Chapter 2

Polygons

One of the most natural and well-studied geometric structures is the polygon. It is defined as a cyclic sequence $P = (p_1, p_2, \ldots, p_n, p_{n+1} = p_1)$ of n points in the plane, such that each consecutive pair determines an edge $\overline{p}_i p_{i+1}$. A polygon is *simple* if no two of its edges have intersecting interiors. In addition to simplicity, the concept of convexity is also central to geometric algorithm design.

2.1 Planar Convex Hulls

Given a set $S = \{p_1, p_2, \ldots, p_n\}$ of n points, a *convex combination* of the points in S is a point defined by

$$p = \sum_{i=1}^{n} \alpha_i p_i,$$

where $\sum_{i=1}^{n} \alpha_i = 1$. The *convex hull* of S, denoted $conv(S)$, is the set of all convex combinations of points in S. It can alternatively be viewed as the smallest convex set containing the points in S. The convex hull can be defined for a set of points in arbitrary d-dimensional space, but let concentrate for the time being on some methods for constructing 2-dimensional hulls.

Theorem 2.1.1 [?]: *A point p is inside the interior of $conv(S)$ if and only if there are three points q, r, and s in S such that p is inside the interior of $\triangle qrs$.*

This elegant characterization theorem immediately implies a simple $O(n^4)$ algorithm for determining those points in S on the boundary of $conv(S)$: for each pont $p \in S$ check if p is inside any triangle determined by three other points in S, of which there are $\binom{n}{3}$. We can do much better than this, however. Indeed, in the remainder of this section we show, in turn, how to apply each of three powerful algorithmic paradigms to design efficient algorithms for 2-d convex hull construction.

2.1.1 Gift wrapping

Let S be a set of n points in the plane. The first method we describe for computing $conv(S)$ is an algorithm due to Jarvis [?] based upon a paradigm of incrementally "wrapping" a line L around S all the while keeping L tangent to $conv(S)$.

An incremental approach

We begin by locating a point $p \in S$ with minimum y-coordinate. If there are more than one, pick the one among them that has largest x-coordinate. If we let L be the horizontal line through p, then L is clearly tangent to $conv(S)$ at p. We orient L from left to right. We then peform the "wrapping" step, where we locate the point q in S that forms the smallest counterclockwise angle with L. This

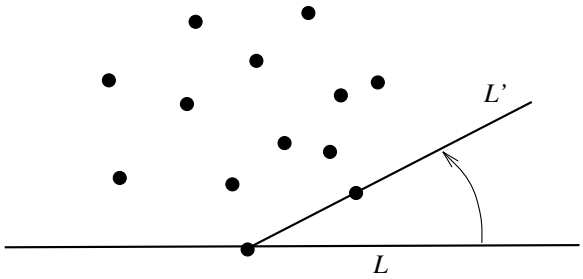

Figure 2.1: The "wrapping" step.

point must also be on $conv(S)$. (See Figure 2.1.) Thus, we can repeat the wrapping step at q with the line $\overleftarrow{pq}\rightarrow$ playing the role of L (oriented from p to q). We continue performing wrapping steps, discovering a new vertex of $conv(S)$ at each iteration until we return to the point in S where we began (the point with minimum y-coordinate).

This algorithm is easy to prove correct, as it incrementally discovers points on $conv(S)$ one after the other in a counterclockwise manner. So, let us analyze this algorithm's running time. Finding the starting point p can easily be done in $O(n)$ time, by a simple minimum-finding procedure. Likewise, each wrapping step can also be performed in $O(n)$ time by a similar procedure. Thus, since there can be $O(n)$ points on the boundary of $conv(S)$, the total running time of the gift wrapping procedure is $O(n^2)$.

Quick-Hull

There is an alternate way to implement the gift wrapping algorithm that could result in a faster algorithm in practice. It is commonly refered to as the *quick-hull* algorithm, due to a slight similarity with the well-known quick-sort algorithm.

