A Simple and Efficient Algorithm for Sorting the Intersection Points between a Jordan Curve and a Line

Eduard Sojka
Department of Computer Science, Technical University of Ostrava,
tr.17.listopadu, 708 33 Ostrava-Poruba, Czech Republic
E-mail: eduard.sojka@vsb.cz

Abstract: In this paper, we focus on the Jordan sorting problem: Given N intersection points of a Jordan curve with the x-axis in the order in which they occur along the curve. The task is to sort these points into the order in which they occur along the x-axis. Contrary to general sorting whose solution (in the algebraic decision-tree model of computation) requires θ(N log N) time in the worst case, the Jordan sorting problem can be solved in θ(N) time. The linear worst-case time algorithms for Jordan sorting were proposed by Hoffman et al., and by Fung et al. Unfortunately, both these algorithms are rather complicated, which makes them difficult to use in practice. In this paper, we propose and analyse a simple algorithm for Jordan sorting. Although the worst-case time complexity of this algorithm is O(N log N), we show that the worst time is achieved only for special inputs. For most inputs, a better performance can be expected. We also show that for a certain class of inputs which may be of practical interest, the algorithm runs even in O(N) expected time. We believe that for many practical applications, the algorithm may be more advantageous than rather complicated worst-case time optimal algorithms. Our main result is the analysis of this otherwise rather straightforward algorithm.

Keywords: Computational geometry, Jordan sorting, polygon clipping.

1. Introduction

Problem 1 (*Jordan sorting*). Given N intersection points of a Jordan curve (Jordan curve is a homeomorphic image of a circle) with the x-axis in the order in which they occur along the curve, sort these points into the order in which they occur along the x-axis (Fig. 1.1a).

In the algebraic decision-tree model of computation, the worst-case time complexity of general sorting is θ(N log N). Contrary to general sorting, the time complexity of Jordan sorting is only θ(N). Hoffman et al. [Hoffman 86] proposed an O(N) worst-case time algorithm for solving the Jordan sorting problem and polygon clipping. In their algorithm, they used a sophisticated data structure, the *level-linked search tree*. Later, Fung et al. [Fung 90] devised another similar O(N) time algorithm. In this algorithm, they replaced the level linked search tree with the *heterogeneous finger tree*. Although their algorithm is somewhat simpler than that proposed by Hoffman, it is still rather complicated, which makes the algorithm difficult to use in practice.

In this paper, we propose and analyse a simple algorithm for Jordan sorting. The underlying idea of the algorithm was outlined in [Hoffman 86], the authors, however, did not study this algorithm in detail, and focused on the O(N) algorithm. Although the worst-case time complexity of the algorithm we propose is O(N log N), we show that the worst time is achieved only for special inputs. For most inputs, a better running time can be expected. We also show that for a certain class of random inputs which may be of practical interest, the algorithm runs even in O(N) expected time. For Jordan sorting problem, the performance of the algorithm is thus better than the performance of the fast general-purpose sorting algorithms. We believe that for many applications, our algorithm may be more advantageous than rather complicated worst-case time optimal algorithms.

