TR 85-008

TRIANGULATION ALGORITHMS FOR SIMPLE, CLOSED, NOT NECESSARILY CONVEX, POLYGONS IN THE PLANE

JOHN H. HALTON

The University of North Carolina
Department of Computer Science
New West Hall 035 A
Chapel Hill, NC 27514 USA

MARCH 31, 1985

TRIANGULATION ALGORITHMS FOR SIMPLE, CLOSED, NOT NECESSARILY CONVEX POLYGONS IN THE PLANE

John H. Halton

Professor of Computer Science
The University of North Carolina
Chapel Hill, NC 27514 USA

ABSTRACT

This paper presents three algorithms for dissecting the interior of an arbitrary simple, closed, not necessarily convex polygon in the plane. The simplest algorithm is shown to have time complexity $O(n^3)$ and the two others, derived from it, while more complicated, have complexity $O(n^2)$. The triangulations obtained are *economical*, in the sense that the number of triangles obtained is as small as possible; but no effort is made to reduce the diameters of the component triangles.

Keywords: Algorithms; data structures; triangulation; polygons; graphics {computers; techniques; performance analysis; complexity}

March 1985

Triangulation Algorithms for Simple, Closed, Not Necessarily Convex Polygons in the Plane

John H. Halton, Chapel Hill, North Carolina, USA

1. Introduction

The problem is a classical one. We are given n points $P_1, P_2, \ldots, P_n$ in the Euclidean plane and interpret other indices *modulo* n, so that $P_0 = P_n$, $P_1 = P_{n+1}$, and in general $P_j = P_{j+kn}$. The points are supposed to be so ordered that

$$\mathfrak{P} \equiv P_1 P_2 \ldots P_n, \tag{1}$$

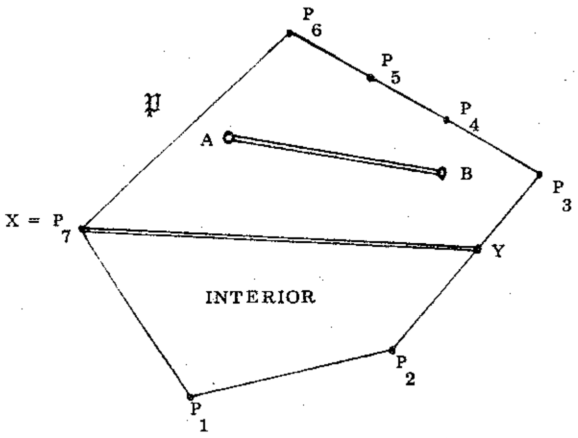
the *polygon* with vertices P_j $(j = 1, 2, \ldots, n)$, consisting of the n line-segments $P_j P_{j+1}$ $(j = 1, 2, \ldots, n)$, is *simple* (i.e., all the P_j are distinct and no two sides $P_i P_{i+1}$ and $P_j P_{j+1}$ have points in common, except when $i = j$ [of course] or $i = j - 1$ [only P_j in common] or $i = j + 1$ [only P_i in common]). In common parlance, we would say that a simple polygon *does not cross itself*. We wish to identify a set of *triangles*, whose interiors are *disjoint*, and whose union is the *interior and boundary* of the polygon $\mathfrak{P}$. This process is referred to as the *triangulation* of the polygon.

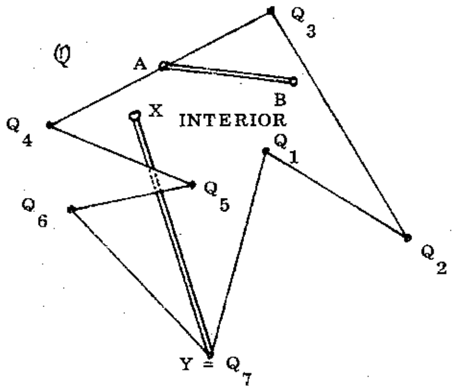
The removal of a simple polygon from the plane leaves exactly two connected open sets, called its *interior* $I_{\mathfrak{P}}$ and its *exterior* $E_{\mathfrak{P}}$, with the interior identified in that it is *bounded* (i.e., there is a circle in the plane which entirely contains $I_{\mathfrak{P}}$). We re-number the vertices (if necessary) so that, as we traverse the polygon $P_1 P_2 \ldots P_n$, the interior is on the *left*.

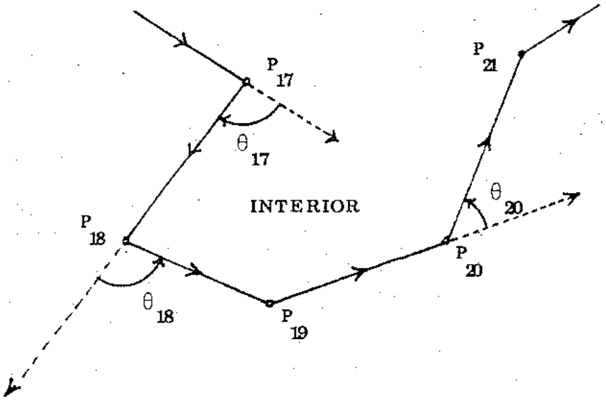
Vertices may be divided into three mutually-exclusive classes, according to the angle by which one turns from the direction of $P_{j-1}P_j$ to that of $P_j P_{j+1}$. If this angle θ_j satisfies $0 < \theta_j < \pi$, we say that P_j is a *convex* vertex; if the angle satisfies $-\pi < \theta_j < 0$, we call P_j a *re-entrant* vertex; and if $\theta_j = 0$, P_j is called *redundant* or *collinear* (and will later be eliminated). If the polygon $\mathfrak{P}$ is such that the line-segment joining any two points in its interior or boundary is entirely contained in the union of $\mathfrak{P}$ and $I_{\mathfrak{P}}$, we shall say that

$\mathfrak{P}$ is a *convex* polygon. We shall not limit ourselves to this simple case.

Figure 1.







$\mathfrak{P} = P_1P_2P_3P_4P_5P_6P_7$ is a convex polygon [or heptagon, since $n = 7$]; line-segments such as AB or XY are entirely in or on $\mathfrak{P}$. On the contrary, $\mathfrak{Q} = Q_1Q_2Q_3Q_4Q_5Q_6Q_7$ is not convex; while the segment AB is in or on $\mathfrak{Q}$, segments such as XY are not (the dotted portion is exterior to the heptagon). All vertices of $\mathfrak{P}$ are convex, as are Q_2, Q_3, Q_4, Q_6, and Q_7; but Q_1 and Q_5 are re-entrant vertices of $\mathfrak{Q}$. In the third illustration, P_{17} is re-entrant ($-\pi < \theta_{17} < 0$), while P_{18} and P_{20} are convex ($0 < \theta_{18} < \pi$ and $0 < \theta_{20} < \pi$). What are P_{19} and P_{21}?

Again in common parlance, if the polygon is traversed as defined above, then *one turns left at a convex vertex* (i.e., *towards the interior*) and *turns right at a re-entrant vertex.*

We seek a *triangulation algorithm* which:

(i) always yields a complete triangulation in a finite number of steps;

(ii) is as fast as possible (i.e., each step is fast, and the total number of steps required is least);

(iii) is as economical as possible (i.e., the final set of triangles has no more than $n - 2$ members —— less than $n - 2$ when certain vertices are collinear, as in the polygon $\mathfrak{P}$ (vertices P_4 and P_5) in Figure 1).

In some cases, a fourth criterion is used also: it is sought to increase the minimum internal angle of the triangles as much as possible, so as to avoid long-thin triangles, which are not desirable for computational triangulations. We shall not consider this criterion here.

Two workable algorithms will be described here. Each has some merits. Both are adequately fast, as will be demonstrated.

2. Preliminary Results

Denote the coordinates of each vertex P_j by $(x_j, y_j, 0)$.

LEMMA 1. *The passage from* P_{j-1} *through* P_j *to* P_{j+1} *is a turn to the left if*

$$x_j(y_{j+1} - y_{j-1}) - y_j(x_{j+1} - x_{j-1}) > x_{j-1}y_{j+1} - x_{j+1}y_{j-1}. \tag{2}$$

Proof. [Proofs will be enclosed in ⟦...⟧ from now on.]

⟦The vector $P_{j-1}P_j = (x_j - x_{j-1}, y_j - y_{j-1}, 0)$ and the vector $P_jP_{j+1} = (x_{j+1} - x_j, y_{j+1} - y_j, 0)$; so that the vector [or cross] product

$$P_{j-1}P_j \wedge P_jP_{j+1} = (0, 0, Z), \tag{3}$$

where

$$Z = (x_j - x_{j-1})(y_{j+1} - y_j) - (x_{j+1} - x_j)(y_j - y_{j-1}), \tag{4}$$

and this quantity will have the same sign as $\sin\theta_j$, where θ_j is the angle defined earlier, from the vector $P_{j-1}P_j$ to the vector P_jP_{j+1}. Thus, the turn is to the left $(0 < \theta_j < \pi)$ if $Z > 0$. It remains to rearrange terms to give the inequality (2).⟧

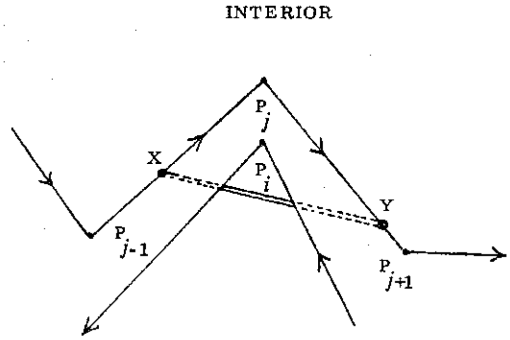
The importance of this result is that it is an easy matter to determine whether there is a turn to the left or to the right at any given vertex.

COROLLARY. *The passage through* P_j *is a turn to the right if '>' is replaced by '<' in* (2).

LEMMA 2. *A convex polygon has only convex vertices.*

⟦Define $\mathfrak{P}$ as in (1). Suppose it is *convex*; then any line-segment joining two points *in or on* $\mathfrak{P}$ [we use this phrase to indicate that the points are either in $\mathfrak{P}$ or in $I_{\mathfrak{P}}$] is *entirely* in or on $\mathfrak{P}$. Let P_j be a *re-entrant vertex* of $\mathfrak{P}$; then there is a *right* turn from $P_{j-1}P_j$ to P_jP_{j+1}, with the interior of $\mathfrak{P}$ on the *left*. It follows that any segment XY, with X interior to the segment $P_{j-1}P_j$ and Y interior to P_jP_{j+1} crosses the *exterior* $E_{\mathfrak{P}}$ of $\mathfrak{P}$ (at least next to X and to Y; there could be vertices of $\mathfrak{P}$ in the triangle P_jXY). This is illustrated in Figure 2, where the exterior portion of XY is shown dotted (as in Figure 1 [$\mathfrak{Q}$]). This result contradicts the definition of convexity for the polygon $\mathfrak{P}$. Therefore there ***cannot*** be any re-entrant vertex of a convex polygon.⟧

Figure 2.



LEMMA 3. *A polygon with only convex vertices is convex.*

⟦Define $\mathfrak{P}$ as in (1). Suppose it is *not* convex; then there is a line-segment joining two points X and Y which are in or on $\mathfrak{P}$, such that *not all* of the segment XY is in or on $\mathfrak{P}$. Therefore we can find a point C between X and Y on the segment XY, such that C is exterior to $\mathfrak{P}$. Since X and Y are interior and C is exterior, XY must *cross* the polygon an even number of times (at least twice). Let A and B be the nearest intersections of XY and $\mathfrak{P}$ on either side of C (see Figure 3). Then let $AP_iP_{i+1} \ldots P_{j-1}P_jB$ be the (properly directed) polygonal sub-arc of $\mathfrak{P}$ from A to B. The linear segment ACB must be to the right of the vector $P_{i-1}P_i$, since C is exterior. Thus, the net turn from $P_{i-1}P_i$ to P_jP_{j+1} must be to the right; and therefore not all angles θ_i, θ_{i+1}, ..., θ_{j-1}, θ_j can be positive; whence at least one of the vertices P_i, P_{i+1}, ..., P_{j-1}, P_j is re-entrant. This contradicts our hypothesis; so $\mathfrak{P}$ must be convex.⟧

Figure 3.

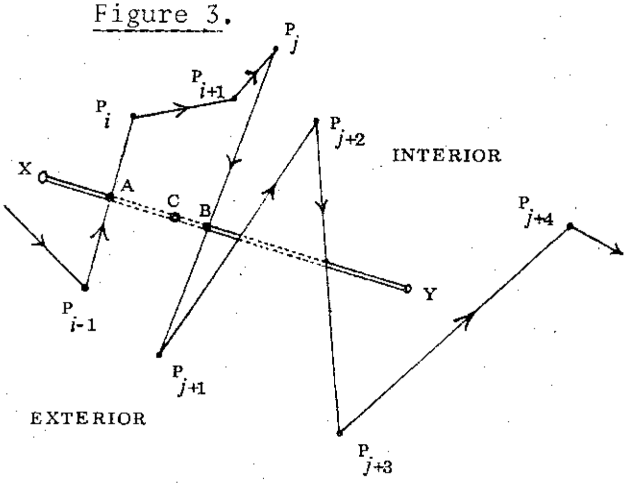
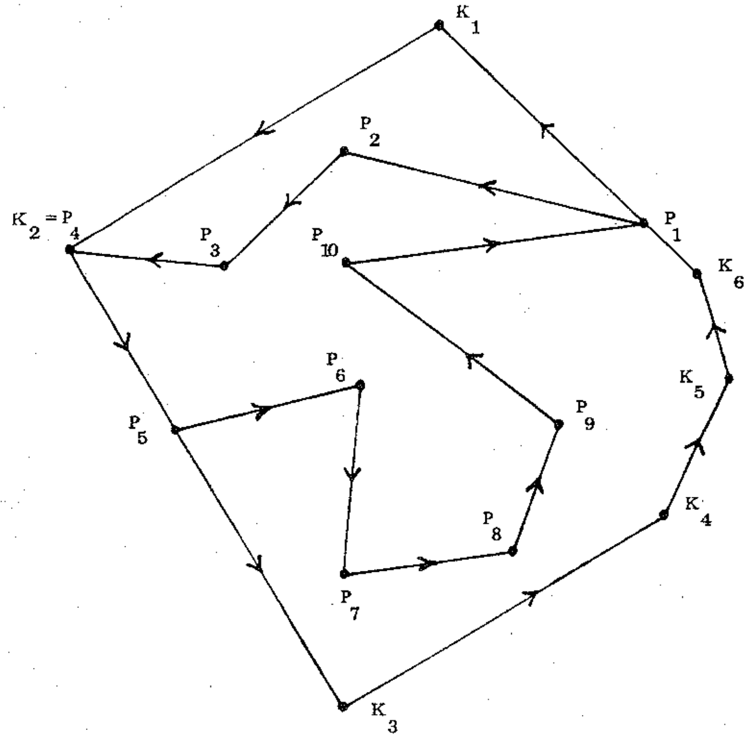


Figure 4.



LEMMA 4. *Given a convex polygon $\mathfrak{K}$ and a general polygon $\mathfrak{P}$ entirely in or on $\mathfrak{K}$, if a vertex P_j of $\mathfrak{P}$ lies on $\mathfrak{K}$, then P_j is a convex vertex of $\mathfrak{P}$.*

⟦The situation is illustrated in Figure 4, where $\mathfrak{K}$ is a convex hexagon and $\mathfrak{P}$ is a decagon; with P_1, P_4, and P_5 lying on $\mathfrak{K}$. If P_j lies on $\mathfrak{K}$, then it is either coincident with a vertex of $\mathfrak{K}$ (like P_4 in Figure 4) or is interior to a side of $\mathfrak{K}$ (like P_5 and P_1 in Figure 4). In either case, we can uniquely identify vertices K_r and K_s, such that K_rP_j and P_jK_s are parts of sides of $\mathfrak{K}$ (for P_1 in Figure 4, we have K_6 and K_1; for P_4, K_1 and K_3; and for P_5, K_2 and K_3), there being no other vertex of $\mathfrak{K}$ between K_r and P_j or between P_j and K_s, the direction being the same as that in which $\mathfrak{K}$ is traversed. Since P_{j-1} and P_{j+1} are both in or on $\mathfrak{K}$, the angle $\angle P_{j-1}P_jP_{j+1}$ is contained in the angle $\angle K_rP_jK_s$ and is therefore of the same sign, namely, positive [$\mathfrak{K}$ is convex; so, by Lemma 2, its vertices are convex, while points in its straight sides subtend angles of π (= 180°); and $\mathfrak{P}$ and $\mathfrak{K}$ are traversed in the same (counterclockwise) direction]. Thus, P_j is a convex vertex.⟧

COROLLARY. *If the vertices of a simple, closed polygon $\mathfrak{P}$ have coordinates*

$$P_j = (x_j, y_j, 0) \quad (j = 1, 2, \ldots, n), \tag{5}$$

then the vertices satisfying

$$x_i = \min_j x_j \quad \text{or} \quad x_i = \max_j x_j \quad \text{or} \quad y_i = \min_j y_j \quad \text{or} \quad y_i = \max_j y_j, \tag{6}$$

are all convex vertices.

⟦The notation is that used in proving Lemma 1. The rectangle $\mathfrak{R}$ with vertices

$$(\min_j x_j,\ \min_j y_j,\ 0),\ (\max_j x_j,\ \min_j y_j,\ 0),\ (\max_j x_j,\ \max_j y_j,\ 0),\ (\min_j x_j,\ \max_j y_j,\ 0) \tag{7}$$

is a convex polygon containing all of $\mathfrak{P}$. Thus, by Lemma 4, vertices satisfying any of the equations (6) lie on the sides of the rectangle and so must be convex vertices.⟧

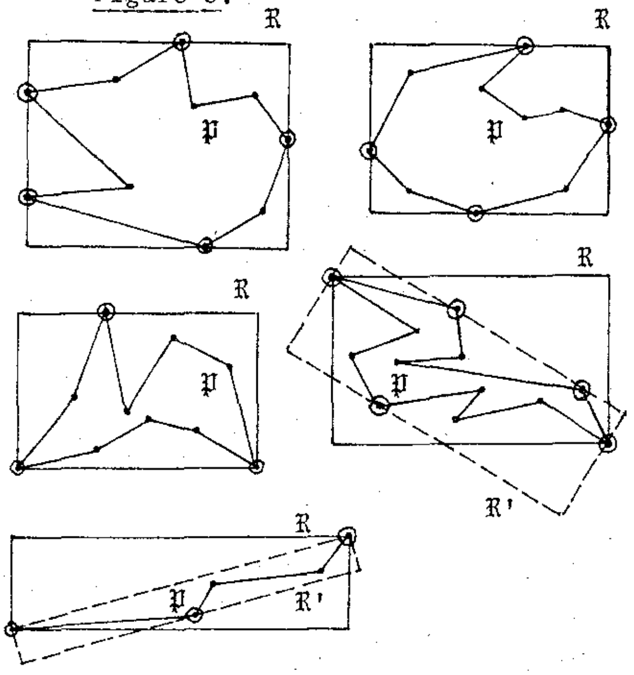
<u>LEMMA 5</u>. *Every polygon with a non-empty interior must have at least three convex vertices.*

⟦Polygons with one or two vertices have no interior. Polygons with all their vertices collinear have no interior. Thus, for a polygon to have a non-empty interior, $n \geqslant 3$. If the interior $I_{\mathfrak{P}}$ of $\mathfrak{P}$ is non-empty, it is defined as an *open* set; that is, every point X of $I_{\mathfrak{P}}$ is surrounded by a circular neighborhood entirely contained in $I_{\mathfrak{P}}$ (such a neighborhood is the set of all points Y distant less than some radius $\rho > 0$ from X); and it follows that

$$\min_j x_j < \max_j x_j \quad \text{and} \quad \min_j y_j < \max_j y_j. \tag{8}$$

Therefore the rectangle $\mathfrak{R}$ defined above, with vertices (7), has sides of positive length (opposite sides are distinct). It takes at least two distinct vertices of $\mathfrak{P}$ on the boundary of the rectangle to define it (see Figure 5). Now either there are three such vertices on the rectangle, and we are through; or there are only two. In the latter case, rotate the coordinate axes of x and y about the z-axis so that the line through the two extreme vertices of $\mathfrak{P}$ is parallel to the new x'-axis. Make a new rectangle $\mathfrak{R}'$ as before, in terms of the new coordinates x' and y'; then, since the interior of $\mathfrak{P}$ is non-empty, at least one more vertex of $\mathfrak{P}$ is on $\mathfrak{R}'$. (The two extreme vertices from $\mathfrak{R}$ are extremes of x' in $\mathfrak{R}'$.)⟧

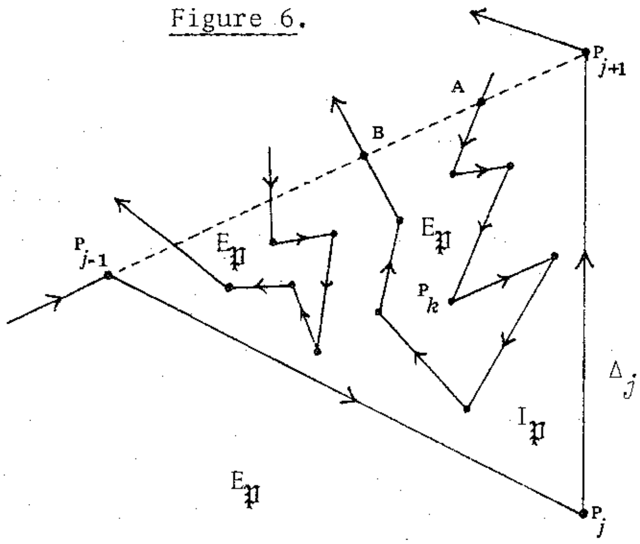
<u>Figure 5.</u>



We shall call the triangle $P_{j-1}P_jP_{j+1}$ formed by three consecutive vertices of a polygon $\mathfrak{P}$ the *triad* Δ_j at P_j. It is a *convex triad* if P_j is a convex vertex of $\mathfrak{P}$.

LEMMA 6. *If* $\Delta_j = P_{j-1}P_jP_{j+1}$ *is a convex triad of a polygon* $\mathfrak{P}$, *and if* Δ_j *contains* *any* *vertex of* $\mathfrak{P}$, *then it must contain at least one* *re-entrant* *vertex of* $\mathfrak{P}$.

Figure 6.



⟦The situation is illustrated in Figure 6. The argument is similar to that used in proving Lemma 3. P_j is a convex vertex, with a *left* turn from $P_{j-1}P_j$ to P_jP_{j+1}. If a vertex P_k of $\mathfrak{P}$ is inside the triad Δ_j, it must bring with it a part of the exterior $E_{\mathfrak{P}}$ of $\mathfrak{P}$. Let A and B be adjacent points in which $\mathfrak{P}$ crosses the side $P_{j-1}P_{j+1}$ of the triad, traversed from A to B.† Then the side of $\mathfrak{P}$ through A must turn *right* in net effect, for the sub-arc of $\mathfrak{P}$ from A to B to reach B, which is on the *right* of the side of $\mathfrak{P}$ through A. It follows that at least one vertex of $\mathfrak{P}$ between A and B (and therefore inside Δ_j) must involve a right turn; that is, must be re-entrant. [† Here, we mean that P_k is part of the sub-arc of $\mathfrak{P}$ traversed from A to B entirely inside Δ_j.]⟧

LEMMA 7. *The vertex* P_k *lies inside the convex triad* Δ_j *if and only if*

$$x_j(y_k - y_{j-1}) - y_j(x_k - x_{j-1}) > x_{j-1}y_k - x_ky_{j-1}, \tag{9}$$

$$x_{j+1}(y_k - y_j) - y_{j+1}(x_k - x_j) > x_jy_k - x_ky_j, \tag{10}$$

and

$$x_{j-1}(y_k - y_{j+1}) - y_{j-1}(x_k - x_{j+1}) > x_{j+1}y_k - x_ky_{j+1}. \tag{11}$$

⟦We argue exactly as in proving Lemma 1. P_k is inside Δ_k if and only if it is to the *left* of each of the vectors $P_{j-1}P_j$, P_jP_{j+1}, and $P_{j+1}P_{j-1}$. Thus, we obtain the conditions (9), (10), and (11) by respectively replacing the indices $(j-1, j, j+1)$ by $(j-1, j, k)$, $(j, j+1, k)$, and $(j+1, j-1, k)$.⟧

As with Lemma 1, the importance of this result is in showing how it is quick and easy to determine inclusion of a vertex in a triad.

ALGORITHM 0. Given a simple, closed polygon 𝔓, defined by the coordinates of its vertices in the xy-plane (as in (5)), we *prepare it for triangulation* as follows: *for each vertex* P_j $(j = 1, 2, \ldots, n)$,

0.1. *compute the discriminant,*

$$\Gamma_j = x_j(y_{j+1} - y_{j-1}) - y_j(x_{j+1} - x_{j-1}) - x_{j-1}y_{j+1} + x_{j+1}y_{j-1}, \tag{12}$$

0.2. *if* $\Gamma_j > 0$, *enter the index* j *into a list* $\mathcal{A}$,

0.3. *if* $\Gamma_j < 0$, *enter the index* j *into a list* $\mathcal{B}$,

0.4. *if* $\Gamma_j = 0$, *omit the index* j, *reducing higher indices by one,*

0.5. *beginning with* $h = 1$ *and* $M = x_1$, *if* $x_j > M$, *put* $h = j$ *and* $M = x_j$, *if* $x_j = M$ *and* $y_j > y_h$, *put* $h = j$, *otherwise do nothing (note* Γ_h*);*

0.6. *if* $\Gamma_h < 0$, *re-number the vertices in lists* $\mathcal{A}$ *and* $\mathcal{B}$ *so that* P_i *becomes* P_{N-i+1}, *where* N *is the number of vertices remaining (last index value entered in one of the two lists), and interchange the lists* $\mathcal{A}$ *and* $\mathcal{B}$.

Explanation. The discriminant Γ_j is just the z-component Z of the vector product (3) (compare (4)). Thus, by Lemma 1, $\Gamma_j = 0$ when the vertices P_{j-1}, P_j, and P_{j+1} are collinear, so that P_j is *redundant*; in this case, P_j is *omitted* in 0.4. If $\Gamma_j > 0$, the polygon makes a *left turn* at P_j, while if $\Gamma_j < 0$, it makes a *right turn* there; hence the lists $\mathcal{A}$ and $\mathcal{B}$ generated by 0.2 and 0.3 are lists of left-turn and right-turn vertices. However, the *interior* of the polygon is not known yet. In 0.5, we progressively seek the vertex with maximum x-coordinate, and in case of a tie, that with maximum y-coordinate among them, and call it P_h. By Lemma 4, P_h is a *convex* vertex; thus, if 𝔓 is being traversed properly (by our convention), with its interior on the left, $\Gamma_j > 0$; otherwise, we reverse the numbering and the roles of the lists $\mathcal{A}$ and $\mathcal{B}$ in 0.6; so that $\mathcal{A}$ is the list of indices of *convex* vertices and $\mathcal{B}$ is the list of *re-entrant* vertices of 𝔓.