In this case we begin by finding points a, b, c, and d in S with smallest y-coordinate, largest x-coordinate, largest y-coordinate, and smallest x-coordiante, respectively. Clearly, any points of S enclosed inside the interior of the polygon $P = (a, b, c, d, a)$ can be removed—they cannot be boundary points of $conv(S)$. In addition, we can define a smallest axis-parallel rectangle R enclosing S by passing horizontal lines through a and c and passing vertical lines through b and d, so that each of the remaining points of S are contained in one of the triangles formed by the region between P and R. (See Figure 2.2(a).) We then recursively enumerate the convex hull points in the triangles with bases $\overline{ab}$, $\overline{bc}$, $\overline{cd}$, and $\overline{da}$, respectively (there may be 2, 3, or 4 recursive calls depending upon the uniqueness of a, b, c, and d). In a generic recursive step we are given a subset S' of S contained in the interior of a given triangle T with base $\overline{pq}$ such that p and q known to be on the boundary of $conv(S)$. If $S' = \emptyset$ then we output the point q and return. Otherwise, we find the point r in S' farthest from $\overline{pq}$. Any points inside the triangle $\triangle prq$ can be removed—they cannot be on the boundary of $conv(S)$. Likewise, taking a line parallel to $\overline{pq}$ through r, together with edges $\overline{qr}$ and $\overline{rp}$ define triangles inside T. We then recurse on the points of S' inside the new triangles with base $\overline{qr}$ and $\overline{rp}$, respectively. (See Figure 2.2.)

Interestingly, the quick-hull procedure outputs the points on the boundary of $conv(S)$ in the same order as in the incremental gift-wrapping algorithm. It may be more advantagious in practice, however, because of the possibility that a large number of points of S will be eliminated with each iteration. Still, the worst case running time is $O(n^2)$, just as in the previous algorithm. The reason for this is that at any level in the recursion we may only be eliminating a single point from S and recursing on a subset of size $n - 1$. Thus, the worst-case running time is characterized by the recurrence relation

$$T(n) = T(n-1) + bn,$$

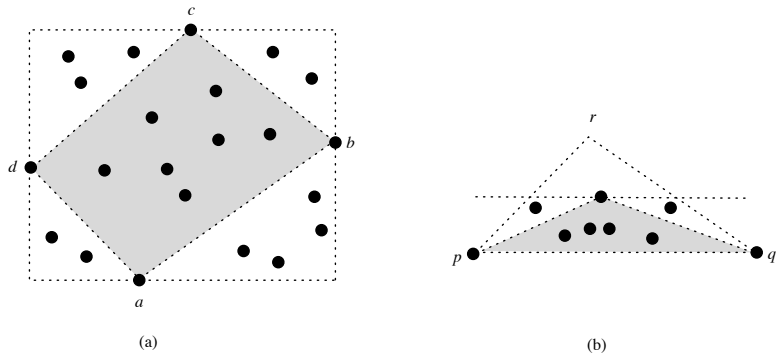

Figure 2.2: The Quick-Hull Algorithm. In (a) we illustrate the starting condition and in (b) a generic recursive step.

for some constant b, which has solution $T(n) = O(n^2)$.

Output Sensitivity. This analysis is perhaps a bit too pessimistic, however, for, as in the incremental gift wrapping algorithm, we output a vertex on the boundary of $conv(S)$ at each major step of the algorithm (be it a gift wrapping step or a recursive call in the quick-hull algorithm). Since each major step can be implemented in $O(n)$ time it is probably more accurate to characterize the running time of these two algorithms as being $O(nh)$, where h is the number of vertices on $conv(S)$. Indeed, this shows another advantage of these two algorithms—they run very fast if the number of points on the convex hull is small compared to the total number of points. Such a situation occurs, for example, if the points of S are randomly and uniformly distributed in a square, in which case the expected number of vertices on the convex hull is $O(\log n)$ [**?**].

2.1.2 Plane Sweeping

Still, there are a number of applications where the number of points on the convex hull will be large. Fortunately, there are alternative approaches to convex hull construction that leads to an efficient algorithm for all values of h. The first such approach we discribe is based upon a powerful algorithmic technique called *plane sweeping*.

Using plane sweeping for convex hull construction

We first describe how this paradigm can be used for convex hull construction and then we later describe the main components for applying this technique in general. So, let S be a set of n points in the plane. As an adaptation of an algorithm due to Graham [**?**], we describe a method for constructing the *upper hull* of S, that is, the set of edges of $conv(S)$ with normal vectors that "point up." Formally, an edge is on the upper hull if the last component of its normal vector is positive.

