COMP4045 Computational Geometry Assignment 1

Enoch Lau
SID 200415765

19 April 2007

Question 1

The following is a rectilinear polygon with $n = 4$ sides that requires $\lfloor n/4 \rfloor = 1$ guard (the x's represent guards).

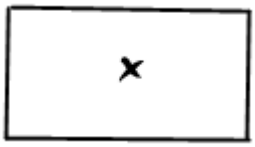

Figure 1: A rectilinear polygon with $n = 4$ sides

The following is a rectilinear polygon with $n = 8$ sides that requires $\lfloor n/4 \rfloor = 2$ guards.

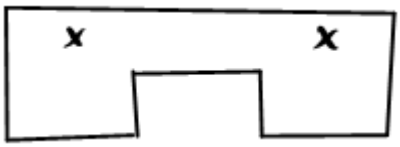

Figure 2: A rectilinear polygon with $n = 8$ sides

Generalising, we obtain the following rectilinear polygon with n sides that requires $\lfloor n/4 \rfloor$ guards.

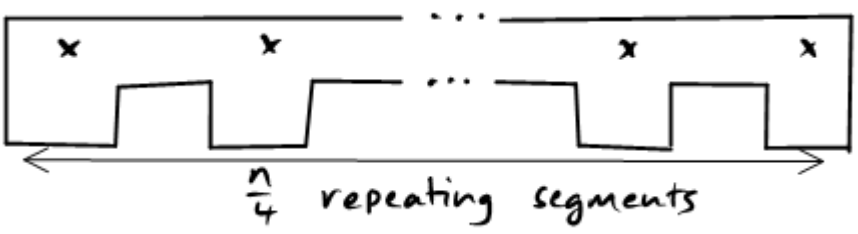

Figure 3: A rectilinear polygon with n sides and $n/4$ repeating segments

Question 2

Theorem 1. *Two x-monotone polygonal chains P and Q intersect $O(n)$ times, where n is the total number of vertices of P and Q.*

Proof. Suppose that P and Q have p and q vertices respectively. The number of edges in P and Q are therefore $p-1$ and $q-1$ respectively (assuming $p, q \geq 1$), and we define the edges in P and Q to be $\{l_{P_1}, l_{P_2}, \ldots, l_{P_{p-1}}\}$ and $\{l_{Q_1}, l_{Q_2}, \ldots, l_{Q_{q-1}}\}$. Define $I(l)$ to be the number of edges that l intersects.

Without loss of generality (this argument can proceed with P and Q swapped around), we take an edge in P, say l_{P_i}, that intersects $I(l_{P_i}) = m \geq 1$ edges in Q, say $l_{Q_{j_1}}, \ldots, l_{Q_{j_m}}$. Of these m edges, at least $m-1$ of these edges will not be able to be intersected by a further edge of P due to x-monotonicity; only the last edge, $l_{Q_{j_m}}$, can be intersected by $l_{P_{i+1}}$ for example. If we sum the number of edges in Q that are unable to be intersected by any further edges of P for each edge in P (taken in order, say, from left to right), this is at most the number of edges in Q, which is $q-1$ (after the $q-1$ edges of Q have been made such that they are unable to be intersected, the edges of P can no longer intersect with anything). Hence:

$$\sum_{i=1}^{p-1} J(l_{P_i}) \leq q-1$$

where, in order to account for those edges in P that do not intersect with any edges in Q, we define:

$$J(l_{P_i}) = \begin{cases} 0 & \text{if } I(l_{P_i}) = 0 \\ I(l_{P_i}) - 1 & \text{otherwise} \end{cases}$$

Hence:

$$\begin{aligned} \sum_{i=1}^{p-1} \left(I(l_{P_i}) - 1\right) &\leq q-1 \\ \sum_{i=1}^{p-1} I(l_{P_i}) &\leq (p-1)+(q-1) \\ \sum_{i=1}^{p-1} I(l_{P_i}) &= O(n) \end{aligned}$$