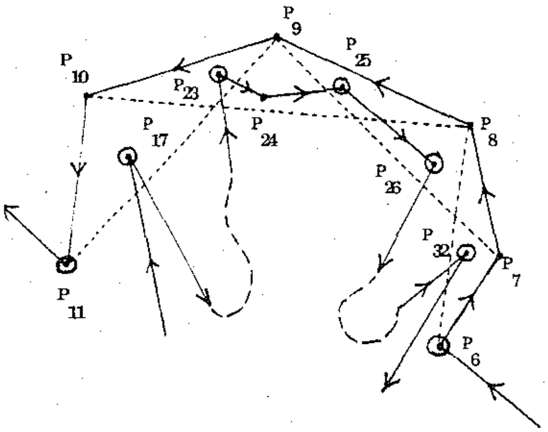
LEMMA 8. *No simple, closed polygon has an empty interior.*
⟦This case is, in fact excluded by the definitions given in the Introduction above. If all the vertices P_j $(j = 1, 2, \ldots, n)$ are distinct and no two sides P_iP_{i+1} and P_jP_{j+1} $(i, j = 1, 2, \ldots, n$, with $P_{n+1} = P_1)$ have points in common, unless $i = j$ or $i = j - 1$ (P_j only) or $i = j + 1$ (P_i only); then it is impossible for a polygon to have less than three vertices or for a polygonal arc (even a single side) to be traversed in both directions (or in the same direction) twice. The passage from any vertex P_i to another P_j in each direction must be along entirely disjoint paths; so the interior of the polygon must be non-empty. Therefore the provision of Lemma 5 is unnecessary.⟧

3. The First Algorithm

THEOREM 1. *Every simple, closed polygon* $\mathfrak{P}$ *has at least two convex triads* Δ_r *and* Δ_s *each containing no other vertex of* $\mathfrak{P}$.

⟦By Lemmas 5 and 8, $\mathfrak{P}$ must have at least three three convex vertices, and so at least three convex triads. By Lemmas 2, 3, and 6; first, if a convex triad contains no re-entrant vertex, then it contains no vertex of $\mathfrak{P}$ at all; and also, if $\mathfrak{P}$ is convex (or equivalently has only convex vertices) every triad is convex and contains no other vertices of $\mathfrak{P}$. Thus our theorem presents a problem only when $\mathfrak{P}$ is not convex. (i) Let $P_i, P_{i+1}, \ldots, P_{j-1}, P_j$ be consecutive convex vertices (as in Figure 7); then, if any of the corresponding convex triads $\Delta_i, \Delta_{i+1}, \ldots, \Delta_{j-1}, \Delta_j$ contains *no* other vertex, we are ahead by that triad. If, on the contrary, *each* of them contains at least one vertex (and so at least one re-entrant vertex), we must search elsewhere. Note that the polygonal arc of $\mathfrak{P}$ containing these re-entrant vertices may itself have one or more *empty* convex triads (which would put us ahead), but it does not have to. (In Figure 7, $i = 7$, $j = 10$, re-entrant vertices are ringed, and only Δ_{24} is explicitly shown as convex and empty.) In the worst case, from the point of view of our theorem, a string of convex

Figure 7.



vertices is flanked by a corresponding string of re-entrant vertices, as in Figure 8, with no branching, such as occurred in Figure 7. (In both figures, the dotted lines indicate the third sides of convex triads and ringed vertices are re-entrant.) It will be seen that this worst-case arrangement presents a "ribbon" of polygonal interior, if not quite parallel-sided, then bounded by polygonal arcs running alongside each other. The less-than-worst case is then either a broadening of the ribbon, which immediately yields empty convex triads, or a branching of the ribbon, which does not change our argument and indeed yields more empty convex triads than does the worst case. (ii) This worst-case ribbon construct is bounded on either side by polygonal sub-arcs of $\mathfrak{P}$, and, since $\mathfrak{P}$ is a simple, closed polygon, these two arcs must join at their ends. This can happen only in two ways, as illustrated in Figure 9, and the first is not permissible, since it separates $\mathfrak{P}$ into several disjoint loops. (We may think of AB and XY as polygonal "sides" of the ribbon, and then the first way is to join B to A and Y to X, completing an annular ribbon, while the second —— and only legitimate —— way is to join B to X and Y to A.) The question then reduces to whether the "ends" of the ribbon must have empty convex triads; and clearly this is so; for the point Q must lie in the triad ALM (with L a convex and Q a re-entrant vertex; or their roles are reversed) and either A is convex and the empty convex triad is YAL, or Y is convex and the empty convex triad is QYA (at least one of A and Y is convex, since otherwise A would be inside LQM, contradicting our assertion that Q is inside ALM). This is illustrated in Figure 10. (iii) Since a ribbon construct

Figure 8.

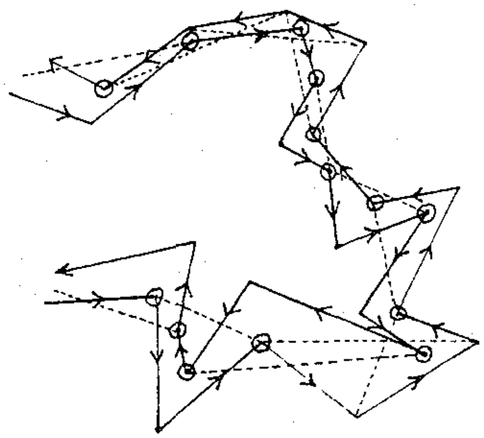
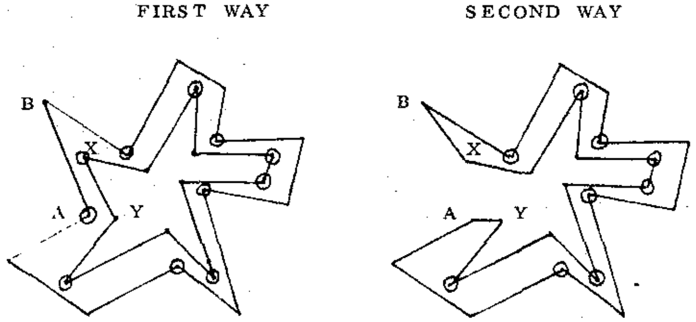


Figure 9.



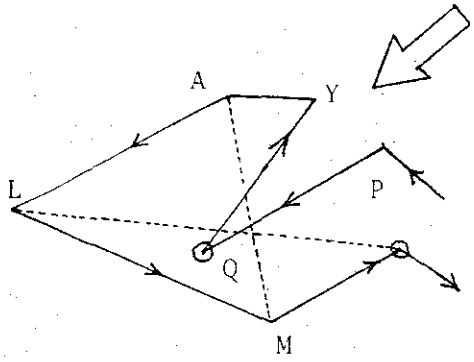
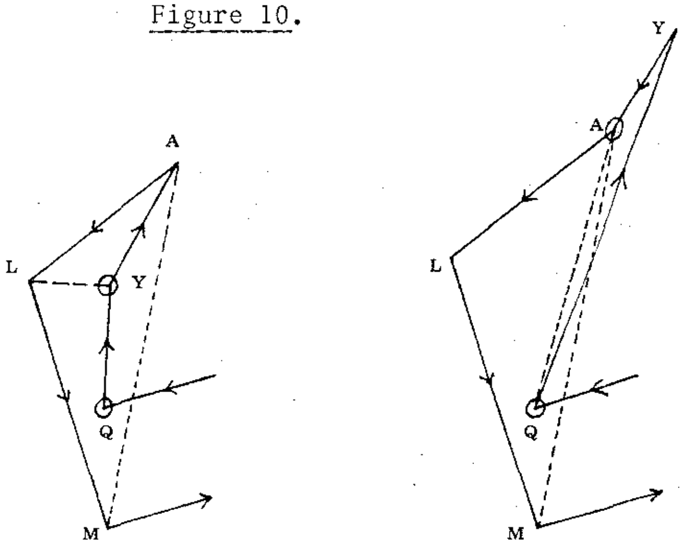
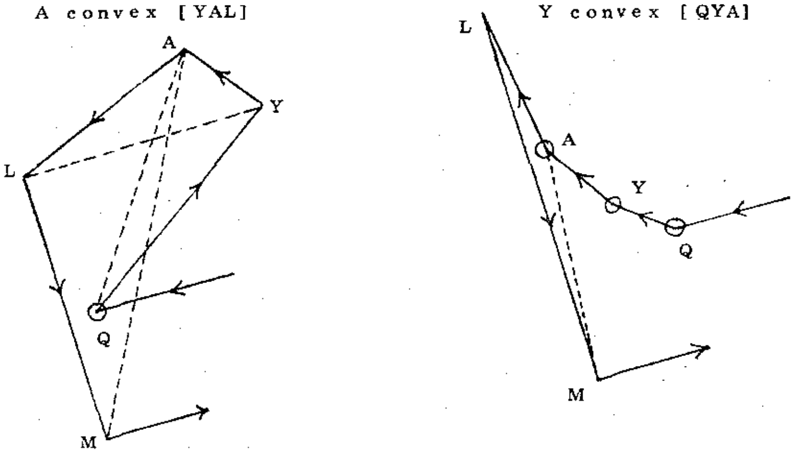


Figure 10.



A convex [YAL]

Y convex [QYA]



A and Y both convex
[YAL and QYA]

A and Y both re-entrant
[Q not in ALM]

such as we have defined above must have at least two ends (more, if there are branches), it follows that any simple, closed polygon must have at least two empty convex triads.]

ALGORITHM 1. We suppose that the simple, closed polygon $\mathfrak{P}$ has been prepared for triangulation by means of Algorithm 0, yielding a reduced set of vertices, properly ordered (so that the interior of $\mathfrak{P}$ is on the left as we traverse the polygon) and without redundant vertices with angle π (= 180°), and partitioned into lists $\mathcal{A}$ and $\mathcal{B}$, the first containing all convex vertices and the second all re-entrant vertices. Now proceed as follows: *treating $\mathcal{A}$ as a circular list* (i.e., last member is immediately followed by first), *for each successive vertex P_j whose index is in the list $\mathcal{A}$,*

1.1. *for every vertex P_k whose index k is in the list $\mathcal{B}$, compute the inequalities* (9), (10), *and* (11) *of Lemma 7,*

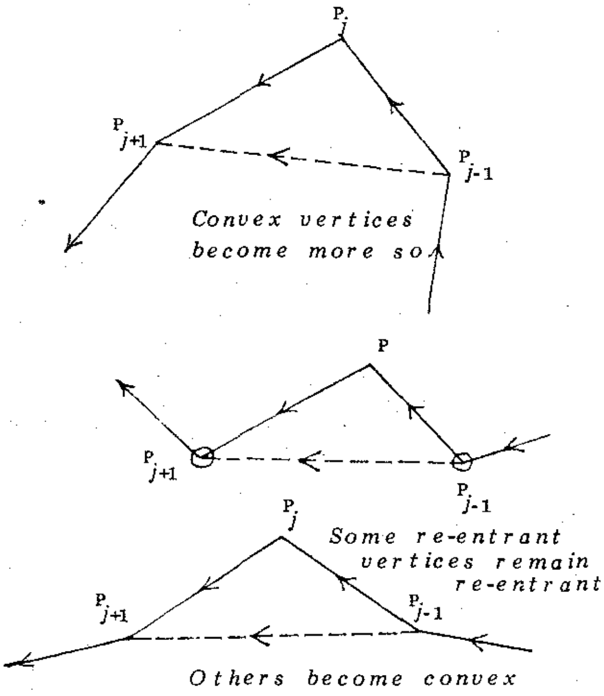
1.2. *if all three inequalities hold for any re-entrant vertex P_k from list $\mathcal{B}$, go on to the next convex vertex from list $\mathcal{A}$* (i.e., iterate to 1.1),

1.3. *if one or more of the inequalities fail, for every P_k from list $\mathcal{B}$, then* (a) *put the triad $\Delta_j = P_{j-1}P_jP_{j+1}$ into a list $\mathcal{C}$ of empty convex triads,* (b) *remove the index of P_j from list $\mathcal{A}$,* (c) *test Γ_{j-1} and Γ_{j+1} as in* 0.1-0.4 *and adjust lists $\mathcal{A}$ and $\mathcal{B}$ accordingly, and then go on to the next vertex from list $\mathcal{A}$;*

1.4. *continue until list $\mathcal{A}$ has only two indices in it.*

Explanation. By Lemma 7, the vertex P_k lies inside the triad Δ_j if and only if all three inequalities tested in step 1.1 hold. We seek empty *convex* triads; so we need only consider j in list $\mathcal{A}$. By Lemma 6, a convex triad will be empty if it contains no *re-entrant* vertex; so we need only test k in list $\mathcal{B}$. As soon as we find a re-entrant vertex in a convex triad, we may go on to the next convex triad; hence 1.2. As stated in 1.3, if all re-entrant vertices fail the test, the convex triad being tested is indeed empty. By Lemmas 5 and 8, the list $\mathcal{A}$ will not be initially empty. By Theorem 1, each pass of the list will yield at least two empty convex triads, so that the list will be reduced at each iterative pass by at least 2; but then as many as four indices may be transferred from list $\mathcal{B}$ to list $\mathcal{A}$. (Re-entrant vertices may become convex by removal of a triad's apex, but the reverse cannot happen. See Figure 11.) Nevertheless, each time a triad is found and put in the list $\mathcal{C}$, at least one vertex is removed (if a flanking vertex becomes redundant, by 0.4, when an apex vertex is removed, it too is removed) from the union of the two lists $\mathcal{A}$ and $\mathcal{B}$. Thus the process will eventually terminate (since, when the list $\mathcal{B}$ is empty, all triads become empty and convex (by Lemma 6).

Figure 11.



THEOREM 2. *Algorithm 1* (i) *always yields a complete triangulation in a finite number of steps;* (ii) *takes* 9 *arithmetic operations* (additions, subtractions, and multiplications) *to compute a discriminant* [of the form (12)], *altogether* $9n$ *arithmetic operations and* $O(n)$ *other operations to execute the preparatory Algorithm 0, and less than*

$$\frac{9}{4}\left(n^3 - \frac{3}{2}n^2 + 7n - \frac{69}{2}\right) = O(n^3) \qquad (13)$$

arithmetic operations and $O(n^3)$ *other operations to perform;* (iii) *is as economical as possible* (i.e., yields at most $n - 2$ triads). ⟦ (i) This result is indicated in the Explanation above; indeed, when (ii) is proved, we get (i) as a conclusion. (ii) Examination of (12) verifies that it takes 9 arithmetic operations ["a.o." hereinafter] to compute a discriminant.

Suppose that $\mathcal{A}$ has p_r indices of convex vertices listed and that $\mathcal{B}$ has q_r indices of re-entrant vertices listed, after r triads have been put in $\mathcal{C}$, Then

$$p_r + q_r = n_r \leqslant n_0 - r, \quad n_0 \leqslant n, \tag{14}$$

since Algorithm 0 may remove some redundant vertices (at 0.4), and whenever an empty convex triad has been identified and its apex removed, the same 0.4 test may lead to the removal of more redundant vertices. The inclusion test performed in 1.1-1.3 takes the checking of three discriminants [*none may be omitted*] and therefore takes 27 a.o. each time. Since, by Theorem 1, the list $\mathcal{A}$ must contain the indices of at least two empty convex triads, it takes at the very most $(p_r - 1)q_r$ inclusion tests to reach success (at 1.3). Given the total number n_r of vertices in $\mathcal{A}$ and $\mathcal{B}$ combined, we seek an upper bound for this expression. Now, $[(p + 1) - 1](q - 1) - (p - 1)q = pq - p - pq + q = q - p > 0$ when $q > p$; so that $(p - 1)q$ increases when p is increased, so long as $q > p$. Thus, $(p_r - 1)q_r$ is greatest, for given n_r, when

$$q_r = \left\lfloor \tfrac{1}{2}n_r \right\rfloor, \; p_r = \left\lceil \tfrac{1}{2}n_r \right\rceil, \tag{15}$$

where $\lfloor \ldots \rfloor$ and $\lceil \ldots \rceil$ respectively denote the "floor" and "roof" functions [the *integer infimum* and *supremum*]. Let us consider the worst case, when 0.4 never leads to the elimination of redundant vertices and success in finding an empty convex triad always takes the maximum number of failures first. Then we may put

$$n_r = n - r. \tag{16}$$

Further suppose that the working of 1.3(c) so balances p_r and q_r that (15) holds for all r. Then the total number of inclusion tests required by the algorithm is (for n even)

$$\frac{n}{2}\left(\frac{n}{2} - 1\right) + \left(\frac{n}{2} - 1\right)^2 + \left(\frac{n}{2} - 1\right)\left(\frac{n}{2} - 2\right) + \left(\frac{n}{2} - 2\right)^2 + \ldots + 3 \times 2 + 2 \times 2 + 2 \times 1$$

$$= \frac{1}{2}[(n - 1)(n - 2) + (n - 3)(n - 4) + \ldots + 7 \times 6 + 5 \times 4] + 2$$

$$= 2\sum_{h=1}^{n/2} \left(h - \frac{1}{2}\right)(h - 1) - 1 = \frac{1}{24}\Big[2n(n + 2)(n + 1) - 9n(n + 2) + 12n\Big] - 1$$

$$= \frac{1}{12}\left(n^3 - \frac{3}{2}n^2 - n - 12\right), \tag{17}$$

or (for n odd)

$$\left(\frac{n-1}{2}\right)^2 + \left(\frac{n-1}{2}\right)\left(\frac{n-1}{2} - 1\right) + \left(\frac{n-1}{2} - 1\right)^2 + \dots + 3 \times 2 + 2 \times 2 + 2 \times 1,$$

which is the same as before, with n replaced by $n - 1$ and the addition of the first term, $[\frac{1}{2}(n - 1)]^2$; this yields the sum, therefore,

$$\begin{aligned} &\frac{1}{12}[(n-1)^3 - \frac{3}{2}(n-1)^2 - (n-1) - 12] + \frac{1}{4}[(n-1)^2] \\ &\quad = \frac{1}{12}(n^3 - \frac{3}{2}n^2 - n - 12 + \frac{3}{2}), \end{aligned} \tag{18}$$

just slightly more (by $\frac{1}{8}$) than (17). The total number of a.o. required for the inclusion tests is thus not greater than

$$\frac{9}{4}(n^3 - \frac{3}{2}n^2 - n - \frac{21}{2}). \tag{19}$$

We must add to this the number of a.o. required to compute the two discriminants in 1.3(c), namely 18, for each success (except the last), for a total of $18(n - 3)$ a.o. The sum of this and (19) is (13). (iii) Finally, to see that the algorithm is *economical*, we need only observe that all triads put in list $\mathcal{C}$ have vertices of the polygon $\mathfrak{P}$ as their vertices, and in addition, any redundant vertices occurring along the way are omitted.⟧

4. The Second Algorithm

This algorithm was prompted by the feeling that much of the scanning of list $\mathcal{A}$ in Algorithm 1 might lead to failures (i.e., convex triads containing re-entrant vertices of the polygon $\mathfrak{P}$), when, in fact, empty convex triads could be found inside such non-empty triads, still with economy as defined above (i.e., triangulation does not generate additional vertices). It was felt that greater speed could thus be generated at the cost of rather more complex programming (without excessive computation).

First, we note that, if we write the *discriminant* Γ_j in (12) as

$$\begin{aligned} \Gamma_j = Z &= |P_{j-1}P_j \wedge P_jP_{j+1}| = |P_{j-1}P_j| \times \delta(P_{j+1}, P_{j-1}P_j) \\ &= \gamma[j - 1, j, j + 1], \end{aligned} \tag{20}$$

where $|\boldsymbol{x}|$ denotes the *magnitude* of the vector $\boldsymbol{x}$ and $\delta(C, AB)$ is the *distance* from the point C to the line AB, then the discriminants in the inequalities

(9), (10), (11) may be written as $\gamma[j-1, j, k]$, $\gamma[j, j+1, k]$, and $\gamma[j+1, j-1, k]$, respectively; and, indeed, the inequalities (2), (9), (10), and (11) then become

$$\gamma[j-1, j, j+1] > 0, \tag{21}$$

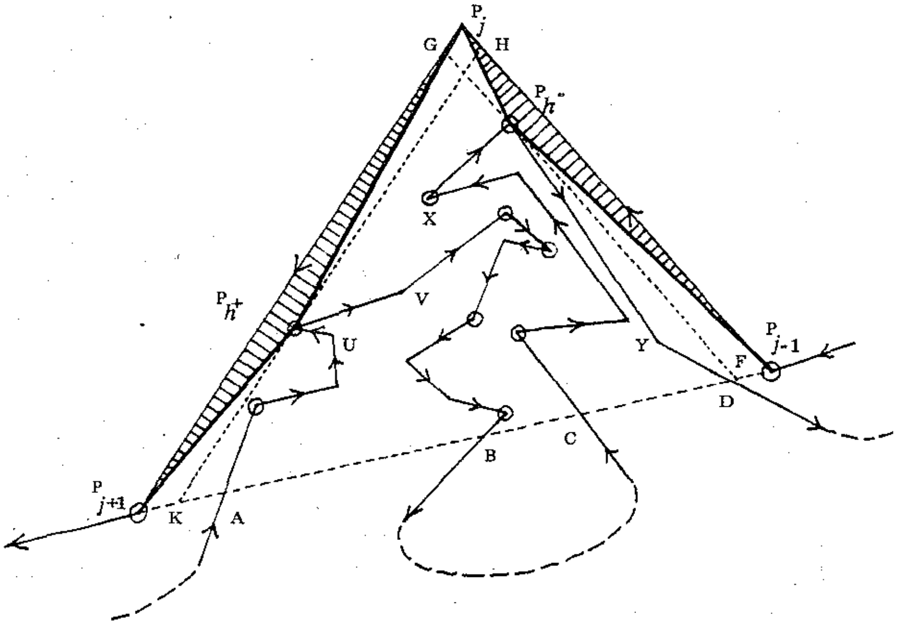
$$\gamma[j-1, j, k] > 0, \quad \gamma[j, j+1, k] > 0, \quad \gamma[j+1, j-1, k] > 0, \tag{22}$$

respectively. Thus, for fixed i and j, as k varies, $\gamma[i, j, k]$ is proportional to the *distance* from the point P_k to the line P_iP_j.

LEMMA 9. *If the convex triad* $\Delta_j = P_{j-1}P_jP_{j+1}$ *does contain certain vertices, then the vertices* P_{h^-} *and* P_{h^+} *among them, respectively having the least values of* $\gamma[j-1, j, h^-]$ *and* $\gamma[j, j+1, h^+]$, *are re-entrant, and the corresponding triads* $P_{j-1}P_jP_{h^-}$ *and* $P_jP_{j+1}P_{h^+}$ *are empty convex triads.*

⟦Since the discriminants $\gamma[j-1, j, k]$ and $\gamma[j, j+1, k]$ are respectively proportional to the distances from vertices P_k to the lines $P_{j-1}P_j$ and P_jP_{j+1}, we see that the vertices P_{h^-} and P_{h^+} are respectively the *closest* to these lines among vertices interior to the triad $P_{j-1}P_jP_{j+1}$. Figure 12 illustrates the situation: the polygon $\mathfrak{P}$ invades the interior of the triad in one or more polygonal sub-arcs (here, two: A...U...V...B and C...X...Y...D; entering the triangle (across the side $P_{j+1}P_{j-1}$) at A and again at C and emerging at B and again at D). P_{h^-} and P_{h^+} are defined as above; so that the dotted lines FG and HK, respectively parallel to $P_{j-1}P_j$ and $P_{j+1}P_j$ through P_{h^-} and P_{h^+} can have no vertices of $\mathfrak{P}$ interior to the triad and between the parallel pairs. It follows immediately that the shaded triads $P_{j-1}P_jP_{h^-}$ and $P_jP_{j+1}P_{h^+}$ are both empty and convex. Finally, the angles $\angle XP_{h^-}Y \leqslant \angle GP_{h^-}F = \pi$ and $\angle UP_{h^+}V \leqslant \angle KP_{h^+}H = \pi$, so that both P_{h^-} and P_{h^+} must be re-entrant, in view of the direction of traversal (marked in Figure 12 by arrow-heads).⟧

Figure 12.



ALGORITHM 2. We suppose, as for Algorithm 1, that the polygon has been prepared for triangulation by means of Algorithm 0, yielding lists $\mathcal{A}$ and $\mathcal{B}$, and that list $\mathcal{A}$ will be scanned, each convex triad Δ_j being tested for included re-entrant vertices P_k from list $\mathcal{B}$.

For each successive vertex P_j *of* $\mathfrak{P}$ *whose index* j *lies in list* $\mathcal{A}$,

2.1. [same as 1.1] *for every vertex* P_k *whose index* k *is in the list* $\mathcal{B}$, *compute the discriminants* $\gamma[j - 1, j, k]$, $\gamma[j, j + 1, k]$, *and* $\gamma[j + 1, j - 1, k]$ *of the inequalities* (9), (10), *and* (11) *of Lemma 7*,

2.2. *if all three discriminants are positive for any re-entrant vertex* P_k *from list* $\mathcal{B}$, *note the index* k *and the values of the discriminants* $\gamma[j - 1, j, k]$ *and* $\gamma[j, j + 1, k]$, *and* (a) *keep track of the indices of the least such discriminants, yielding the indices* h^- *and* h^+ *when all of list* $\mathcal{B}$ *has been traversed, then* (b) *put the triads* $P_{j-1}P_jP_{h^-}$ *and* $P_{j+1}P_jP_{h^+}$ *into list* $\mathcal{C}$, *and* (c) *recursively apply Algorithm 2 to each of the simple closed polygons thereby separated* [in Figure 12, these would be the polygons $\ldots P_{j-1}P_{h^-}Y\ldots$, $\ldots XP_{h^-}P_jP_{h^+}V\ldots$, and $\ldots UP_{h^+}P_{j+1}\ldots$, the dots denoting remaining connected vertices of $\mathfrak{P}$, in the same order as they appear in $\mathfrak{P}$],

2.3. [same as 1.3] *if one or more of the discriminants in* 2.1 *are non-positive, for every* P_k *from list* $\mathcal{B}$, *then* (a) *put the triad* $P_{j-1}P_jP_{j+1}$ *into the list* $\mathcal{C}$, (b) *remove the index of* P_j *from list* $\mathcal{B}$, (c) *test* $\Gamma_{j-1} = \gamma[j - 2, j - 1, j]$ *and* $\Gamma_{j+1} = \gamma[j, j + 1, j + 2]$ *as in* 0.1-0.4 *and adjust lists* $\mathcal{A}$ *and* $\mathcal{B}$ *accordingly, and then go on to the next vertex in list* $\mathcal{A}$;

2.4. *continue (with recursion, as needed) until each list* $\mathcal{A}$ *has only two indices in it.*

Explanation. 2.2 is the case when the triad does contain vertices of $\mathfrak{P}$; we now diverge from Algorithm 1 by recursively calling Algorithm 2 to each of the three sub-polygons into the original one is split, as explained above and illustrated in Figure 12. Lemma 9 ensures that the two triads added to list $\mathcal{C}$ in doing this always exist and are empty convex triads, as required. In 2.3, note that the discriminants $\gamma[j - 1, j, k]$ and $\gamma[j, j + 1, k]$ cannot vanish (because of the elimination of redundant vertices by 0.4); and if $\gamma[j + 1, j - 1, k] = 0$, then the triad Δ_j is empty and the vertex P_k is redundant in the residual polygon.