We begin by sorting the points of S by their x-coordinates. Then, we imagine a sweeping of the plane from left to right by a vertical line L. During this sweep we maintain the *invariant* of always maintaining a representation of the upper hull of all the points of S to the left of L. Initially, when L is to the left of the first point S, this hull is empty, and at the end, when L is to the right of the last point in S, we will have a representation of the entire upper hull of S.

Even though we describe this sweep as a continuous process, it actually proceeds in a sequence of discrete *events*, where we may need to update our invariant. In particular, we must update our invariant as our sweep line L encounters each in S. In order to facilitate efficient processing of each event, we make use of a *data structure*, which, in this case, will be a stack storing the upper hull of all the points to the left of L, listed left to right on the stack (so that the rightmost hull vertex is at the top).

Let us therefore examine the computations needed to process an event, where L is sweeping past some point $p \in S$. Let q be the top element on the stack (if it exists), and let r be the stack element below q (if it exists). There are three cases:

1. q or r doesn't exist. Then we push p onto the stack, for it must be part of the upper hull of points now to the left of L.

2. r, q, and p form a right turn. Then we push p onto the stack, for it must be part of the upper hull of points now to the left of L.

3. r, q, and p form a left turn. Then we pop q from the stack and repeat the above test for p.

Note that we may perform a number of stack pops during the processing of the event for p, but eventually we must end up in case 1 or 2. When we have completed processing for the last point in S, we are done—the stack holds a representation of the upper hull of S.

Analysis. The initial pre-sorting step can be implemented in $O(n \log n)$ time, using, say, a heapsort or mergesort procedure. We claim that the rest of the computation requires just $O(n)$ time. The reason is that we perform a constant amount of work when we push a point onto the stack and a constant amount of work when we pop a point from the stack, and a point will be pushed and popped at most once during the entire computation. Thus, by using the plane sweeping paradigm, we can compute the upper hull (or a lower hull) of a set of sorted points in $O(n)$ time. Therefore, we can compute the convex hull of n points in the plane in $O(n \log n)$ time in total.

The three components of plane sweeping

In describing our plane sweeping solution we highlighted three important components of any sweep-type algorithm:

1. An invariant. A sweep algorithm should maintain some invariant involving the problem to be solved. This invariant should be true before sweeping begins, be able to be maintained throughout the sweep, and should be sufficient to solve the problem being addressed when the sweeping has completed.

2. A set of events. There should be a discrete set of events where processing needs to be done in order to maintain the invariant. This set of events does not necessarily need to be known completely in advance, but at each stage in the sweep the next event to be processed should be clear.

3. Data structure(s). In order to efficiently process the events good data structures should be employed to maintain the invariant and the set of events.

Although it results in an elegant algorithm that should also run very fast in practice, this plane sweeping algorithm does not extend very nicely into higher dimensions. There is an alternate method, however, based upon another powerful algorithmic technique—divide-and-conquer—which does.

2.1.3 Divide-and-conquer

In applying the *divide-and-conquer* technique to solve a particular problem, one divides the problem into a number of smaller subproblems, recursively solves each subproblem in turn, and then joins all the subproblem solutions together to produce a solution to the entire problem.

In applying this technique to the problem of computing the upper hull of a set S of n points in the plane we can derive a convex hull algorithm similar to one designed by Preparata and Hong [**?**]. We partition S into two sets S_1 and S_2 of roughly equal size (we can be off by one) using a vertical dividing line L, recursively find the upper hull of S_1 and S_2, and then "merge" these two hulls to form an upper hull for S. The crucial step in this process is, of course, the "merge" step, and the crucial computation in this step is to find a common supporting tangent line for the upper hulls of

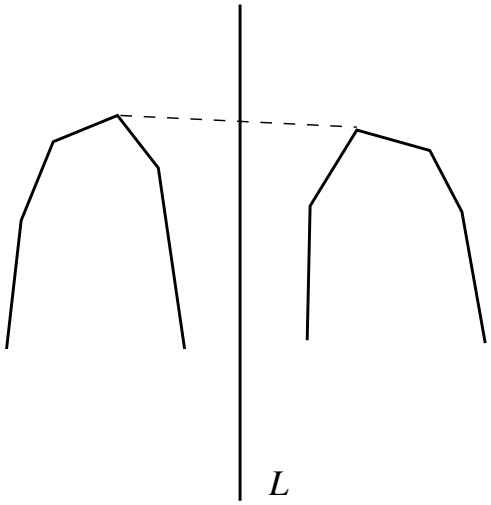

Figure 2.3: A Divide-and-Conquer Upper-Hull Algorithm. The crucial step is finding the common supporting tangent, shown above as a dashed line.