Noting that the number of times that edges in P intersect edges in Q is the same as the number of times that edges in Q intersect with edges in P, we have shown that the number of times that P and Q intersect is $O(n)$. □

Question 3

a)

Convex hull merge algorithm

Algorithm 1 demonstrates CONVEX-HULL-MERGE, a $O(n)$ algorithm to merge two convex hulls[1] P_1 and P_2, assuming that P_1 and P_2 can be separated by a vertical line. Without loss of generality, we assume that P_1 is to the left of P_2 (we can swap them around otherwise). P_1 and P_2 are presumed to be represented as lists of points that, regardless of whether the points are stored in clockwise or counterclockwise order, can be traversed in both directions in constant time (if they can only be traversed in one direction efficiently, it is trivial to create a corresponding doubly-linked list in linear time). The return value of the algorithm is list in clockwise order.

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

Algorithm 1 CONVEX-HULL-MERGE(P_1, P_2)

▷ *Find the right-most point of* P_1
$u \leftarrow$ POLYGON-RIGHT-MOST-POINT(P_1)
▷ *Find the left-most point of* P_2
$v \leftarrow$ POLYGON-LEFT-MOST-POINT(P_2)
▷ *Find the line that joins the upper hulls of* P_1 *and* P_2
$\overrightarrow{u_u v_u} \leftarrow$ CONVEX-HULL-MERGE-TOP-LINE(P_1, P_2, u, v)
▷ *Find the line that joins the lower hulls of* P_1 *and* P_2
$\overrightarrow{u_l v_l} \leftarrow$ CONVEX-HULL-MERGE-BOTTOM-LINE(P_1, P_2, u, v)
▷ *Construct the complete convex hull*
$H \leftarrow \emptyset$
$p \leftarrow u_u$
repeat
    $H \leftarrow H \cup \{p\}$
    if $p = u_u$ **then**
        $p \leftarrow v_u$
    else if $p = v_l$ **then**
        $p \leftarrow u_l$
    else if p is on P_1 **then**
        $p \leftarrow$ the point on P_1 in the clockwise direction from p
    else {p is on P_2}
        $p \leftarrow$ the point on P_2 in the clockwise direction from p
    end if
until $p = u_u$
return H

[1] A convex polygon is the convex hull of its own vertices.

Algorithm 2 POLYGON-RIGHT-MOST-POINT(P)

$r \leftarrow p_1$
for $i = 2$ to $|P|$ **do**
  if $x[p_i] > x[r]$ **then**
    $r \leftarrow p_i$
  end if
end for
return r

Algorithm 3 POLYGON-LEFT-MOST-POINT(P)

$l \leftarrow p_1$
for $i = 2$ to $|P|$ **do**
  if $x[p_i] < x[l]$ **then**
    $l \leftarrow p_i$
  end if
end for
return l

Algorithm 4 CONVEX-HULL-MERGE-TOP-LINE(P_1, P_2, u, v)

line_changed $\leftarrow$ **true**
$u_u \leftarrow u$
$v_u \leftarrow v$
while *line_changed* **do**
  line_changed $\leftarrow$ **false**
  ▷ *Try and move u_u up on the left polygon*
  $u'_u \leftarrow$ *the vertex in P_1 anticlockwise from u_u*
  if *gradient* $\left[\overrightarrow{u'_u v_u}\right] <$ *gradient* $\left[\overrightarrow{u_u v_u}\right]$ **then**
    $u_u \leftarrow u'_u$
    line_changed $\leftarrow$ **true**
  end if
  ▷ *Try and move v_u up on the right polygon*
  $v'_u \leftarrow$ *the vertex in P_2 clockwise from v_u*
  if *gradient* $\left[\overrightarrow{u_u v'_u}\right] >$ *gradient* $\left[\overrightarrow{u_u v_u}\right]$ **then**
    $v_u \leftarrow v'_u$
    line_changed $\leftarrow$ **true**
  end if
end while
return $\overrightarrow{u_u v_u}$