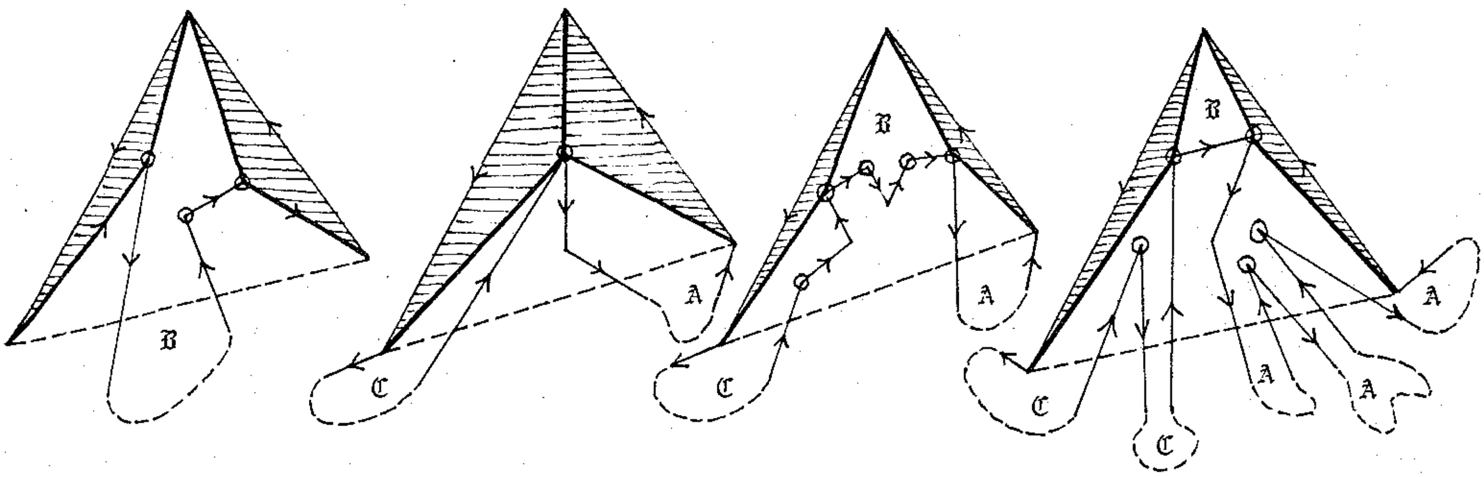
The analysis of this algorithm is a little more tricky than that of Algorithm 1, proving Theorem 2 (and, in particular, the a.o. count given by (13)). Again, we seek an upper bound for the number of a.o. required to perform the algorithm, and therefore look throughout at worst-case situations. The first postulate, therefore, would be that no redundant vertices are ever found, since these would shorten the work. The algorithm bifurcates at 2.2 and 2.3; so that, if 2.2 is more laborious, we should assume that this is the path taken every time; while, if 2.3 takes more a.o., we should similarly assume that this is the choice at every step.

First consider 2.3. Let lists $\mathcal{A}$ and $\mathcal{B}$ have p and q entries, respectively, with $p + q = n$. Then, if option 2.3 occurs every time, the first set of tests will lead to it; so that only q inclusion tests ($27q$ a.o.) need be computed [compare $(p - 1)q$ in the analysis of Algorithm 1]. The worst case is given by Lemma 5, with $p = 3$ and $q = n - 3$. This gives a count of $27(n - 3)$ a.o. We can now add-up the counts, much as before (at each step, we need 18 more a.o. to test the two discriminants Γ_{j-1} and Γ_{j+1}), to yield

$$27(n - 3) + 18 + 27(n - 4) + 18 + \dots + 18 + 27(2) + 18 + 27$$
$$= \frac{27}{2}(n - 2)(n - 3) + 18(n - 4) = \frac{27}{2}\left(n^2 - \frac{11}{3}n + \frac{2}{3}\right). \qquad (23)$$

Now suppose instead that 2.2 is chosen each time. We first note that, however a convex triad turns out to be non-empty, the situation is essentially the same. This is illustrated in Figure 13, which shows all possible arrangements, in essence. Any of the sub-polygons may be degenerate; but there cannot be more than three. The three sub-polygons are marked $\mathfrak{A}$, $\mathfrak{B}$, and $\mathfrak{C}$ in the figure, and they are easy to identify. In the first example, $\mathfrak{A}$ and $\mathfrak{C}$ disappear (each may degenerate separately), and in

Figure 13.



the second, $\mathfrak{B}$ is degenerate; the third example shows that, even when there is only one incursion into the interior of the triad, all three sub-polygons are generated; and the last example shows, on the one hand, that only three sub-polygons occur, even with many incursions, and, on the other hand, that the sub-polygon $\mathfrak{B}$ may reduce to a single triangle. Observe, too, that, if $\mathfrak{A}$, $\mathfrak{B}$, and $\mathfrak{C}$ respectively have n_1, n_2, and n_3 vertices, then

$$n_1 + n_2 + n_3 = n + 2, \tag{24}$$

because P_h- and P_h+ are counted twice. Having divided our polygon into three, we must make three new lists $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, and three new lists $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$ (the list $\mathcal{C}$ remains unique and comprehensive); to do this takes $9(n + 2)$ a.o. Thus, if we suppose that $f(n)$ denotes the upper bound we are seeking, for the number of a.o. required to perform the algorithm, it necessarily follows that

$$f(n) = \max_{n_1+n_2+n_3=n+2}[f(n_1) + f(n_2) + f(n_3)] + 9(n + 2). \tag{25}$$

If any $n_i = 3$, the corresponding lists are unnecessary; so $9(n + 2)$ becomes $9(n - 1)$ or $9(n - 4)$; and $f(3) = 0$; while we see by the construction that no $n_i \leqslant 2$. Taking these cases one-by-one, we see that, if $n_1 = n_2 = 3$,

$$f(n) = f(n - 4) + 9(n - 4) \tag{26}$$

has a solution of the form $an^2 + bn + c$; and the equation (26) shows that $an^2 + bn + c = an^2 - 8an + 16a + bn - 4b + c + 9n - 36$, or $8a = 9$, $16a - 4b = 36$; whence $a = 9/8$ and $b = 4a - 9 = -9/2$. Now, $f(4) = 27$ [there can only be one re-entrant vertex, by Lemma 5, and so one inclusion test suffices, and 2.2 yields empty convex triads only], so that $16a + 4b + c = 27$, whence $c = 27 - 18 + 18 = 27$, yielding the solution

$$f(n) = \frac{9}{8}(n^2 - 4n + 24). \tag{27}$$

Similarly, if $n_1 = 3$ and $n_2 = 4$, we get

$$f(n) = 27 + f(n - 5) + 9(n - 1) = f(n - 5) + 9(n + 2), \tag{28}$$

which will have a similar solution with $an^2 + bn + c = an^2 - 10an + 25a + bn - 5b + c + 9n + 18$, or $10a = 9$, $25a - 5b + 18 = 0$, and $16a + 4b + c = 27$; whence $a = 9/10$, $b = 81/10$, and $c = -22$, yielding the solution

$$f(n) = \frac{9}{10}(n^2 + 9n - 22). \tag{29}$$

Again, if $n_1 = n_2 = 4$, we get

$$f(n) = 54 + f(n - 6) + 9(n + 2) = f(n - 6) + 9(n + 8), \tag{30}$$

which will have a similar solution with $an^2 + bn + c = an^2 - 12an + 36a + bn - 6b + c + 9n + 72$, or $12a = 9$, $36a - 6b + 72 = 0$, and $16a + 4b + c = 27$; whence $a = 3/4$, $b = 33/2$, and $c = -51$, yielding the solution

$$f(n) = \frac{3}{4}(n^2 + 22n - 68). \tag{31}$$

In degenerate cases, such as are illustrated in Figure 13, there may be *no* sub-polygons at all [$n_1 = n_2 = n_3 = 0$ and $n = 4$]; or *one* sub-polygon [$n_2 = n_3 = 0$ and $n_1 = n - 2$], when we have

$$f(n) = f(n - 2) + 9(n - 2), \tag{32}$$

with the solution

$$f(n) = \frac{9}{4}(n^2 - 2n + 4); \tag{33}$$

or *two* sub-polygons [$n_3 = 0$], when we either have $n_2 = 3$ and $n_1 = n - 3$, or $n_2 = 4$ and $n_1 = n - 4$, the former yielding

$$f(n) = f(n - 3) + 9(n - 3), \tag{34}$$

with the solution

$$f(n) = \frac{3}{2}(n^2 - 3n + 14), \tag{35}$$

and the latter yielding

$$f(n) = f(n - 4) + 9(n + 3), \tag{36}$$

with the solution

$$f(n) = \frac{9}{8}(n^2 + 10n - 32). \tag{37}$$

These cases have all dealt in extremely skewed values of n_1, n_2, and n_3. It is apparent that $f(n)$ is monotonically increasing with n, and faster than linearly; and in such circumstances, it is advantageous to make the three n_i as equal as possible. To illustrate this, we may consider the case when we suppose the equation to be

$$f(n) = 3f(\frac{n + 2}{3}) + 9(n + 2). \tag{38}$$

In this case, we can see that the solution is asymptotic to some $kn \log n$; for then we get that $kn \log n \sim k(n + 2)[\log(n + 2) - \log 3] + 9n + 18$, which demonstrates the correctness of the general form, and yields that $k \log 3 = 9$, whence $k = 9/(\log 3)$. [A further term is then seen to be asymptotic to $k' \log n$, yielding that $kn \log n + k' \log n \sim k(n + 2)[\log(n + 2) - \log 3] + 9n + 18 + 3k'[\log(n + 2) - \log 3]$, whence $kn \log n + k' \log n - kn \log n - 2k \log n - kn \log(1 + \frac{2}{n}) - 2k \log(1 + \frac{2}{n}) + kn \log 3 + 2k \log 3 - 9n - 18 - 3k' \log n - 3k' \log(1 + \frac{2}{n}) + 3k' \log 3 \sim (k \log 3 - 9)n - 2(k' + k)\log n + O(1) \sim 0$. This gives $k = 9/(\log 3)$ and $k' = -k = -9/(\log 3)$. Further terms can be obtained similarly.] The point here is that making the n_i almost equal gives much faster execution of the algorithm; and since we are seeking worst-case situations, we are right [unfortunately!] in concentrating on the skewed cases considered earlier.

To make our conclusions rigorous, we need some results in *convexity*. Let us consider functions $f(x)$ defined for $x \geqslant 0$, such that $f(x) \geqslant 0$.

LEMMA 10. *If* $f(x)$ [as above] *is differentiable, monotonically increasing with* x*, faster than* x*, so that*

$$f'(x) \nearrow \infty \quad \text{as} \quad x \to \infty, \tag{39}$$

then f *is convex for* $x \geqslant 0$*; i.e., for all* $0 \leqslant x_1 \leqslant x_2$ *and all* $0 \leqslant \lambda \leqslant 1$,

$$\lambda f(x_1) + (1 - \lambda)f(x_2) \geqslant f(\lambda x_1 + (1 - \lambda)x_2). \tag{40}$$

⟦The inequality degenerates to an equality when $x_1 = x_2$ or $\lambda = 0$ or $\lambda = 1$, as is immediately obvious. Therefore fix $x_1 \geqslant 0$ and $0 < \lambda < 1$, and vary $x_2 \geqslant x_1$. By the Mean Value Theorem, there is a ξ such that $x_1 \leqslant \xi \leqslant x_2$ and

$$\begin{aligned}
&\lambda f(x_1) + (1 - \lambda)f(x_2) - f(\lambda x_1 + (1 - \lambda)x_2) \\
&\quad = \lambda f(x_1) + (1 - \lambda)f(x_1) - f(\lambda x_1 + (1 - \lambda)x_1) \\
&\qquad + (1 - \lambda)f'(\xi) - (1 - \lambda)f'(\lambda x_1 + (1 - \lambda)\xi) \\
&\quad = (1 - \lambda)[f'(\xi) - f'(\lambda x_1 + (1 - \lambda)\xi) \geqslant 0,
\end{aligned}$$

by (39), which states that f' is monotonically increasing, since (because $x_1 \leqslant \xi$) $\xi \geqslant \lambda x_1 + (1 - \lambda)\xi$. This proves (40).⟧

Note that the form of the function $f(n)$ in the discussion of Algorithm 2 is that specified by (39) above (since f increases at least as fast as the equations (26) - (38) suggest.

LEMMA 11. *If $f(x)$ is a convex function for $x \geqslant 0$, then*

$$F(x_1, x_2, \ldots, x_k) = \sum_{i=1}^{k} f(x_i) = f(x_1) + f(x_2) + \ldots + f(x_k) \tag{41}$$

is a convex function over all $x_i \geqslant 0$ ($i = 1, 2, \ldots, k$), and the same is true if we impose the condition [i.e., limit points $(x_1, x_2, \ldots, x_k)$ to the hyperplane]

$$x_1 + x_2 + \ldots + x_k = X. \tag{42}$$

⟦Since f is convex, we have that, for all $0 \leqslant x_1 \leqslant x_2$ and all $0 \leqslant \lambda \leqslant 1$, the inequality (40) holds. Taking k-dimensional vectors $(x_{11}, x_{12}, \ldots, x_{1k})$ and $(x_{21}, x_{22}, \ldots, x_{2k})$ in the positive orthant, we see that, by (40), for each $i = 1, 2, \ldots, k$,

$$\lambda f(x_{1i}) + (1 - \lambda)f(x_{2i}) \geqslant f(\lambda x_{1i} + (1 - \lambda)x_{2i}), \tag{43}$$

Summing these equations over all i, we get that

$$\begin{aligned}\lambda F(x_{11}, x_{12}, \ldots, x_{1k}) &+ (1 - \lambda)F(x_{21}, x_{22}, \ldots, x_{2k}) \\ &= \lambda \sum_{i=1}^{k} f(x_{1i}) + (1 - \lambda) \sum_{i=1}^{k} f(x_{2i}) = \sum_{i=1}^{k} [\lambda f(x_{1i}) + (1 - \lambda)f(x_{2i})] \\ &\geqslant \sum_{i=1}^{k} f(\lambda x_{1i} + (1 - \lambda)x_{2i}) = F(\lambda \boldsymbol{x}_1 + (1 - \lambda)\boldsymbol{x}_2)\end{aligned} \tag{44}$$

with the usual vector notation; and this is the defining inequality of convexity of the function F in k-dimensional Euclidean space. If we limit ourselves to vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ satisfying (42), then we see that the vector $\lambda\boldsymbol{x}_1 + (1 - \lambda)\boldsymbol{x}_2$ also satisfies (43), and this proves that F is convex in the hyperplane also.⟧

Now note that again the function f in the discussion of Algorithm 2 is indeed convex (as was pointed out above) and so the function

$$F(n_1, n_2, n_3) = f(n_1) + f(n_2) + f(n_3) \tag{45}$$

occurring in the crucial equation (25) is convex, even on the plane (24). Now, a convex function attains its maximum at the boundary of the domain of permitted values [see, e.g., A. W. Roberts & D. E. Varberg, *Convex Functions* (Academic Press, New York, 1973) p. 124, Theorems D and E], provided this is a compact convex set [and the set of $\boldsymbol{x}$ satisfying (42) with non-negative coordinates is precisely so]. Thus we get the necessary result:

LEMMA 12. *The function* (45) *attains its global maximum under the condition* (24) *at an extreme point of the allowable values of* n_1, n_2, *and* n_3.

This lemma completes the proof that indeed the bounds obtained for all the extreme cases of 2.2 in (26) - (37) contain among them the global bound $f(n)$ for the a.o. count. Of the bounds obtained, all quadratic in behavior, that with the largest coefficient of n^2 is (33). The corresponding coefficient in the bound for 2.3 in (23) is 27/2, which is larger; so that we may conclude that this is the worst case of all. The advantage of this asymptotic behavior over that given in (13) for Algorithm 1 is evident.

Thus, we have established the next main result:

THEOREM 3. *Algorithm 2* (i) *always yields a complete triangulation in a finite number of steps;* (ii) *takes* $9n$ *a.o. and* $O(n)$ *other operations to execute the preparatory Algorithm 0, and less than*

$$\frac{27}{2}\left(n^2 - \frac{11}{3}n + \frac{2}{3}\right) = O(n^2) \qquad (46)$$

a.o. and $O(n^2)$ *other operations to perform;* (iii) *is as economical as possible.*

Note that (ii) implies (i). The reference, here and in Theorem 2, to the "other operations" is a reminder that bookkeeping operations and tests are of the same order of number as the a.o. (in certain algorithms, though these "other operations" are quick, they become so numerous as to overshadow the a.o.: this is not the case here). The step 2.3 is economical (i.e., does not introduce new triangles, beyond the $n - 2$ necessary ones, as has already been explained in Theorem 2. A count of vertices shows that the net number of triads arising before and after step 2.2 is the same $[(n_1 - 2) + (n_2 - 2) + (n_3 - 2) + 2 = n - 2]$.

A comparison of the bounds of the two algorithms for smaller values of n is also instructive:

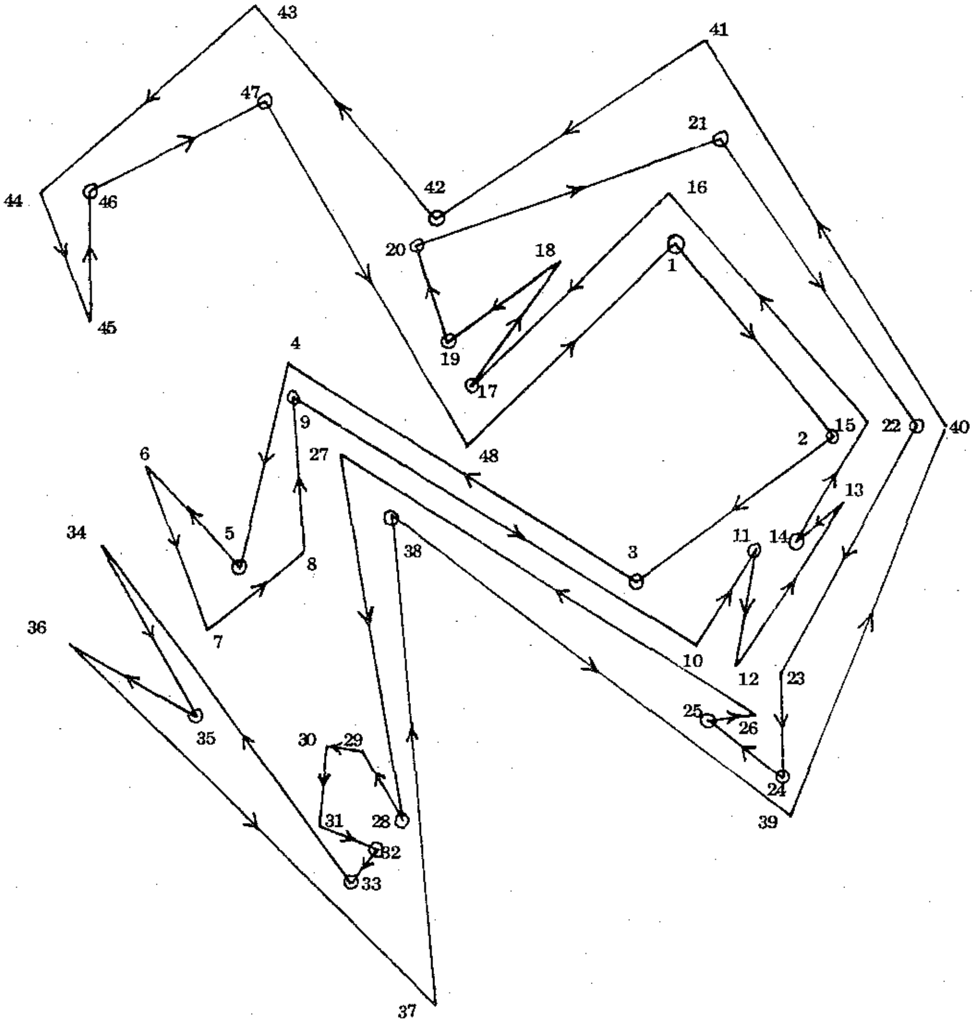
n =	4	8	20	100
(13)	75⅜	984⅜	16 887⅜	2 217 747⅜
(45)	27	477	4 419	130 059

(47)

5. Example

Figure 14. Example of a non-convex polygon with n = 48 vertices.

List $\mathcal{A}$: {4, 6, 7, 8, 10, 12, 13, 15, 16, 18, 23, 26, 27, 29, 30, 31, 34, 36, 37, 39, 40, 41, 43, 44, 45, 48}; p = 26.



List $\mathcal{B}$: {1, 2, 3, 5, 9, 11, 14, 17, 19, 20, 21, 22, 24, 25, 28, 32, 33, 35, 38, 42, 46, 47}; q = 22.

Algorithm 1: Empty convex triads at first pass, to List $\mathcal{C}$: (5,6,7), (5,7,8), (5,8,9), (12, 13, 14), (17,18,19), (22,23,24), (25,26,27), (28,29,30), (28,30,31), (28,31,32), (33,34,35), (33,35,36), (33,36,37), (44,45,46), (44,46, 47). Note that, in updating the lists, we remove 6, 7, 8, 13, 18, 23, 26, 29, 30, 31, 34, 35, 36, 45, 46 from list $\mathcal{A}$ (with 35 and 46 having been transferred from list $\mathcal{B}$ to list $\mathcal{A}$), remove 32 from list $\mathcal{B}$ by redundancy (collinearity), and further transfer 5, 25, 33 from list $\mathcal{B}$ to list $\mathcal{A}$. The results are shown in

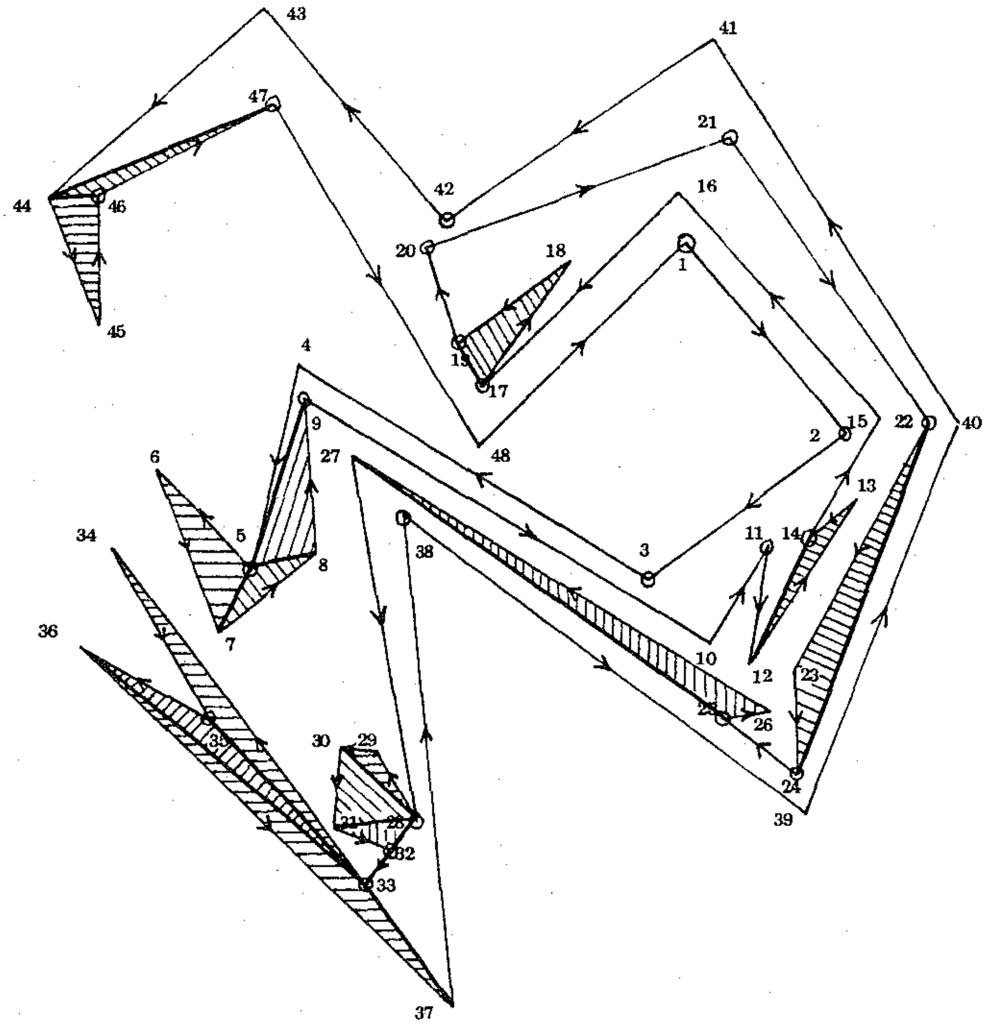


Figure 15.

List $\mathcal{A}$: {4, 5, 10, 12, 15, 16, 25, 27, 33, 37, 39, 40, 41, 43, 44, 48}; with p = 16.

List $\mathcal{B}$: {1, 2, 3, 9, 11, 14, 17, 19, 20, 21, 22, 24, 28, 38, 42, 47}; with q = 16.

Empty convex triads at second pass, to List $\mathcal{C}$: (4,5,9), (4,9,10), (11,12,14), (11,14,15), (24,25,27), (28,33,37), (28,37,38), (43,44,47), (43,47,48). In updating lists, we remove 5, 9, 12, 14, 25, 33, 37, 44, 47 from list $\mathcal{A}$ (9, 14, and 47 having been transferred from list $\mathcal{B}$ to list $\mathcal{A}$), and further transfer 28 from list $\mathcal{B}$ to list $\mathcal{A}$. The results are shown in Figure 16; for which we have:

List $\mathcal{A}$: {4, 10, 15, 16, 27, 28, 39, 40, 41, 43, 48}; with p = 11.

List $\mathcal{B}$: {1, 2, 3, 11, 17, 19, 20, 21, 22, 24, 38, 42}; with q = 12.

Empty convex triads at third pass, to List $\mathcal{C}$: (3,4,10), (3,10,11), (3,11,15), (27,28,38), (27,38,39). In updating lists, we remove 4, 10, 11, 28, 38

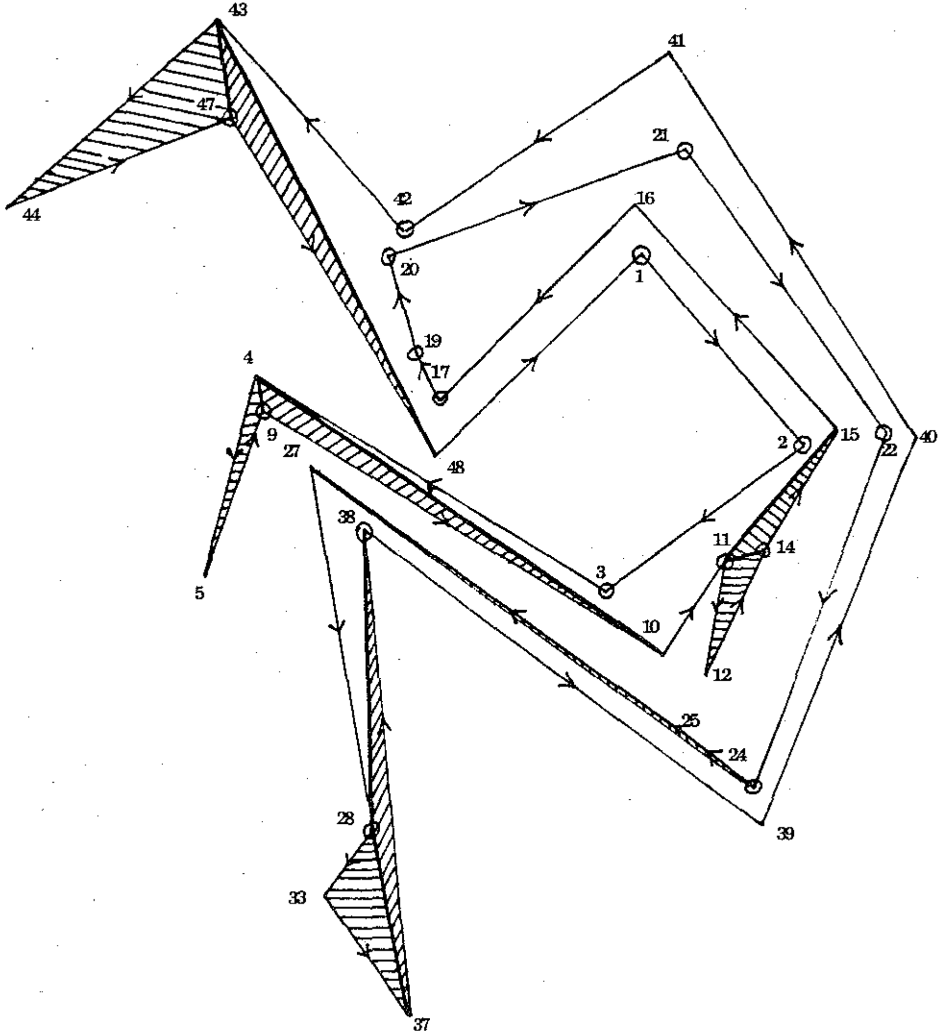


Figure 16.

from list $\mathcal{A}$ (11 and 38 having been transferred from list $\mathcal{B}$ to list $\mathcal{A}$), and further transfer 2 from list $\mathcal{B}$ to list $\mathcal{A}$, and remove 3 from list $\mathcal{B}$ by redundancy. The result is shown in Figure 17; for which we have:

List $\mathcal{A}$: {2, 15, 16, 27, 39, 40, 41, 43, 48}; with $p = 9$.