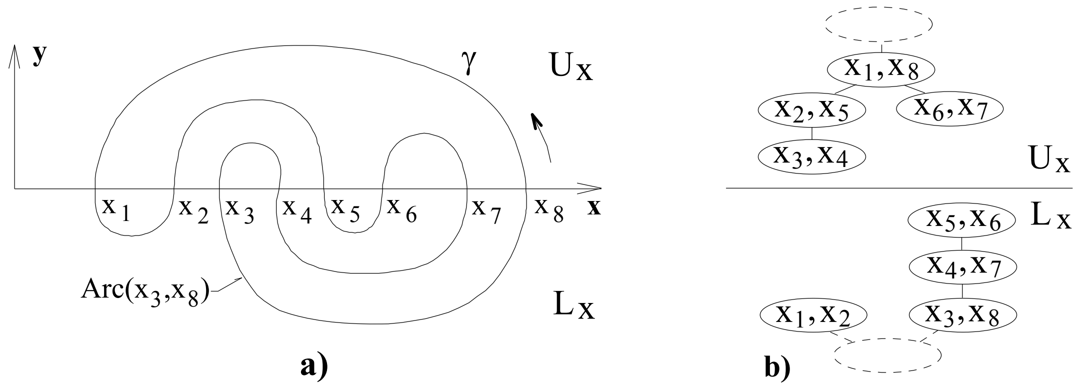

Figure 1.1. a) *Cutting a Jordan curve by the x-axis;* $x_8,x_1,x_2,x_5,x_6,x_7,x_4,x_3$ *is an example of the input sequence for Jordan sorting.* ***b)*** *The corresponding Hasse diagram of the relation of enclosing. The dotted nodes correspond to the additional dummy pair* $\{-\infty,\infty\}$.

The algorithms that solve the Jordan sorting problem can be used in a variety of practical applications. The most straightforward application is in cutting and clipping a simple polygon by a line. The solution to this problem can be divided into the following three steps: (i) Go along the boundary of the polygon and find the intersections between this boundary and the line. (ii) Find the order in which the intersections that were found in the previous step appear along the line. (iii) Assemble the boundaries of the resulting polygons. Although sorting intersection points is the most difficult (in the sense of time complexity) step of this problem, the majority of authors do not discuss this step (e.g., [Liang 83], [Sutherland 74]).

The paper is organised as follows. In Section 2, we explain the basic concepts and the needed terminology. The algorithm is described and analysed in Section 3. Section 4 is a conclusion.

2. Preliminaries

Although the term *Jordan curve* is usually used for closed curves, the algorithm we present does not require this property. It only requires curves without self-intersection points (an open Jordan curve is a homeomorphic image of a line segment). For clarity and brevity of presentation, we exclude such intersection points in which the x-axis is tangent to the curve. This restriction influences neither the principle of the algorithm nor its time complexity. Without loss of generality we also suppose that in the first given intersection point, the curve passes from the lower to the upper half-plane determined by the x-axis.

Let γ be a Jordan curve (Fig. 1.1a). The x-axis divides the plane into two half-planes (upper and lower), denoted by U_x and L_x, respectively. We suppose that the curve and the x-axis intersect each other at N intersection points, denoted by $x_1,x_2,\dots, x_N$. For presenting the algorithm, the intersection points are numbered in such a way that the sequence $x_1,x_2,\dots, x_N$ is ordered along the x-axis (Fig. 1.1a). We will use the notation $x_i<x_j$ to express the fact that, along the x-axis, the point x_i precedes the point x_j. The intersection points divide the x-axis into intervals. We will use the term *segment* to refer to such an interval. At the same time, the

intersection points divide the curve into parts which we will call the *arcs*. We will use the notation arc(x_p,x_q) to refer to the arc whose endpoints are x_p and x_q.

The algorithm we propose is based on successively constructing a planar map $\mathfrak{m}(\gamma)$ that corresponds to the given input of the Jordan sorting problem (Fig. 2.1a). The final map $\mathfrak{m}(\gamma)$ contains just N+2 vertices. N vertices correspond to the intersection points $x_1, x_2, ..., x_N$. The remaining two vertices, denoted by x_0 and x_{N+1}, are added on the x-axis such that $x_0 < x_1$ and $x_N < x_{N+1}$. In the map $\mathfrak{m}(\gamma)$, N+1 edges correspond to the segments, and N edges correspond to the arcs of the curve. Furthermore, the vertices x_0, x_{N+1} are connected by the two edges lying in U_x and L_x, respectively (Fig. 2.1a). During the computation, the sequence $\mathfrak{m}^1, \mathfrak{m}^2, ..., \mathfrak{m}^N, \mathfrak{m}(\gamma)$ of the maps is constructed (Fig. 2.2). The computation starts with the map $\mathfrak{m}^1$ containing x_0, x_{N+1}, and the intersection point that was read from the first position of the input sequence. The algorithm then successively processes the remaining intersection points and updates the map. Once $\mathfrak{m}(\gamma)$ is found, the ordered sequence $x_1, x_2, ..., x_N$ is read from this map.