S_1 and S_2. (See Figure 2.3.) It is fairly straightforward to find such a tangent line in $O(n)$ time (in fact, we give it as an exercise). Given this line it is a simple matter to then "prune" away the portions of the upper hulls of S_1 and S_2 under this line, which cannot belong to the upper hull of S.

What is perhaps a bit more surprising is that, using a two-way binary search method of Overmars and VanLeeuwen [**?**], the common supporting tangent line can actually be found in $O(\log n)$ time, assuming that the upper hulls of S_1 and S_2 are stored in arrays. The idea is to perform coordinated binary searches simultaneously in the two upper hulls. With each constant-time "probe" we can guarantee that we will make progress in one of the two binary searches, possibly both; hence, we will complete both searches in $O(\log n)$ time. Let U_1 denote the portion of the upper hull of S_1 in which we have restricted the left tangent vertex to lie (initially, U_1 is the entire upper hull). Likewise, let U_2 denote the portion of the upper hull of S_1 in which we have restricted the right tangent vertex to lie. A generic probe consists of selecting the median vertex, p, in U_1 and the median vertex, q, in U_2. The line segment $\overline{pq}$ can touch each upper hull in one of three ways—it can be locally *beneath* it, locally *tangent*, or locally *above* it—and we can test which state it is in simply by examining the neighbors of p and q on their respective upper hulls in constant time. How we proceed, then, depends upon the outcomes of these tests. The portions of U_1 and U_2 that we can eliminate from consideration as a result of this probe are illustrated in Figure 2.4. The method for resolving a probe is immediate in most cases. The only exception is when the line $\overleftrightarrow{pq}$ is locally beneath U_1 and U_2. In this case we can resolve the probe by extending to a ray $\vec{r}_1$ the upper hull edge on U_1 incident to p on the right and extending to a ray $\vec{r}_2$ the upper hull edge on U_2 incident to q on the left and seeing where they intersect. If the intersection point of $\vec{r}_1$ and $\vec{r}_2$ lies to the left of the vertical dividing line L, then we can eliminate the portion of U_1 to the left of p, for all the points of S_2 must be below the ray $\vec{r}_1$. Likewise, if this intersection point is to the right of L, then we can eliminate the portion of U_2 to the right of q. In each case, then, with a constant amount of work we can make progress on one of the two sides, reducing the size of U_1 or U_2 by half. Thus, in $O(\log n)$ time we can locate the common upper tangent of the upper hulls of S_1 and S_2.

Analysis. The median dividing line L can be found in linear time. This can either be done directly using a rather complicated procedure, or we can presort the set S from left-to-right (by x-coordinates if represented in Cartesian coordinates) and thereafter simply divide the current list in two in linear time. We then recursively solve the two subproblem solutions and merge them together in linear time (with the crucial tangent-finding step possibly implemented in $O(\log n)$ time). Thus,