List $\mathcal{B}$: {1, 17, 19, 20, 21, 22, 24, 42}; with $q = 8$.

Empty convex triads at fourth pass, to List $\mathcal{C}$: (1,2,15), (1,15,16), (1,16,17), (24,27,39), (24,39,40). In updating lists, we remove 2, 15, 16, 27, 39 from list $\mathcal{A}$, and transfer 1 and 24 from list $\mathcal{B}$ to list $\mathcal{A}$. The result is shown in Figure 18; for which we have:

List $\mathcal{A}$: {1, 24, 40, 41, 43, 48}; with $p = 6$.

List $\mathcal{B}$: {17, 19, 20, 21, 22, 42}; with $q = 6$.

Figure 17.

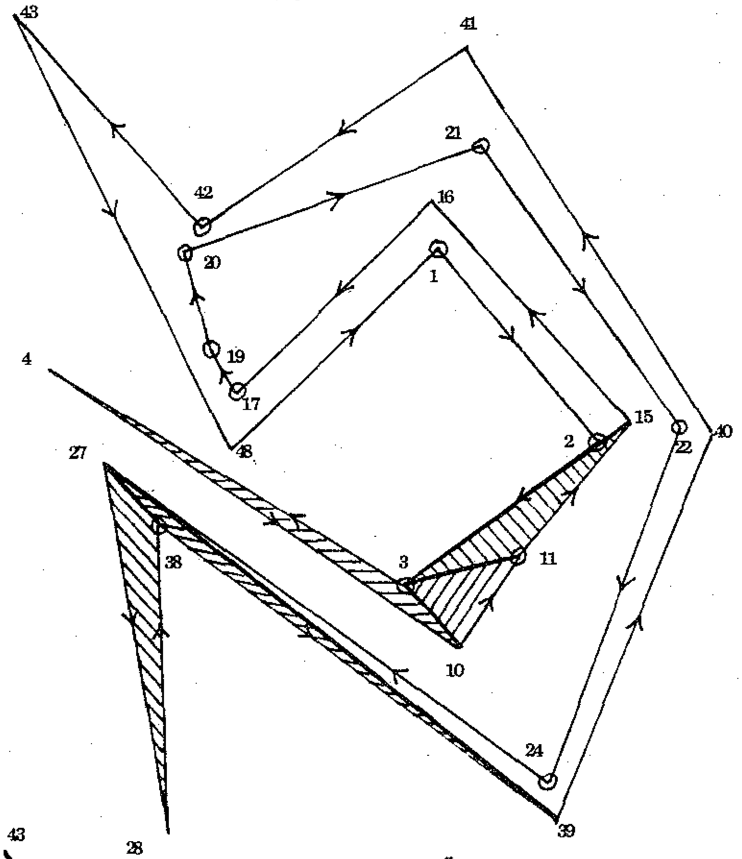


Figure 18.

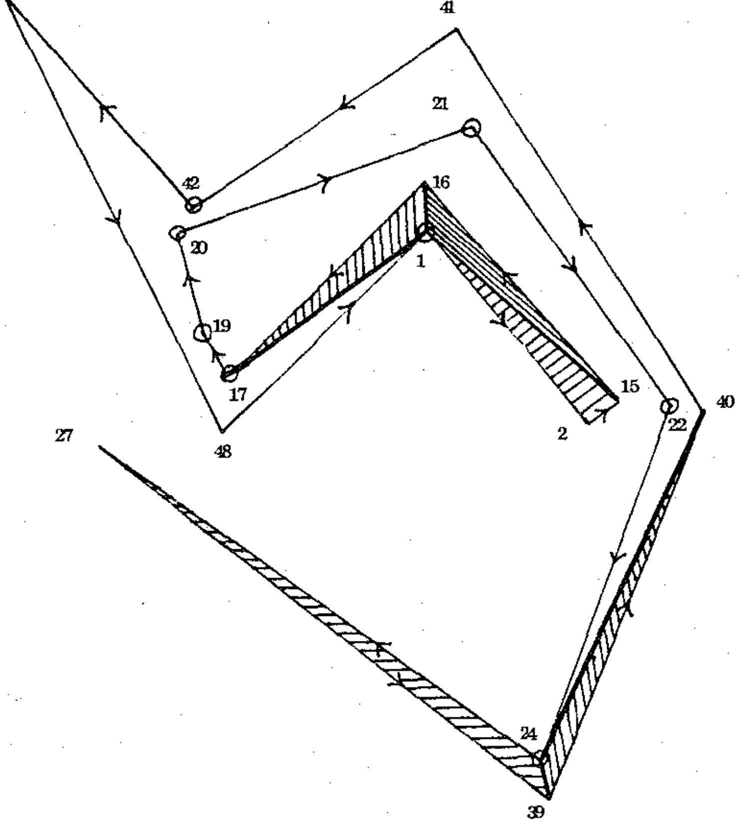


Figure 19.

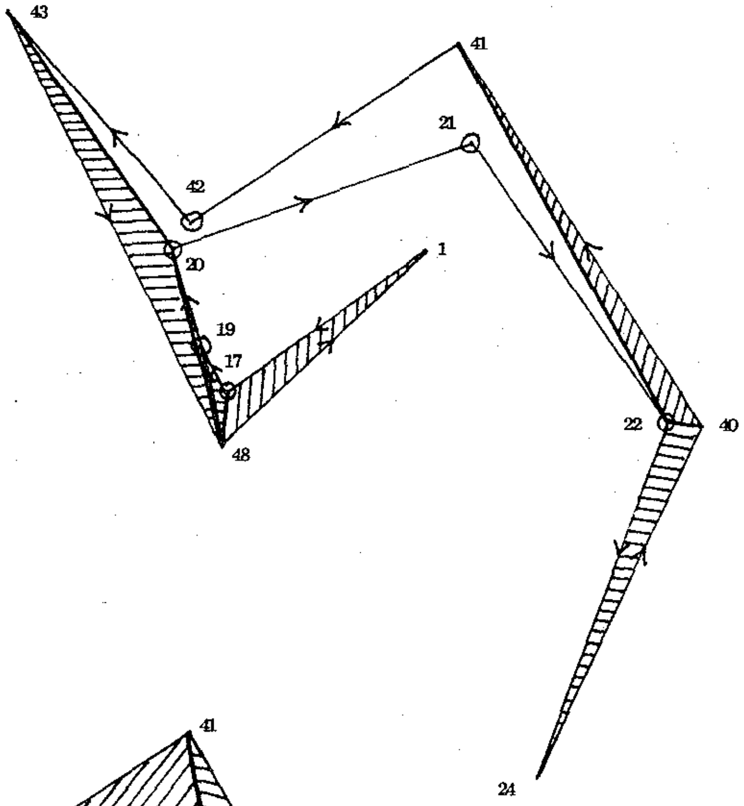
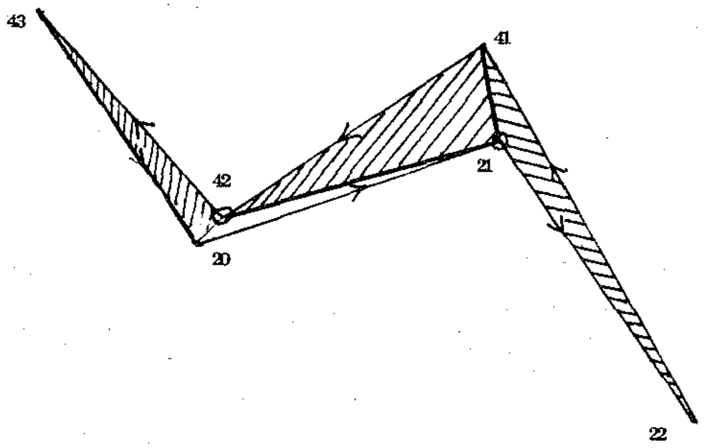


Figure 20.



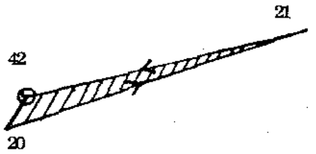


Figure 21.

Empty convex triads at fifth pass, to list $\mathcal{C}$: (48,1,17), (48,17,19), (48,19,20), (22,24,40), (22,40,41), (43,48,20). Inupdating lists, we remove 1, 17, 19, 24, 40, 48 from list $\mathcal{A}$ (17 and 19 having been transferred from list $\mathcal{B}$ to list $\mathcal{A}$), and further transfer 20 and 22 from list $\mathcal{B}$ to list $\mathcal{A}$. The result is shown in Figure 19; for which we have:

List $\mathcal{A}$: {20, 22, 41, 43}; with $p = 4$.
List $\mathcal{B}$: {21, 42}; with $q = 2$.

Empty convex triads at sixth pass, to list $\mathcal{C}$: (21,22,41), (21,41,42), (42,43,20). In updating lists, we remove 22, 41, 43 from list $\mathcal{A}$, and transfer 21 and 42 from list $\mathcal{B}$ to list $\mathcal{A}$, leaving list $\mathcal{B}$ empty. The result is shown in Figure 20; for which we have:

List $\mathcal{A}$: {20, 21, 42}; with $p = 3$.
List $\mathcal{B}$: empty; $q = 0$.

The final situation is shown in Figure 21, where the single remaining triad is removed into list $\mathcal{C}$. Just 15 + 9 + 5 + 5 + 6 + 3 + 1 = 44 triads are in list $\mathcal{C}$, being $(n - 2) - 2$, the deficit of 2 being attributable to the two vertices removed by the exercise of 0.4 (redundancy by collinearity) in the first (P_{32}) and third (P_3) passes.

We now turn to the *a.o. count*. First, note that two discriminants are computed, under 1.3(c) for every triad put into list $\mathcal{C}$, excepting the last two; so there is a count of

$$18 \times 42 = 756 \tag{48}$$

a.o. for this, in all. The remaining a.o. arise from discriminant computation for inclusion tests, 27 a.o. for each test. The number of tests is obtained as follows. We begin with $q = 22$ indices in list $\mathcal{B}$:

$22 \times 3 = 66$	4, {6}, {7} tested [{...} denotes removal to list $\mathcal{C}$]; 5 transferred from list $\mathcal{B}$ to list $\mathcal{A}$.
$21 \times 9 = 189$	{8}, 10, 12, {13}, 15, 16, {18}, {23}, {26}; 25 transferred.
$20 \times 4 = 80$	27, {29}, {30}, {31}; 32 eliminated by redundancy.
$19 \times 1 = 19$	{34}; 35 transferred.
$18 \times 2 = 36$	{35}, {36}; 33 transferred.
$17 \times 7 = 119$	37, 39, 40, 41, 43, 44, {45}; 46 transferred.
$16 \times 4 = 64$	{46}, 48, 4, {5}; 9 transferred.
$15 \times 3 = 45$	{9}, 10, {12}; 14 transferred.
$14 \times 6 = 84$	{14}, 15, 16, {25}, 27, {33}; 28 transferred.
$13 \times 6 = 78$	{37}, 39, 40, 41, 43, {44}; 47 transferred.

$12 \times 3 = 36$ {47}, 48, {4}; 3 transferred.
$11 \times 1 = 11$ {10}; 11 transferred.
$10 \times 5 = 50$ {11}, 15, 16, 27, {28}; 38 transferred.
$9 \times 7 = 63$ {38}, 39, 40, 41, 43, 48, {3} [null triad]; 2 transferred.
$8 \times 2 = 16$ {2}, {15}; 1 transferred.
$7 \times 2 = 14$ {16}, {27}; 24 transferred.
$6 \times 2 = 12$ {39}, {1}; 17 transferred.
$5 \times 1 = 5$ {17}; 19 transferred.
$4 \times 2 = 8$ {19},{24}; 22 transferred.
$3 \times 4 = 12$ {40}, 41, 43, {48}; 20 transferred.
$2 \times 2 = 4$ 20, {22}; 21 transferred.
$1 \times 2 = 2$ {41}, {43}; 42 transferred.

The total is thus 1,013 tests = 27,351 a.o., plus (48) for a grand total of

$$28{,}107 \text{ a.o.} \tag{49}$$

For comparison, the bound (13) yields the result that (49) should be

$$\leqslant 241{,}734\tfrac{3}{8} \text{ a.o.}; \tag{50}$$

so that we see how much of a "worst case estimate" it is!

Returning to Figure 14, we now apply <u>Algorithm 2</u>: Initial lists $\mathcal{A}$ and $\mathcal{B}$ are as before (see page 23 above). Triads (3,4,9) and (4,5,9) are put in list $\mathcal{C}$, by <u>2.2</u>, and we get two polygons:

$\mathfrak{P}_1 = [1, 2, 3, 9, 10, \ldots, 47, 48]$ and $\mathfrak{P}_3 = [5, 6, 7, 8, 9]$,

with new lists,

$\mathcal{A}_1$: {9, 10, 12, 13, 15, 16, 18, 23, 26, 27, 29, 30, 31, 34, 36, 37, 39, 40, 41, 43, 44, 45, 48}; with $p_1 = 23$;
$\mathcal{B}_1$: {1, 2, 3, 11, 14, 17, 19, 20, 21, 22, 24, 25, 28, 32, 33, 35, 38, 42, 46, 47}; with $q_1 = 20$;
$\mathcal{A}_3$: {6, 7, 8, 9}; with $p_3 = 4$;
$\mathcal{B}_3$: {5}; with $q_3 = 1$;

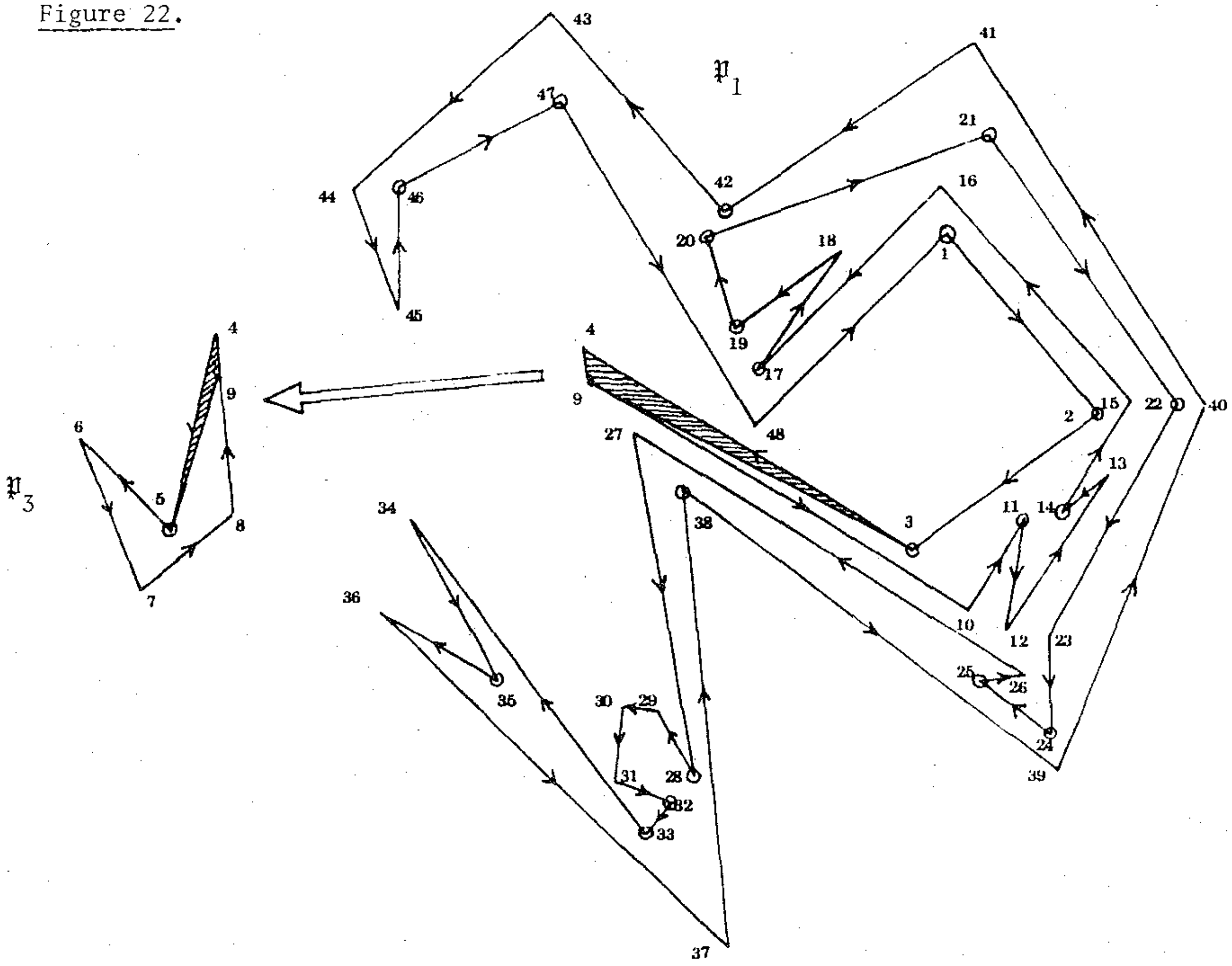
as illustrated in Figure 22. Take $\mathfrak{P}_3$ first: triads (5,6,7), (5,7,8), and (5,8,9) successively go to list $\mathcal{C}$, terminating this branch, by <u>2.3</u> only. In $\mathfrak{P}_1$, triads (3,9,10) and (3,10,11) are empty; then (11,12,14) and (12,13,14) are removed, by <u>2.2</u>, leaving just one new polygon:

$\mathfrak{P}_{11} = [1, 2, 3, 11, 14, 15, 16, \ldots, 47, 48]$,

with new lists [11 being removed by redundancy],

$\mathcal{A}_{11}$: {3, 14, 15, 16, 18, 23, 26, 27, ..., 43, 44, 45, 48};

Figure 22.



with $p_{11} = 21$;

$\mathcal{B}_{11}$: $\{1, 2, 17, 19, 20, 21, \ldots, 42, 46, 47\}$; with $q_{11} = 17$;

as illustrated in Figure 23. Proceeding, empty triads are found at (2,3,14), (2,14,15), (2,15,16), and the next split occurs at (2,16,1) and (16,17,1), again yielding a single polygon:

$\mathfrak{P}_{113} = [1, 17, 18, 19, 20, \ldots, 47, 48]$,

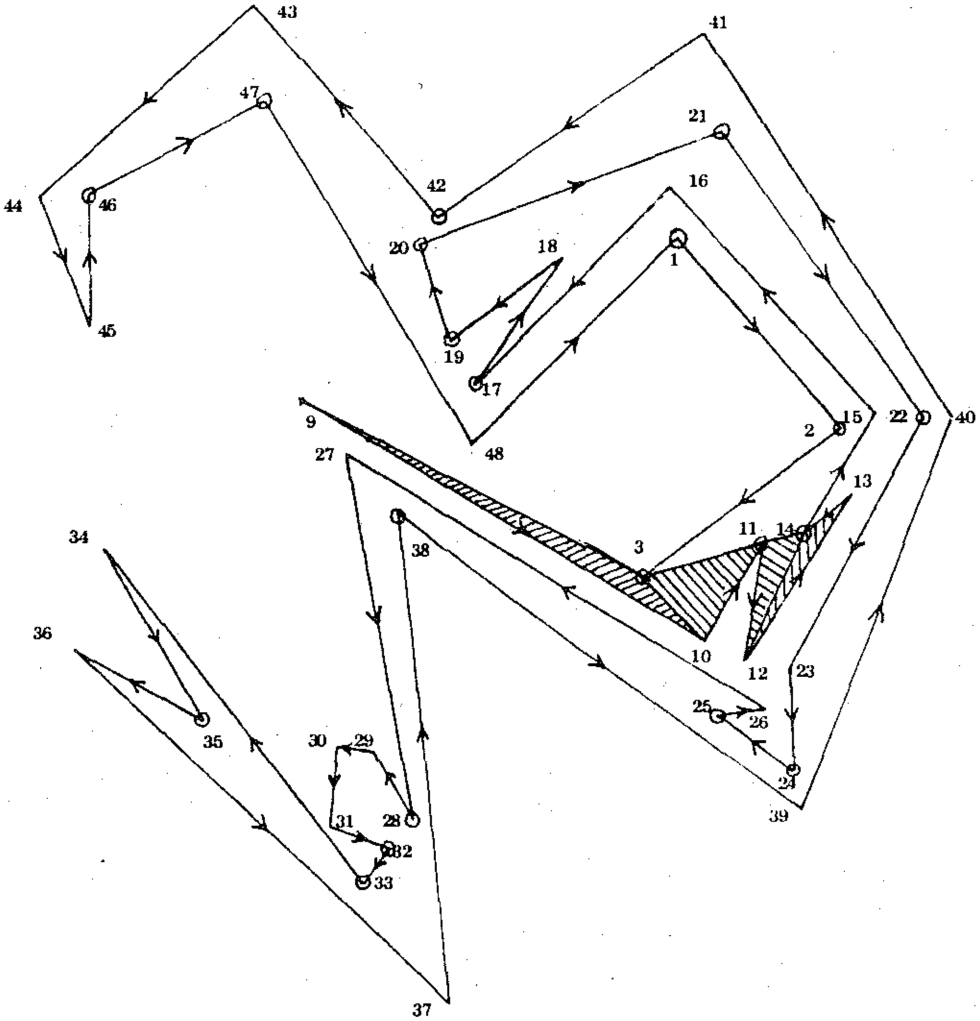
with new lists,

$\mathcal{A}_{113}$: $\{1, 18, 23, 26, 27, \ldots, 44, 45, 48\}$; with $p_{113} = 18$;

$\mathcal{B}_{113}$: $\{17, 19, 20, 21, \ldots, 42, 46, 47\}$; with $q_{113} = 15$;

as illustrated in Figure 24.

Figure 23.

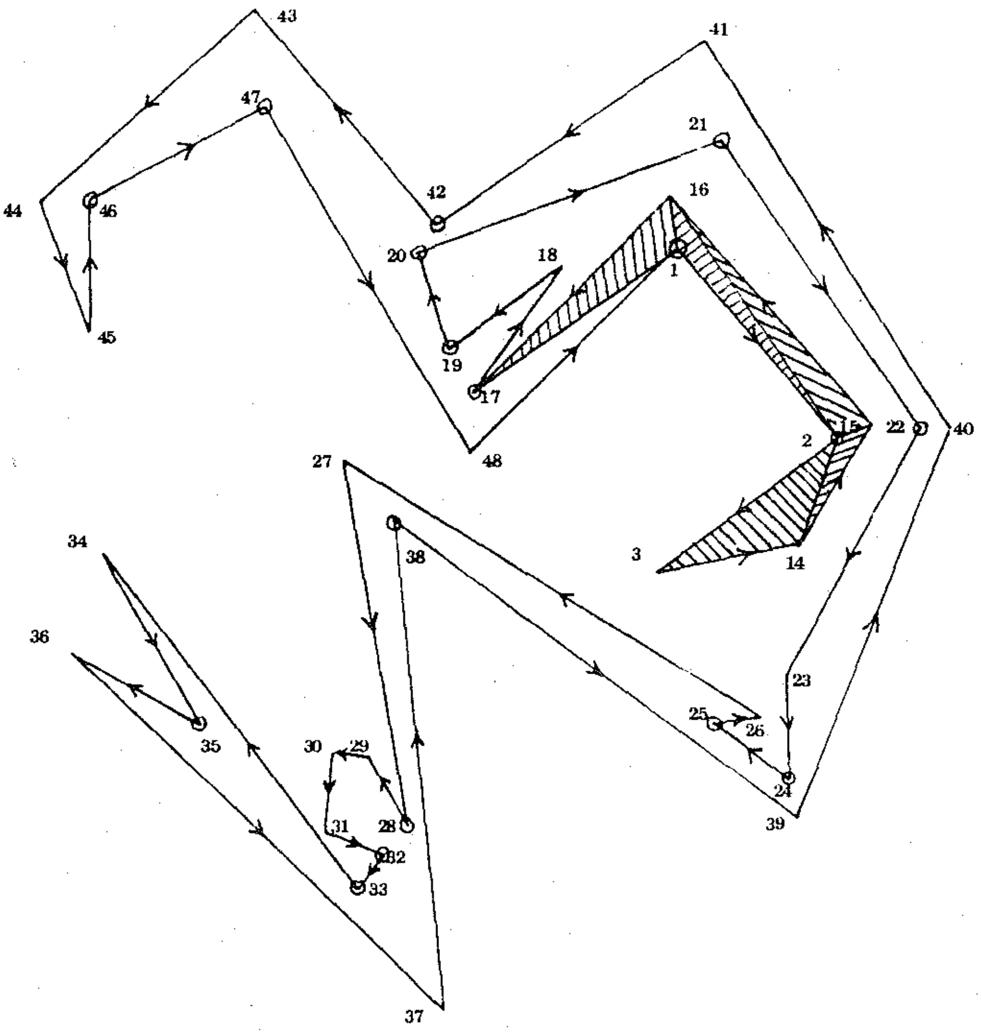


Note: We use subscripts to refer to the sub-polygons on the h^- [1], middle [2], and h^+ [3] sides of the triad in question. There are no middle polygons so far (cases have been degenerate as Figure 13, second example, or worse).