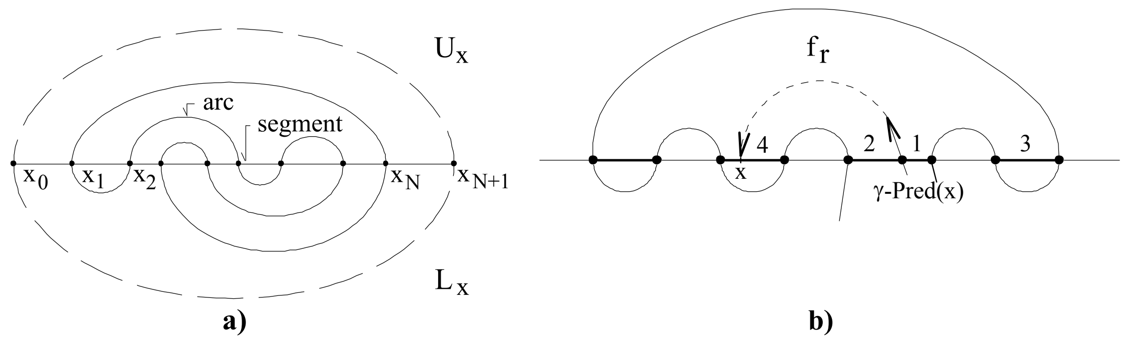

Figure 2.1. ***a)*** *The map* $\mathfrak{m}$ *(γ) corresponding to the final stage of solving the Jordan sorting problem for a certain input.* ***b)*** *Illustration of Lemma 1. The numbers show the order in which the segments are tested in the algorithm. Before splitting, the size of the face* f_r *is* $Size(f_r)=5$.

Consider the situation in which the map $\mathfrak{m}^{k-1}$ has already been constructed. Let x denote the intersection point that is being processed at this moment, and let *γ-Pred*(x) denote the intersection point that was processed immediately before x. The process of updating the map from $\mathfrak{m}^{k-1}$ to $\mathfrak{m}^k$ is based on the following lemma, which follows directly from the fact that the curve does not intersect itself (Fig. 2.1b).

Lemma 1. If in γ-Pred(x), the curve enters a certain face f_r of the map $\mathfrak{m}^{k-1}$, then the intersection point x lies inside a segment which is a part of the boundary of f_r. •

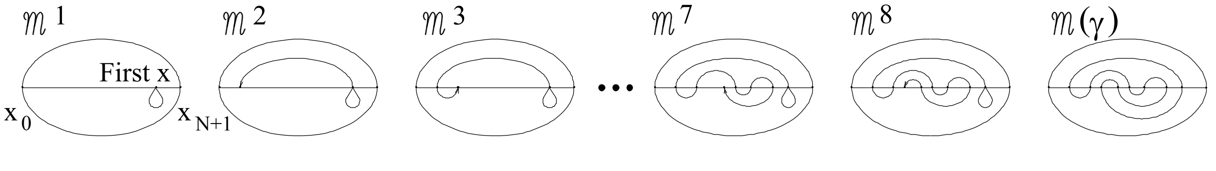

Figure 2.2. *An example of the sequence of the maps that are constructed during the computation.*

In the k-th step of the algorithm, a certain face of the map $\mathfrak{m}^k$ is split, which gives rise to the map $\mathfrak{m}^{k+1}$. We will use the term *transition* to refer to this action. The transition updating the map from $\mathcal{M}^{k-1}$ to $\mathcal{M}^k$ involves: (1) determining the segment containing x and splitting this segment into the two segments, (2) splitting the face f_r containing the arc(γ-Pred(x),x) into the two faces f_u, f_v. We use *Size*(f_i) to denote the number of the segments that lie on the boundary of the face f_i, and we will use the term *size of face* for this number (Fig. 2.1b). Since each transition gives rise to a new segment, the following equation holds

$$\mathrm{Size}(f_u)+\mathrm{Size}(f_v) = \mathrm{Size}(f_r)+1. \qquad (2.1)$$

Note that the size of the faces can be considered at two different moments: (a) at the moment when the face arose by splitting another face, (b) at the moment when the face was chosen for further splitting (if any). In the latter case, the size increases by one, which is due to the fact that during the transition in which a face was chosen for further splitting, the segment in which the curve enters the face was split (Fig. 2.1b). Therefore, if the size of all faces is to be considered just before their splitting, Eq. 2.1 should be adapted as follows

$$\mathrm{Size}(f_u) + \mathrm{Size}(f_v) = \mathrm{Size}(f_r) + 3. \qquad (2.2)$$

