24 Sample problems and algorithms

The nature of geometric computation: Three problems and algorithms chosen to illustrate the variety of issues encountered. (1) Convex hull yields to simple and efficient algorithms, straightforward to implement and analyze. (2) Objects with special properties, such as convexity, are often much simpler to process than are general objects. (3) Visibility problems are surprisingly complex; even if this complexity does not show in the design of an algorithm, it sneaks into its analysis.

24.1 Geometry and geometric computation

Classical geometry, shaped by the ancient Greeks, is more axiomatic than constructive: It emphasizes axioms, theorems, and proofs, rather than algorithms. The typical statement of Euclidean geometry is an assertion about all geometric configurations with certain properties (e.g., the theorem of Pythagoras: "In a right-angled triangle, the square on the hypotenuse c is equal to the sum of the squares on the two catheti a and b: $c^2 = a^2 + b^2$") or an assertion of existence (e.g., the parallel axiom: "Given a line L and a point $P \notin L$, there is exactly one line parallel to L passing through P"). Constructive solutions to problems do occur, but the theorems about the *impossibility* of constructive solutions steal the glory: "You cannot trisect an arbitrary angle using ruler and compass only," and the proverbial "It is impossible to square the circle."

Computational geometry, on the other hand, starts out with problems of construction so simple that, until the 1970s, they were dismissed as trivial: "Given n line segments in the plane, are they free of intersections? If not, compute (construct) all intersections." But this problem is only trivial with respect to the *existence* of a constructive solution. As we will soon see, the question is far from trivial if interpreted as: How *efficiently* can we obtain the answer?

Computational geometry has some appealing features that make it ideal for learning about algorithms and data structures: (1) The problem statements are easily understood, intuitively meaningful, and mathematically rigorous; right away the student can try his own hand at solving them, without having to worry about hidden subtleties or a lot of required background knowledge. (2) Problem statement, solution, and every step of the construction have natural visual representations that support abstract thinking and help in detecting errors of reasoning. (3) These algorithms are practical; it is easy to come up with examples where they can be applied.

Appealing as geometric computation is, writing geometric programs is a demanding task. Two traps lie hiding behind the obvious combinatorial intricacies that must be mastered, and they are particularly dangerous when they occur together: (1) degenerate configurations, and (2) the pitfalls of numerical computation due to discretization and rounding errors. Degenerate configurations, such as those we discussed in Chapter 14 on intersecting line segments, are special cases that often require special code. It is not always easy to envision all the kinds of degeneracies that may occur in a given problem. A configuration may be degenerate for a specific algorithm, whereas it may be nondegenerate for a different algorithm solving the same problem. Rounding errors tend to cause more obviously disastrous consequences in geometric computation than, say, in linear algebra or differential equations. Whereas the traditional analysis of rounding errors focuses on bounding their cumulative value, geometry is concerned primarily with a stringent all-or-nothing question: Have errors impaired the topological consistency of the data? (Remember the pathology of the braided straight lines.)

In this Part VI we aim to introduce the reader to some of the central ideas and techniques of computational geometry. For simplicity's sake we limit coverage to two-dimensional Euclidean geometry - most problems become a lot more complicated when we go from two- to three-dimensional configurations. And we focus on a type of algorithm that is remarkably well suited for solving two-dimensional problems efficiently: sweep algorithms. To illustrate their generality and effectiveness, we use plane-sweep to solve several rather distinct problems. We will see that sweep algorithms for different problems can be assembled from the same building blocks: a skeleton sweep program that sweeps a line across the plane based on a queue of events to be processed, and transition procedures that update the data structures (a dictionary or table, and perhaps other structures) at each event and maintain a geometric invariant. Sweeps show convincingly how the dynamic data structures of Part V are essential for the efficiency.

The problems and algorithms we discuss deal with very simple objects: points and line segments. Applications of geometric computation such as CAD, on the other hand, typically deal with very complex objects made up of thousands of polygons. But the simplicity of these algorithms does not deter from their utility. Complex objects get processed by being broken into their primitive parts, such as points, line segments, and triangles. The algorithms we present are some of the most basic subroutines of geometric computation, which play a role analogous to that of a square root routine for numerical computation: As they are called untold times, they must be correct and efficient.

24.2 Convex hull: A multitude of algorithms

The problem of computing the convex hull H(S) of a set S consisting of n points in the plane serves as an example to demonstrate how the techniques of computational geometry yield the concise and elegant solution that we presented in Chapter 3. The convex hull of a set S of points in the plane is the smallest convex polygon that contains the points of S in its interior or on its boundary. Imagine a nail sticking out above each point and a tight rubber band surrounding the set of nails.

Many different algorithms solve this simple problem. Before we present in detail the algorithm that forms the basis of the program 'ConvexHull' of Chaper 3, we briefly illustrate the main ideas behind three others. Most convex hull algorithms have an initialization step that uses the fact that we can easily identify two points of S that lie on the convex hull H(S): for example, two points P_{min} and P_{max} with minimal and maximal x-coordinate, respectively. Algorithms that grow convex hulls over increasing subsets can use the segment $\overline{P_{min}P_{max}}$ as a (degenerate) convex hull to start with. Other algorithms use the segment $\overline{P_{min}P_{max}}$ to partition S into an upper and a lower subset, and compute the upper and the lower part of the hull H(S) separately.

1. *Jarvis's march* [Jar 73] starts at a point on H(S), say P_{min}, and 'walks around' by computing, at each point P, the next tangent $\overline{PQ}$ to S, characterized by the property that all points of S lie on the same side of $\overline{PQ}$ (Fig. 24.1) .

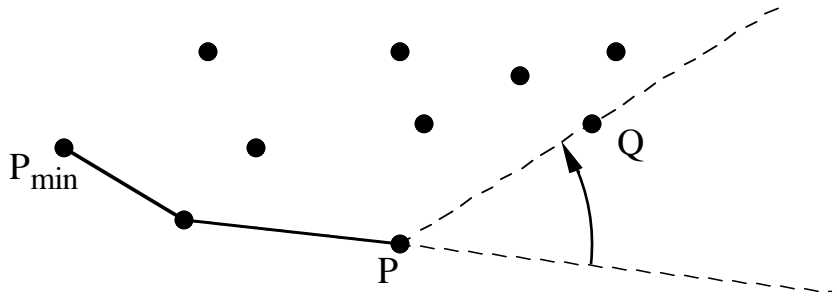

Figure 24.1: The "gift-wrapping" approach to building the convex hull.

