGRAM(P)
▷ *Map the Voronoi diagram to a sorted list*
Find the vertex of the V that is at the origin, and locate the edge e incident on the vertex that goes down the negative y-axis by a linear search
$e' \leftarrow e$
$l \leftarrow \emptyset$
repeat
    $p \leftarrow$ the site belonging to the face anticlockwise from e
    $\theta \leftarrow \cos^{-1} p_x$
    $l \leftarrow l \cup \left\{\frac{\theta}{\pi}(max - min) + min\right\}$ (if we allow duplicate values, add this value enough times to cover its multiplicity)
    $e' \leftarrow$ the next edge in anticlockwise order around the origin
until e' = e
return l

The algorithm first transforms the input into sites of a Voronoi diagram, such that the points are evenly spread out on the top half of the unit circle, with the smallest number placed at $(1,0)$ and the largest number placed at $(-1,0)$. The Voronoi diagram V is then computed. We are given that for every vertex of V, we able to find the incident edges in cyclic order. Starting from the cell with $(1,0)$ as the site, we visit the cells of V in anticlockwise order, and transform the sites back into the original values. This causes the original values to come out in sorted order, because the function that maps values to angles is linearly increasing.



Lemma 1. *The Voronoi diagram of a set of sites P where all the sites lie on the unit circle and $|P| \geq 3$ consists of a vertex at the origin, and $|P|$ edges that all go through the origin, one between each pair of sites adjacent on the circle.*

Proof. Consider two sites p_i and p_j that are adjacent to each other on the unit circle. The bisector between p_i and p_j must go through the origin, using an elementary circle geometry theorem.



Only bisectors between adjacent sites are included in the Voronoi diagram, because a bisector between, say p_i and p_k, where p_i and p_k are not adjacent, would lie away from the cell for p_i. This leaves one edge between each pair of sites adjacent on the circle, that is, $|P|$ edges that go through the origin. □

Lemma 2. VORONOI-SORT *sorts the given list of values X in ascending order.*

Proof. The function used to map a value x to its polar angle is

$$\theta(x) = \frac{x - min}{max - min}$$

and it is an increasing function in x. Therefore, if you visit the sites in an anticlockwise fashion, starting from $(1,0)$, they correspond to the values in increasing order.

By Lemma 1, these sites create a Voronoi diagram with a vertex at the origin and one edge between each site. Because the Voronoi diagram allows us to read off values in cyclic order, we have completed the task of sorting the list of values. □

Theorem 5. *Computing a Voronoi diagram has a lower bound of* $\Omega(n \log n)$.

Proof. We examine VORONOI-SORT, and by Lemma 2, this sorts a given list of values of length n. The first part of the algorithm creates a corresponding site for each value in the list, and this requires $O(n)$ time. The last part of the algorithm, where we visit each site once in clockwise order, also requires $O(n)$ time. However, we know that sorting (on the usual model of computation) has a lower bound of $\Omega(n \log n)$, and therefore the intermediate step, namely the VORONOI-DIAGRAM algorithm must take $\Omega(n \log n)$. □