In the algorithm, the maps $\mathcal{M}^1,\mathcal{M}^2,\ldots, \mathcal{M}^N,\mathcal{M}(\gamma)$ are represented by a *doubly linked list*. The implementation is supposed to support the inquiry functions *Pred*(x_i), *Succ*(x_i), *Upper*(x_i), *Lower*(x_i). These functions are defined as follows: Pred(x_i)=x_{i-1}, Succ(x_i)=x_{i+1}, Upper(x_i)=x_j if x_i,x_j are connected by the edge representing an arc lying in U_x (if this edge does not exist, Upper(x_i) is not defined), similarly, Lower(x_i)=x_j if x_i,x_j are connected by the edge representing an arc lying in L_x. In addition, the following updating operations are available: The operation *Split segment*(x_i,x_{i+1}, x) splits the segment x_ix_{i+1} into the two segments x_ix and xx_{i+1} (it is assumed that $x_i<x<x_{i+1}$). The operations *Insert upper edge*(x_i,x_j) and *Insert lower edge*(x_i,x_j) create the edge connecting the vertices x_i,x_j and representing the arc arc(x_i,x_j) lying in U_x and L_x, respectively. It is easy to check that all the mentioned operations can be carried out in constant time.

3. The Algorithm

In this section, we will describe a simple algorithm for solving Problem 1. Although the worst-case time complexity of the algorithm is O(N log N), the theorems presented in this section show that the worst time is achieved only for a special input. For most inputs, better running times can be expected. The main goal of this section is to present the analysis of running time for a certain class of random inputs.

Algorithm 1

Input: The sequence of N intersection points between a Jordan curve and the x-axis. The points are ordered as they occur along the curve.

Output: The sequence of intersection points sorted along the x-axis. (If the algorithm is used for solving the problem of curve or polygon cutting or clipping, then also the map $\mathcal{M}(\gamma)$ can be the output).

```
1   Read the first intersection point from the input sequence, and create the initial map $\mathcal{M}^1$ as
    depicted in Fig. 2.2.
    repeat
2       Read the next intersection point x from the input sequence. $\gamma$-Pred(x) now denotes the
        intersection point that was processed immediately before x, $f_r$ denotes the face
        containing the arc arc($\gamma$-Pred(x),x) (this face was identified when the curve entered this
        face in $\gamma$-Pred(x)).
3       Beginning with $\gamma$-Pred(x), go sequentially and simultaneously in both directions along
        the boundary of $f_r$ (Fig. 2.1b, the order in which the segments are tested is important
        for the time complexity of the algorithm) For each segment on this boundary, test
        whether the segment contains x. The process stops when the segment containing x is
        found.
4       Split the segment containing x into the two segments, and insert the new edge
        representing the arc arc($\gamma$-Pred(x),x) into the map (the edges representing the upper
        and the lower arcs alternate as the intersection points are processed).
    until all the intersection points are processed
5   From the map $\mathcal{M}(\gamma)$, read the output sequence of sorted points.
```

In the rest of this section, the analysis of the algorithm will be presented. The time complexity of one transition and the time complexity of the whole algorithm will be measured by the number of tests deciding whether x lies inside a segment (Step 3, Fig. 2.1b). The time complexity of the transition from $\mathcal{M}^{k-1}$ to $\mathcal{M}^{k}$ that splits a certain face f_r into two faces f_u, f_v is thus

$$t = 2\min\{\text{Size}(f_u), \text{Size}(f_v)\} \quad \text{or} \quad t = 2\min\{\text{Size}(f_u), \text{Size}(f_v)\}-1. \tag{3.1}$$

Note that if the size of one of the faces resulting from splitting (either f_u or f_v) is 1, i.e., if the time complexity of the transition is 1 or 2 (Eq. 3.1), then the size of the other resulting face is Size(f_r) (Eq. 2.1), which gives the size Size(f_r)+1 before the next splitting. The transitions with time complexity 1 or 2 thus give rise to the faces with higher sizes.