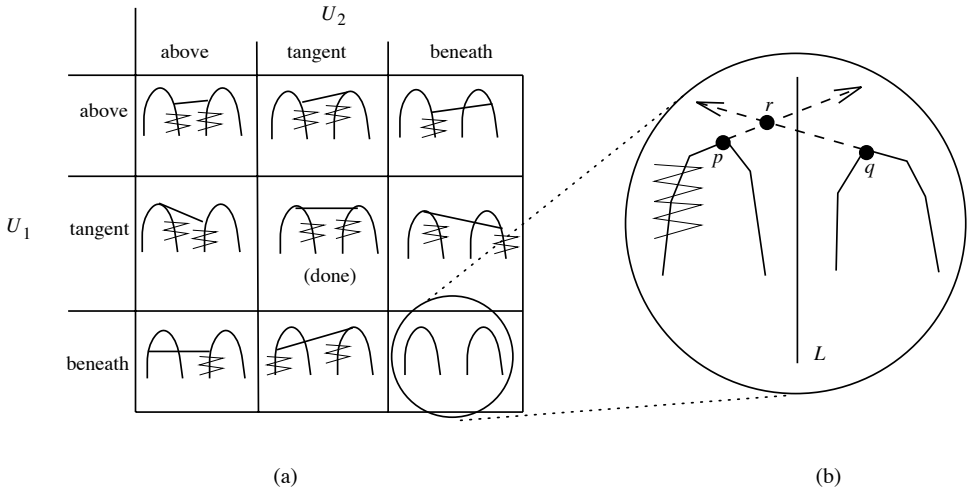

Figure 2.4: The Cases for Upper Common Tangent Finding Algorithm. In (a) we illustrate the eight most simple cases, and in (b) how to resolve the beneath-beneath case.

if we let $T(n)$ denote the running time of this method, we can characterize its running time as $T(1) = b$ and, for $n > 1$,

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + bn,$$

for some constant b. Assuming for the sake of analysis that n is a power of 2, we can simplify this to $T(n) = 2T(n/2) + bn$. This implies that $T(n)$ is $O(n \log n)$.

Thus, we have derived a second convex hull algorithm running in $O(n \log n)$ time. Is this the best we can do?

2.1.4 A Simple Lower Bound

In terms of worst-case complexity it indeed is. For we can easily reduce the well-known sorting problem to the convex hull problem in $O(n)$ time. Since sorting has an $\Omega(n \log n)$ lower bound in the comparison model, this implies that any comparison-based convex hull algorithm must take time $Omega(n \log n)$. For otherwise we could derive a faster sorting algorithm.

So, let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of numbers we wish to sort. We transform this to a set S of n points in the plane by the simple assignment $p_i := (x_i, x_i^2)$. Geometrically, this involves projecting all the points of X up onto the parabola $y = x^2$. Since each point projected onto this curve must be on the convex hull, a listing of the points around boundary of the convex hull of S can easily be transformed into a sorted listing of the numbers in X. Therefore, we have just given an $O(n)$-time reduction of the sorting problem to the convex hull problem.

2.1.5 Problems

1. A naive implementation of the gift wrapping algorithm would require the computation angles (e.g., using arc-cosines). Show how to implement a wrapping step in $O(n)$ time using the *left-of* function.

2. Give a simple $O(n)$-time procedure for finding the upper common tangent between two upper hulls U_1 and U_2 of total size n.

3. Why is the above resolution of the beneath-beneath case in the $O(\log n)$-time tangent-finding algorithm correct?

4. Describe an efficient method for determining if two planar point sets A and B can be separated by a line. Characterize the running time of your method in terms of $n = |A|$ and $m = |B|$.

5. Given a set S of n points in the plane, define the *width* of S as the smallest distance between a pair of parallel lines that have all the points of S on or between them (taken over all such pairs). Describe an efficient algorithm for determining the width of S, and characterize the running time of your method.

2.1.6 FastHull – A Worst-Case Optimal Output-Sensitive Algorithm

From the previous discussion one might be tempted to think that one must always choose between achieving output sensitivity and worst-case optimality. This is not necessary, however, for, as we show in this section, one can design a planar convex hull algorithm that has a worst-case running time that is $O(n \log h)$, which is optimal. The method we describe is an ingenious variation due to Chan [?] on the QuickHull procedure.

Let us assume, without loss of generality, that we are again interested in computing the upper hull of a set S of n points in the plane. Furthermore, let s and t respectively denote the leftmost and rightmost points in S (taking the ones with maximum y-coordinates if there are ties). If there are no other points in S besides s and t then we can return the segment $\overline{st}$ and we are done. So suppose there are at least three points in S. Arbitrarily pair up all the points in S (leaving one left over if n is odd). Each such pair determines a line. Using an $O(n)$-time selection method of XX [?], find the pair (p, q) that determines the line ℓ with median slope from this group. Let $y = mx + b$ be the equation of this line (if the median line is vertical, use the equation for the line $\overleftrightarrow{ab}$). Find the point $r = (r_x, r_y)$ in S such that $r_y - mr_x$ is maximum. This point must necessarily be on the boundary of the convex hull of S. Let S_1 denote the points of S that have x-coordinate less than or equal to r_x, and let S_2 denote the points of S that have x-coordinate greater than or equal to r_x. Next, we perform a pruning process, where we remove each point p such that p is directly below the line segment $\overline{qr}$, where q denotes the point p was paired with. We complete the algorithm by recursively solving the problem for the pruned subsets S_1 and S_2.