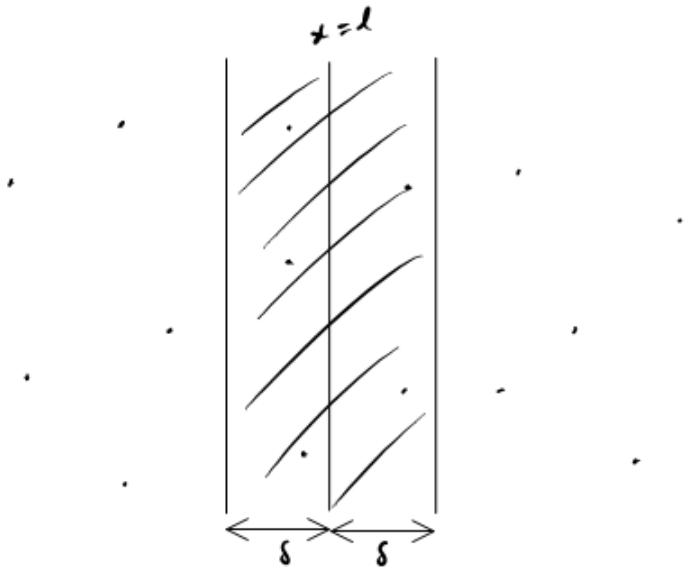
3 Question 3: Closest Pair

3.1 Algorithm

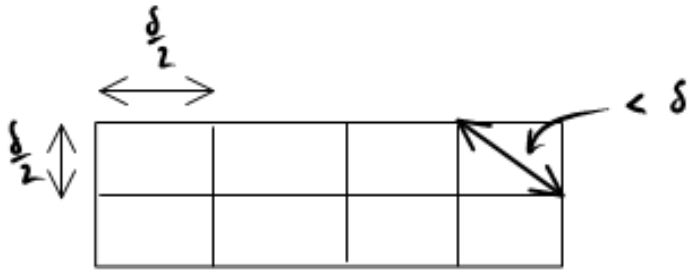
We present a divide-and-conquer algorithm, CLOSEST-PAIR, that finds the points in a given set that are closest together in Algorithm 4. It makes use of CLOSEST-PAIR-ADJACENT in Algorithm 5 to handle the case where all the points share the same y-coordinate.

Theorem 6. CLOSEST-PAIR *returns the closest pair of points in the given set.*

Proof. The algorithm divides the points into those that are left or on the dividing line, and those that are to the right of the dividing line. There are two possibilities for the closest pair: both points are on the same side, or the pair straddles the dividing line. The recursive calls handle the first possibility, and the merge step handles the second possibility. If we have a pair straddling the dividing line, then the x values of both points must be within δ of the dividing line, where δ is the smallest distance found on either left or right.



For a particular point in the strip, if there was a point closer to it than δ distance, the other point must lie within a $(2\delta, \delta)$ window. This window can be broke up into 8 squares, where each square can contain at most 1 point.



Algorithm 4 CLOSEST-PAIR(P, X, Y)

Input: A set of points P, a list of the points in P sorted by x value X, and a list of the points in P sorted by y value Y

Output: A tuple consisting of the two points that are the closest in P, and the distance δ between them

if $|P| \leq 3$ **then**
  return answer by brute force
end if
▷ *Divide the points into two halves*
$l \leftarrow$ the $\lfloor n/2 \rfloor$-th smallest x value
$P_l \leftarrow \{p \in P : p_x \leq l\}$; $P_r \leftarrow \{p \in P : p_x > l\}$
$X_l \leftarrow \{p \in X : p_x \leq l\}$; $X_r \leftarrow \{p \in X : p_x > l\}$ ▷ *Assume that these operations preserve order between elements*
$Y_l \leftarrow \{p \in Y : p_x \leq l\}$; $Y_r \leftarrow \{p \in Y : p_x > l\}$
if $|P_l| = 0$ **then**
  return CLOSEST-PAIR-ADJACENT(Y_r)
else if $|P_r| = 0$ **then**
  return CLOSEST-PAIR-ADJACENT(Y_l)
end if
▷ *Find closest pair in subcases*
$(p_{1l}, p_{2l}, \delta_l) \leftarrow$ CLOSEST-PAIR(P_l, X_l, Y_l)
$(p_{1r}, p_{2r}, \delta_r) \leftarrow$ CLOSEST-PAIR(P_r, X_r, Y_r)
if $\delta_l < \delta_r$ **then**
  $p_1 \leftarrow p_{1l}$
  $p_2 \leftarrow p_{2l}$
  $\delta \leftarrow \delta_l$
else
  $p_1 \leftarrow p_{1r}$
  $p_2 \leftarrow p_{2r}$
  $\delta \leftarrow \delta_r$
end if
▷ *Find points straddling the two halves*
$Y_\delta \leftarrow \{p \in Y : \left|p_x - l\right| \leq \delta\}$
for all $p \in Y_\delta$ **do**
  Find the smallest distance between p and (up to) 7 points preceding it (Y_δ is sorted by y value)
  $\delta' \leftarrow$ the smallest distance found
  $p' \leftarrow$ the preceding point
  if $\delta' < \delta$ **then**
    $p_1 \leftarrow p$
    $p_2 \leftarrow p'$
    $\delta \leftarrow \delta'$
  end if
end for
return (p_1, p_2, δ)