Proceeding again, empty triads are found at (48,1,17), (17,18,19), (22,23,24), (25,26,26), before we encounter a split at (25,27,38) and (27, 28,38), yielding the two polygons:

$\mathfrak{P}_{1131}$ = [17, 19, 20, 21, 22, 24, 25, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48];

Figure 24.



with lists

$\mathcal{A}_{1131}$: {17, 25, 38, 39, 40, 41, 43, 44, 45, 48}; with $p_{1131} = 10$;
$\mathcal{B}_{1131}$: {19, 20, 21, 22, 24, 42, 46, 47}; with $q_{1131} = 8$;

and

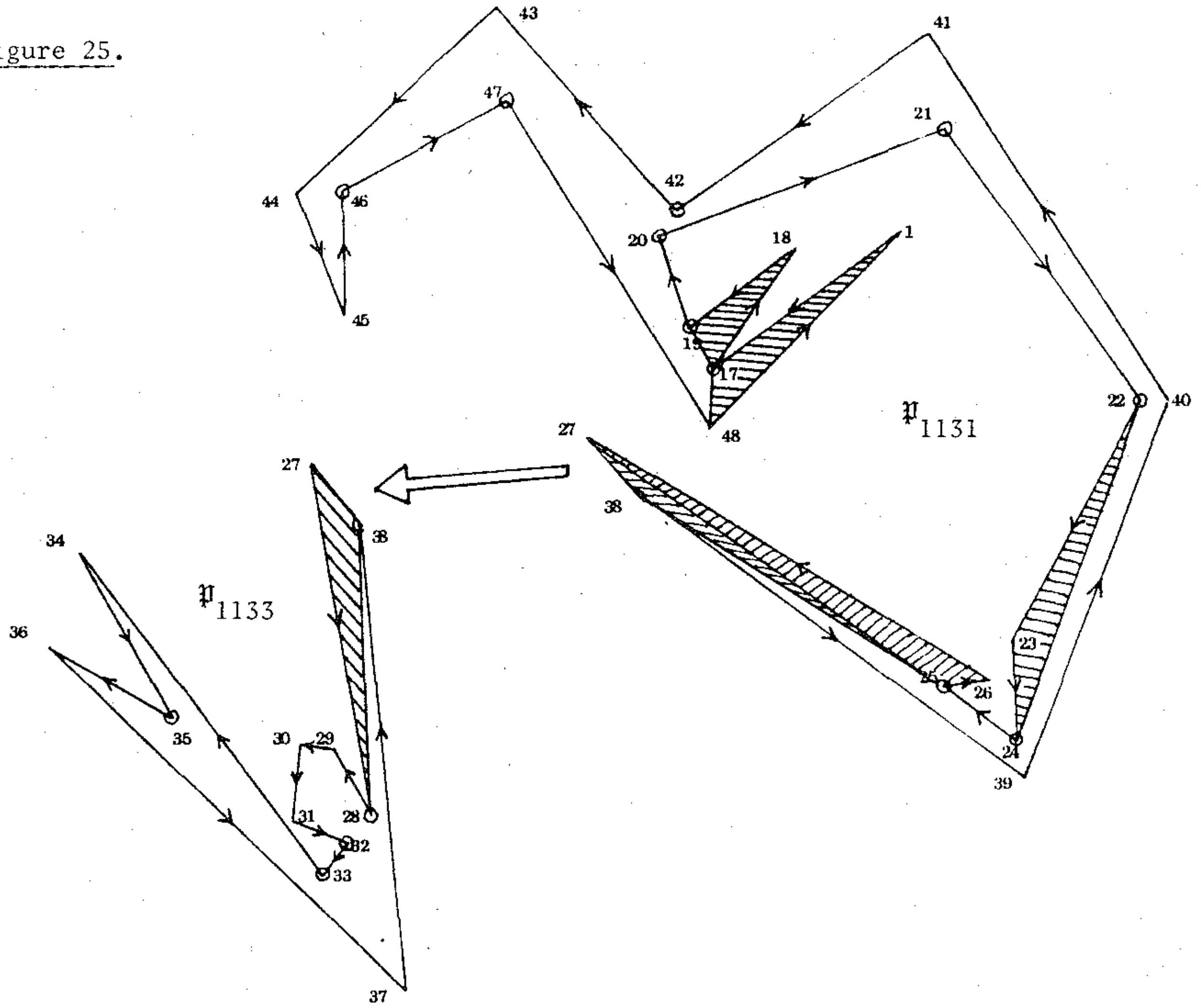
$\mathfrak{P}_{1133} = [28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38]$;

with lists

$\mathcal{A}_{1133}$: {29, 30, 31, 34, 36, 37, 38}; with $p_{1133} = 7$;
$\mathcal{B}_{1133}$: {28, 32, 33, 35}; with $q_{1133} = 4$;

as illustrated in Figure 25. Take $\mathfrak{P}_{1133}$ first: triads (28,29,30), (28,30,31), (28,31,32) are found empty and 32 becomes redundant; then (33,34,35), (33,35,36), (33,36,37) are also found empty; then a fully degenerate split yields (33,37,28) and (37,38,28), terminating this branch.

Figure 25.



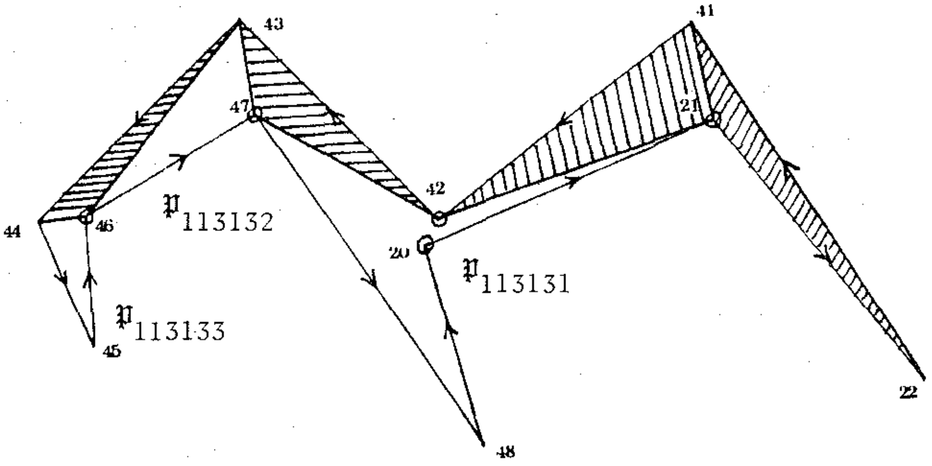
In $\mathfrak{P}_{1131}$, empty triads are removed at (48,17,19), (48,19,20), (24,25,38), (24,38,39), (24,39,40), before we appeal to <u>2.2</u> and split off (24,40,22) and (40,41,22), yielding the single polygon:

$\mathfrak{P}_{11313}$ = [20, 21, 22, 41, 42, 43, 44, 45, 46, 47, 48],
with lists

$\mathcal{A}_{11313}$: {22, 41, 43, 44, 45, 48}; with $p_{11313} = 6$;
$\mathcal{B}_{11313}$: {20, 21, 42, 46, 47}; with $q_{11313} = 5$;
as illustrated in Figure 26. Empty triads are found at (21,22,41) and (21,41, 42), before our first and only three-way split (in this example) at (42,43,47) and (43,44,46), leaving the three polygons:

$\mathfrak{P}_{113131}$ = [20, 21, 42, 47, 48],
with lists

Figure 26.



$\mathcal{A}_{113131}$: {21, 47, 48}; with $p_{113131} = 3$; $\mathcal{B}_{113131}$: {20, 42}; with $q_{113131} = 2$; $\mathfrak{P}_{113132} = [43, 46, 47]$; and $\mathfrak{P}_{113133} = [44, 45, 46]$.

The last two polygons are empty triads; so they terminate immediately. For $\mathfrak{P}_{113131}$, we find empty triads (20,21,42), (20,42,47), and (20,47,48), completing the triangulation.

We now turn to the *a.o. count* for this algorithm. Counting the triads removed, we find 44 = (48 - 2) - 2 again, with P_{11} and P_{32} found redundant. Of these, 30 are found empty through 2.3 and there are seven splits by 2.2, for 14 more triads. Thus the discriminant-pairs under 2.3(c) number just 24 (we recall that the last two or less triads of a polygon do not require the calculation of these test-discriminants. Thus we use

$$18 \times 24 = 432 \text{ a.o.} \tag{51}$$

for this purpose. In computing the a.o. count for the first algorithm, we did not count the work required to set up the initial lists $\mathcal{A}$ and $\mathcal{B}$; so neither do we do so here; but we now must compute the a.o. required to get the new lists, at each split. In all, there are seven splits, requiring in all

$$\begin{aligned} 9 \times (&p_1 + q_1 + p_3 + q_3 + p_{11} + q_{11} + p_{113} + q_{113} + p_{1131} + q_{1131} + \\ &p_{1133} + q_{1133} + p_{11313} + q_{11313} + p_{113131} + q_{113131}) \\ &= 9 \times (23 + 20 + 4 + 1 + 21 + 17 + 18 + 15 + 10 + 8 + 7 + 4 + \\ &\quad 6 + 5 + 3 + 2) = 9 \times 164 = 1{,}476 \text{ a.o.} \end{aligned} \tag{52}$$

Finally, we must count inclusion tests, performed at each step of 2.1 for all members of the current list $\mathcal{B}$ and taking 27 a.o. each. We count as we did before.

$22 \times 1 = 22$ (4) [(...) denotes a split after this test.]
$1 \times 2 = 2$ {6}, {7}.
$20 \times 1 = 20$ {9}; 3 transferred.
$19 \times 2 = 38$ {10}, (12).
$17 \times 2 = 34$ {3}, {14}; 2 transferred.
$16 \times 2 = 32$ {15}, (16).
$15 \times 2 = 30$ {1}, {18}; 17 transferred.
$14 \times 2 = 28$ {23}, {26}; 25 transferred.
$13 \times 1 = 13$ (27).
$4 \times 3 = 12$ {29}, {30}, {31}; 32 redundant.
$3 \times 1 = 3$ {34}; 35 transferred.
$2 \times 2 = 4$ {35}, {36}; 33 transferred; (37).
$8 \times 1 = 8$ {17}; 19 transferred.
$7 \times 5 = 35$ {19},{25}, {38}, {39}, (40).
$5 \times 1 = 5$ {22}; 21 transferred.
$4 \times 2 = 8$ {41}, (43).
$2 \times 1 = 2$ {21}; 42 transferred.

The total is thus 296 tests = 7,992 a.o., plus (51) and (52), for a grand total of

$$9{,}900 \text{ a.o.}, \tag{53}$$

or *about one-third* of the work required by Algorithm 1. For comparison, the bound (46) yields the result that (53) should be

$$\leqslant 28{,}737 \text{ a.o.} \tag{54}$$

so that the bound is somewhat closer for this example with Algorithm 2 than with Algorithm 1.

6. The Third Algorithm

A reconsideration of the first two algorithms, as described above, indicates that no use is made of the fact, that, when a triad is processed, the rest of the polygon changes relatively little; the procedure prescribed requires the computation, at each iteration, of numerous discriminants γ (as defined in (20) - (22)); and indeed, these make up the bulk of the computational work of the algorithms. It is evident that there is an irreducible residue of inclusion-testing of the order of $\frac{1}{4}n^2$ tests, or $\frac{27}{4}n^2$ a.o., in the worst case. Since the second algorithm takes time of the order of about twice this, it does not seem very promising to seek improvement of this along this line of thought; but, by the same token, since the first algorithm takes time of the order of $\frac{9}{4}n^3$, it is a much likelier candidate.

We therefore reconstruct Algorithm 1, in a way that seeks to minimize the duplication of effort, by keeping a record of all vertices contained in each convex triad under consideration. We shall specify the data structures used in a little more detail. We assume that, initially, the polygon $\mathfrak{P}$ is given as an array [see (5)]

$$\left.\begin{array}{l} P = [\mathrm{P}_1, \mathrm{P}_2, \mathrm{P}_3, \ldots, \mathrm{P}_n], \quad \text{with} \quad \mathrm{P}_j = [x_j, y_j]; \\ \begin{array}{|c|c|c|c|c|} \hline x_1 & x_2 & x_3 & \cdots & x_n \\ \hline y_1 & y_2 & y_3 & \cdots & y_n \\ \hline \end{array} \begin{array}{l} 1 \\ 2 \end{array} \downarrow \text{first index (row)} \;\vdots\; \begin{array}{l} P(1, j) = x_j, \\ P(2, j) = y_j. \end{array} \\ \quad 1 \quad 2 \quad 3 \quad \ldots \quad n \quad \longrightarrow \text{second index (column)} \end{array}\right\} \qquad (55)$$

We also assume that n is too large to allow space allocation of $\Omega(n^2)$ or more; so that some economy of storage must be adopted.

We set up data-structures as follows:

(a) Real array G of size n [to hold discriminants for each vertex].

(b) Pointer (address) array S of size n [pointer $S(k)$ points to list $\mathcal{L}_k$].

(c) Integer array C of size $(3 \times n)$ [Successive $C(1, r)$, $C(2, r)$, and $C(3, r)$ hold indices h, i, and j of empty triads $\mathrm{P}_h\mathrm{P}_i\mathrm{P}_j$ as they are identified. This corresponds to 'List $\mathcal{C}$' of Algorithm 1.]

(d) *Linked lists* will be structured as follows. There will be an *identifier*, which is a pointer, *id*, whose name is the name of the list; there will be a *header cell*, of the form [*lp*, *ls*], where *lp* points to the first cell of the list and *ls* points to the last cell of the list; and then the cells making up the body of the list will be of the form [*lp*, *content*], where each pointer *lp* points to the next cell in sequence and *content* denotes the content of the cell. When the list is initialized, the header cell takes the form [NIL, NIL], and the last cell will always take the form [NIL, *content*]. Two *operations* on lists will be required here: *append*(*id*, *entries*) attaches a cell with the given *entries* at the end of the list with identifier *id*. The procedure is:

```
A*1    if id:lp = NIL, then id:ls ← id:lp ← newcell {newcell is a pointer
           to a new cell, pointed to by header and old last-cell};
A*2    else, id:ls ← id:ls:lp ← newcell {assign right-to-left};
A*3    id:ls:lp ← NIL {list-pointer of new last-cell is NIL};
A*4    id:ls:content ← entries {e.g., if content ≡ (a, b, c), entries ≡
           (x, y, z), then id:ls:a ← x, id:ls:b ← y, id:ls:c ← z}.
```

In our pseudo-code, the notation 'A ← B' means that the expression or variable B is evaluated, and the result is inserted into the variable (or memory-location) A [*assignment* operation]; if Q is a pointer to a cell with components a, b, c, ..., then the notation 'Q:x' denotes the component x of the cell pointed to by Q; if the x-component is itself a cell-pointer, then 'Q:x:y' means the component y of the cell pointed to by Q:x. Thus, above, *id*:*ls* is the last-cell pointer of the header, *id*:*ls*:*lp* is the list-pointer of the last cell, and *id*:*ls*:*lp*:*lp* is the list-pointer in the cell pointed to by what was the last cell, i.e., the list-pointer in the new (last) cell. As usual, assignment overwrites and supersedes previous content. The operation *delete*(*id*, *ptr*) removes from the list with identifier *id* the cell next after that to which the pointer *ptr* points. The procedure is:

```
D-1    if id:ls = ptr:lp, then id:ls ← ptr  {if the cell to be deleted
                   is the last, then the last-cell pointer in the header
                   should point to the predecessor cell; otherwise the
                   last-cell pointer is unchanged};
D-2    ptr:lp ← ptr:lp:lp  {the list-pointer in the cell preceding that
                   to be deleted should point directly to the cell to
                   which the deleted cell points}.
```

What must be noted is that both of these procedures take time $O(1)$ to execute.

(e) Linked list with identifier $\mathcal{D}$ and cells of the form [*lp*, *cp*, *up*, *x*] in the body of the list; so that *content* = [*cp*, *up*, *x*], where *cp* and *up* are pointers, and *x* is an integer index [$\mathcal{D}$ is a list of all *active* vertices of the polygon $\mathfrak{P}$; initially, $\mathcal{D}$ is constructed as a list of all *convex* and *re-entrant* vertices (*x* denoting the index of the vertex P_x), in the order in which they occur in a tour of $\mathfrak{P}$ in the direction in which the interior $I_{\mathfrak{P}}$ of $\mathfrak{P}$ is on the *left*. In each cell, the pointer *lp* points to the next cell in the list $\mathcal{D}$; if P_x is a *convex* vertex, and if a pointer *ptr* points to the *predecessor* of the cell referring to P_x, then the pointer *ptr*:*cp* points to the predecessor of the cell referring to the *next* convex vertex; similarly, if P_x is a *re-entrant* vertex and *ptr* points to the predecessor of the cell referring to P_x, then *ptr*:*cp* points to the predecessor of the cell referring to the *next* re-entrant vertex. A pointer $\mathcal{A}$ is initially set to point to the predecessor of the first cell referring to a convex vertex; so that the cells pointed to by

$$\mathcal{A}{:}lp, \quad \mathcal{A}{:}cp{:}lp, \quad \mathcal{A}{:}cp{:}cp{:}lp, \quad \mathcal{A}{:}cp{:}cp{:}cp{:}lp, \quad \ldots \tag{56}$$

form the complete list of convex vertices in the cyclic order ['List $\mathcal{A}$']; and similarly a pointer $\mathcal{B}$ is initially set to point to the predecessor of the first cell referring to a re-entrant vertex, and the cells pointed to by

$$\mathcal{B}{:}lp, \quad \mathcal{B}{:}cp{:}lp, \quad \mathcal{B}{:}cp{:}cp{:}lp, \quad \mathcal{B}{:}cp{:}cp{:}cp{:}lp, \quad \ldots \tag{57}$$

form the complete list of re-entrant vertices in the cyclic order ['List $\mathcal{B}$']. When A points to the predecessor of a cell referring to the convex vertex P_i, say, so that $A{:}lp{:}x = i$; then (if $A{:}x = h$ and $A{:}lp{:}lp{:}x = j$, say) $P_hP_iP_j$ forms a *convex triad*, and if it is *empty* of other vertices, it can be transferred to the array C (i.e., to 'List $\mathcal{C}$': see (c) above). All the pointers

$$\mathcal{B}{:}up = \mathcal{B}{:}cp{:}up = \mathcal{B}{:}cp{:}cp{:}up = \mathcal{B}{:}cp{:}cp{:}cp{:}up = \ldots = \mathrm{NIL}, \tag{58}$$

and if $A{:}lp{:}x = i$, then $A{:}up$ points to the list $\boldsymbol{u}_i$, for every i.]

(f) For every k, linked list with identifier $\boldsymbol{t}_k$ and cells of the form $[lp,\ tp]$, where lp is the list-pointer, as usual, and tp is a pointer which points to the predecessor in some List $\boldsymbol{u}_i$ of a cell referring to index k [$S(k) = \boldsymbol{t}_k$; see (b) above].

(g) For every i such that P_i is a convex vertex, linked list with identifier $\boldsymbol{u}_i$ and cells of the form $[up,\ k]$, where up is the list-pointer and k is the index of a vertex contained in the convex triad associated with P_i. [If $A{:}lp{:}x = i$, then the triad is $P_hP_iP_j$, where $h = A{:}x$ and $j = A{:}lp{:}lp{:}x$.]

(h) After the list $\mathcal{D}$ has been constructed, we apply

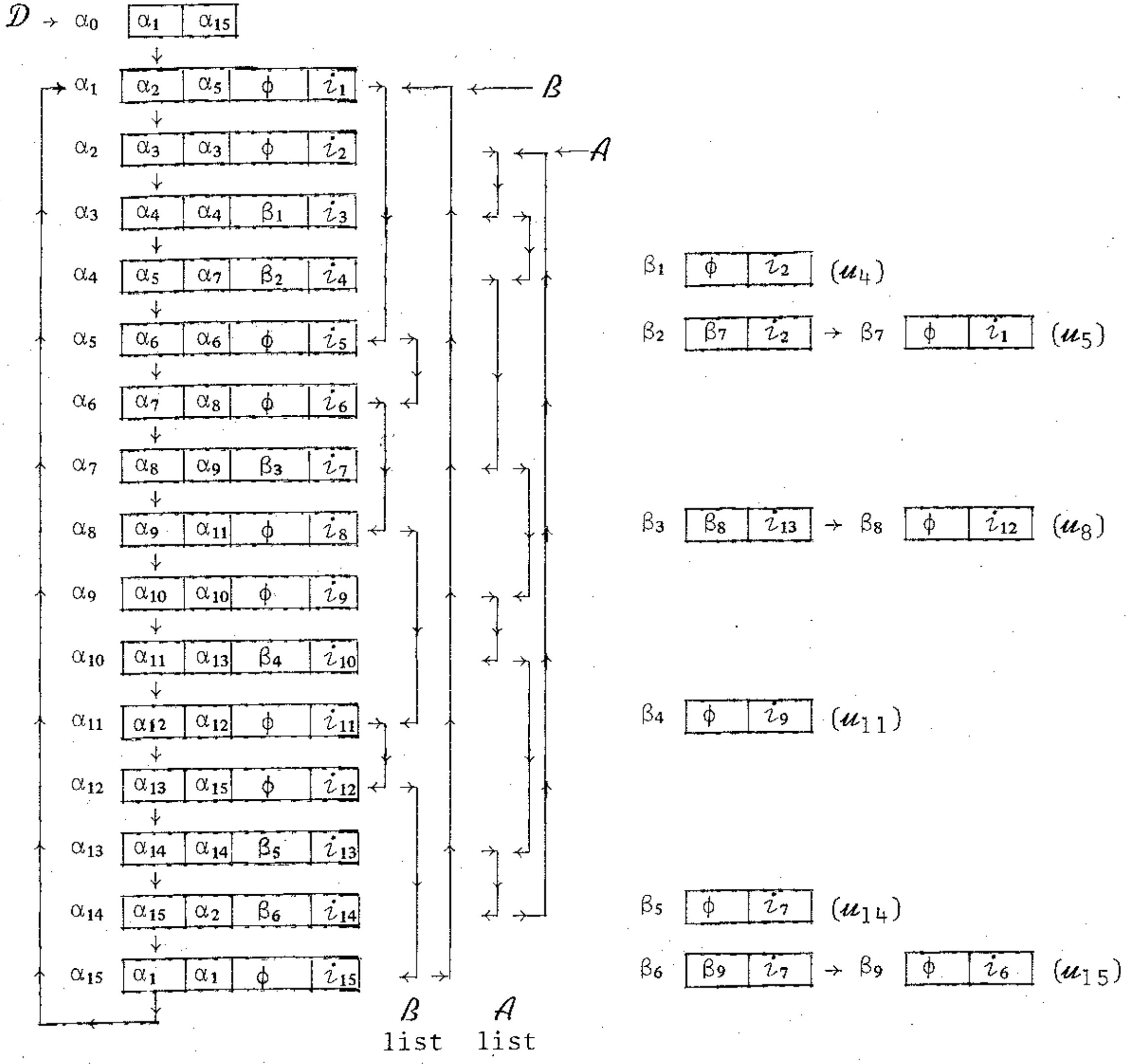
C -1 $\quad \mathcal{D}{:}ls{:}lp \leftarrow \mathcal{D}{:}lp$ {pointer in last cell now points to first cell};

to make the list *circular*. We also note that A and B are initially equal to $\mathcal{A}$ and $\mathcal{B}$, and advance as we construct the list $\mathcal{D}$ until A points to the predecessor of the last cell referring to a convex vertex and B points to the predecessor of the last cell referring to a re-entrant vertex. We then do

C -2 $\quad A{:}cp \leftarrow \mathcal{A}$;
C -3 $\quad B{:}cp \leftarrow \mathcal{B}$;

making lists $\mathcal{A}$ and $\mathcal{B}$ circular, too.

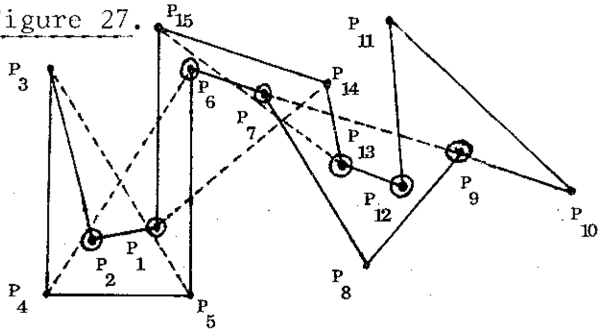
The diagram below illustrates the structures described above.



$S(1) = t_1 = \gamma_1 \rightarrow \boxed{\phi \mid \beta_2}$, $S(2) = t_2 = \gamma_2 \rightarrow \boxed{\gamma_8 \mid \alpha_4} \rightarrow \gamma_8\ \boxed{\phi \mid \alpha_5}$,
$S(6) = t_6 = \gamma_3 \rightarrow \boxed{\phi \mid \beta_6}$, $S(7) = t_7 = \gamma_4 \rightarrow \boxed{\gamma_9 \mid \alpha_{14}} \rightarrow \gamma_9\ \boxed{\phi \mid \alpha_{15}}$,
$S(9) = t_9 = \gamma_5 \rightarrow \boxed{\phi \mid \alpha_{11}}$, $S(12) = t_{12} = \gamma_6 \rightarrow \boxed{\phi \mid \beta_3}$, $S(13) = t_{13} = \gamma_7 \rightarrow \boxed{\phi \mid \alpha_8}$.

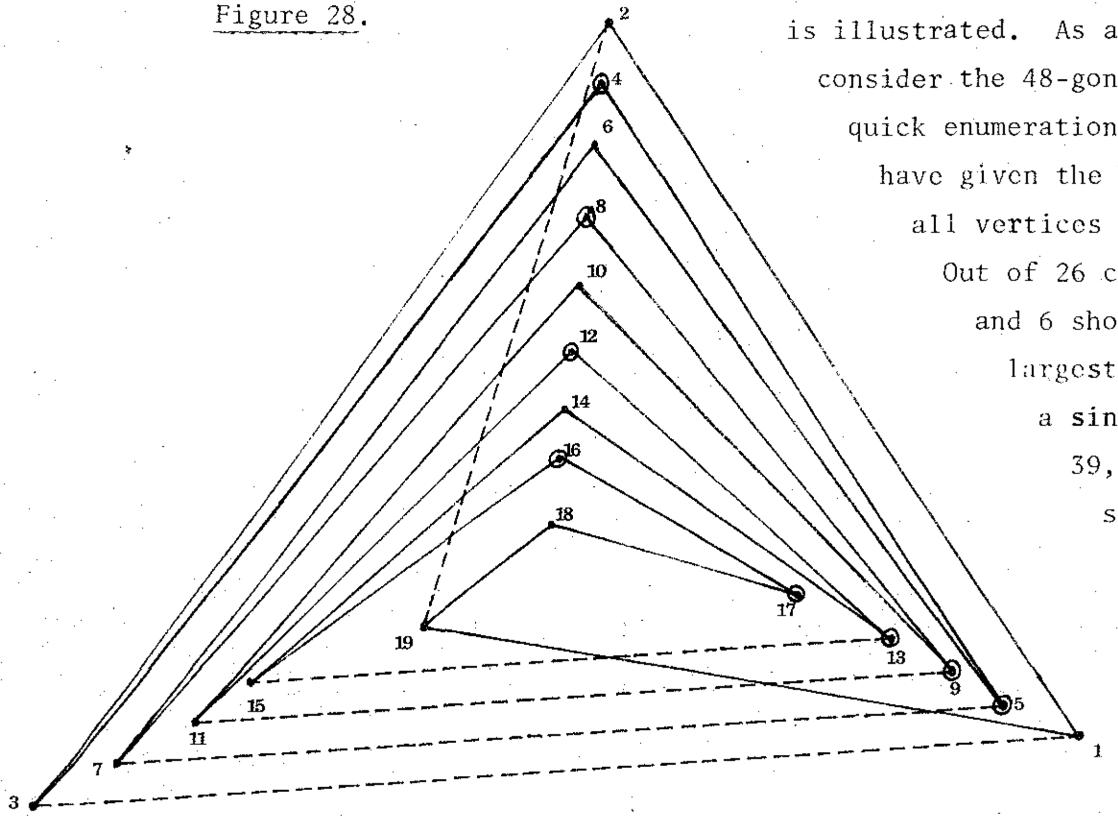
Figure 27 below shows a corresponding polygon.

Figure 27.



Turning to space requirements, we see that the arrays G, S, and C, and the array $\mathcal{D}$ will take up memory space $O(n)$. The problem lies with the lists t_k and u_i, each of which may, in the worst case, take space $O(n^2)$, when summed over all values of the index. Each collection of lists takes up $2m$ memory locations, where m is the total number of *inclusions* (i.e., relations of a vertex being inside a convex triad); and it is possible to construct polygons for which $m = O(n^2)$. For example, Figure 28 illustrates a class of $(4k - 1)$-gons, in which the triad $(4k - 1, 1, 2)$ contains $3(k - 1)$ vertices; while the triads $(1, 2, 3)$, $(5, 6, 7)$, ..., $(4i - 3, 4i - 2, 4i - 1)$, ..., $(4k - 3, 4k - 2, 4k - 1)$ contain respectively $4(k - 1)$, $4(k - 2)$, ..., $4(k - i)$, ..., $4(k - k)$; for a total of $(2k + 3)(k - 1)$ inclusions. The case of $k = 5$ is illustrated. As a more realistic example, consider the 48-gon in Figure 14. Here, a quick enumeration shows that $m = 57$ (we have given the benefit of the doubt to all vertices nearly included in triads). Out of 26 convex triads, 10 are empty and 6 show only one inclusion; the largest number of inclusions in a single triad is 11 in (38, 39, 40). Here, $m < 1.19\,n$; so that $O(n^2)$ behavior is not in evidence. The two sets of lists would require $4m = 228$ memory locations; while a plain $(n \times n)$ array would take up $48^2 = 2304$ memory locations. Returning to the extreme case of Figure 28, we see that the lists would require $4 \times 13 \times 4 = 208$ memory locations, while a simple square array would need $19^2 = 361$: still in favor of the list-structure. Indeed, for all k, $4(2k + 3)(k - 1) < (4k - 1)^2$.