2. *Divide-and-conquer* comes to mind: Sort the points of S according to their x-coordinate, use the median x-coordinate to partition S into a left half S_L and a right half S_R, apply this convex hull algorithm recursively to each half, and merge the two solutions $H(S_L)$ and $H(S_R)$ by computing the two common exterior tangents to $H(S_L)$ and $H(S_R)$ (Fig. 24.2). Terminate the recursion when a set has at most three points.

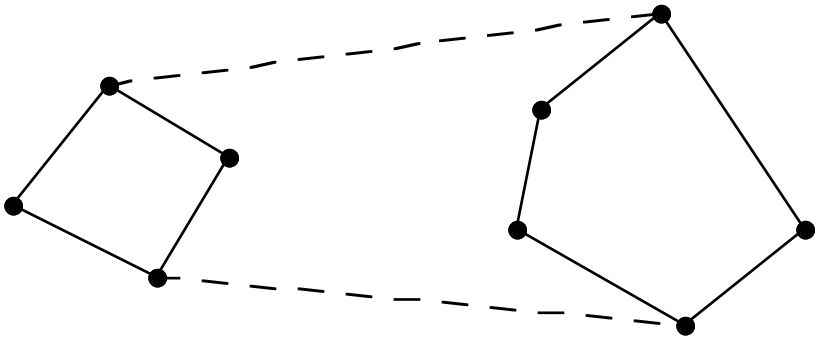

Figure 24.2: Divide-and-conquer applies to many problems on spatial data.

3. *Quickhull* [Byk 78], [Edd 77], [GS 79] uses divide-and-conquer in a different way. We start with two points on the convex hull H(S), say P_{min} and P_{max}. In general, if we know ≥ 2 points on H(S), say P, Q, R in Fig. 24.3, these define a convex polygon contained in H(S). (Draw the appropriate picture for just two points P_{min} and P_{max} on the convex hull.) There can be no points of S in the shaded sectors that extend outward from the vertices of the current polygon, PQR in the example. Any other points of S must lie either in the polygon PQR or in the regions extending outward from the sides.

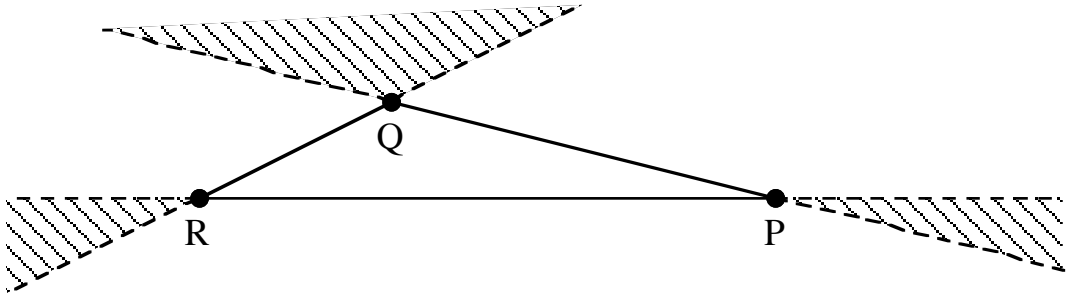

Figure 24.3: Three points known to lie on the convex hull identify regions devoid of points.

For each side, such as $\overline{PQ}$ in Fig. 24.4, let T be a point *farthest* from $\overline{PQ}$ among all those in the region extending outward from $\overline{PQ}$, if there are any. T must lie on the convex hull, as is easily seen by considering the parallel to $\overline{PQ}$ that passes through T. Having processed the side $\overline{PQ}$, we extend the convex polygon to include T, and we now must process 2 additional sides, $\overline{PT}$ and $\overline{TQ}$. The reader will observe a formal analogy between quicksort (Chapter 17) and quickhull, which has given the latter its name.

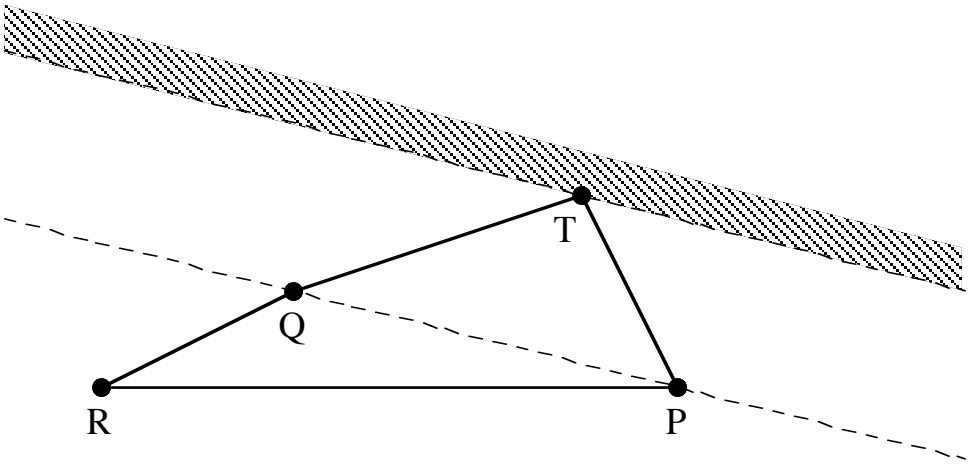

Figure 24.4: The point T farthest from $\overline{PQ}$ identifies a new region of exclusion (shaded).

4. In an *incremental scan* or *sweep* we sort the points of S according to their x-coordinates, and use the segment $\overline{P_{min}P_{max}}$ to partition S into an upper subset and a lower subset, as shown in Fig. 24.5. For simplicity of presentation, we reduce the problem of computing H(S) to the two separate problems of computing the upper hull U(S) [i.e., the upper part of H(S)], shown in bold, and the lower hull L(S), drawn as a thin line. Our notation and pictures are chosen to describe U(S).