Algorithm 5 CLOSEST-PAIR-ADJACENT(Y)

Input: A list of points all sharing the same y-coordinate Y

Output: A tuple consisting of the two points that are the closest in Y, and the distance δ between them

```
1: p1 ← first point
2: p2 ← second point
3: δ ← dist(p1, p2)
4: for all pairs of points qi and qi+1 adjacent in Y do
5:     if qi and qi+1 are closer than δ then
6:         p1 ← qi
7:         p2 ← qi+1
8:         δ ← dist(p1, p2)
9:     end if
10: end for
11: return (p1, p2, δ)
```

This shows why scanning a fixed number (7) of points back for each point in the vertical strip will always find any pairs closer than δ distance inside the vertical strip.

The algorithm always terminates, because each recursive call deals with a smaller number of points, until it reaches the base case. In the event that all the points lie on the same line, this is handled separately by CLOSEST-PAIR-ADJACENT, which works out the closest pair of points if all the points lie in a vertical line. □

3.2 Time Complexity

The base case with $n \leq 3$ is clearly $O(1)$. For $n > 3$, we either use CLOSEST-PAIR-ADJACENT, which runs in $O(n)$ time, or call the algorithm recursively and use the merge step, which takes $2T(n/2) + O(n)$. Therefore, we have the recurrence

$$T(n) = \begin{cases} O(1) & \text{if } n \leq 3 \\ 2T(n/2) + O(n) & \text{if } n > 3 \end{cases}$$

which solves to $T(n) = O(n \log n)$.

4 Question 4: Star-Shaped Polygons

4.1 Algorithm

We present an algorithm RANDOMIZED-STAR-SHAPED-POLYGON in Algorithm 6 that is an modification of the 2D-RANDOMIZED-BOUNDED-LP algorithm in the textbook.

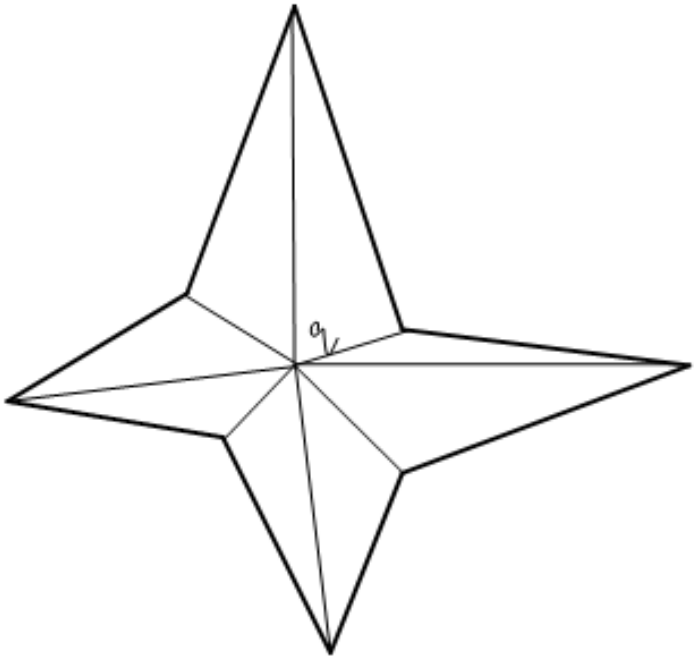
Algorithm 6 RANDOMIZED-STAR-SHAPED-POLYGON(P)

Input: A simple polygon P
Output: Returns true if P is star-shaped, and false otherwise

▷ *Create the linear program*
Create a set of half-planes H by taking each edge of the polygon, extending the segment to infinity and taking the side of the line where the inside of the polygon is
Choose a large M that exceeds any of the coordinates of the polygon, and use this to create the constraints m_1 and m_2
Pick $c_x = 1$ and $c_y = 1$ for the objective function
▷ *Solve the linear program*
Let v_0 be the corner of C_0 calculated from the constraints m_1 and m_2
Compute a random permutation $h_1, \ldots, h_n$ of the half-planes using RANDOM-PERMUTATION($H[1 \cdots n]$)
for $i \leftarrow 1 \text{to} n$ **do**
  if $v_{i-1} \in h_i$ **then**
    $v_i \leftarrow v_{i-1}$
  else
    $v_i \leftarrow$ the point p on l_i (the line that bounds half-plane h_i) that maximises $f_{\vec{c}}(p)$ (the objective function) subject to the constraints in H_{i-1}
    if p does not exist **then**
      return false
    end if
  end if
end for
return true