Figure 28.



We can now proceed to modify and refine the algorithms. We first adjust Algorithm 0.

ALGORITHM 0*.

```
0*1   $h \leftarrow 1$;  $M \leftarrow x_1$;
0*2   for $j \leftarrow 1$ to $n$ (step 1), do
0*3       $C(1, j) \leftarrow C(2, j) \leftarrow C(3, j) \leftarrow 0$;   $S(j){:}lp \leftarrow S(j){:}ls \leftarrow$ NIL  {initialize};
0*4       compute the discriminant $\Gamma_j = \gamma(j - 1, j, j + 1)$  {see (12), (20)};
0*5       $G(j) \leftarrow \Gamma_j$;
0*6       if $x_j > M$, then do
0*7           $h \leftarrow j$;  $M \leftarrow x_j$;
0*8       else, if $x_j = M$ and $y_j > y_h$, then $h \leftarrow j$;
0*9   end {for j}
      {By Lemma 4 Corollary, $P_h$ is now an extreme vertex of the polygon $\mathfrak{P}$ and so
       must be convex; if $\Gamma_h > 0$, the polygon is correctly indexed for touring it
       with interior on the left; if $\Gamma_h < 0$, the order must be reversed.}
0*10  $A \leftarrow B \leftarrow \mathcal{A} \leftarrow \mathcal{B} \leftarrow \mathcal{D}{:}lp \leftarrow \mathcal{D}{:}ls \leftarrow ptr \leftarrow z \leftarrow$ NIL;  $p \leftarrow q \leftarrow 0$  {initialize};
0*11  if $G(h) > 0$, then, for $j \leftarrow 1$ to $n$ (step 1), fill_lists  {right ordering};
0*12  else, for $j \leftarrow n$ to 1 (step -1), fill_lists  {wrong ordering of vertices}.
```

In the pseudo-code, multiple assignments are done in the direction of the arrows, from right to left [in A-2 of *append*, this is crucial, since *id*:*ls*:*lp* ← *newcell* is done first, with the old pointer *id*:*ls*, and then *id*:*ls* ← *id*:*ls*:*lp* updates this pointer to its new value; here, it is not so important]; *for* $i \leftarrow$ a *to* b (*step* c) repeats all subsequent material (either a single instruction, or all instructions from *do* to *end*) with i taking successive values a, a + c, a + 2c, ..., a + kc, ..., as long as $(j - \mathrm{b})/\mathrm{c} \leqslant 0$ (c must not be 0), with *no* execution if $(\mathrm{a} - \mathrm{b})/\mathrm{c} > 0$; *if* K *then* will execute all subsequent material (either a single instruction, or all instructions from *do* to either *end* or *else*) *once* only, if and only if K is TRUE; should there be an *else*, all subsequent material (single instruction, or everything from *do* to *end*) will be executed only once, if and only if K is FALSE.

The procedure *fill_lists* is as follows.

```
F-1  append($\mathcal{D}$, NIL, NIL, $j$)  {add a cell referring to $P_j$ at the end of List $\mathcal{D}$};
F-2  if $G(j) > 0$, then increment($\mathcal{A}$, $A$, $p$)  {vertex $P_j$ is convex; add to List $\mathcal{A}$};
F-3  if $G(j) < 0$, then increment($\mathcal{B}$, $B$, $q$)  {vertex $P_j$ is re-entrant; add to $\mathcal{B}$};
F-4  ptr ← $\mathcal{D}$:ls  {ptr now points to new previous-cell}.
```

The procedure *increment*($\mathcal{Z}$, Z, w) is as follows.

```
I-1  if $w = 0$ and ptr ≠ NIL, then $\mathcal{Z}$ ← ptr  {$P_j$ is the first vertex in List $\mathcal{Z}$;
          ptr points to the previous cell; make $\mathcal{Z}$ point to the predecessor of the
          first cell referring to a vertex in the current list};
I-2  if $w = 1$ and $\mathcal{Z}$ = NIL, then z ← ptr  {first cell in List $\mathcal{D}$ is in List $\mathcal{Z}$,
          and current vertex is second in List $\mathcal{Z}$};
I-3  if $w > 0$, then do  {$P_j$ is not the first vertex in List $\mathcal{Z}$};
I-4    if Z ≠ NIL, then Z:cp ← ptr  {Z:cp points to the previous cell};
I-5    Z ← ptr  {Z points to the predecessor of the latest vertex in List $\mathcal{Z}$};
I-6  end {if}
I-7  if ($G(h) > 0$ and $j = n$) or ($G(h) < 0$ and $j = 1$), then do  {end of search}
I-8    if $\mathcal{A}$ = NIL, then do  {first cell in List $\mathcal{D}$ is in List $\mathcal{A}$}
I-9      $\mathcal{A}$ ← $\mathcal{D}$:ls  {last cell is predecessor of first cell in List $\mathcal{A}$};
I-10     $\mathcal{A}$:cp ← z  {$\mathcal{A}$:cp points to predecessor of second cell in List $\mathcal{A}$};
I-11   end {if}
I-12   if $\mathcal{B}$ = NIL, then do  {first cell in List $\mathcal{D}$ is in List $\mathcal{B}$}
I-13     $\mathcal{B}$ ← $\mathcal{D}$:ls  {last cell is predecessor of first cell in List $\mathcal{B}$};
I-14     $\mathcal{B}$:cp ← z  {$\mathcal{B}$:cp points to predecessor of second cell in List $\mathcal{B}$};
I-15   end {if}
I-16   $\mathcal{D}$:ls:lp ← $\mathcal{D}$:lp  {circularize List $\mathcal{D}$; see C-1};
I-17   A:cp ← $\mathcal{A}$  {circularize List $\mathcal{A}$; see C-2};
I-18   B:cp ← $\mathcal{B}$  {circularize List $\mathcal{B}$; see C-3};
I-19 end {if}
I-20 $w$ ← $w + 1$  {$w$ counts vertices in List $\mathcal{Z}$}.
```

On termination of this algorithm, we have a circular linked list of all convex vertices in List $\mathcal{A}$, a circular linked list of all re-entrant vertices in List $\mathcal{B}$, both ordered so as to make a tour of the polygon $\mathfrak{P}$ with its interior on the left, and both incorporated in the circular linked list $\mathcal{D}$ of all active vertices. Apart from the use of linked lists and the considerably greater detail given above than in the earlier algorithms, the only change is that we have not altered the original indices given to the vertices of the polygon.

We can now modify and expand Algorithm 1. The new algorithm will have two parts: first, a setting-up part, which we shall call Algorithm 1*, will form the collections of lists $\mathfrak{t}_k$ and $\mathfrak{u}_i$; then an iterative part will extract successive empty triads: this we shall call Algorithm 3.

ALGORITHM 1*.

```
$ap \leftarrow \mathcal{A}$  {initialize the $\mathcal{A}$-list pointer};
loop
  $h \leftarrow ap{:}x$;  $i \leftarrow ap{:}lp{:}x$;  $j \leftarrow ap{:}lp{:}lp{:}x$  {$P_hP_iP_j$ is convex triad};
  $ap{:}up{:}ls \leftarrow ap{:}up{:}lp \leftarrow$ NIL  {initialize $\mathfrak{u}_i$-list header};
  $mt \leftarrow 0$    {initially suppose the triad is empty};
  $bp \leftarrow \mathcal{B}$  {initialize the $\mathcal{B}$-list pointer};
  if $bp \neq$ NIL, then do
    loop
      $k \leftarrow bp{:}lp{:}x$  {$P_k$ is a re-entrant vertex};
      compute the three discriminants $\gamma_1 = \gamma(h, i, k)$, $\gamma_2 = \gamma(i, j, k)$, and
            $\gamma_3 = \gamma(j, h, k)$  {see (9), (10), (11), (22)};
      if $\gamma_1 > 0$ and $\gamma_2 > 0$ and $\gamma_3 > 0$, then do  {vertex $P_k$ is in triad $P_hP_iP_j$}
        $mt \leftarrow 1$  {i.e., the triad is not empty};
        append($ap{:}up$, $k$)  {add $P_k$ to List $\mathfrak{u}_i$};
        append($S(k)$, $ap{:}up{:}ls$)  {add pointer to new cell in $\mathfrak{u}_i$ to List $\mathfrak{t}_k$};
      end {if}
      $bp \leftarrow bp{:}cp$  {go to next re-entrant vertex};
    until $bp = \mathcal{B}$  {continue to end of List $\mathcal{B}$};
  end {if}
```

Algorithm 1* (continued):

```
if mt = 1, then do  {i.e., triad contains at least one re-entrant vertex}
  bp ← $\mathcal{A}$  {initialize an $\mathcal{A}$-list pointer};
  loop
    k ← bp:lp:x  {$P_k$ is a convex vertex};
    compute the three discriminants $\gamma_1$, $\gamma_2$, and $\gamma_3$;
    if $\gamma_1 > 0$ and $\gamma_2 > 0$ and $\gamma_3 > 0$, then do  {vertex $P_k$ is in triad $P_h P_i P_j$};
      append(ap:up, k)  {add $P_k$ to List $\boldsymbol{u}_i$};
      append(S(k), ap:up:ls)  {add pointer to new cell in $\boldsymbol{u}_i$ to List $\boldsymbol{t}_k$};
    end {if}
    bp ← bp:cp  {go to next convex vertex};
  until bp = $\mathcal{A}$  {continue to end of List $\mathcal{A}$};
end {if}
{List $\boldsymbol{u}_i$ is now complete.}
ap ← ap:cp  {go to next triad};
until ap = $\mathcal{A}$  {continue to end of List $\mathcal{A}$}.
```

Only one new pseudo-code construct appears above; namely, *loop* ... *until* M; which means that the body of ... is repeated so long as, at its end, M is FALSE [this piece of code is therefore necessarily executed at least once].

The entire structure is now complete, and we can proceed to Algorithm 3.

ALGORITHM 3.

```
ap ← $\mathcal{A}$;  r ← 0  {initialize};
loop
  if ap:up:ls = NIL, then do  {i.e., the triad is empty}
    r ← r + 1  {increment position in array C};
    C(1, r) ← h ← ap:x;  C(2, r) ← i ← ap:lp:x;  C(3, r) ← j ← ap:lp:lp:x
          {put the triad $P_h P_i P_j$ into the array C of empty triads};
    bp ← S(i):lp  {initialize a $\boldsymbol{t}_i$-list pointer};
    while bp ≠ NIL, do
      bp:tp:lp ← bp:tp:lp:lp  {delete the cell next after that to which bp:tp
            points [this destroys the value of the corresponding $\boldsymbol{u}_g$:ls
            pointer; but this will not matter]};
      bp ← bp:lp  {go to next cell in $\boldsymbol{t}_i$};
    end {while}
```

To ensure the viability of a full implementation of the third algorithm, a program in 'C' was written and tested, following the procedures outlined above. The fully-annotated program is listed in §7 and four examples of triangulations are given in §8.

Since this algorithm essentially does the same thing as Algorithm 1, we know from Theorem 2 that the procedure will always yield a complete, economical triangulation in a finite number of steps. It remains only to obtain the worst-case order of magnitude of the time taken.

The program is divided into four principal parts:

(1) Preliminary Definitions (pages 47 - 51);
(2) Main Program (pages 52 - 57);
(3) Find Included Vertices (pages 58 - 59);
and (4) Output Lists (pages 60 - 61).

The last of these is concerned with presenting the results, and the time taken in doing so is not a proper part of the timing calculation. The Preliminary Definitions consist of preprocessor instructions and storage declarations, which are used by the compiler and do not affect execution time of the compiled or 'object' code, together with functions,

app_u(h, j), app_t(k, u), del_S(i),
and fill_D(j, G),

which are used in the Main Program. The functions app_u() and app_t() take constant time (they append a single cell to a linked list equipped with a header which points to the last cell); del_S(i) deletes from u-lists all references to P_{i+1}. Since del_S(i) is invoked at most once, for each i, and since the total size of all the u-lists cannot exceed n^2, the time taken by all calls to del_S(i) is definitely no more than $O(n^2)$. Finally, fill_D(), which appends a D_cell to the D-list, adjusting all appropriate D-, A-, and B-pointers, takes constant time. The section titled "Find Included Vertices" consists of the function

find_u(a),

which constructs the u-list for the D_cell pointed to by the pointer a. Each call to this function takes the computation of inclusion conditions for, at worst, every vertex in the D-list (first, the B-list is tested; but then, if a re-entrant vertex is found to be included, the A-list is tested too); so that the expenditure of time is $O(p + q)$, where p is the number of vertices in the A-list and q the number of vertices in the B-list; and this includes $27(p + q)$ a.o., involved in computing three discriminants for each possible included vertex. Of course, p and q will diminish, as each vertex is removed. This estimate is slightly excessive, since somewhat less computation is required for empty triads (only $27q$ a.o.), and 9 or 18 a.o. may suffice (rather than 27 a.o.) to eliminate many vertices.

We may now turn to the Main Program. Input (like output) is not included in the timing calculation. The time required to initialize the D-, A-, and B-lists (essentially Algorithm 0*) is clearly $O(n)$, including $9n$ a.o. to compute the n discriminants. Since Algorithm 1* now calls find_u() for each convex vertex, the total time here is $O(p(p + q)) = O(n^2)$, including at most $27p(p + q) < 27n^2$ a.o. This brings us to Algorithm 3 proper: the elimination of successive empty convex triads. In the worst case, there are no redundant (collinear) vertices at any stage; so that we eliminate triads in $n - 2$ iterations, with $n = p + q$ initially and $p + q$ diminishing by one at each iteration. The search for the next empty triad takes a worst-case time $O(p)$, as we cycle through the A-list; and the calls to del_S() will contribute to a total $O(n^2)$ overall, as has already been explained. In each iteration, two new discriminants must be computed, taking 18 a.o., and there may be, at worst, as many as four calls to find_u(), involving not more than $108(p + q)$ a.o. (if the vertices flanking the vertex to be removed from the apex of the triad in question are thereby made redundant, two more discriminants will change in value, but not in sign, and therefore need not be recomputed). As careful perusal of the program will bear out, all other operations take constant time, for each iteration. It therefore follows that the time for each iteration is $O(p + q)$, including $108(p + q) + 18$ a.o. In sum, the iterations together take time $O(n^2)$, including $54(n^2 + \frac{4}{3}n - \frac{20}{3})$ a.o. With the $9n$ and $27n^2$ above, this yields:

THEOREM 4. *Algorithms 0*, 1*, and 3 together* (i) *always yield a complete, economical triangulation, and* (ii) *take less than*

$$81\,n(n + 1) - 360 = O(n^2) \tag{59}$$

a.o. and $O(n^2)$ other operations to perform.

7. The Program

```
/**************************************************************************
                         PRELIMINARY DEFINITIONS
**************************************************************************/

#include <stdio.h>

     /***    We are given a simple closed polygon P, with vertices
             P(1), P(2), ..., P(n).  For j = 0, 1, 2, ..., n - 1,
             P[0][j] contains the x-component x(j+1), and P[1][j]
             the y-component y(j+1) of the vertex P(j+1).  The
             discriminant (see below) of the triad whose middle
             vertex is P(j+1) is stored in G[j].                      ***/

float P[2][100], G[100];

     /***    gamma(h, i, j) is the discriminant,

                         P(h+1)P(i+1) ^ P(i+1)P(j+1),

             of the triad P(h+1)P(i+1)P(j+1).                         ***/

#define gamma(h, i, j)   (g++,    P[0][i] * (P[1][j] - P[1][h]) \
                                - P[1][i] * (P[0][j] - P[0][h]) \
                                + P[1][h] * P[0][j] - P[0][h] * P[1][j])

     /***    The polygon P has n vertices:  p are convex, q are
             re-entrant, and the rest (if any) are redundant
             (i.e., collinear with their neighbors).  Discriminant
             evaluations are counted in g as they occur.  As empty
             convex triads P(h+1)P(i+1)P(j+1) are found, they are
             stored in the array C:  h in C[0][r], i in C[1][r],
             and j in C[2][r], with r = 0, 1, 2, ...                  ***/

int g = 0, n, p = 0, q = 0, C[3][100];

     /***    The u-lists have identifying pointers in the "up"
             components of cells in the D-list (see below); these
             point to header-cells "head_u" of the form {uf, us},
             with "uf" a pointer pointing to the first, and "us" a
             pointer pointing to the last, "u_cell".  Every u_cell
             = {ul, udex}, where "ul" is a list-pointer, and the
             index "udex" identifies a vertex P(udex + 1) of the
             polygon P, contained inside the convex triad to which
             the D_cell (whose "up" component points to the current
             u-list) refers.  Each u-list has a first cell, of the
             form {ul, udex} = {ul, 0}.                               ***/

struct u_cell { struct u_cell *ul;
                int udex;
              } ;

struct head_u { struct u_cell *uf, *us; } ;
```

```
     /***   malloc(L) allocates a free memory space of length L
            and returns a (character) pointer to it.                   ***/

char *malloc();

     /***   NEW_u returns a pointer to a new u_cell, for addition
            to an existing u_list.  NEW_Hu returns a pointer to
            a new header-cell head_u, for initializing a u-list.       ***/

#define NEW_u   (struct u_cell *) malloc(sizeof(struct u_cell))

#define NEW_Hu   (struct head_u *) malloc(sizeof(struct head_u))

     /***   app_u(h, j) appends a new u_cell {0, j} with index
            "udex" = j to the end of the u_list with identifying
            pointer h.                                                 ***/

app_u(h, j)

  struct head_u *h;
  int j;

  { struct u_cell *u;

    u = h -> us = h -> us -> ul = NEW_u;
    u -> ul = 0;
    u -> udex = j;
  }

     /***   The t-lists have identifying pointers S[k], pointing to
            header-cells "head_t" = {tf, ts}, with "tf" pointing to
            the first, and "ts" to the last, "t_cell".  Every t_cell
            = {tl, tu}, where "tl" is a list-pointer and "tu" points
            to a u_cell, which is the predecessor of a u_cell whose
            index is k (the index of the t-list S[k]).                 ***/

struct t_cell { struct t_cell *tl;
                struct u_cell *tu;
              } ;

struct head_t { struct t_cell *tf, *ts; } *S[100];

     /***   NEW_t returns a pointer to a new t_cell, for addition
            to an existing t_list.  NEW_Ht returns a pointer to
            a new header-cell head_t, for initializing a t-list.       ***/

#define NEW_t   (struct t_cell *) malloc(sizeof(struct t_cell))

#define NEW_Ht   (struct head_t *) malloc(sizeof(struct head_t))
```

```
    /***   app_t(k, u) appends, to the end of the t_list S[k], a
           new t_cell {0, u}, with "tu" = u pointing to the
           predecessor, in some u-list, of a u cell with "index"
           = k.                                                      ***/

app_t(k, u)

  int k;
  struct u_cell *u;

  { struct t_cell *t;

    if (S[k] -> tf == 0) t = S[k] -> ts = S[k] -> tf = NEW_t;
    else                 t = S[k] -> ts = S[k] -> ts -> tl = NEW_t;
    t -> tl = 0;
    t -> tu = u;
  }

    /***   del_S(i) deletes cells referring to vertex P(i+1) from
           all u-lists, using the listing of their predecessors in
           S[i]; then voids S[i].  [NOTE:  Once del_S(i) has been
           used, it is no longer possible to rely on the values of
           the u-list header-pointers d -> up -> us (where d is a
           pointer to any D_cell), since these are not updated by
           del_S(i).]                                                ***/

del_S(i)

  int i;

  { struct t_cell *t;

    t = S[i] -> tf;
    while (t != 0)
      { t -> tu -> ul = t -> tu -> ul -> ul;
        t = t -> tl;
      }
    S[i] -> tf = S[i] -> ts = 0;
  }
```

```
        /***    The "D"-list has identifying pointer D, pointing to the
                first "D_cell".  Each D_cell = {pp, np, f, b, up, index},
                where "pp" is a list-pointer, "np" is a reverse-sense
                list-pointer, "f" and "b" are other pointers to D_cells
                (see below), "up" is the identifying pointer to a u-list
                (see above), and "index" is the index of the vertex
                P(index + 1) of the polygon, to which the D_cell refers.

                The D-list incorporates two other lists, the A-list and
                the B-list. All three of these lists (unlike the u- and
                t-lists) have no header-cells.  The identifying pointer
                of the A-list (which points directly to the first D_cell
                in the A-list) is A, and that of the B-list (which points
                to the first D_cell in the B-list) is B; the pointers AA,
                BB, and DD respectively point to the last D_cells of the
                A-, B-, and D-lists.  The D_cells in the A-list are those
                referring to convex vertices; the D_cells in the B-list
                are those referring to re-entrant vertices.  The "f" and
                "b" pointers are forward and backward list-pointers for
                D_cells of like kind (both in the A-list, or both in the
                B-list).

                When the construction of the A-, B-, and D-lists is
                completed, the list-pointers of the last cells are made
                to point to the first cells of the respective lists,
                making them circular.                                   ***/

struct D_cell { struct D_cell *pp, *np, *f, *b;
                struct head_u *up;
                int index;
              } *ap, *A, *AA, *bp, *B, *BB, *D, *DD, *mtt;

        /***    NEW_D returns a pointer to a new D_cell, for addition
                to the D_list.                                          ***/

#define NEW_D   (struct D_cell *) malloc(sizeof(struct D_cell))
```

```
    /***   fill_D(j, G) appends a D_cell {0, np, 0, b, up, j} to
           the D-list, and increments the A-list if P(j+1) is
           convex, and the B-list if P(j+1) is re-entrant.         ***/

fill_D(j, G)

  int j;
  float G;

  { struct D_cell *d;
    char *malloc();

    d = NEW_D;
    d -> pp = d -> f = 0;
    d -> up = 0;
    d -> index = j;
    if (DD != 0)
      { DD -> pp = d;
        d -> np = DD;
      }
    else
      { d -> np = 0;
        D = d;
      }
    DD = d;

    if (G > 0)

      { if (AA != 0)
          { AA -> f = DD;
            DD -> b = AA;
          }
        else
          { DD -> b = 0;
            A = DD;
          }
        AA = DD;
      }
    if (G < 0)

      { if (BB != 0)
          { BB -> f = DD;
            DD -> b = BB;
          }
        else
          { DD -> b = 0;
            B = DD;
          }
        BB = DD;
      }
  }
```

```
/**************************************************************************
                              MAIN PROGRAM
**************************************************************************/

main()
  { int h, hh, i, j, jj, k, mt, r;
    float x, y;
    struct D_cell *find_u();

     /***   Read in the vertices of the polynomial.                  ***/

    do scanf("%d ", &n); while (n < 3);
    for (i = 0; i < n; i++)
      { scanf("%f %f ", &x, &y);
        P[0][i] = x;
        P[1][i] = y;
      }

     /***   Find the vertex with maximum x-coordinate (if several,
            find that with maximum y-coordinate).  (This is an
            extreme vertex, and so is convex.)  Also compute gamma
            values and initialize the C-array and the t-lists.       ***/

    h = 0; x = P[0][0];
    for (i = 0; i < n; i++)
      { if (P[0][i] > x)
          { h = i;
            x = P[0][i];
          }
        if (P[0][i] == x && P[1][i] > P[1][h]) h = i;
        if (i == n - 1)  G[i] = gamma(n - 2, n - 1, 0);
        else if (i == 0) G[i] = gamma(n - 1, 0, 1);
        else             G[i] = gamma(i - 1, i, i + 1);
        C[0][i] = C[1][i] = C[2][i] = 0;
        S[i] = NEW_Ht;
        S[i] -> tf = S[i] -> ts = 0;
      }

     /***   G[h] is the discriminant of a vertex guaranteed to be
            convex.  Thus, if G[h] < 0 (it cannot vanish), the
            polygon is numbered in the wrong sense (correct sense
            has the interior on the left as we tour the polygon).
            For the correct sense, all discriminants computed
            above must have signs changed.  Count the convex
            vertices in p and the re-entrant vertices in q.          ***/

    x = ((G[h] > 0) ? 1 : (-1));
    for (i = 0; i < n; i++)
      { G[i] = x * G[i];
        if (G[i] > 0)      p++;
        else if (G[i] < 0) q++;
      }
```

```
 /***   Print out the polygon.                                   ***/

printf("Polygon P:  %d vertices; %d convex, %d re-entrant.\n\n",
  n, p, q);
printf("Vertex       x              y          Discriminant     \n\n");
for (i = 0; i < n; i++)
  { printf("P(%3d):  %12.7f   %12.7f   %12.7f      ",
      i+1, P[0][i], P[1][i], G[i]);
    if (G[i] > 0)       printf("convex\n");
    else if (G[i] < 0) printf("re-entrant\n");
    else                printf("redundant (collinear)\n");
  }
printf("\n");

 /***   Initialize all A-, B-, and D-list pointers.              ***/

A = AA = B = BB = D = DD = 0;

 /***   In correct interior-on-left cyclic order, append D_cells
        for each convex or re-entrant vertex to the D-list and
        update A- and B-lists accordingly.                       ***/

if (x > 0) for (i = 0;     i < n;   i++) fill_D(i, G[i]);
else       for (i = n - 1; i > - 1; i--) fill_D(i, G[i]);

 /***   After completing the D-, A, and B- lists, now
        circularize all three lists.                             ***/

DD -> pp = D;
D -> np = DD;
AA -> f = A;
A -> b = AA;
BB -> f = B;
B -> b = BB;

 /***   Examine each convex triad P(h+1)P(i+1)P(j+1) to make up
        a u-list of all contained vertices.  At least one such
        triad must be empty.  find_u returns a pointer to the
        triad it has examined, if that triad is empty; or else
        it returns mtt (the pointer to the last empty triad).    ***/

ap = A;
mtt = 0;
do
  { mtt = find_u(ap);
    ap = ap -> f;
  }
while (ap != A) ;
```

```
 /***   Print out the lists.                                      ***/

LIST();
EMPTY();

 /***   Proceed to search for empty convex triads and remove
        them from the D-list to the C-list.  Position in the
        array C is initialized to r = 0.  We begin at the first
        empty triad in the A-list.                                ***/

r = 0;
AA = mtt -> f;
while (p > 2)

 /***   Search for next empty triad, cycling forward through
        circular A-list.                                          ***/

  { mt = 1;
    h = 0;

    while (mt == 1)
      { if (AA -> up -> uf -> ul == 0)
          { mtt = AA;
            mt = 0;
          }
        else
          { h++;
            AA = AA -> f;
          }
      }

 /***   Put indices h, i, and j of empty convex triad into
        C-list and decrement A-list count p.  Vertex P(i+1)
        will be removed from the D-list.                          ***/

    C[0][r] = h = mtt -> np -> index;
    C[1][r] = i = mtt -> index;
    C[2][r] = j = mtt -> pp -> index;
    p--;

      printf("\n   %3d >>>>  Remove vertex P(%d) from P(%d)P(%d)P(%d)\n",
        r + 1, i + 1, h + 1, i + 1, j + 1);

 /***   Delete cells referring to vertex P(i+1) from all
        u-lists, using the listing of their predecessors in
        S[i]; then void S[i].                                     ***/

    del_S(i);
```

```
/***   Remove P(i+1) from A- and D-lists.                         ***/

   mtt -> pp -> np = mtt -> np;
   mtt -> np -> pp = mtt -> pp;
   if (mtt == D) D = mtt -> pp;
   mtt -> f -> b = mtt -> b;
   mtt -> b -> f = mtt -> f;
   if (mtt == A) A = mtt -> f;
   AA = mtt -> f;

/***   Put old discriminants of adjacent vertices to P(i+1)
       in x and y, and recalculate them without P(i+1).           ***/

   G[i] = 0;
   x = G[h];
   y = G[j];
   hh = mtt -> np -> np -> index;
   jj = mtt -> pp -> pp -> index;
   G[h] = gamma(hh, h, j);
   G[j] = gamma(h, j, jj);

/***   Reconstruct u-lists for any convex adjacent vertices.      ***/

   if (G[h] > 0) find_u(mtt -> np);
   if (G[j] > 0) find_u(mtt -> pp);

/***   Check adjacent vertices for change from re-entrant to
       convex (the reverse is not possible).                      ***/

   if (x < 0 && G[h] >= 0 ¦¦ y < 0 && G[j] >= 0)

/***   Put into ap, bp, AA, and BB pointers to the previous
       convex and re-entrant, and the next convex and
       re-entrant, vertices, respectively.                        ***/

     { ap = mtt -> b;
       ap -> f = AA;
       AA -> b = ap;
       if (x < 0) bp = mtt -> np -> b;
       else       bp = mtt -> pp -> b;
       if (y < 0) BB = mtt -> pp -> f;
       else       BB = mtt -> np -> f;
```

```
/***   Adjust to each side-vertex in turn.                       ***/

       if (x < 0 && G[h] >= 0)
         { q--;

             printf("\n   Vertex P(%3d) changes from re-entrant",
               h + 1);

           if (q == 0) B = 0;
           bp -> f = mtt -> np -> f;
           if (y < 0) mtt -> pp -> b = bp;
           else       BB -> b = bp;
           if (mtt -> np == B) B = mtt -> np -> f;
           if (G[h] > 0)
             { p++;

                 printf(" to convex.\n");

               mtt -> np -> b = ap;
               ap = ap -> f = mtt -> np;
               ap -> f = AA;
               AA -> b = ap;
             }
           else
             { mtt -> np -> np -> pp = mtt -> pp;
               mtt -> pp -> np = mtt -> np -> np;
               if (mtt -> np == D) D = mtt -> pp;

                 printf(" to redundant (collinear).  Remove it.\n");

               if (mtt -> np -> np == ap) find_u(ap);
               if (mtt -> pp == AA) find_u(AA);

/***   Delete cells referring to vertex P(h+1) from all
       u-lists, using the listing of their predecessors in
       S[h]; then void S[h].                                     ***/

               del_S(h);
             }
         }
       else if (x < 0) bp = mtt -> np;
       if (y < 0 && G[j] >= 0)
         { q--;

             printf("\n   Vertex P(%3d) changes from re-entrant",
               j + 1);

           if (q == 0) B = 0;
           bp -> f = BB;
           BB -> b = bp;
           if (mtt -> pp == B) B = BB;
```

```
              if (G[j] > 0)
                { p++;

                    printf(" to convex.\n");

                  find_u(mtt -> pp);
                  mtt -> pp -> f = AA;
                  AA = AA -> b = mtt -> pp;
                  AA -> b = ap;
                  ap -> f = AA;
                }
              else
                { mtt -> pp -> pp -> np = mtt -> pp -> np;
                  mtt -> pp -> np -> pp = mtt -> pp -> pp;
                  if (mtt -> pp == D) D = mtt -> pp -> pp;

                    printf(" to redundant (collinear).  Remove it.\n");

                  if (mtt -> pp -> pp == AA) find_u(AA);
                  if (mtt -> pp -> np == ap) find_u(ap);

   /***   Delete cells referring to vertex P(j+1) from all
          u-lists, using the listing of their predecessors in
          S[j]; then void S[j].                                       ***/

                  del_S(j);
                }
            }
        }

   /***   Increment position in C-array.                              ***/

      r++;
      EMPTY();
    }

  printf("\n%d Discriminants Evaluated:  %d a.o.\n", g, 9 * g);

   /***   Print out the C-array.                                      ***/

  printf("\nArray C of empty convex triads as found by the program.\n\n");
  for (h = 0; h <= r / 13; h++)
    { for (i = 0; i < 3; i++)
        { for (j = 0; j < 13 && (k = 13 * h + j) < r; j++)
            printf("%3d  ", C[i][k] + 1);
          printf("\n");
        }
      printf("\n");
    }
}
```

```
/***************************************************************************
                            FIND INCLUDED VERTICES
***************************************************************************/

struct D_cell *find_u(a)

  struct D_cell *a;

  { int h, i, j, k, mt, app_t(), app_u();
    struct u_cell *u;
    struct head_u *hu;
    struct D_cell *d;

     /***   Initialize an empty u-list for the D-cell pointed to by
            the pointer a.                                          ***/

    hu = a -> up = NEW_Hu;
    u = hu -> uf = hu -> us = NEW_u;
    u -> ul = 0;
    u -> udex = 0;

    h = a -> np -> index;
    i = a -> index;
    j = a -> pp -> index;

     /***   mt is the "empty" flag, initially 0 (empty).            ***/

    mt = 0;

     /***   Examine each re-entrant vertex P(k+1) for inclusion.    ***/

    d = B;
    if (d != 0)
      do
        { k = d -> index;

     /***   Compute the three discriminants; if all three are
            non-negative, then P(k+1) lies in the triad.            ***/

          if (gamma(h, i, k) >= 0 && k != h)
            if (gamma(i, j, k) >= 0 && k != i)
              if (gamma(j, h, k) >= 0 && k != j)
                { mt = 1;
```

```
    /***   Add pointer to last cell in current u-list to t-list at
           S[k]; add k to current u-list.                                ***/

                 app_t(k, a -> up -> us);
                 app_u(a -> up, k);
               }
         d = d -> f;
       }
     while (d != B) ;

    /***   If the triad contains at least one re-entrant vertex,
           it may contain convex vertices also.  If so, examine
           each convex vertex P(k+1) for inclusion.                      ***/

   if (mt == 1)
     { d = A;
       do
         { k = d -> index;
           if (gamma(h, i, k) >= 0 && k != h)
             if (gamma(i, j, k) >= 0 && k != i)
               if (gamma(j, h, k) >= 0 && k != j)
                 { app_t(k, a -> up -> us);
                   app_u(a -> up, k);
                 }
           d = d -> f;
         }
       while (d != A) ;
     }

    /***   If still mt = 0, the triad is empty.   If so, return a
           pointer to the triad.                                         ***/

   if (mt == 0) return(a);
   else         return(mtt);
}
```