First, for completeness, we recall several theorems concerning the worst-case time complexity of the algorithm. Since these theorems have already been published, we omit the proofs (they can be found in [Sojka 96]). Theorem 1 shows that in the worst case, Algorithm 1 is at least as good as the fast general-purpose sorting algorithms. The remaining theorems suggest that a lower time complexity can be expected for some inputs.

Theorem 1. In the worst case, Algorithm 1 requires no more than $N(4+\log_2 N)$ tests deciding whether x lies inside a segment. •

Theorem 2. In the best possible case, the number of the tests that are required by Algorithm 1 is N. •

Theorem 3. If during the computation, the size of the faces in the map is bounded, i.e., the size is never greater than a certain constant C, then no more than $N(4+\log_2 C)$ tests deciding whether x lies inside a segment are needed in Algorithm 1. •

Theorem 4. If during the computation, the number of transitions with time complexity 1 or 2 is not greater than D, then no more than $N(3+(D/N)\log_2 N)$ tests deciding whether the intersection point lies inside a segment are needed in Algorithm 1. •

Note that the transitions with time complexity 1 or 2 occur if in the sequence of the points that have already been sorted, x (i.e., the point that is being sorted) directly follows or directly precedes the point that was sorted immediately before x (Fig. 2.1b). Theorem 4 shows that the constant before the term $\log_2 N$ decreases with the decreasing probability of this event.

Theorem 5. Consider the pair of Hasse diagrams corresponding to a given input of size N (Fig. 1.1b). Suppose that in total, the diagrams have Q inner nodes and, therefore, N-Q+2 leaves. Let n_i denote the i-th inner node, let d_i be the degree of this node (i.e., the number of its child nodes), let L_i denote the number of leaves of the sub-tree whose root lies in n_i, and let h_j denote the depth of the j-th leaf in the Hasse diagram. For the total number of tests, denoted by T, the following inequalities hold

$$\text{a)}\quad T \le 2N + 2\sum_{i=1}^{Q} L_i\,, \qquad\qquad \text{b)}\quad T \le 2N + 2\sum_{i=1}^{N-Q} h_j\,. \qquad\qquad \bullet$$

Theorem 5 shows that if either the depth or the width of the Hasse diagrams that correspond to the given input are bounded, i.e., are never higher than a certain constant that does not depend on the size of problem, then Algorithm 1 runs in constant time. Less formally, if the input sequence is "simple" in the sense that one of the mentioned dimensions of the corresponding Hasse diagrams is low, then a short running time can be expected.

From the worst-case study, it follows that the worst running time is achieved only under rather special circumstances consisting in: (a) First, both in U_X and L_X, certain D transitions with time complexity 1 or 2 create a face of size D+2 (the number D leading to the worst possible running time depends on the size of the problem and increases with this size). (b) In each of the remaining transitions, the biggest face in the appropriate half-plane is chosen and split (we suppose that the biggest face can always be chosen). If s denotes the size of this face, then the face must be split into the faces of size $\lceil (s+1)/2 \rceil$, $\lfloor (s+1)/2 \rfloor$. In Fig. 3.1, the process of splitting in the upper half-plane is illustrated by a tree. In this tree, the inner nodes represent the faces that were split during a certain transition. The leaves represent the faces that have not been split and that are thus present in the final map. The process leading to the worst running time is highly organised. Any deviation from the rules described above leads to a lower running time. This suggests the idea that in the case of a random input, a better running time can be expected. Informally, this expectation can be explained by the two facts: (a) It is possible that in the case of a random input, the sizes of the faces in the map will not tend to grow too much, i.e., the probability of existence of big faces will be low. (b) In the case that big faces will appear, there is still a good chance that they will not be split in the worst possible manner.