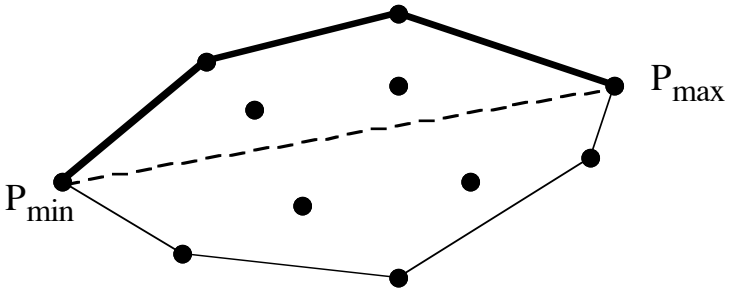

Figure 24.5: Separate computations for the upper hull and the lower hull.

Let $P_1, \ldots, P_n$ be the points of S sorted by x-coordinate, and let $U_i = U(P_1, \ldots, P_i)$ be the upper hull of the first i points. $U_1 = P_1$ may serve as an initialization. For i = 2 to n we compute U_i from U_{i-1}, as Fig. 24.6 shows. Starting with the tentative tangent $\overline{P_iP_{i-1}}$ shown as a thin dashed line, we retrace the upper hull U_{i-1} until we reach the actual tangent: in our example, the bold dashed line $\overline{P_iP_2}$. The tangent is characterized by the fact that for $j = 1, \ldots, i-1$, it minimizes the angle $A_{i,j}$ between $\overline{P_iP_j}$ and the vertical.

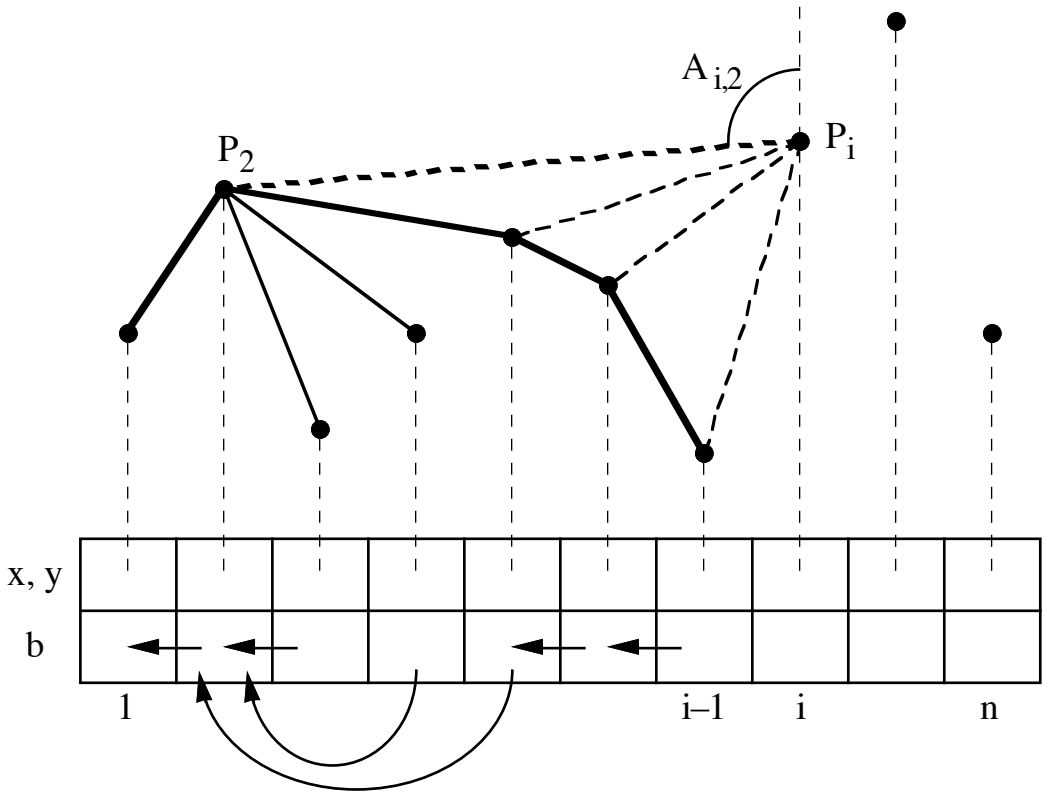

Figure 24.6: Extending the partial upper hull $U(P_1, \ldots, P_{i-1})$ to the next point P_i.

The program 'ConvexHull' presented in Chapter 3 as an example for algorithm animation is written as an on-line algorithm: Rather than reading all the data before starting the computation, it accepts one point at a time, which must lie to the right of all previous ones, and immediately extends the hull U_{i-1} to obtain U_i. Thanks to the input restriction that the points are entered in sorted order, 'ConvexHull' becomes simpler and runs in linear time. This explains the two-line main body:

```
PointZero; { sets first point and initializes all necessary variables }
while NextRight do  ComputeTangent;
```

There remain a few programming details that are best explained by relating Fig. 24.6 to the declarations:

```
var  x, y, dx, dy: array[0 .. nmax] of integer;
     b: array[0 .. nmax] of integer; { backpointer }
     n: integer; { number of points entered so far }
     px, py: integer; { new point }
```

The coordinates of the points P_i are stored in the arrays x and y. Rather than storing angles such as $A_{i,j}$, we store quantities proportional to $\cos(A_{i,j})$ and $\sin(A_{i,j})$ in the arrays dx and dy. The array b holds back pointers for retracing the upper hull back toward the left: b[i] = j implies that P_j is the predecessor of P_i in U_i. This explains the key procedure of the program:

```
procedure ComputeTangent;  { from P_n = (px, py) to U_n–1 }
var i: integer;
begin
  i := b[n];
  while  dy[n] · dx[i] > dy[i] · dx[n]  do  begin  { dy[n]/dx[n] > dy[i]/dx[i] }
    i := b[i];
    dx[n] := x[n] – x[i];  dy[n] := y[n] – y[i];
    MoveTo(px, py);  Line(–dx[n], –dy[n]);
    b[n] := i
  end;
  MoveTo(px, py);  PenSize(2, 2);  Line(–dx[n], –dy[n]);  PenNormal
end;  { ComputeTangent }
```

The algorithm implemented by 'ConvexHull' is based on Graham's scan [Gra 72], where the points are ordered according to the angle as seen from a fixed internal point, and on [And 79].

24.3 The uses of convexity: Basic operations on polygons

The convex hull of a set of points or objects (i.e., the smallest convex set that contains all objects) is a model problem in geometric computation, with many algorithms and applications. Why? As we stated in the introductory section, applications of geometric computation tend to deal with complex objects that often consist of thousands of primitive parts, such as points, line segments, and triangles. It is often effective to approximate a complex configuration by a simpler one, in particular, to package it in a container of simple shape. Many proximity queries can be answered by processing the container only. One of the most frequent queries in computer graphics, for example, asks what object, if any, is first struck by a given ray. If we find that the ray misses a container, we infer that it misses all objects in it without looking at them; only if the ray hits the container do we start the costly analysis of all the objects in it.