Analysis. Let us analyze the worst-case running time, $T(n, h)$, of the FastHull method, where h denotes the number of vertices in the upper hull. From the description above we can write

$$T(n, h) \leq T(n_1, h_1) + T(n_2, h_2) + bn,$$

where b is a constant and h_i is the number of vertices in the upper hull of the recursive problem S_i, with $n_i = |S_i|$, for $i = 1, 2$. Immediately from defintions, we have that

$$n_1 + n_2 \leq n,$$

and

$$h_1 + h_2 \leq h + 1.$$

In addition,

$$n_i \leq \frac{3}{4}n.$$

For example, note that there are $n/2$ pairs (p, q) defining lines with slopes greater than that of ℓ. For each such pair, either p or q is not in S_1 or we have that one of p or q was removed in the pruning step. Thus, there are at least $n/4$ points that cannot be in the subproblem for S_1. And a similar argument holds for S_2.

Let us now show how these inequalities can be used to prove, by induction on n and h, that $T(n, h) \leq cn \log h$, for some constant $c > 0$. There are two base cases, $n \leq 2$ and $h \leq 2$, which both yield a running time of $O(n)$ in the FastHull procedure (which satisfies the induction hypothesis). So, let us assume that the induction hypothesis holds for values smaller than n and h. Consider $T(n, h)$ for $n > 2$ and $h > 2$. As we have already observed, in this case,

$$T(n, h) \leq T(n_1, h_1) + T(n_2, h_2) + bn,$$

which, by induction, can be written

$$T(n,h) \leq cn_1 \log h_1 + cn_2 \log h_2 + bn.$$

Without loss of generality, let us assume that $h_1 \leq h_2$, so $h_1 \leq h/2$. Then we can write

$$\begin{aligned} T(n,h) &\leq cn_1 \log h/2 + cn_2 \log h + bn \\ &\leq c(n_1+n_2)\log h - cn_1 + bn \\ &\leq cn \log h, \end{aligned}$$

for $c \geq 8b$, because either $n_1 + n_2 \leq 7n/8$ or $n_1 \geq n/8$. Therefore, the worst case running time of FastHull is $O(n \log h)$.

2.1.7 Problems

1. There is a simple-to-implement version of FastHull in which the call to the method of XX [?] for finding the line with median slope is replaced by a step of simply selecting a random pair (p,q). Show that this method has an expected running time that is $O(n \log h)$, where the expectation is taken over all possible runnings of the algorithm (not any assumptions about the input distribution).

2.2 2-Dimensional Linear Programming

We begin by describing the 2-dimensional version of this problem. Suppose one is given a collection S of n half planes in $\mathcal{R}^2$, each of which is oriented to be above its bounding line. The 2-dimensional *linear programming* problem can be viewed as the problem of finding the vertex with largest y-coordinate in the intersection of these n half planes. Let us further assume, for the sake of simplicity, that this point is bounded (the method we describe can be easily modified to handle general types of 2-dimensional linear programs).

The method we describe is based upon a powerful searching strategy, developed by Megiddo, known as *prune-and-search*. The main idea, at a high level, is to perform a binary-search style of search for the optimal value in such a way that with each "probe" in this search we eliminate a constant fraction of the remaining half planes. In particular, we do the following computation. We pair up the half planes in S arbitrarily. Each pair of bounding lines, of course, intersect in a point. Let p_m be the median such intersection point, which can be found in $O(n)$ time. We then test for the vertical line L containing p_m if the optimal solution lies to the left or right of L (or on L, in which case we will be done). Suppose the optimal value is to the right of L. Then for each pair with an intersection point to the left of L we can eliminate one of the half planes determining this intersection (the one with boundling line of larger slope). Likewise, if the optimal value is to the left of L, then we can eliminate 1/2 of the half planes to the right of L. Since we chose L to contain p_m, this implies that we will eliminate one fourth of the half planes with this probe. We complete the algorithm, then, by recursively searching on the remaining collection of half planes.