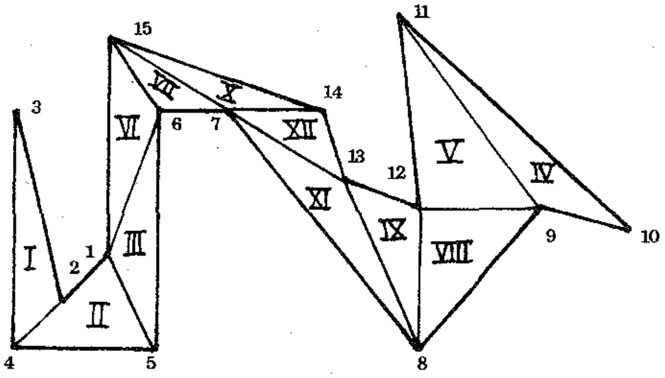
```
/***************************************************************************
                              OUTPUT LISTS
***************************************************************************/

LIST()

  { int v;
    struct u_cell *u;
    struct head_u *h;
    struct D_cell *d;

    printf("\nA-list:  %3d vertices:  { ", p);
    v = 0;
    d = A;
    do
      { if (v % 8 == 6) printf("\n            ");
        printf("P(%3d) ", (d -> index) + 1);
        v++;
        d = d -> f;
      }
    while (d != A) ;
    printf("}\n");

    if (q > 0)
      { printf("\nB-list:  %3d vertices:  { ", q);
        v = 0;
        d = B;
        do
          { if (v % 8 == 6) printf("\n            ");
            printf("P(%3d) ", (d -> index) + 1);
            v++;
            d = d -> f;
          }
        while (d != B) ;
        printf("}\n");
      }
  }
```

```
EMPTY()

  { int v;
    struct D_cell *d;

    printf("\nEmpty Convex Triads at  { ");
    v = 0;
    d = A;
    do
      { if(d -> up -> uf -> ul == 0)
          { if (v % 8 == 6) printf("\n                ");
            printf("P(%3d) ", d -> index + 1);
            v++;
          }
        d = d -> f;
      }
    while (d != A) ;
    printf("}\n");
  }
```

8. Examples

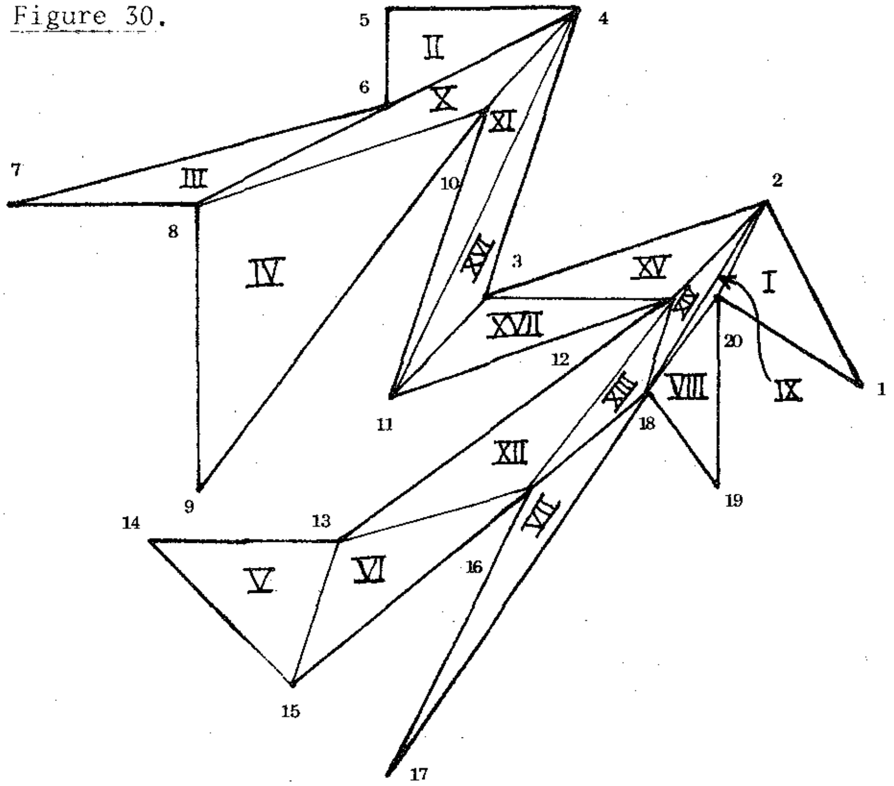
Four examples were run. The first was the 15-gon in Figure 27, whose resulting triangulation is shown in Figure 29 below. The computer output is shown on pages 63 - 64. Twelve triangles are formed (n - 3; because the vertex P_2 becomes collinear after the first triad ($P_2P_3P_4$) is removed. The total number of a.o. (i.e., nine times the number of discriminants evaluated) comes to 4,257, as compared with the bound (59) of 81 × 15 × 16 - 360 = 19,080 (a factor of almost 5 too big; compare the factors of almost 9 and about 3 in Algorithms 1 and 2).

Figure 29.



The second example was the 20-gon shown in Figure 30. The computer output is given on pages 65 - 67. Seventeen triangles are formed (n - 3; because the vertex P_6 becomes collinear after the removal of the first three triads ($P_{20}P_1P_2$, $P_4P_5P_6$, and $P_6P_7P_8$). This time, the total number of a.o. is 6,579, as compared with the bound (59) of 81 × 20 × 21 - 360 = 33,660 (a factor of about 5 too big).

Figure 30.



Example 1.

```
Polygon P:  15 vertices; 8 convex, 7 re-entrant.

Vertex        x              y            Discriminant

P(  1):    4.0000000     4.0000000    -18.0000000     re-entrant
P(  2):    2.0000000     2.0000000    -20.0000000     re-entrant
P(  3):    0.0000000    10.0000000     20.0000000     convex
P(  4):    0.0000000     0.0000000     60.0000000     convex
P(  5):    6.0000000     0.0000000     60.0000000     convex
P(  6):    6.0000000    10.0000000    -30.0000000     re-entrant
P(  7):    9.0000000    10.0000000    -30.0000000     re-entrant
P(  8):   17.0000000     0.0000000     98.0000000     convex
P(  9):   22.0000000     6.0000000    -29.0000000     re-entrant
P( 10):   26.0000000     5.0000000     26.0000000     convex
P( 11):   16.0000000    14.0000000     71.0000000     convex
P( 12):   17.0000000     6.0000000    -23.0000000     re-entrant
P( 13):   14.0000000     7.0000000     -8.0000000     re-entrant
P( 14):   13.0000000    10.0000000     24.0000000     convex
P( 15):    4.0000000    13.0000000     81.0000000     convex


A-list:    8 vertices:  { P(  3) P(  4) P(  5) P(  8) P( 10) P( 11)
            P( 14) P( 15) }

B-list:    7 vertices:  { P(  1) P(  2) P(  6) P(  7) P(  9) P( 12)
            P( 13) }

Empty Convex Triads at  { P(  3) P( 10) }

      1 >>>>  Remove vertex P(3) from P(2)P(3)P(4)

   Vertex P(  2) changes from re-entrant to redundant (collinear).  Remove it.

Empty Convex Triads at  { P(  4) P( 10) }

      2 >>>>  Remove vertex P(4) from P(1)P(4)P(5)

   Vertex P(  1) changes from re-entrant to convex.

Empty Convex Triads at  { P(  5) P( 10) P(  1) }

      3 >>>>  Remove vertex P(5) from P(1)P(5)P(6)

Empty Convex Triads at  { P( 10) P(  1) }

      4 >>>>  Remove vertex P(10) from P(9)P(10)P(11)

   Vertex P(  9) changes from re-entrant to convex.

Empty Convex Triads at  { P( 11) P(  1) }
```

```
     5 >>>>  Remove vertex P(11) from P(9)P(11)P(12)

Empty Convex Triads at  { P(  9) P(  1) }

     6 >>>>  Remove vertex P(1) from P(15)P(1)P(6)

   Vertex P(  6) changes from re-entrant to convex.

Empty Convex Triads at  { P(  9) P(  6) }

     7 >>>>  Remove vertex P(6) from P(15)P(6)P(7)

Empty Convex Triads at  { P(  9) P( 15) }

     8 >>>>  Remove vertex P(9) from P(8)P(9)P(12)

   Vertex P( 12) changes from re-entrant to convex.

Empty Convex Triads at  { P( 12) P( 15) }

     9 >>>>  Remove vertex P(12) from P(8)P(12)P(13)

Empty Convex Triads at  { P(  8) P( 15) }

    10 >>>>  Remove vertex P(15) from P(14)P(15)P(7)

   Vertex P(  7) changes from re-entrant to convex.

Empty Convex Triads at  { P(  8) P( 14) }

    11 >>>>  Remove vertex P(8) from P(7)P(8)P(13)

   Vertex P( 13) changes from re-entrant to convex.

Empty Convex Triads at  { P( 14) P(  7) P( 13) }

    12 >>>>  Remove vertex P(13) from P(7)P(13)P(14)

Empty Convex Triads at  { P( 14) P(  7) }

473 Discriminants Evaluated:  4257 a.o.

Array C of empty convex triads as found by the program.

   2    1    1    9    9   15   15    8    8   14    7    7
   3    4    5   10   11    1    6    9   12   15    8   13
   4    5    6   11   12    6    7   12   13    7   13   14
```

Example 2.

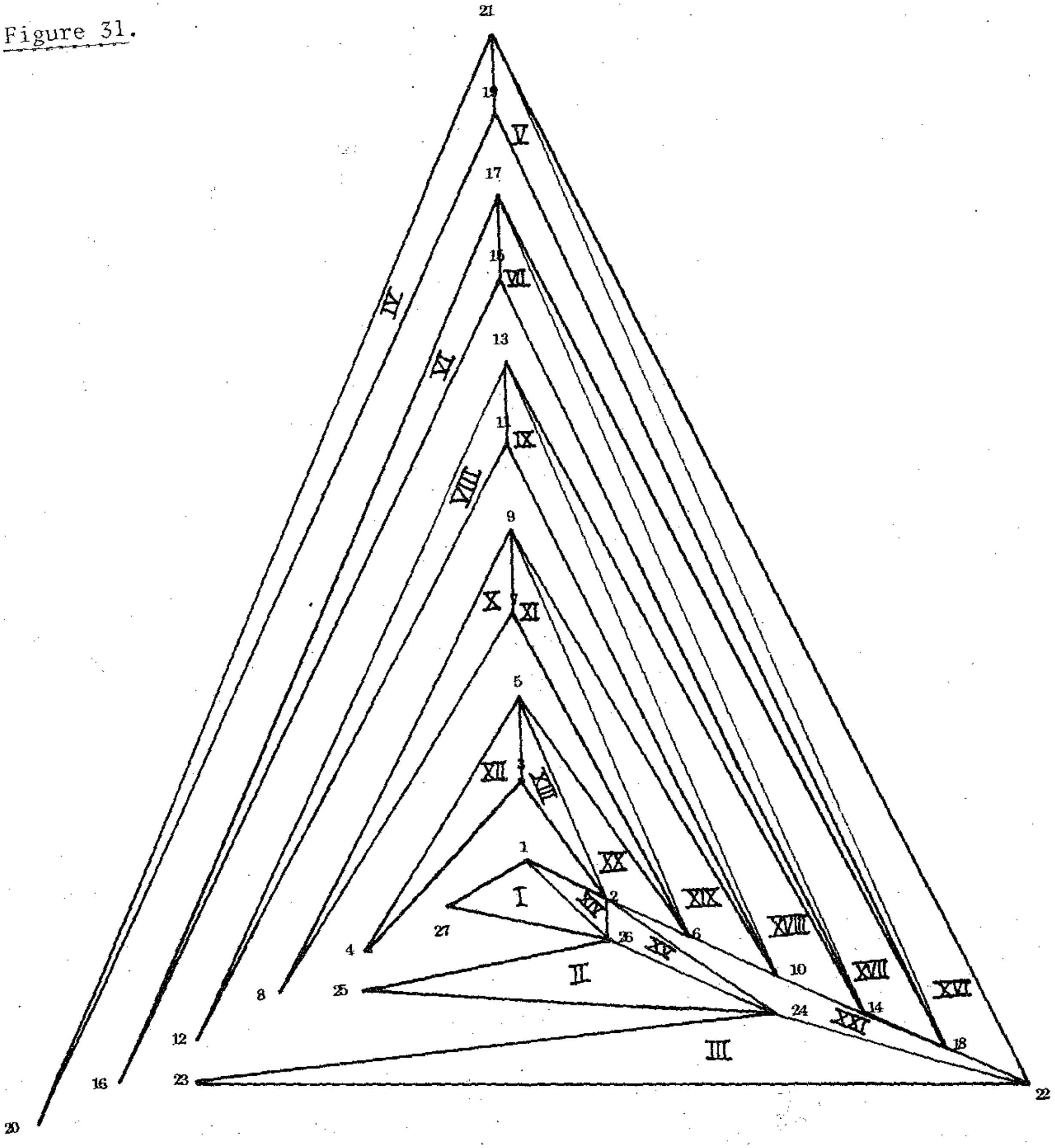
```
Polygon P:  20 vertices; 11 convex, 9 re-entrant.

Vertex        x              y           Discriminant

P(  1):    5.0000000      0.0000000      2.0000000     convex
P(  2):    4.0000000      2.0000000      7.0000000     convex
P(  3):    1.0000000      1.0000000     -8.0000000     re-entrant
P(  4):    2.0000000      4.0000000      6.0000000     convex
P(  5):    0.0000000      4.0000000      2.0000000     convex
P(  6):    0.0000000      3.0000000     -4.0000000     re-entrant
P(  7):   -4.0000000      2.0000000      2.0000000     convex
P(  8):   -2.0000000      2.0000000     -6.0000000     re-entrant
P(  9):   -2.0000000     -1.0000000      9.0000000     convex
P( 10):    1.0000000      3.0000000     -5.0000000     re-entrant
P( 11):    0.0000000      0.0000000      8.0000000     convex
P( 12):    3.0000000      1.0000000     -4.0000000     re-entrant
P( 13):   -0.5000000     -1.5000000     -5.0000000     re-entrant
P( 14):   -2.5000000     -1.5000000      3.0000000     convex
P( 15):   -1.0000000     -3.0000000      6.7500000     convex
P( 16):    1.5000000     -1.0000000     -4.5000000     re-entrant
P( 17):    0.0000000     -4.0000000      2.2500000     convex
P( 18):    2.7500000      0.0000000     -5.7500000     re-entrant
P( 19):    3.5000000     -1.0000000      1.5000000     convex
P( 20):    3.5000000      1.0000000     -3.0000000     re-entrant


A-list:   11 vertices:  { P(  1) P(  2) P(  4) P(  5) P(  7) P(  9)
           P( 11) P( 14) P( 15) P( 17) P( 19) }

B-list:    9 vertices:  { P(  3) P(  6) P(  8) P( 10) P( 12) P( 13)
           P( 16) P( 18) P( 20) }

Empty Convex Triads at  { P(  1) P(  5) P(  7) P(  9) P( 14) P( 17)
           P( 19) }

     1 >>>>  Remove vertex P(1) from P(20)P(1)P(2)

Empty Convex Triads at  { P(  5) P(  7) P(  9) P( 14) P( 17) P( 19) }

     2 >>>>  Remove vertex P(5) from P(4)P(5)P(6)

Empty Convex Triads at  { P(  7) P(  9) P( 14) P( 17) P( 19) }

     3 >>>>  Remove vertex P(7) from P(6)P(7)P(8)

   Vertex P(  6) changes from re-entrant to redundant (collinear).  Remove it.

   Vertex P(  8) changes from re-entrant to convex.

Empty Convex Triads at  { P(  9) P( 14) P( 17) P( 19) }
```

```
     4 >>>>  Remove vertex P(9) from P(8)P(9)P(10)
Empty Convex Triads at  { P(  8) P( 14) P( 17) P( 19) }
     5 >>>>  Remove vertex P(14) from P(13)P(14)P(15)
  Vertex P( 13) changes from re-entrant to convex.
Empty Convex Triads at  { P(  8) P( 13) P( 15) P( 17) P( 19) }
     6 >>>>  Remove vertex P(15) from P(13)P(15)P(16)
Empty Convex Triads at  { P(  8) P( 13) P( 17) P( 19) }
     7 >>>>  Remove vertex P(17) from P(16)P(17)P(18)
  Vertex P( 16) changes from re-entrant to convex.
Empty Convex Triads at  { P(  8) P( 13) P( 16) P( 19) }
     8 >>>>  Remove vertex P(19) from P(18)P(19)P(20)
  Vertex P( 18) changes from re-entrant to convex.
  Vertex P( 20) changes from re-entrant to convex.
Empty Convex Triads at  { P(  8) P( 13) P( 16) P( 18) P( 20) }
     9 >>>>  Remove vertex P(20) from P(18)P(20)P(2)
Empty Convex Triads at  { P(  8) P( 13) P( 16) P( 18) }
    10 >>>>  Remove vertex P(8) from P(4)P(8)P(10)
  Vertex P( 10) changes from re-entrant to convex.
Empty Convex Triads at  { P(  4) P( 10) P( 13) P( 16) P( 18) }
    11 >>>>  Remove vertex P(10) from P(4)P(10)P(11)
Empty Convex Triads at  { P(  4) P( 13) P( 16) P( 18) }
    12 >>>>  Remove vertex P(13) from P(12)P(13)P(16)
Empty Convex Triads at  { P(  4) P( 16) P( 18) }
    13 >>>>  Remove vertex P(16) from P(12)P(16)P(18)
Empty Convex Triads at  { P(  4) P( 18) }
```

```
    14 >>>>  Remove vertex P(18) from P(12)P(18)P(2)

   Vertex P( 12) changes from re-entrant to convex.

Empty Convex Triads at  { P(  2) P(  4) P( 12) }

    15 >>>>  Remove vertex P(2) from P(12)P(2)P(3)

Empty Convex Triads at  { P(  4) P( 12) }

    16 >>>>  Remove vertex P(4) from P(3)P(4)P(11)

   Vertex P(  3) changes from re-entrant to convex.

Empty Convex Triads at  { P( 11) P( 12) P(  3) }

    17 >>>>  Remove vertex P(11) from P(3)P(11)P(12)

Empty Convex Triads at  { P( 12) P(  3) }

731 Discriminants Evaluated:  6579 a.o.

Array C of empty convex triads as found by the program.

 20    4    6    8   13   13   16   18   18    4    4   12   12
  1    5    7    9   14   15   17   19   20    8   10   13   16
  2    6    8   10   15   16   18   20    2   10   11   16   18

 12   12    3    3
 18    2    4   11
  2    3   11   12
```

The third example was a variant on the highly-involuted 19-gon shown in Figure 28. This was a 27-gon, shown triangulated in Figure 31. The computer output is shown on pages 69 - 72. Twenty-one triads are formed ($n - 6$; because the vertices P_{18}, P_{14}, P_{10}, and P_6 are successively removed upon becoming collinear). The total number of a.o. is 11,367, as compared with the bound (59) of 81 × 27 × 28 - 360 = 60,876 (too big by a factor of about 5).

Figure 31.