The convex hull is often a very effective container. Although not as simple as a rectangular box, say, convexity is such a strong geometric property that many algorithms that take time O(n) on an arbitrary polygon of n vertices require only time O(log n) on convex polygons. Let us list several such examples. We assume that a polygon G is given as a (cyclic) sequence of n vertices and/or n edges that trace a closed path in the plane. Polygons may be self-intersecting, whereas simple polygons may not. A simple polygon partitions the plane into two regions: the interior, which is simply connected, and the exterior, which has a hole.

Point-in-polygon test. Given a simple polygon G and a query point P (not on G), determine whether P lies inside or outside the polygon.

Two closely related algorithms that walk around the polygon solve this problem in time O(n). The first one computes the *winding number* of G around P. Imagine an observer at P looking at a vertex, say V, where the walk starts, and turning on her heels to keep watching the walker (Fig. 24.7). The observer will make a first (positive) turn α, followed by a (negative) turn β, followed by … , until the walker returns to the starting vertex V. The sum $\alpha + \beta + \ldots$ of all turning angles during one complete tour of G is: $2 \cdot \pi$ if P is inside G, and 0 if P is outside G.

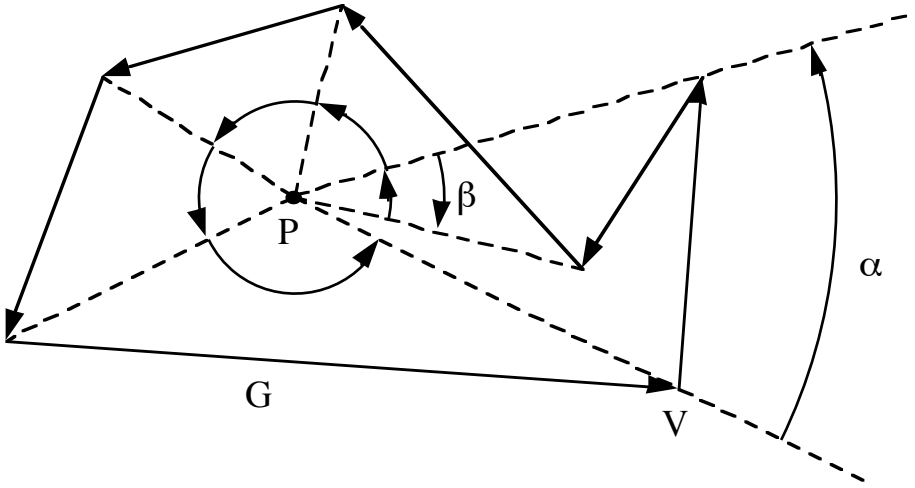

Figure 24.7: Point-in polygon test by adding up all turning angles.

The second algorithm computes the *crossing number* of G with respect to P. Draw a semi-infinite ray R from P in any direction (Fig. 24.8). During the walk around the polygon G from an arbitrary starting vertex V back to V, keep track of whether the current oriented edge intersects R, and if so, whether the edge crosses R from below (+1) or from above (–1). The sum of all these numbers is +1 if P is inside G, and 0 if P is outside G.

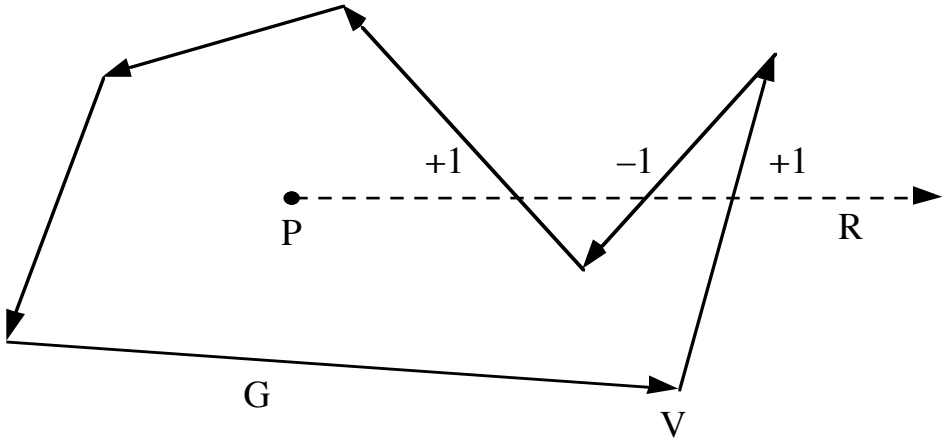

Figure 24.8: Point-in polygon test by adding up crossing numbers.

Point-in-convex-polygon test. For a convex polygon Q we use binary search to perform a point-in-polygon test in time O(log n). Consider the hierarchical decomposition of Q illustrated by the convex 12-gon shown in Fig. 24.9. We choose three (approximately) equidistant vertices as the vertices of an innermost core triangle, painted black. "Equidistant" here refers not to any Euclidean distance, but rather to the number of vertices to be traversed by traveling along the perimeter of Q. For a query point P we first ask, in time O(1), which of the seven regions defined by the extended edges of this triangular core contains P. These seven regions shown in Fig. 24.10 are all "triangles" (albeit six of them extend to infinity), in the sense that each one is defined as the intersection of three half-spaces. Four of these regions provide a definite answer to the query "Is P inside Q, or outside Q?" One region (shown hatched in Fig. 24.10) provides the answer 'In', three the answer 'Out'. The remaining three regions, labeled 'Uncertain', lead recursively to a new point-in-convex-polygon test, for the same query point P, but a new convex polygon Q' which is the intersection of Q with one of the uncertain regions. As Q' has only about $n/3$ vertices, the depth of recursion is O(log n). Actually, after the first comparison against the innermost triangular core of Q, we have no longer a general point-in-convex-polygon problem, but one with additional information that makes all but the first test steps of a binary search.