We can characterize the running time of this method using the recurrence relation:

$$T(n) \leq T(3n/4) + bn,$$

for some constant $b > 0$, which implies that $T(n)$ is $O(n)$. Incidentally, this approach can be extended to $\mathcal{R}^d$, for any fixed dimension d, resulting in an $O(n)$-time algorithm in each case (albeit with a constant fact that depends exponentially on d).

2.3 Polygon Triangulation

Another fundamental problem involving polygons is that of polygon triangulation. In this problem one is given an n-node simple polygon P and asked to add non-crossing diagonal edges between various pairs of vertices of P so that the interior of P is partitioned into triangles.

2.3.1 Trapezoidal Decomposition

As we show in the next subsection, the polygon triangulation problem is closely related to another decomposition problem–trapezoidal decomposition. In this problem one is given a collection of non-crossing line segments (which may or may not form a simple polygon) and asked to partition the plane by extending a vertical ray up and down from each segment endpoint until it hits another segment (or tends off to positive or negative infinity). Our method for solving the trapezoidal decomposition problem is by the plane sweeping paradigm.

We sweep a vertical line L from left to right across the set of segments maintaining the invarant that we keep the segments intersected by L ordered by the above relation. We stop at various event points, which in this case are defined by segment endpoints. By maintaining the current set of intersected segments in a balanced binary search tree B (such as a red-black tree) we can process each event in $O(\log n)$ time. If an event corresponds to a left endpoint of some segment s, then we locate the new segment in B, and report as its endpoint's vertical neighbors the segments directly above and below s in B. If an event corresponds to a right endpoint of some segment s, then we report as its endpoint's vertical neighbors the segments directly above and below s in B, and then we delete s from B. When the sweep completes we will have decomposed the plane into trapezoids. The total time for this sweep is $O(\log n)$ per event; hence, $O(n \log n)$ overall.

In the case that the segments form a simple polygon, then Chazelle shows that this trapezoidal decomposition can actually be performed in $O(n)$ time. We do not give the details of this method, however, and instead we focus on how a trapezoidal decomposition can be converted into a triangulation.

2.3.2 Converting a Trapezoidal Decomposition into a Triangulation

Suppose we are given a collection S of n line segments, which may or may not form a simple polygon. Suppose further that we are given a trapezoidal decomposition for S. In this section we show how to convet this decomposition into a triangulation in $O(n)$ time. Our method is to process each segment in S in turn.

For each segment $s = (v, w)$, we identify the ordered list of vertices $(v_1, v_2, \ldots, v_k)$ that have s as a downward vertical neighbor in linear ($O(k)$) time, given the list of trapezoids. We then add the diagonal edges $\overline{vv_1}$ and $\overline{v_k w}$ as well as $\overline{v_i v_{i+1}}$, for each $i = 1, 2, \ldots, k-1$, provided that the edge to add does not already exist as a segment of S. We perform a similar computation for the vertices that have s as an upwared vertical neighbor. This decomposes the region we wish to triangulate into *one-sided monotone* polygons, that is, simple polygons with a distinguished edge e such that any vertical line intersecting the polygon intersects e and one other edge. We triangulate each one-sided monotone polygon in linear time via simple plane sweep. Specifically, we sweep the polygon (which, without loss of generality, we assume to be oriented so that its interior is above its distinguished edge) from left to right, maintain the lower hull of the edges encountered so far, as well as a triangulation of the portion between this lower hull and the upper polygon chain of the one-sided monotone polygon.. If the next vertex v in this sweep forms a left turn with the previous edge, then we add v to the current hull. If the next vertex v in this sweep forms a right turn with the previous edge, however, then we add an edge from v to the previous vertex in the hull, remove this vertex, and repeat the test for v. Thus, we add an edge from v to each vertex on the previous hull that v "sees." Since we insert and delete a vertex of a one-sided monotone polygon at most once, this computation completes in linear time. Moreover, since the lower hull of the vertices of this one-sided monotone polygon is the distinguished edge itself, our invariant implies that we have

triangulated its interior upon completion of the plane sweep procedure. The total time is $O(n)$ (plus the time to compute a trapezoidal decomposition).