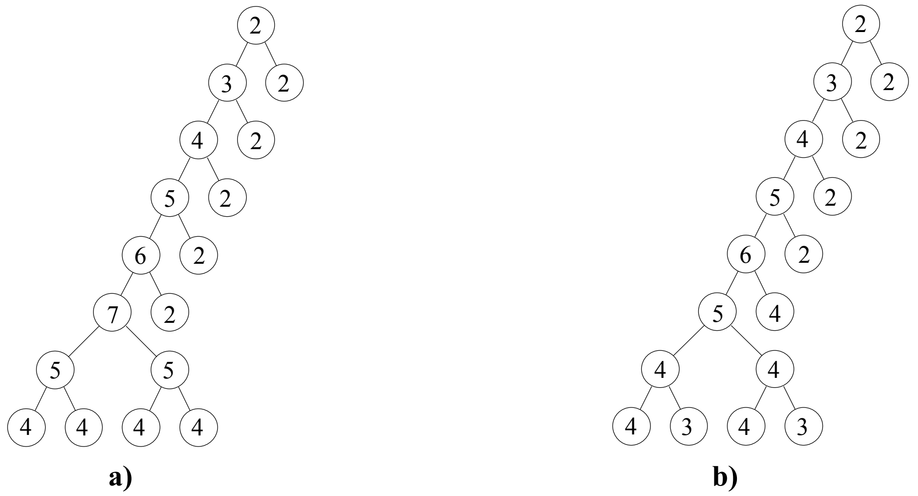

Figure 3.1. a) *An example of the tree that depicts the history of splitting. The numbers inscribed in the nodes are the sizes of the faces (measured at the moment before their further splitting). The tree corresponds to the worst case for M=N/2=8 (T=5*2+7+2*5=27). However, there exist another trees leading to the same time* ***(b)****(T=4*2+6+5+2*4=27).*

To study the time complexity of Algorithm 1 for random inputs, we will introduce a certain model of generating the Jordan sequences. In this model, the curves are supposed to be open. The model is based on the assumption that for each of the segments lying on the boundary of the face containing the arc arc(γ-Pred(x),x) (according to Lemma 1 no other segment can contain x), it makes a sense to think about the probability of the event that x will fall just into this segment. We will introduce the numeration of the segments lying on the boundary of the face as depicted in Fig. 3.2a. Furthermore, we will introduce the function $\pi(s,u)$ expressing the probability that in the face of size s, the intersection point will fall into the segment whose number is u. Different functions $\pi(s,u)$ cause that the generator produces the curves of different "nature". (However, we do not claim that the model can generate the curves of all shapes. The assumption that the functions $\pi(s,u)$ exist and that they do not depend on the size of the problem seems to be restrictive.) Fig. 3.2b shows an example of the curve that was generated by the generator.

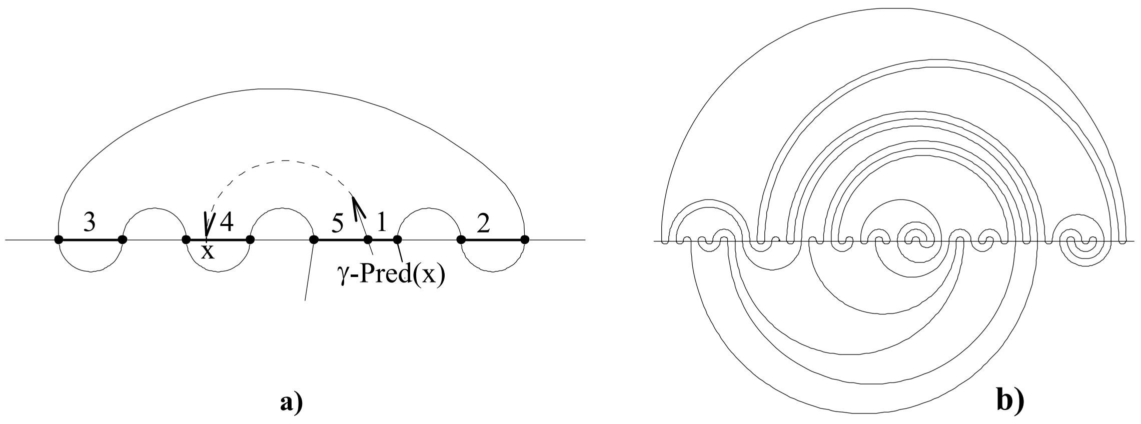

Figure 3.2. ***a)*** *The numeration of the segments lying on the boundary of a face. In this case, $s=5$, $s_1=3$, $s_2=5$ (see further text).* ***b)*** *An example of the curve that was generated randomly. The distribution of probability was $\pi(s,u)=1/s$.*