Algorithm 5 CONVEX-HULL-MERGE-BOTTOM-LINE(P_1, P_2, u, v)

```
line_changed ← true
u_l ← u
v_l ← v
while line_changed do
    line_changed ← false
    ▷ Try and move u_l down on the left polygon
    u'_l ← the vertex in P_1 clockwise from u_l
    if gradient[u'_l v_l] > gradient[u_l v_l] then
        u_l ← u'_l
        line_changed ← true
    end if
    ▷ Try and move v_l down on the right polygon
    v'_l ← the vertex in P_2 anticlockwise from v_l
    if gradient[u_l v'_l] < gradient[u_l v_l] then
        v_l ← v'_l
        line_changed ← true
    end if
end while
return u_l v_l
```

Correctness of algorithm

Theorem 2. CONVEX-HULL-MERGE *computes the convex hull of* $P_1 \cup P_2$ *(with the assumptions as described above).*

Proof. We begin by finding the right-most vertex in P_1 (if there are multiple choices, then any one will do), and the left-most vertex in P_2 (again, if there are multiple choices, then any one will do). This ensures that an edge drawn between these two points does not intersect with any existing edge. We call this line $\overrightarrow{uv}$.

Let us examine what occurs for the line joining the tops of the two polygons. This is computed by CONVEX-HULL-MERGE-TOP-LINE in Algorithm 4. We start with $\overrightarrow{uv}$ and progressively shuffle the two endpoints upwards. This results in a convex upper part of the hull because:

- If the gradient of the line can be lowered by shifting the point u_u to u'_u, this means that from $\overrightarrow{u'_u u_u}$ to $\overrightarrow{u_u v_u}$, we take a left turn. Left turns are not permitted on upper hulls.

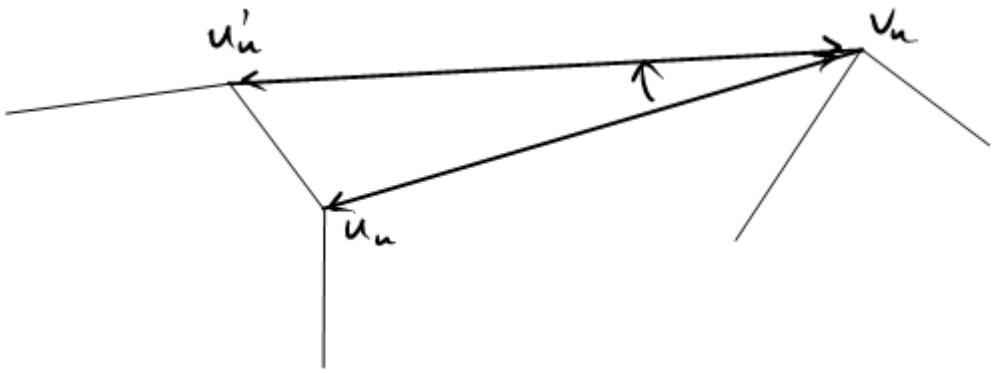

Figure 4: Shifting u_u to u'_u

- If the gradient of the line can be increased by shifting the point v_u to v'_u, this means that from $\overrightarrow{u_u v_u}$ to $\overrightarrow{v_u v'_u}$, we take a left turn. Left turns are not permitted on upper hulls.