Example 3.

```
Polygon P:  27 vertices; 15 convex, 12 re-entrant.

Vertex         x                y           Discriminant

P(  1):     0.0000000      0.0000000      4.0000000      convex
P(  2):     2.0000000     -1.0000000     -4.0000000      re-entrant
P(  3):     0.0000000      2.0000000    -20.0000000      re-entrant
P(  4):    -4.0000000     -2.0000000      8.0000000      convex
P(  5):     0.0000000      4.0000000     48.0000000      convex
P(  6):     4.0000000     -2.0000000     -8.0000000      re-entrant
P(  7):     0.0000000      6.0000000    -84.0000000      re-entrant
P(  8):    -6.0000000     -3.0000000     12.0000000      convex
P(  9):     0.0000000      8.0000000    132.0000000      convex
P( 10):     6.0000000     -3.0000000    -12.0000000      re-entrant
P( 11):     0.0000000     10.0000000   -188.0000000      re-entrant
P( 12):    -8.0000000     -4.0000000     16.0000000      convex
P( 13):     0.0000000     12.0000000    256.0000000      convex
P( 14):     8.0000000     -4.0000000    -16.0000000      re-entrant
P( 15):     0.0000000     14.0000000   -332.0000000      re-entrant
P( 16):   -10.0000000     -5.0000000     20.0000000      convex
P( 17):     0.0000000     16.0000000    420.0000000      convex
P( 18):    10.0000000     -5.0000000    -20.0000000      re-entrant
P( 19):     0.0000000     18.0000000   -516.0000000      re-entrant
P( 20):   -12.0000000     -6.0000000     24.0000000      convex
P( 21):     0.0000000     20.0000000    624.0000000      convex
P( 22):    12.0000000     -6.0000000    508.0000000      convex
P( 23):    -8.0000000     -5.0000000     34.0000000      convex
P( 24):     6.0000000     -4.0000000    -24.0000000      re-entrant
P( 25):    -4.0000000     -3.0000000     16.0000000      convex
P( 26):     2.0000000     -2.0000000    -10.0000000      re-entrant
P( 27):    -2.0000000     -1.0000000      6.0000000      convex


A-list:   15 vertices:  { P( 27) P( 25) P( 23) P( 22) P( 21) P( 20)
            P( 17) P( 16) P( 13) P( 12) P(  9) P(  8) P(  5) P(  4)
            P(  1) }

B-list:   12 vertices:  { P( 26) P( 24) P( 19) P( 18) P( 15) P( 14)
            P( 11) P( 10) P(  7) P(  6) P(  3) P(  2) }

Empty Convex Triads at  { P( 27) P( 25) P( 23) P( 20) P( 16) P( 12)
            P(  8) P(  4) P(  1) }

   1 >>>>  Remove vertex P(27) from P(1)P(27)P(26)

Empty Convex Triads at  { P( 25) P( 23) P( 20) P( 16) P( 12) P(  8)
            P(  4) P(  1) }
```

```
       2 >>>>  Remove vertex P(25) from P(26)P(25)P(24)

    Vertex P( 26) changes from re-entrant to convex.

 Empty Convex Triads at  { P( 23) P( 20) P( 16) P( 12) P(  8) P(  4)
            P(  1) P( 26) }

       3 >>>>  Remove vertex P(23) from P(24)P(23)P(22)

    Vertex P( 24) changes from re-entrant to convex.

 Empty Convex Triads at  { P( 20) P( 16) P( 12) P(  8) P(  4) P(  1)
            P( 26) P( 24) }

       4 >>>>  Remove vertex P(20) from P(21)P(20)P(19)

    Vertex P( 19) changes from re-entrant to convex.

 Empty Convex Triads at  { P( 21) P( 19) P( 16) P( 12) P(  8) P(  4)
            P(  1) P( 26) P( 24) }

       5 >>>>  Remove vertex P(19) from P(21)P(19)P(18)

 Empty Convex Triads at  { P( 21) P( 16) P( 12) P(  8) P(  4) P(  1)
            P( 26) P( 24) }

       6 >>>>  Remove vertex P(16) from P(17)P(16)P(15)

    Vertex P( 15) changes from re-entrant to convex.

 Empty Convex Triads at  { P( 21) P( 17) P( 15) P( 12) P(  8) P(  4)
            P(  1) P( 26) P( 24) }

       7 >>>>  Remove vertex P(15) from P(17)P(15)P(14)

 Empty Convex Triads at  { P( 21) P( 17) P( 12) P(  8) P(  4) P(  1)
            P( 26) P( 24) }

       8 >>>>  Remove vertex P(12) from P(13)P(12)P(11)

    Vertex P( 11) changes from re-entrant to convex.

 Empty Convex Triads at  { P( 21) P( 17) P( 13) P( 11) P(  8) P(  4)
            P(  1) P( 26) P( 24) }

       9 >>>>  Remove vertex P(11) from P(13)P(11)P(10)

 Empty Convex Triads at  { P( 21) P( 17) P( 13) P(  8) P(  4) P(  1)
            P( 26) P( 24) }
```

```
    10 >>>>  Remove vertex P(8) from P(9)P(8)P(7)

  Vertex P(  7) changes from re-entrant to convex.

Empty Convex Triads at  { P( 21) P( 17) P( 13) P(  9) P(  7) P(  4)
                 P(  1) P( 26) P( 24) }

    11 >>>>  Remove vertex P(7) from P(9)P(7)P(6)

Empty Convex Triads at  { P( 21) P( 17) P( 13) P(  9) P(  4) P(  1)
                 P( 26) P( 24) }

    12 >>>>  Remove vertex P(4) from P(5)P(4)P(3)

  Vertex P(  3) changes from re-entrant to convex.

Empty Convex Triads at  { P( 21) P( 17) P( 13) P(  9) P(  5) P(  3)
                 P(  1) P( 26) P( 24) }

    13 >>>>  Remove vertex P(3) from P(5)P(3)P(2)

Empty Convex Triads at  { P( 21) P( 17) P( 13) P(  9) P(  5) P(  1)
                 P( 26) P( 24) }

    14 >>>>  Remove vertex P(1) from P(2)P(1)P(26)

Empty Convex Triads at  { P( 21) P( 17) P( 13) P(  9) P(  5) P( 26)
                 P( 24) }

    15 >>>>  Remove vertex P(26) from P(2)P(26)P(24)

  Vertex P(  2) changes from re-entrant to convex.

Empty Convex Triads at  { P( 21) P( 17) P( 13) P(  9) P(  5) }

    16 >>>>  Remove vertex P(21) from P(22)P(21)P(18)

Empty Convex Triads at  { P( 22) P( 17) P( 13) P(  9) P(  5) }

    17 >>>>  Remove vertex P(17) from P(18)P(17)P(14)

  Vertex P( 18) changes from re-entrant to redundant (collinear).  Remove it.

Empty Convex Triads at  { P( 22) P( 13) P(  9) P(  5) }

    18 >>>>  Remove vertex P(13) from P(14)P(13)P(10)

  Vertex P( 14) changes from re-entrant to redundant (collinear).  Remove it.

Empty Convex Triads at  { P( 22) P(  9) P(  5) }
```

```
    19 >>>>  Remove vertex P(9) from P(10)P(9)P(6)

   Vertex P( 10) changes from re-entrant to redundant (collinear).  Remove it.

Empty Convex Triads at  { P( 22) P(  5) }

    20 >>>>  Remove vertex P(5) from P(6)P(5)P(2)

   Vertex P(  6) changes from re-entrant to redundant (collinear).  Remove it.

Empty Convex Triads at  { P( 22) P(  2) }

    21 >>>>  Remove vertex P(2) from P(22)P(2)P(24)

Empty Convex Triads at  { P( 22) }

1263 Discriminants Evaluated:  11367 a.o.

Array C of empty convex triads as found by the program.

  1   26   24   21   21   17   17   13   13    9    9    5    5
 27   25   23   20   19   16   15   12   11    8    7    4    3
 26   24   22   19   18   15   14   11   10    7    6    3    2

  2    2   22   18   14   10    6   22
  1   26   21   17   13    9    5    2
 26   24   18   14   10    6    2   24
```

The final example was the 48-gon treated earlier and shown in Figure 14. The computer output for this is shown on pages 74 - 80. Forty-five (n - 3) triads are formed (P_{32} becoming redundant). The algorithm took 36,306 a.o. to complete. The corresponding bound (59) is 81 × 48 × 49 - 360 = 190,152 (again about five times too big). Algorithm 1 took 28,107 a.o. and Algorithm 2 took 9,900 a.o. to complete, for the same polygon.

Despite this last, at first sight unfavorable, comparison, it is important to realize that Algorithm 3 is preferable to Algorithm 1. First, we see that the asymptotic behavior of the former is (by (13)) $\frac{9}{4}n^3$, while that of the latter is (by (59)) $81\,n^2$; so that a crossover around $n = 36$ might be expected, with the third algorithm preferable for greater values of n. (More precisely, the bounds (13) and (59) cross over at $n = 39$.) Secondly, we see that Algorithm 1 repeatedly tests each convex triad for the inclusion of at least one re-entrant vertex. The worst case occurs when (i) $p + q - 1$ triads must be tested for each empty triad found, (ii) q re-entrant vertices must be tested to find one that is included in any given triad, and (iii) q remains as large as possible, i.e., $p = 3$, at every stage; and this is extremely unlikely to occur. On the other hand, Algorithm 3 maintains u-lists of all (both re-entrant and convex) vertices included in each convex triad; so that, while the worst case surpasses the worst case for Algorithm 1 at $n = 39$, it is clear that the probable situation must be closer to the worst case, here. Very roughly speaking, we could expect factors $\frac{1}{4}$, $\frac{1}{2}$, and $\frac{1}{2}$ to enter in (i), (ii), (iii) above; for a ratio of actual to worst-case a.o. of about $\frac{1}{16}$. The actual ratio observed is $28{,}107/241{,}734 = \frac{1}{8.6}$. In Algorithm 3, we bound $p(p + q)$ with n^2, for a probable factor of perhaps $\frac{1}{2}$ on $\frac{1}{3}$ of the total bound (59), and the remaining $\frac{2}{3}$ of the bound assumes four calls to find_u(), when perhaps two are nearer to the truth; and, in testing for inclusion, on average, only two and not three discriminants need be computed, so the factor here is about $\frac{1}{3}$; for a net factor of $\frac{7}{18}$. The actual ratio observed is $36{,}306/190{,}152 = \frac{7}{36.7}$. Combining our estimates, we would expect Algorithm 3 to compare with Algorithm 1 about six times less favorably than is indicated by the bounds; combining the observed ratios for our 48-gon, the number is about two. The crossover point would then be $n = 75$ (factor, 2) to $n = 219$ (factor, 6).

Example 4.

```
Polygon P:  48 vertices; 26 convex, 22 re-entrant.

Vertex          x              y          Discriminant

P(  1):     2.5000000      3.5000000      -6.7500000       re-entrant
P(  2):     3.5000000      1.5000000      -4.5000000       re-entrant
P(  3):     2.0000000      0.0000000      -9.7500000       re-entrant
P(  4):    -2.0000000      2.5000000      11.2500000       convex
P(  5):    -2.5000000      0.0000000      -3.2500000       re-entrant
P(  6):    -3.5000000      1.5000000       0.5000000       convex
P(  7):    -2.5000000     -0.5000000       2.7500000       convex
P(  8):    -1.5000000      0.2500000       1.5000000       convex
P(  9):    -1.5000000      1.7500000      -6.0000000       re-entrant
P( 10):     2.5000000     -0.7500000       5.8750000       convex
P( 11):     3.2500000      0.2500000      -1.1875000       re-entrant
P( 12):     3.5000000     -1.0000000       1.3750000       convex
P( 13):     4.2500000      0.7500000       0.9375000       convex
P( 14):     3.5000000      0.2500000      -1.0625000       re-entrant
P( 15):     4.0000000      2.0000000       3.6250000       convex
P( 16):     2.5000000      4.0000000       7.1250000       convex
P( 17):     0.2500000      2.2500000      -0.6250000       re-entrant
P( 18):     1.5000000      3.5000000       0.6250000       convex
P( 19):     0.0000000      2.5000000      -2.0000000       re-entrant
P( 20):    -0.5000000      3.5000000      -4.0000000       re-entrant
P( 21):     3.0000000      4.5000000     -12.5000000       re-entrant
P( 22):     5.0000000      1.5000000      -9.5000000       re-entrant
P( 23):     3.5000000     -1.0000000       1.5000000       convex
P( 24):     3.5000000     -2.0000000      -0.7500000       re-entrant
P( 25):     2.7500000     -1.2500000      -0.1875000       re-entrant
P( 26):     3.2500000     -1.5000000       0.3125000       convex
P( 27):    -1.0000000      1.2500000      14.5625000       convex
P( 28):    -0.5000000     -2.5000000      -0.5625000       re-entrant
P( 29):    -0.7500000     -1.7500000       0.3750000       convex
P( 30):    -1.2500000     -1.7500000       0.3750000       convex
P( 31):    -1.2500000     -2.5000000       0.3750000       convex
P( 32):    -0.7500000     -2.7500000      -0.1875000       re-entrant
P( 33):    -1.0000000     -3.0000000      -1.5625000       re-entrant
P( 34):    -3.7500000      0.5000000       2.0000000       convex
P( 35):    -2.7500000     -1.5000000      -1.7500000       re-entrant
P( 36):    -4.0000000     -0.7500000       1.6875000       convex
P( 37):     0.0000000     -4.5000000      18.1250000       convex
P( 38):    -0.5000000      0.5000000     -18.5000000       re-entrant
P( 39):     3.5000000     -2.5000000      22.0000000       convex
P( 40):     5.5000000      1.5000000      18.0000000       convex
P( 41):     3.0000000      5.5000000      15.7500000       convex
P( 42):     0.0000000      4.0000000      -9.0000000       re-entrant
P( 43):    -2.0000000      6.0000000       9.0000000       convex
P( 44):    -4.5000000      4.0000000       4.7500000       convex
P( 45):    -4.0000000      2.5000000       0.7500000       convex
P( 46):    -4.0000000      4.0000000      -3.0000000       re-entrant
P( 47):    -2.0000000      5.0000000      -8.5000000       re-entrant
P( 48):     0.0000000      1.7500000      11.6250000       convex
```

```
A-list:   26 vertices:  { P(  4) P(  6) P(  7) P(  8) P( 10) P( 12)
              P( 13) P( 15) P( 16) P( 18) P( 23) P( 26) P( 27) P( 29)
              P( 30) P( 31) P( 34) P( 36) P( 37) P( 39) P( 40) P( 41)
              P( 43) P( 44) P( 45) P( 48) }

B-list:   22 vertices:  { P(  1) P(  2) P(  3) P(  5) P(  9) P( 11)
              P( 14) P( 17) P( 19) P( 20) P( 21) P( 22) P( 24) P( 25)
              P( 28) P( 32) P( 33) P( 35) P( 38) P( 42) P( 46) P( 47) }

Empty Convex Triads at  { P(  6) P(  8) P( 13) P( 18) P( 23) P( 26)
              P( 29) P( 30) P( 31) P( 34) P( 36) P( 45) }

       1 >>>>  Remove vertex P(6) from P(5)P(6)P(7)

     Vertex P(  5) changes from re-entrant to convex.

Empty Convex Triads at  { P(  5) P(  7) P(  8) P( 13) P( 18) P( 23)
              P( 26) P( 29) P( 30) P( 31) P( 34) P( 36) P( 45) }

       2 >>>>  Remove vertex P(7) from P(5)P(7)P(8)

Empty Convex Triads at  { P(  5) P(  8) P( 13) P( 18) P( 23) P( 26)
              P( 29) P( 30) P( 31) P( 34) P( 36) P( 45) }

       3 >>>>  Remove vertex P(8) from P(5)P(8)P(9)

Empty Convex Triads at  { P(  5) P( 13) P( 18) P( 23) P( 26) P( 29)
              P( 30) P( 31) P( 34) P( 36) P( 45) }

       4 >>>>  Remove vertex P(13) from P(12)P(13)P(14)

Empty Convex Triads at  { P(  5) P( 12) P( 18) P( 23) P( 26) P( 29)
              P( 30) P( 31) P( 34) P( 36) P( 45) }

       5 >>>>  Remove vertex P(18) from P(17)P(18)P(19)

Empty Convex Triads at  { P(  5) P( 12) P( 23) P( 26) P( 29) P( 30)
              P( 31) P( 34) P( 36) P( 45) }

       6 >>>>  Remove vertex P(23) from P(22)P(23)P(24)

Empty Convex Triads at  { P(  5) P( 12) P( 26) P( 29) P( 30) P( 31)
              P( 34) P( 36) P( 45) }

       7 >>>>  Remove vertex P(26) from P(25)P(26)P(27)

     Vertex P( 25) changes from re-entrant to convex.

Empty Convex Triads at  { P(  5) P( 12) P( 25) P( 29) P( 30) P( 31)
              P( 34) P( 36) P( 45) }
```

```
     8 >>>>  Remove vertex P(29) from P(28)P(29)P(30)

 Empty Convex Triads at  { P(  5) P( 12) P( 25) P( 30) P( 31) P( 34)
          P( 36) P( 45) }

     9 >>>>  Remove vertex P(30) from P(28)P(30)P(31)

 Empty Convex Triads at  { P(  5) P( 12) P( 25) P( 31) P( 34) P( 36)
          P( 45) }

    10 >>>>  Remove vertex P(31) from P(28)P(31)P(32)

   Vertex P( 32) changes from re-entrant to redundant (collinear).  Remove it.

 Empty Convex Triads at  { P(  5) P( 12) P( 25) P( 34) P( 36) P( 45) }

    11 >>>>  Remove vertex P(34) from P(33)P(34)P(35)

   Vertex P( 35) changes from re-entrant to convex.

 Empty Convex Triads at  { P(  5) P( 12) P( 25) P( 35) P( 36) P( 45) }

    12 >>>>  Remove vertex P(35) from P(33)P(35)P(36)

 Empty Convex Triads at  { P(  5) P( 12) P( 25) P( 36) P( 45) }

    13 >>>>  Remove vertex P(36) from P(33)P(36)P(37)

   Vertex P( 33) changes from re-entrant to convex.

 Empty Convex Triads at  { P(  5) P( 12) P( 25) P( 33) P( 45) }

    14 >>>>  Remove vertex P(45) from P(44)P(45)P(46)

   Vertex P( 46) changes from re-entrant to convex.

 Empty Convex Triads at  { P(  5) P( 12) P( 25) P( 33) P( 44) P( 46) }

    15 >>>>  Remove vertex P(46) from P(44)P(46)P(47)

 Empty Convex Triads at  { P(  5) P( 12) P( 25) P( 33) P( 44) }

    16 >>>>  Remove vertex P(5) from P(4)P(5)P(9)

   Vertex P(  9) changes from re-entrant to convex.

 Empty Convex Triads at  { P(  4) P(  9) P( 12) P( 25) P( 33) P( 44) }

    17 >>>>  Remove vertex P(9) from P(4)P(9)P(10)

 Empty Convex Triads at  { P(  4) P( 12) P( 25) P( 33) P( 44) }
```

```
    18 >>>>  Remove vertex P(12) from P(11)P(12)P(14)

   Vertex P( 14) changes from re-entrant to convex.

Empty Convex Triads at  { P(  4) P( 14) P( 25) P( 33) P( 44) }

    19 >>>>  Remove vertex P(14) from P(11)P(14)P(15)

   Vertex P( 11) changes from re-entrant to convex.

Empty Convex Triads at  { P(  4) P( 11) P( 25) P( 33) P( 44) }

    20 >>>>  Remove vertex P(25) from P(24)P(25)P(27)

Empty Convex Triads at  { P(  4) P( 11) P( 33) P( 44) }

    21 >>>>  Remove vertex P(33) from P(28)P(33)P(37)

   Vertex P( 28) changes from re-entrant to convex.

Empty Convex Triads at  { P(  4) P( 11) P( 28) P( 37) P( 44) }

    22 >>>>  Remove vertex P(37) from P(28)P(37)P(38)

Empty Convex Triads at  { P(  4) P( 11) P( 28) P( 44) }

    23 >>>>  Remove vertex P(44) from P(43)P(44)P(47)

   Vertex P( 47) changes from re-entrant to convex.

Empty Convex Triads at  { P(  4) P( 11) P( 28) P( 43) P( 47) }

    24 >>>>  Remove vertex P(47) from P(43)P(47)P(48)

Empty Convex Triads at  { P(  4) P( 11) P( 28) }

    25 >>>>  Remove vertex P(4) from P(3)P(4)P(10)

   Vertex P(  3) changes from re-entrant to convex.

Empty Convex Triads at  { P( 10) P( 11) P( 28) P(  3) }

    26 >>>>  Remove vertex P(10) from P(3)P(10)P(11)

Empty Convex Triads at  { P( 28) P(  3) }

    27 >>>>  Remove vertex P(28) from P(27)P(28)P(38)

   Vertex P( 38) changes from re-entrant to convex.

Empty Convex Triads at  { P( 27) P( 38) P(  3) }
```

```
    28 >>>>  Remove vertex P(38) from P(27)P(38)P(39)

Empty Convex Triads at  { P( 27) P(  3) }

    29 >>>>  Remove vertex P(3) from P(2)P(3)P(11)

Empty Convex Triads at  { P( 11) P( 27) }

    30 >>>>  Remove vertex P(11) from P(2)P(11)P(15)

  Vertex P(  2) changes from re-entrant to convex.

Empty Convex Triads at  { P( 15) P( 27) P(  2) }

    31 >>>>  Remove vertex P(15) from P(2)P(15)P(16)

Empty Convex Triads at  { P( 27) P(  2) }

    32 >>>>  Remove vertex P(27) from P(24)P(27)P(39)

  Vertex P( 24) changes from re-entrant to convex.

Empty Convex Triads at  { P( 24) P( 39) P(  2) }

    33 >>>>  Remove vertex P(39) from P(24)P(39)P(40)

Empty Convex Triads at  { P( 24) P(  2) }

    34 >>>>  Remove vertex P(2) from P(1)P(2)P(16)

  Vertex P(  1) changes from re-entrant to convex.

Empty Convex Triads at  { P( 16) P( 24) P(  1) }

    35 >>>>  Remove vertex P(16) from P(1)P(16)P(17)

Empty Convex Triads at  { P( 24) P(  1) }

    36 >>>>  Remove vertex P(24) from P(22)P(24)P(40)

  Vertex P( 22) changes from re-entrant to convex.

Empty Convex Triads at  { P( 40) P(  1) P( 22) }

    37 >>>>  Remove vertex P(40) from P(22)P(40)P(41)

Empty Convex Triads at  { P(  1) P( 22) }
```

```
    38 >>>>  Remove vertex P(1) from P(48)P(1)P(17)

   Vertex P( 17) changes from re-entrant to convex.

Empty Convex Triads at  { P( 17) P( 22) }

    39 >>>>  Remove vertex P(17) from P(48)P(17)P(19)

   Vertex P( 19) changes from re-entrant to convex.

Empty Convex Triads at  { P( 48) P( 19) P( 22) }

    40 >>>>  Remove vertex P(19) from P(48)P(19)P(20)

Empty Convex Triads at  { P( 48) P( 22) }

    41 >>>>  Remove vertex P(22) from P(21)P(22)P(41)

   Vertex P( 21) changes from re-entrant to convex.

Empty Convex Triads at  { P( 41) P( 48) P( 21) }

    42 >>>>  Remove vertex P(41) from P(21)P(41)P(42)

Empty Convex Triads at  { P( 48) P( 21) }

    43 >>>>  Remove vertex P(48) from P(43)P(48)P(20)

   Vertex P( 20) changes from re-entrant to convex.

Empty Convex Triads at  { P( 43) P( 21) }

    44 >>>>  Remove vertex P(21) from P(20)P(21)P(42)

   Vertex P( 42) changes from re-entrant to convex.

Empty Convex Triads at  { P( 43) P( 20) P( 42) }

    45 >>>>  Remove vertex P(42) from P(20)P(42)P(43)

Empty Convex Triads at  { P( 43) P( 20) }
```

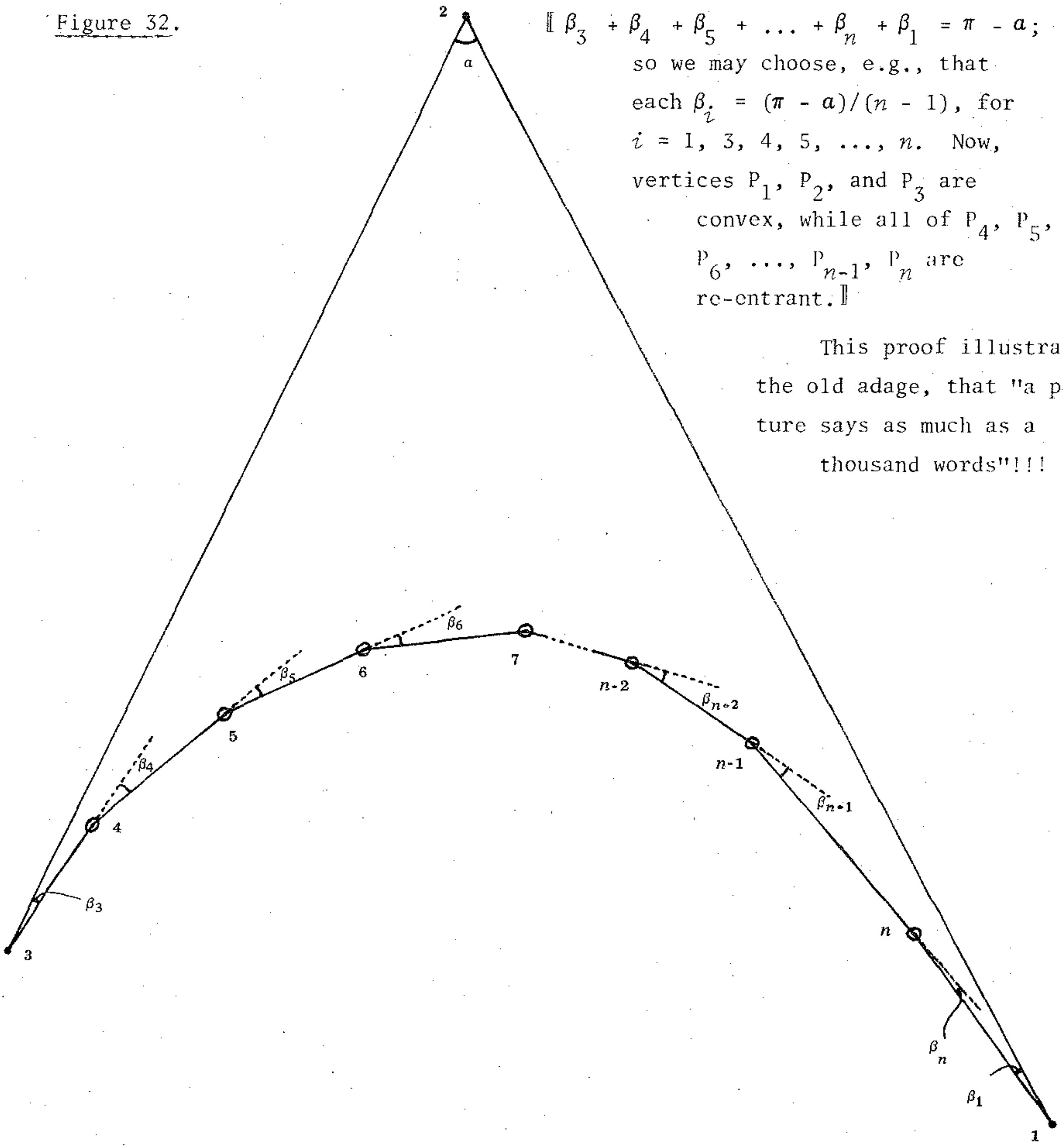
```
4034 Discriminants Evaluated:  36306 a.o.

Array C of empty convex triads as found by the program.

  5   5   5  12  17  22  25  28  28  28  33  33  33
  6   7   8  13  18  23  26  29  30  31  34  35  36
  7   8   9  14  19  24  27  30  31  32  35  36  37

 44  44   4   4  11  11  24  28  28  43  43   3   3
 45  46   5   9  12  14  25  33  37  44  47   4  10
 46  47   9  10  14  15  27  37  38  47  48  10  11

 27  27   2   2   2  24  24   1   1  22  22  48  48
 28  38   3  11  15  27  39   2  16  24  40   1  17
 38  39  11  15  16  39  40  16  17  40  41  17  19

 48  21  21  43  20  20
 19  22  41  48  21  42
 20  41  42  20  42  43
```

9. Maximally Re-Entrant Polygons

As a final note, we add the following result, as a caution against the thought that the computational timing bounds given in the theorems are grossly exaggerated.

<u>LEMMA 13</u>. *The bound in Lemma 5 is tight: there are polygons of any number of vertices $n \geqslant 3$ with only three convex vertices.*

<u>Figure 32</u>.



⟦ $\beta_3 + \beta_4 + \beta_5 + \ldots + \beta_n + \beta_1 = \pi - \alpha$; so we may choose, e.g., that each $\beta_i = (\pi - \alpha)/(n - 1)$, for $i = 1, 3, 4, 5, \ldots, n$. Now, vertices P_1, P_2, and P_3 are convex, while all of P_4, P_5, P_6, $\ldots$, P_{n-1}, P_n are re-entrant. ⟧

This proof illustrates the old adage, that "a picture says as much as a thousand words"!!!

10. Acknowledgement

I wish to thank Dr George C. Clark of the Harris Corporation, Melbourne, Florida, for bringing this problem to my attention, and for several stimulating discussions. The problem arose in seeking an efficient way to fill irregular polygonal shapes, given an efficient and fast triangle-filling command, as part of computer graphics involved in the automation of VLSI design ("C.A.D.")

I also thank Dr Henry Fuchs of The University of North Carolina for encouraging me to reconsider Algorithm 1, in a way that led to Algorithm 3.

Chapel Hill, North Carolina.