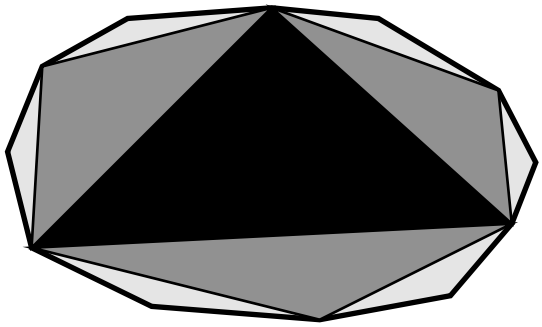

Figure 24.9: Hierarchical approximation of a convex 12-gon
as a 3-level tree of triangles. The root is in black,
its children are in dark grey, grandchildren in light grey.

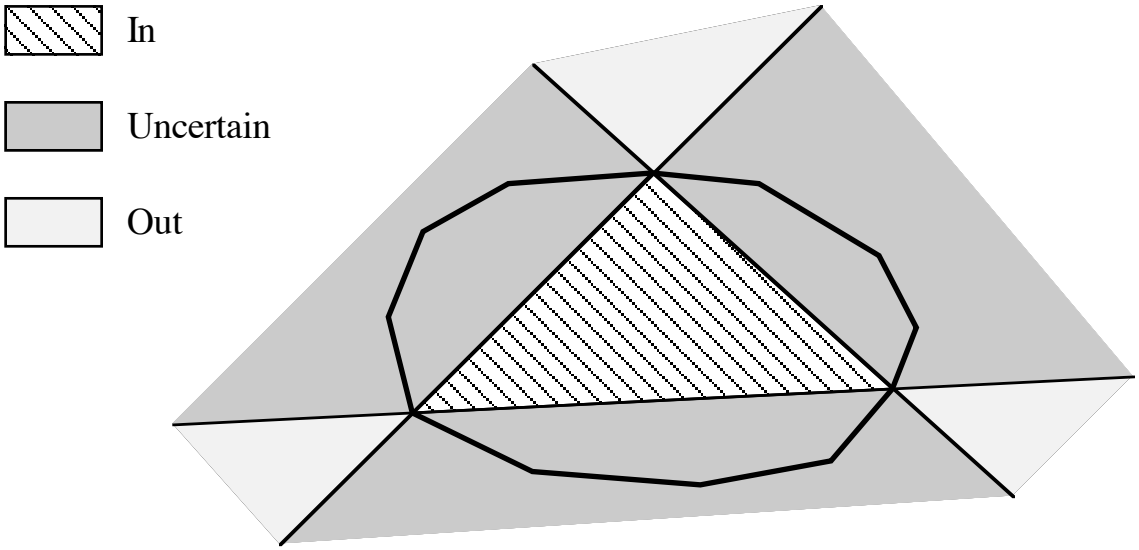

Figure 24.10: The plane partitioned into four regions of certainty and three of uncertainty. The latter are processed recursively.

24.4 Visibility in the plane: A simple algorithm whose analysis is not

Many computer graphics programs are dominated by visibility problems: Given a configuration of objects in three-dimensional space, and given a point of view, what is visible? Dozens of algorithms for hidden-line or hidden-surface elimination have been developed to solve this everyday problem that our visual system performs "at a glance". In contrast to the problems discussed above, visibility is surprisingly complex. We give a hint of this complexity by describing some of the details buried below the smooth surface of a "simple" version: computing the visibility of line segments in the plane.

Problem: Given n line segments in the plane, compute the sequence of (sub)segments seen by an observer at infinity (say, at $y = -\infty$).

The complexity of this problem was unexpected until discovered in 1986 [WS 88]. Fortunately, this complexity is revealed not by requiring complicated algorithms, but in the analysis of the inherent complexity of the geometric problem. The example shown in Fig. 24.11 illustrates the input data. The endpoints (P_1, P_{10}), (P_2, P_8), (P_5, P_{12}) of the three line segments labeled 1, 2, 3 are given; other points are computed by the algorithm. The required result is a list of visible segments, each segment described by its endpoints and by the identifier of the line of which it is a part:

$(P_1, P_3, 1)$, $(P_3, P_4, 2)$, $(P_5, P_6, 3)$, $(P_6, P_8, 2)$, $(P_7, P_9, 3)$, $(P_9, P_{10}, 1)$, $(P_{11}, P_{12}, 3)$

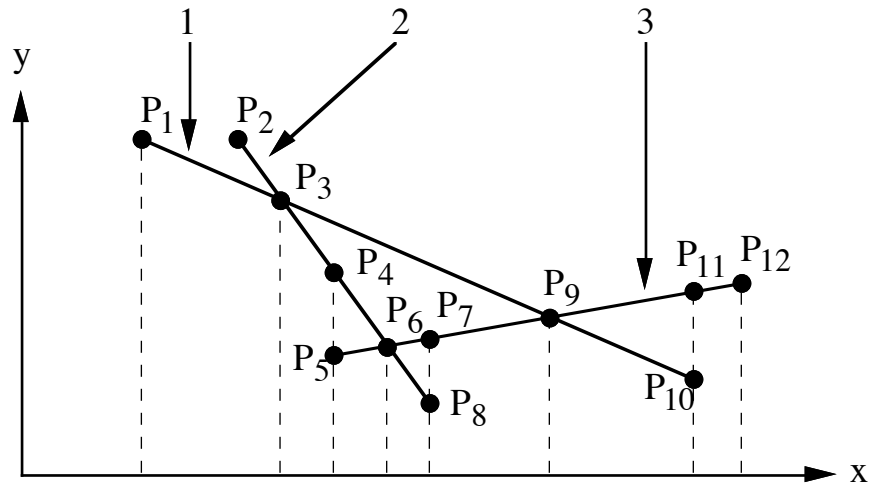

Figure 24.11: Example: Three line segments seen from below generate seven visible subsegments.

In search of algorithms, the reader is encouraged to work out the details of the first idea that might come to mind: For each of the n^2 ordered pairs (L_i, L_j) of line segments, remove from L_i the subsegment occluded by L_j. Because L_i can get cut into as many as n pieces, it must be managed as a sequence of subsegments. Finding the endpoints of L_j in this sequence will take time $O(\log n)$, leading to an overall algorithm of time complexity $O(n^2 \cdot \log n)$.