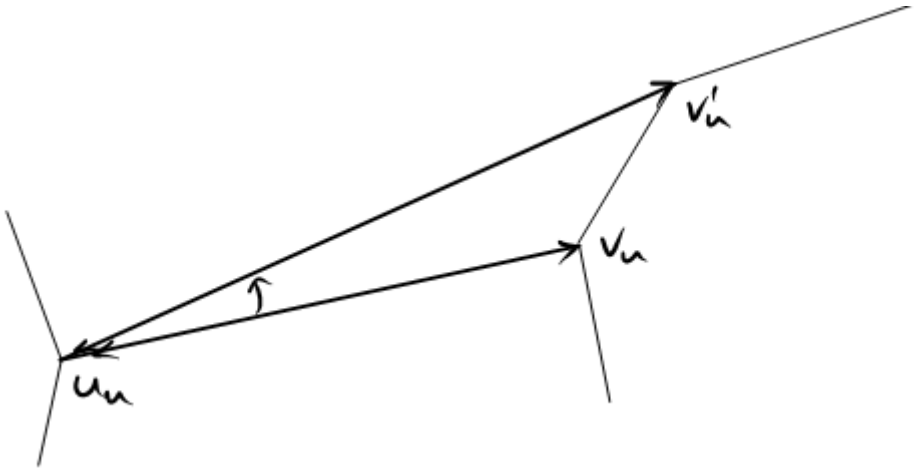

Figure 5: Shifting v_u to v'_u

When we can no longer find a left turn, the upper section of the hull now consists entirely of right turns, and is thus constructed as required.

Similarly, for the lower half of the hull, we start with $\overrightarrow{uv}$ and progressively shuffle the two endpoints downwards. This is computed by CONVEX-HULL-MERGE-BOTTOM-LINE in Algorithm 5. This results in a convex lower part of the hull because:

- If the gradient of the line can be increased by shifting the point u_l to u'_l, this means that from $\overrightarrow{u'_l u_l}$ to $\overrightarrow{u_l v_l}$, we take a right turn. Right turns are not permitted on lower hulls.

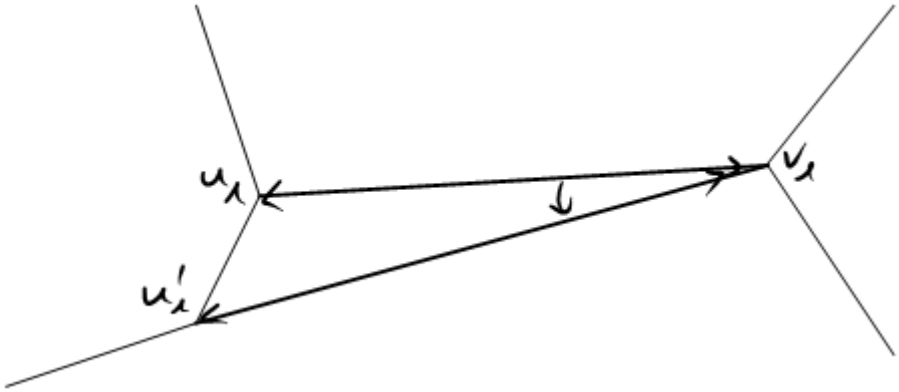

Figure 6: Shifting u_l to u'_l

- If the gradient of the line can be lowered by shifting the point v_l to v'_l, this means that from $\overrightarrow{u_l v_l}$ to $\overrightarrow{v_l v'_l}$, we take a rigth turn. Right turns are not permitted on upper hulls.