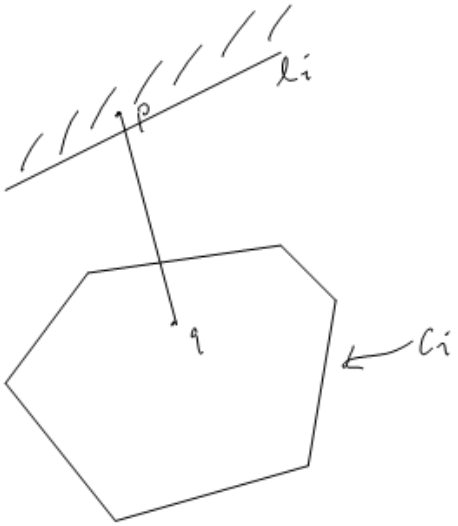
Lemma 3. *The half-planes created from the edges of a simple polygon have a non-empty intersection iff the polygon is star-shaped.*

Proof. Let us assume first that we have a non-empty intersection of half-planes. For any point q in the non-empty intersection, it is possible to draw a line segment pq to any point p on any of the edges of the polygon without it intersecting any edges, by the convexity of the half-planes. This suggests that we are able to triangulate the polygon by drawing a line from q to each vertex of the polygon.



Because it is possible to draw a line from any point in a triangle to any other point in the same triangle, it is therefore possible to draw line segments pq from q to any point p in the entire polygon, and the polygon is star-shaped.

Let us now assume that we have an empty intersection of half-planes. Let C_{i-1} be the last non-empty intersection before we added the half-plane h_i that made C_i empty. In C_{i-1}, there exists a point q such that a line segment pq can be drawn to any point p in the polygon consisting of edges chosen thus far. The fact that h_i made the intersection empty suggests that the "inside" of the polygon for that edge faces away from C_{i-1}, and is therefore impossible to take any q in C_{i-1} and draw a line to a p just inside of the edge, thus making the polygon non-star-shaped.



□

4.2 Time Complexity

Theorem 7. RANDOMIZED-STAR-SHAPED-POLYGON *runs in expected* $O(n)$ *time.*

Proof. Creating the half-planes from the edges of the polygon takes $O(n)$ time. Creating the randomized permutation of the half-planes takes $O(n)$ time. The solution to the linear program takes expected $O(n)$ time, as outlined below.

Let X_i be a random variable, such that

$$X_i = \begin{cases} 1 & \text{if } v_{i-1} \notin h_i \\ 0 & \text{otherwise} \end{cases}$$

We solve a 1-dimensional linear program on line 11, and a linear program on i constraints can be solved in $O(i)$ time. The total time spent computing 1-dimensional linear programs is

$$\sum_{i=1}^{n} O(i) \cdot X_i$$

The probability that the optimal vertex changes when we add h_i is the same as the probability that the optimal vertex changes when we remove a half-plane. At most, the probability of this occurring is $2/i$ because there are i half-planes at that point in time, and we could pick one of the two of the half-planes that affect the optimal vertex. Thus, $E[X_i] = 2/i$. The expected running time for solving the sequence of 1-dimensional linear equations is therefore

$$E\left[\sum_{i=1}^{n} O(i) \cdot X_i\right] = \sum_{i=1}^{n} O(i) \cdot E[X_i] = \sum_{i=1}^{n} O(i) \cdot \frac{2}{i} = O(n)$$

Because the rest of the algorithm takes $O(n)$ time, the expected running time of the entire algorithm is $O(n)$. □

5 Question 5: Assigning Subdivisions to Points

5.1 Algorithm

We present a sweep-line algorithm ASSIGN-SUBDIVISION in Algorithm 7 that will, given a subdivision S and a set of points P, assign the points P to the subdivision in S in which it is contained.