After the reader has mastered the sweep algorithm for line intersection presented in Chapter 25, he will see that its straightforward application to the line visibility problem requires time $O((n + k) \cdot \log n)$, where $k \in O(n^2)$ is the number of intersections. Thus plane-sweep appears to do all the work the brute-force algorithm above does, organized in a systematic left-to-right fashion. It keeps track of all intersections, most of which may be invisible. It has the potential to work in time $O(n \cdot \log n)$ for many realistic data configurations characterized by $k \in O(n)$, but not in the worst case.

Divide-and-conquer yields a simple two-dimensional visibility algorithm with a better worst-case performance. If $n = 0$ or 1, the problem is trivial. If $n > 1$, partition the set of n line segments into two (approximate) halves, solve both subproblems, and merge the results. There is no constraint on how the set is halved, so the divide step is easy. The conquer step is taken care of by recursion. Merging amounts to computing the minimum of two piecewise (not necessarily continuous) linear functions, in time linear in the number of pieces. The example with $n = 4$ shown in Fig. 24.12 illustrates the algorithm. f_{12} is the visible front of segments 1 and 2, f_{34} of segments 3 and 4, $\min(f_{12}, f_{34})$ of all four segments (Fig. 24.13).

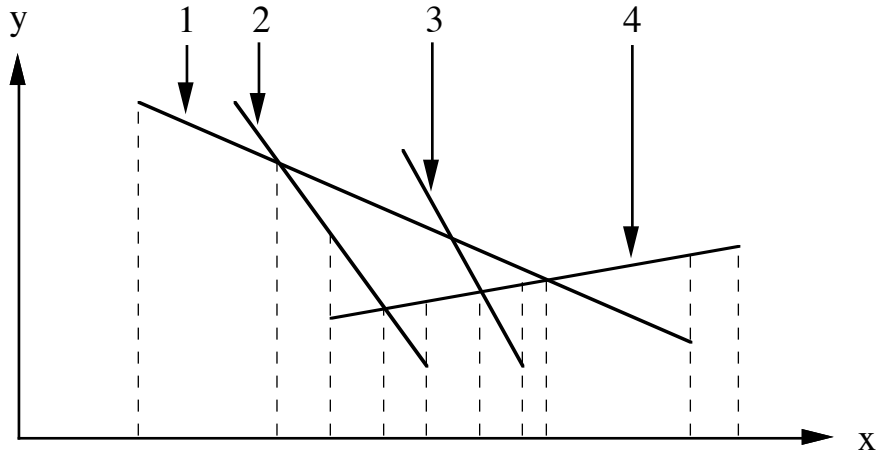

Figure 24.12: The four line segments will be partitioned into subsets {1, 2} and {3, 4}.

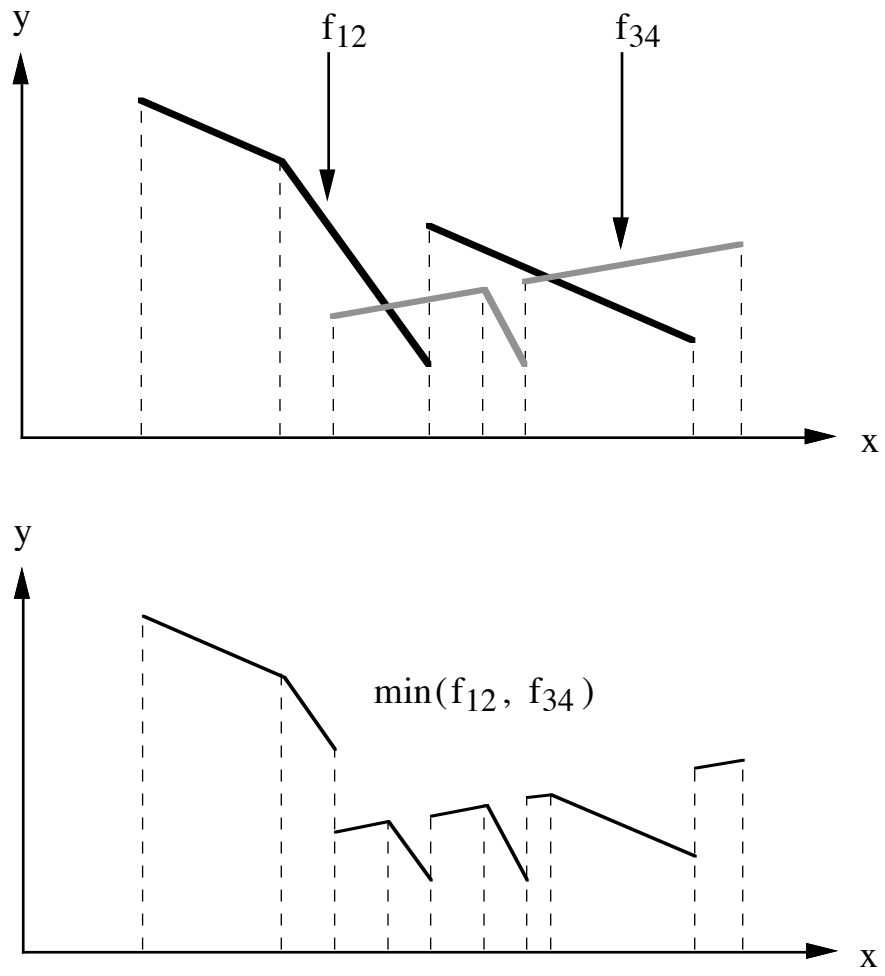

Figure 24.13: The min operation merges the solutions of this divide-and-conquer algorithm.

The time complexity of this divide-and-conquer algorithm is obtained as follows. Given that at each level of recursion the relevant sets of line segments can be partitioned into (approximate) halves, the depth of recursion is O(log n). A merge step that processes v visible subsegments takes linear time O(v). Together, all the merge steps at a given depth process at most V subsegments, where V is the total number of visible subsegments. Thus the total time is bounded by $O(V \cdot \log n)$. How large can V be?

Surprising theoretical results

Let V(n) be the number of visible subsegments in a given configuration of n lines, i.e. the size of the output of the visibility computation. For tiny n, the worst cases [V(2) = 4, V(3) = 8] are shown in Fig. 24.14. An attempt to find worst-case configurations for general n leads to examples such as that shown in Fig. 24.15, with V(n) = 5·n – 8.