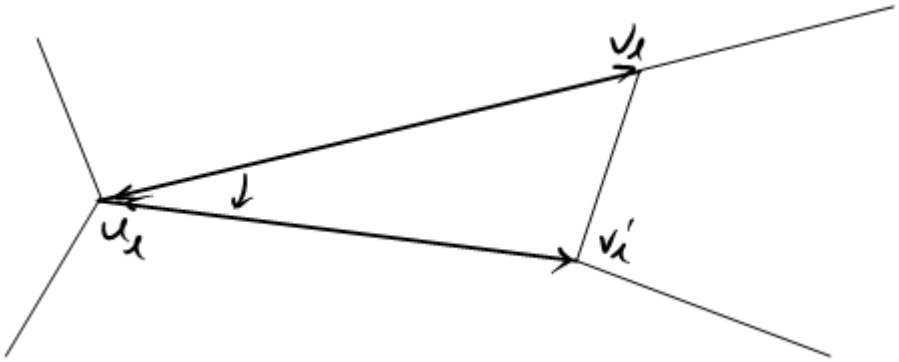

Figure 7: Shifting v_l to v'_l

For the other points not visited by either CONVEX-HULL-MERGE-TOP-LINE or CONVEX-HULL-MERGE-BOTTOM-LINE, they can be included in the final convex hull, because the original polygons are convex. The points that are excluded from the final convex hull are those that CONVEX-HULL-MERGE-TOP-LINE and CONVEX-HULL-MERGE-BOTTOM-LINE determined to cause a left turn on the upper hull and a right turn on the lower hull. The final stage is to select the points on the convex hull by ignoring those bad points; we start at a point known to be on the convex hull, u_u, and follow $\overrightarrow{u_u v_u}$ and $\overrightarrow{u_l v_l}$ where possible, and the polygons' edges otherwise. The result is the convex hull of $P_1 \cup P_2$. □

Asymptotic time complexity

Theorem 3. CONVEX-HULL-MERGE *runs in $O(n)$ time, where n is the total number of points in $P_1 \cup P_2$.*

Proof. POLYGON-RIGHT-MOST-POINT and POLYGON-LEFT-MOST-POINT both run in $O(n)$ time, because they perform linear searches. CONVEX-HULL-MERGE-TOP-LINE and CONVEX-HULL-MERGE-BOTTOM-LINE both run in $O(n)$ time, because no point in the two polygons is ever examined more than once (it is not possible for u_u, v_u, u_l and v_l to cycle in values). The final construction step again examines each point in the two polygons at most once (the worst case is where all the points in P_1 and P_2 are returned in some order), and this runs in $O(n)$ time. In total, this gives us $O(n)$ time. □

b)

Divide-and-conquer convex hull algorithm

Algorithm 6 demonstrates DIVIDE-AND-CONQUER-CONVEX-HULL, a $O(n \log n)$ algorithm that finds the convex hull of a set of points, $P = \{p_1, p_2, \ldots, p_n\}$. It makes use of CONVEX-HULL-MERGE from Algorithm 1, and returns the hull points in clockwise order.

Algorithm 6 DIVIDE-AND-CONQUER-CONVEX-HULL(P)

if $|P| \leq 2$ **then**
  ▷ *P is the convex hull trivially*
  return P
else if $|P| = 3$ **then**
  ▷ *For consistency, ensure points are clockwise*
  if you make a right turn from $\overrightarrow{p_1 p_2}$ to $\overrightarrow{p_2 p_3}$ **then**
    return P
  else
    Swap p_2 and p_3
    return P
  end if
else
  ▷ *Divide the points into two (roughly) equal halves*
  $x_m \leftarrow$ median x value
  $P_l \leftarrow \emptyset$
  $P_r \leftarrow \emptyset$
  for all points p in P **do**
    if $x[p] \leq x_m$ **then**
      $P_l \leftarrow P_l \cup \{p\}$
    else
      $P_r \leftarrow P_r \cup \{p\}$
    end if
  end for
  ▷ *Guard against having all the points in a vertical line*
  if $P_l = \emptyset$ **then**
    return P_r
  else if $P_r = \emptyset$ **then**
    return P_l
  end if
  ▷ *Find the convex hull of each half and merge*
  $H_l \leftarrow$ DIVIDE-AND-CONQUER-CONVEX-HULL(P_l)
  $H_r \leftarrow$ DIVIDE-AND-CONQUER-CONVEX-HULL(P_r)
  return CONVEX-HULL-MERGE(H_l, H_r)
end if

Correctness of algorithm

Theorem 4. DIVIDE-AND-CONQUER-CONVEX-HULL *computes the convex hull of P.*

Proof. For the base case, the algorithm trivially computes the convex hull of P.

For the recursive case, the algorithm creates convex hulls for the left and right halves recursively, and assuming the correctness of CONVEX-HULL-MERGE, which will create the convex hulls for P_1 and P_2 separately, the return value is the convex hull for $P_1 \cup P_2 = P$. The algorithm always terminates, because subcases are guaranteed to be smaller due to the explicit check of whether either the left or right half is the empty set. □

Asymptotic time complexity

Theorem 5. DIVIDE-AND-CONQUER-CONVEX-HULL *runs in* $O(n \log n)$ *time, where* n *is the number of vertices in* P.