Algorithm 7 ASSIGN-SUBDIVISION(S, P)

Input: A subdivision S with n vertices, and a set P of m points
Output: An assignment of each point in P to the subdivision in S in which it is contained

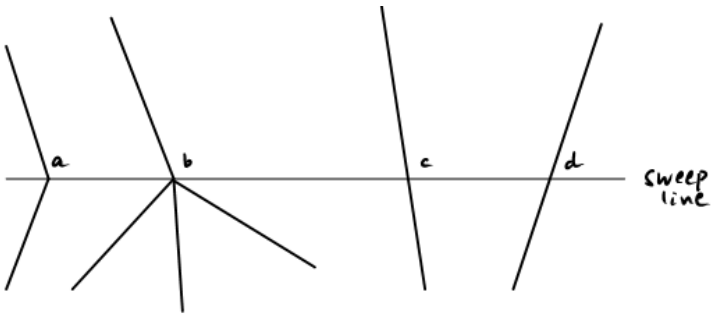
```
▷ Create a bounding box
Create a bounding rectangle that surrounds the vertices in S and the points in P
Turn all half-infinite lines into line segments using the bounding box
▷ Create the events
E ← ∅
for all edges e ∈ S do
    if e is not horizontal then
        Add a "begin line" event to E for the top-most point of e
        Add an "end line" event to E for the bottom-most point of e
    end if
end for
for all points m ∈ P do
    Add a "point" event to E for m
end for
Sort the events E by decreasing y-coordinate
▷ Process the events
Create a new sweep-line l that stores the edges it currently intersects with from left to right
while there are events in E do
    Pop the top event off
    if the event is a "begin line" event then
        ▷ Insert the line into the sweep-line data structure
        Perform a binary search on the x-coordinate of the event in the sweep-line data structure
        and insert the associated line segment into the appropriate location (explained later)
    else if the event is an "end line" event then
        ▷ Remove the line into the sweep-line data structure
        Perform a binary search on the x-coordinate of the event in the sweep-line data structure
        and remove the associated line segment (explained later)
    else
        ▷ Find which subdivision a point belongs in
        Perform a binary search on the x-coordinate of the event in the sweep-line data structure
        to find the two line segments between which the point lines (explained later)
        Associate the point with the subdivision demarcated by the two line segments on the left
        and right
    end if
end while
return assignments of subdivisions to points
```

The event queue contains events for the beginning and end of a line segment, and points for

which we need to calculate a subdivision. The events are processed from top to bottom.

The sweep-line stores information about the line segments with which it currently intersects. Because it is not possible for line segments to cross without causing an event, the order of line segments in the sweep-line data structure only changes at "begin line" and "end line" events. We assume that line segments carry information about their start and end points, are able to calculate the x-coordinate of a point on the line if queried with the y-coordinate.

When inserting a new line segment, the algorithm states that we perform a binary search on the line segments already in the sweep-line to work out where we should insert the new line. The binary search is performed using the x-coordinate of the event as the search key, and this search key is compared against the x-coordinate of the lines already in the data structure given the y-coordinate of the insertion event. As mentioned above, two line segments cannot cross without causing an event, and thus the x-coordinate values are guaranteed to be non-decreasing if you queried the lines from left to right. If two lines share the same start point, then they should be ordered such that if the sweep line moves a small $\epsilon > 0$ downwards, the lines are ordered from left to right.



Deletion is similar. We perform a binary search, where the search key is the x-coordinate of the event, and the search key is compared against the x-coordinate of the lines in the data structure given the y-coordinate of the deletion event.

Assigning subdivisions to points is also similar. A binary search is performed with the x-coordinate of the event as the search key, and the search key is compared against the x-coordinate of the lines in the data structure given the y-coordinate of the point event. The binary search is able to reveal the two segments to the left and right of the point (or just the one segment if the point lies to the left of the left-most line segment, or if the point lies to the right of the right-most line segment). The subdivision S will then be able to inform us about the face of S that lies between the two line segments.

5.2 Time Complexity

Theorem 8. ASSIGN-SUBDIVISION *runs in* $O((n+m)\log(n+m))$ *time, where* n *is the number of vertices of* S *and* m *is the number of points in* P.