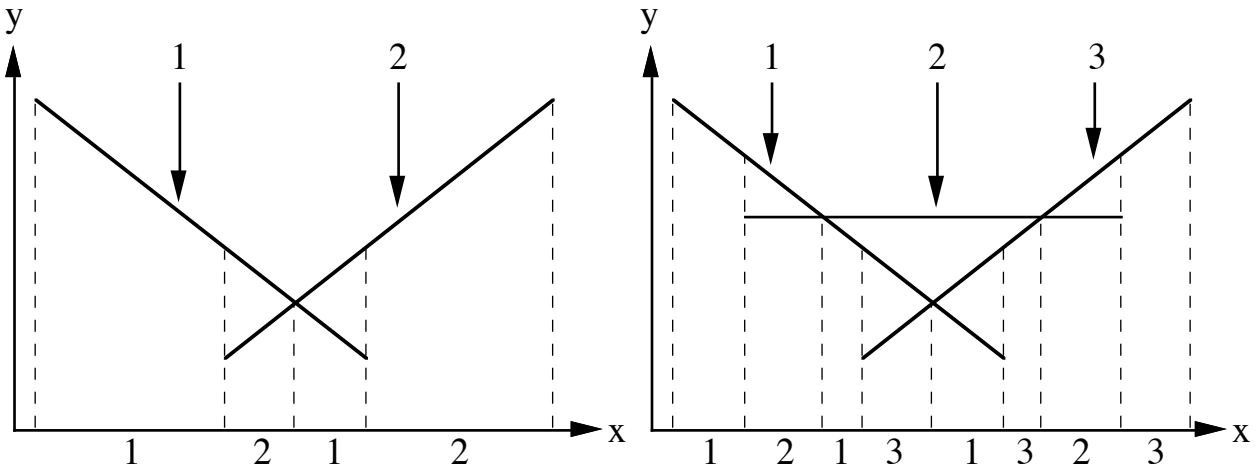

Figure 24.14: Configurations with the largest number of visible subsegments.

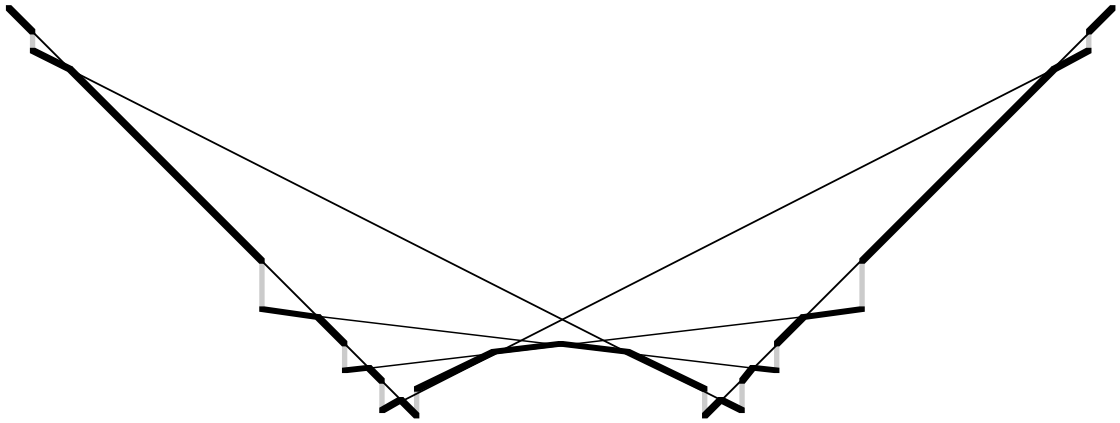

Figure 24.15: A family of configurations with 5·n – 8 visible subsegments.

You will find it difficult to come up with a class of configurations for which V(n) grows faster. It is tempting to conjecture that $V(n) \in O(n)$, but this conjecture is very hard to prove - for the good reason that it is false, as was discovered in [WS 88]. It turns out that $V(n) \in \Theta(n \cdot \alpha(n))$, where $\alpha(n)$, the inverse of Ackermann's function (see Chapter 15, Exercise 2), is a monotonically increasing function that grows so slowly that for practical purposes it can be treated as a constant, call it α.

Let us present some of the steps of how this surprising result was arrived at. Occasionally, simple geometric problems can be tied to deep results in other branches of mathematics. We transform the two-dimensional visibility problem into a combinatorial string problem. By numbering the given line segments, walking along the x-axis from left to right, and writing down the number of the line segment that is currently visible, we obtain a sequence of numbers (Fig. 24.16).

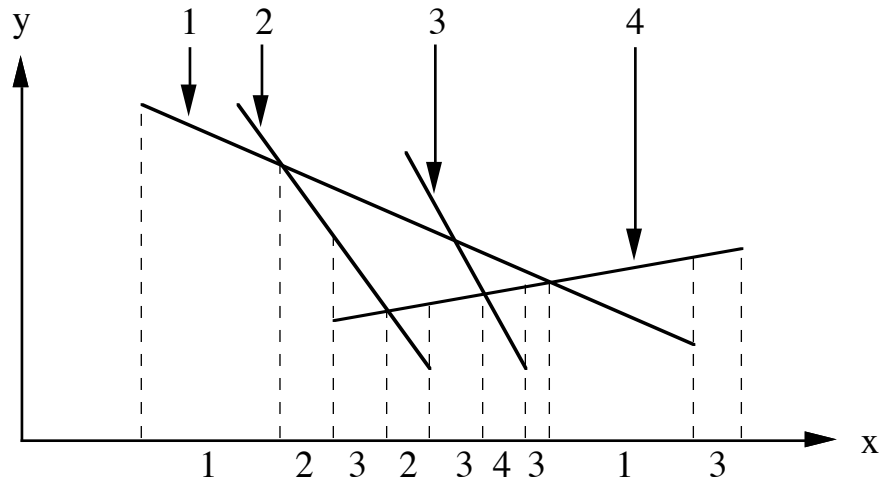

Figure 24.16: The Davenport-Schinzel sequence associated with a configuration of segments.

A geometric configuration gives rise to a sequence $u_1, u_2, \ldots, u_m$ with the following properties:

1. $1 \le u_i \le n$ for $1 \le i \le m$ (numbers identify line segments).

2. $u_i \ne u_{i+1}$ for $1 \le i \le m - 1$ (no two consecutive numbers are equal).

3. There are no five indices $1 \le a < b < c < d < e \le m$ such that $u_a = u_c = u_e = r$ and $u_b = u_d = s$, $r \ne s$. This condition captures the geometric properties of two intersecting straight lines: If we ever see r, s, r, s (possibly separated), we will never see r again, as this would imply that r and s intersect more than once (Fig. 24.17).