Proof. The base case trivially consists of constant time statements; hence, $T(n) = O(1)$ for $n \leq 3$.

For the recursive case, dividing the points into two halves takes $O(n)$ time: there exists a linear time algorithm for finding the median of set of values[2] on line 14, and the for loop starting on line 17 (with $O(1)$ statements in its body) runs over n elements. The infinite loop check takes constant time. The convex hull of the left and right halves can each be computed in $T(n/2)$ time. The merge step takes $O(|H_l| + |H_r|) = O(n)$ time. In total, this gives $T(n) = O(n) + 2T(n/2)$, and hence $T(n) = O(n \log n)$. □

[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.

Question 4

Lemma 1. *For all vertices on the convex hull of any polygonal hole inside a polygon (which may possess other holes), it is possible to draw a straight line to a vertex on another hole or the bounding polygon that does not intersect any edges.*

Proof. Suppose we have a point p on the convex hull of some hole in the polygon. Because it is on the convex hull of the hole, as long as the line we construct goes out of the hole and not into the hole, it will never intersect with an edge of the hole. In addition, there are always some vertices that we can connect to in the reflex sector defined by the two edges adjacent to p; at the very least, these come from the bounding polygon, because the bounding polygon must be such that it contains the hole. This region is illustrated below:

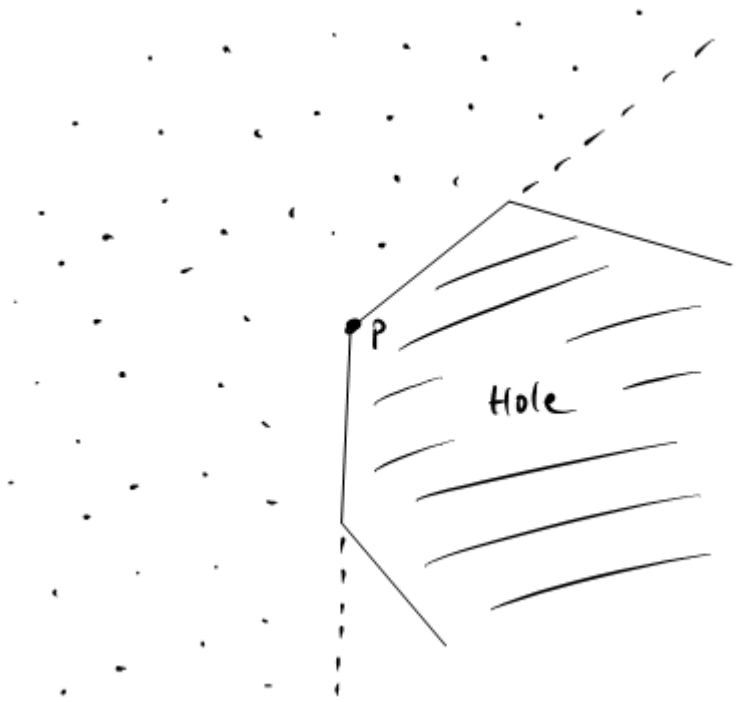

Figure 8: In the reflex sector (dotted region) defined by the two edges adjacent to p, there must be some vertices from the bounding polygon

We then proceed by way of contradiction. Suppose that we are unable to draw a straight line connecting p to a vertex on another hole or on the bounding polygon. If this is so, it must be because there is at least one edge between every other such point and p. Let one of these edges be l_1. However, is p able to see the endpoints of l_1? If not, then there must be at least one edge between the endpoints of l_1 and p. Let one of these edges be l_2. Is p able to see the endpoints of l_2? This argument cannot continue forever, because P has a finite number of edges. Hence, eventually, we will be able to draw a straight line from p to such a vertex on the bounding polygon or on another hole. □

Theorem 6. *Any polygon P with n vertices admits a triangulation, even if it has h holes. The number of triangles in the triangulation is $n+2h-2$.*

Proof. We proceed by induction. We first consider the case where $h = 0$. From the lectures, it was shown that there exists a triangulation of the simple polygon, and the number of triangles in the triangulation is $n-2$.