Proof. We perform the following operations in the algorithm:

- Create a bounding box: $O(n+m)$
- Create the events: $O(n+m)$
- Sort the events: $O((n+m)\log(n+m))$

- Process the events, all of which involve a binary search on the line segments in the sweep-line: $O((n+m)\log n)$

In total, this gives us $O((n+m)\log(n+m))$ time. □

6 Question 6: Line Segment Visibility

6.1 Algorithm

We present an algorithm VISIBLE-LINE-SEGMENTS in Algorithm 8 that will, given a point p and a set of n disjoint line segments S, determine the line segments of S that p is able to see.

Algorithm 8 VISIBLE-LINE-SEGMENTS(S, p)

Input: A set of n disjoint line segments S, and a point p not on any of the line segments of S
Output: All line segments of S that p can see

▷ *Create the events*
$E \leftarrow \emptyset$
for all segments $s \in S$ **do**
    Store an event corresponding to each of the two endpoints of s
end for
Sort the events E according to polar angle $\theta \in [0, 2\pi)$ relative to p
▷ *Create the (radial) sweep-line and then populate it*
Create a new sweep-line, which is a half-line with its endpoint at p and stores the line segments that it intersects in order of closest to furthest
for all segments $s \in S$ **do**
    if s intersects positive x-axis (which is the starting position of the sweep-line) **then**
        Insert s into the sweep-line data structure, maintaining order by finding the correct position using binary search
        Mark s as currently being on the sweep-line
    end if
end for
▷ *Process the events*
$v \leftarrow \emptyset$ ▷ *the visible line segments*
Insert the closest line segment currently on the sweep-line into v, if there is one
while there are events in E **do**
    Pop the top event off E
    $s \leftarrow$ the corresponding line segment in S for the event
    if s is currently on the sweep-line **then**
        ▷ *Remove it from the sweep-line*
        Perform a binary search (explained later) for s in the sweep-line data structure and remove it
        if we removed the closest line segment along the sweep-line ray **then**
            $s' \leftarrow$ the new closest line segment, if there is one
            if $s' \notin v$ **then**
                $v \leftarrow v \cup \{s'\}$
            end if
        end if
    else
        ▷ *Add it to the sweep-line*
        Find the correct place to put s by performing a binary search (explained later) for s in the sweep-line data structure and insert it
        if s has become the new closest line segment along the sweep-line ray **then**
            $s' \leftarrow$ the new closest line segment, if there is one
            if $s' \notin v$ **then**
                $v \leftarrow v \cup \{s'\}$
            end if
        end if
    end if
end while
return v

The algorithm creates an event for each of the two endpoints for a line segment. We start off by initialising the half-line by placing it on the positive x-axis, and inserting the lines that intersect it into the sweep-line data structure, which maintains the line segments that the sweep-line intersects in order from closest to furthest.

Similarly to the previous question, we assume that the line segments store information about their endpoints, and are able to calculate the Euclidean distance from p given the polar angle/angle with respect to the positive x-axis. Furthermore, we are able to use the property once again that line segments cannot change position within the sweep-line data structure. If s_1 is closer than s_2 to p, then it is not possible for s_2 to be closer than s_1 to p at some future time, because that would imply that the two line segments intersect, which is not true.

When we insert a line segment into the sweep-line, we perform a binary search for the position along the sweep-line that we need to insert into; the search key is the distance of the event point from p, and the search key is compared with the distances of points along line segments currently in the sweep-line data structure when given the current angle of the sweep-line. Note that this distance is only computed when requested in the binary search, otherwise we would get a linear time complexity per event. Again, deletion is similar, for we need to search for the line segment in the data structure given the current event point.

For the set v of line segments that are visible from p, this can be implemented as a heap, if we give the line segments an id number that can be used for ordering.

6.2 Time Complexity

Theorem 9. VISIBLE-LINE-SEGMENTS *runs in* $O(n \log n)$ *time, where* n *is the number of line segments in* S.

Proof. We perform the following operations in the algorithm:

- Creating the events: $O(n)$
- Sorting the events by angle: $O(n \log n)$
- Creating and populating the radial sweep-line: $O(n \log n)$ (we potentially have n insertions involving n binary searches)
- Processing the events: $O(n \log n)$ (we have $O(n)$ events, and each event involves binary searches for the correct position along the sweep-line data structure, and searching the heap v to avoid duplicates)

In total, this gives $O(n \log n)$ time. □