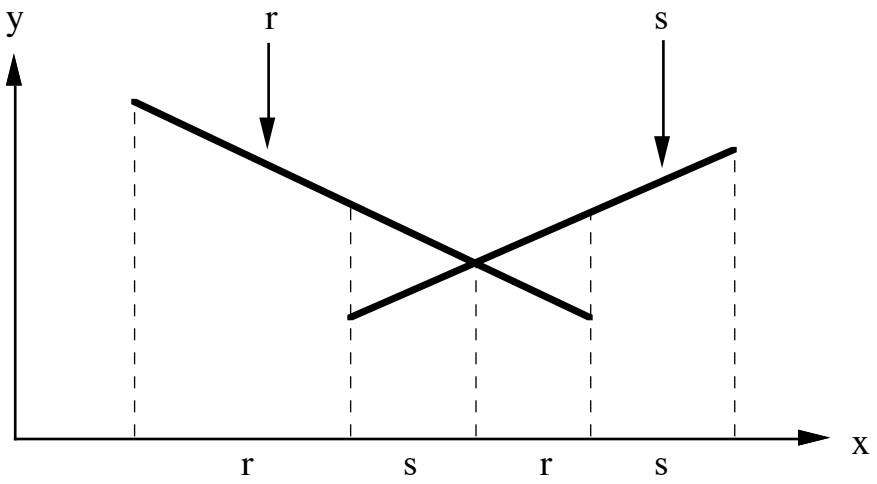

Figure 24.17: The subsequence r, s, r, s excludes further occurrences of r.

Example

The sequence for the example above that shows $m \geq 5\,n - 8$ is
1, 2, 1, 3, 1, … , 1, n–1, 1, n–1, n–2, n–3, … , 3, 2, n, 2, n, 3, n, … , n, n–2, n, n–1, n.

Sequences with the properties 1 to 3, called *Davenport-Schinzel sequences*, have been studied in the context of linear differential equations. The maximal length of a Davenport-Schinzel sequence is $k \cdot n \cdot \alpha(n)$, where k is a constant and $\alpha(n)$ is the inverse of Ackermann's function (see Chapter 15, Exercise 2) [HS 86]. With increasing n, $\alpha(n)$ approaches infinity, albeit very slowly. This dampens the hope for a linear upper bound for the visibility problem, but does not yet disprove the conjecture. For the latter, we need an inverse: For any given Davenport-Schinzel sequence there exists a corresponding geometric configuration which yields this sequence. An explicit construction is given in [WS 88]. This establishes an isomorphism between the two-dimensional visibility problem and the Davenport-Schinzel sequences, and shows that the size of the output of the two-dimensional visibility problem can be superlinear - a result that challenges our geometric intuition.

Exercises

1. Given a set of points S, prove that the pair of points farthest from each other must be vertices of the convex hull H(S).

2. Assume a model of computation in which the operations addition, multiplication, and comparison are available at unit cost. Prove that in such a model $\Omega(n \cdot \log n)$ is a lower bound for computing, in order, the vertices of the convex hull H(S) of a set S of n points. *Hint:* Show that every algorithm which computes the convex hull of n given points can be used to sort n numbers.

3. Complete the second algorithm for the point-in-polygon test in Section 24.3 which computes the crossing number of the polygon G around point P by adressing the special cases that arise when the semi-infinite ray R emanating from P intersects a vertex of G or overlaps an edge of G.

4. Consider an arbitrary (not necessarily simple) polygon G (Fig. 24.18). Provide an interpretation for the winding number w(G, P) of G around an arbitrary point P not on G, and prove that $w(G, P) / 2 \cdot \pi$ of P is always equal to the crossing number of P with respect to any ray R emanating from P.

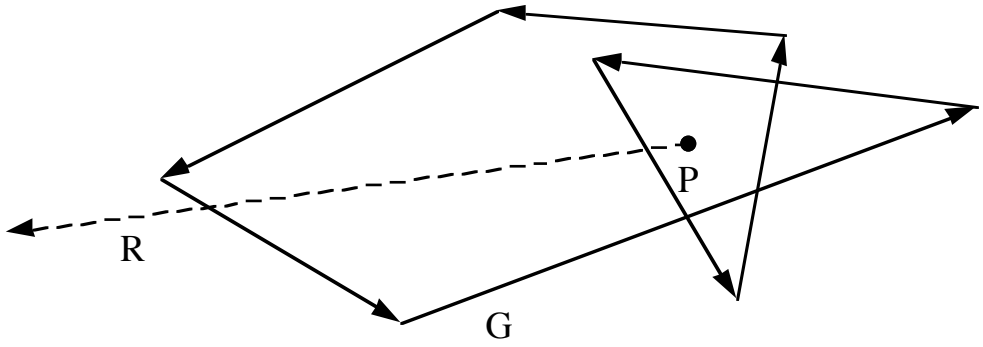

Figure 24.18: Winding number and crossing number
of a polygon G with respect to P.

5. Design an algorithm that computes the area of an n-vertex simple, but not necessarily convex polygon in $\Theta(n)$ time.

6. We consider the problem of computing the intersection of two convex polygons which are given by their lists of vertices in cyclic order.
(a) Show that the intersection is again a convex polygon.
(b) Design an algorithm that computes the intersection. What is the time complexity of your algorithm?

7. *Intersection test for line L and [convex] polygon Q* If an (infinitely extended) line L intersects a polygon Q, it must intersect one of Q's edges. Thus a test for intersection of a given line L with a polygon can be reduced to repeated test of L for intersection with [some of] Q's edges.

(a) Prove that, in general, a test for line-polygon intersection must check at least $n - 2$ of Q's edges. *Hint:* Use an adversary argument. If two edges remain unchecked, they could be moved so as to invalidate the answer.

(b) Design a test that works in time $O(\log n)$ for decoding whether a line L intersects a convex polygon Q.

8. Divide-and-conquer algorithms may divide the space in which the data is embedded, rather than the set of data (the set of lines). Describe an algorithm for computing the sequence of visible segments that partitions the space recursively into vertical stripes, until each stripe is "simple enough"; describe how you choose the boundaries of the stripes; state advantages and disadvantages of this algorithm as compared to the one described in Section 24.4. Analyze the asymptotic time complexity of this algorithm.