Now, assume that there exists a triangulation for a polygon of n vertices with h holes, and that the number of triangles in the triangulation is $n+2h-2$. We seek to show that there exists a triangulation for a polygon of n vertices with $h+1$ holes, and that the number of triangles in the triangulation is $n+2(h+1)-2$.

From Lemma 1, we can take any of the polygonal holes, find its convex hull (which contains at least 3 points), and take any point p, and have a straight line drawn from p to some point q that lies on another polygonal hole or on the bounding polygon. We then perform a change to the polygon as illustrated below:

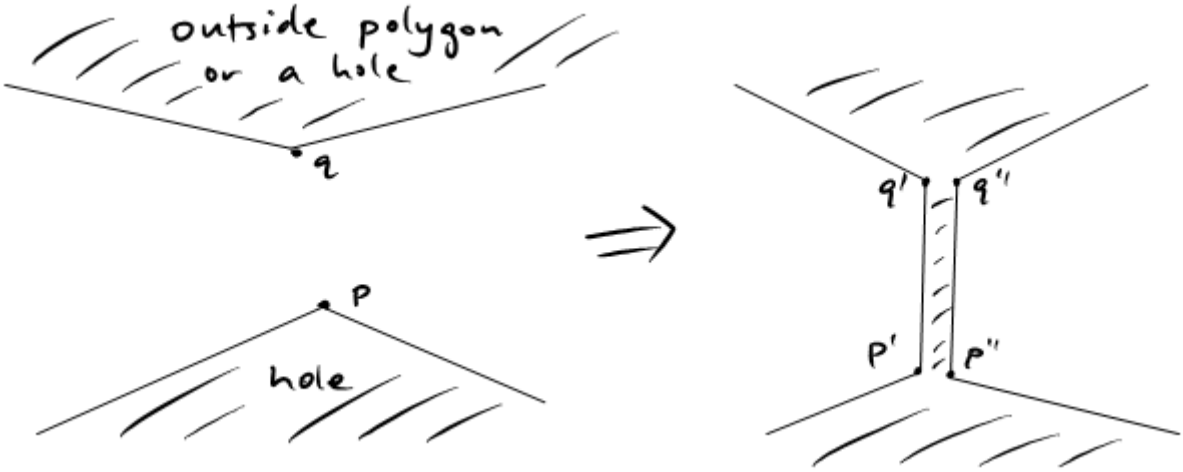

Figure 9: Splitting p and q and joining up the resulting vertices

The change as illustrated is performed as follows. We split p into p' and p'', and q into q' and q'', where p' and p'', and q' and q'' are separated very slightly. We then create edges between p' and q', and between p'' and q'' (if the edges cross, relabel the points). If q lies on another polygonal hole, the two holes have been merged together, and overall, we have one fewer hole, but two more edges. If q lies on the bounding polygon, we have one fewer hole, but two additional edges.

For both of these cases, we triangulate the resultant polygon with h holes. This is possible by assumption. The number of triangles in the triangulation is $(n+2)+2h-2$, also by assumption. We identify p' and p'', and q' and q'', together, hence fusing the two edges $p'q'$ and $p''q''$ together. Taking the newly combined edge as a line in the triangulation, the polygon with $h+1$ holes is now triangulated. Additionally, the number of triangles in the triangulation is unaffected by the fuse of the two edges, and thus it is $n+2(h+1)-2$.

Hence, because the theorem is true for $h = 0$, it is true for $h = 1$, $h = 2$, and so on, $\forall\, h \in \mathbb{Z}^+ \cup \{0\}$. □

Question 5

a)

1-dimensional range count query algorithm

We will take the 1-dimensional range tree as presented in the lectures and modify it so that each node stores the number of leaves underneath it. The range counting query is then performed as in Algorithm 7.

Algorithm 7 1-dimensional range counting query

Find the first node greater than or equal to the beginning of the range, μ_a.
Find the first node less than or equal to the end of the range, μ_b.
▷ *Add up values on right subtrees on the way up from* μ_a
$count \leftarrow 0$
$v = \mu_a$
while $v \neq$ split node **do**
  if v is the left child of its parent and the parent is not the split node **then**
    $count \leftarrow count + number\_of\_leaves$[right child of parent]
  end if
  $v \leftarrow parent[v]$
end while
▷ *Add the values on left subtrees on the way up from* μ_b
$v = \mu_b$
while $v \neq$ split node **do**
  if v is the right child of its parent and the parent is not the split node **then**
    $count \leftarrow count + number\_of\_leaves$[left child of parent]
  end if
  $v \leftarrow parent[v]$
end while
return *count*

Asymptotic time complexity

Theorem 7. *The 1-dimensional range count query as presented above runs in* $O(\log n)$ *time.*

Proof. As with the original range query, finding μ_a and μ_b both take $O(\log n)$ time. The counting process on the way up from μ_a to the split node runs in $O(\log n)$ time as well, because the height of the tree is $O(\log n)$. We avoid the k constant penalty of returning the actual values, because we store the number of leaves under each node at the node during construction, and this can be accessed in constant time. Similarly, the counting process on the way up from μ_b to the split node also runs in $O(\log n)$ time. The total running time is hence $O(\log n)$. □

b)

Theorem 8. *The d-dimensional range count query can be performed in* $O(\log^d n)$ *time.*

Proof. We will take the d-dimensional range tree, using auxiliary data structures, as presented in the lectures, and modify it so that the bottom-most tree (the 1-dimensional case) uses the construction and algorithm described above. The 2-dimensional range query will hence run in $O(\log^2 n)$ time, as the $O(\log n)$ nodes from μ_a and μ_b incur a $O(\log n)$ cost each for searching the 1-dimensional auxiliary data structure. Similarly, a d-dimensional range query will incur a $O(\log^d n)$ cost due to the $O(\log n)$ nodes from μ_a and μ_b each incurring a $O(\log^{d-1} n)$ cost. □

Question 6

a)

Theorem 9. *A range query of the form* $[a,a] \times [b,b]$ *on a* kd*-tree results in a search time of* $O(\log n)$.

Proof. We assume that the kd-tree has been constructed so that we alternate between x and y split lines, and the split lines are chosen using the median (so that the two halves created are (roughly) the same size). If we have a query window that is a point $[a,a] \times [b,b]$ instead of a region with non-zero area, at each split line, the query window must lie on one side or the other (the split line is either shifted slightly from the median position so that it does not intersect any points, or the tree is constructed so that points falling on the split line get assigned to one particular half consistently). It takes a constant amount of time to decide which half the point (a,b) belongs to. Hence, the running time at each level is $T(n) = O(1) + T(n/2)$, which means that $T(n) = O(\log n)$. □

b)

Theorem 10. *A range query of the form* $[a,a] \times [b,b]$ *on a 2-dimensional range tree results in a search time of* $O(\log n)$.

Proof. In the lectures, a range tree that can be queried in $O(\log^{d-1} n + k)$ (where k is the number of points to return) was demonstrated; this involves the construction of pointers from the auxiliary data structures in parent nodes to corresponding places in the auxiliary data structures in the children nodes. This allows for $O(\log n)$ operations to find the start and end of the y range, and the number of pointers that will be traversed is $O(\log n)$; these operations allow us to avoid the $O(\log n)$ cost at each node on the way up from μ_a to μ_b. In particular, the question stipulates that no two points have the same coordinate, in which case $k = 1$, and the query executes in $O(\log n)$, as $d = 2$. (However, we do not absolutely need to assume that no two points are at the same coordinate. Because we are reporting existence or non-existence of such a point, once we have found the required point, we can terminate the query; if the point does not exist, no overhead was incurred in reporting a point. This modified query can execute in $O(\log n)$ time even if there are multiple points at the same coordinate.) □