We will evaluate the time complexity of the transitions in U_x, i.e., we will consider the upper half of the problem. Let M (M=N/2) be the number of these transitions. Consider a face in the map $\mathcal{M}^i$. Each such face is a product of a certain sequence of splits. On the beginning of each such sequence, there is the first initial face in the map $\mathcal{M}^1$. Consider the tree describing the history of splitting (Fig. 3.1). Let k denote the number of splits (splitting levels) that lie on the branch leading from the root to the desired face. The value of k can vary from 0 (k=0 is the level of the initial face in $\mathcal{M}^1$) to M. Let $p_k(s)$ be the probability of the event that after k levels of splitting, the size of the face will be just s. Since the map $\mathcal{M}^1$ contains only one face and since the size of this face is 2, $p_0(s)$ is $p_0(2)=1$, and $p_0(s)=0$ for $s\neq2$. In the rest of this section, by size of face we mean the size that is measured at the moment when the face is chosen for further splitting (see Section 2). Consider the situation in which a face of size s was split, which gave rise to two faces of sizes s_1 and s_2. We will introduce the rule that the face whose size is denoted by s_1, is the face that contains the segment lying to the left of γ-Pred(x) (Fig. 3.2a). The biggest face that can occur after M transitions in U_x (and the same number of transitions in L_x) is a face of size M+2. The face of size s_1 can occur in such a way that in the face whose size is $s\geq s_1-1$, the intersection point x will fall into the segment whose number is $s-s_1+2$. This gives the following recurrent expressions for the probabilities $p_k(s)$:

$$
\begin{aligned}
p_k(2) &= \pi(2,2)p_{k-1}(2) + \pi(3,3)p_{k-1}(3) + \pi(4,4)p_{k-1}(4) + \ldots + \pi(M+2,M+2)p_{k-1}(M+2)\\
p_k(3) &= \pi(2,1)p_{k-1}(2) + \pi(3,2)p_{k-1}(3) + \pi(4,3)p_{k-1}(4) + \ldots + \pi(M+2,M+1)p_{k-1}(M+2)\\
p_k(4) &= \pi(3,1)p_{k-1}(3) + \pi(4,2)p_{k-1}(4) + \ldots + \pi(M+2,M)p_{k-1}(M+2)\\
&\ldots\\
p_k(M+2) &= \pi(M+1,1)p_{k-1}(M+1) + \pi(M+2,2)p_{k-1}(M+2) \qquad (3.2)
\end{aligned}
$$

The probabilities $p_k(2),p_k(3),p_k(4),\ldots$ can be arranged into the vector $\mathbf{p}_k=(p_k(2),p_k(3),p_k(4),\ldots)^T$ (^{T} indicates the transposition). The vector $\mathbf{p}_0$ is $\mathbf{p}_0=(1,0,0,\ldots)$. The values of the function $\pi(s,u)$ can be arranged into the matrix, denoted by Π,

$$\Pi = \begin{bmatrix} \pi(2,2) & \pi(3,3) & \pi(4,4) & \pi(5,5) & \pi(6,6) & \ldots \\ \pi(2,1) & \pi(3,2) & \pi(4,3) & \pi(5,4) & \pi(6,5) & \ldots \\ 0 & \pi(3,1) & \pi(4,2) & \pi(5,3) & \pi(6,4) & \ldots \\ 0 & 0 & \pi(4,1) & \pi(5,2) & \pi(6,3) & \ldots \\ 0 & 0 & 0 & \pi(5,1) & \pi(6,2) & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{bmatrix}. \qquad (3.3)$$

Eq. 3.2 can now be rewritten in the matrix form

$$\mathbf{p}_k = \Pi\mathbf{p}_{k-1}. \qquad (3.4)$$

Substituting for $\mathbf{p}_{k-1}$, $\mathbf{p}_{k-2}$, ..., $\mathbf{p}_1$, we have

$$\mathbf{p}_k = \Pi^k\mathbf{p}_0. \qquad (3.5)$$

Note that the dimensions of the vectors and the matrix can be supposed to be higher than M. The fact that in the case in which only M transitions are to be done, no face whose size is greater than M+2 can occur is expressed by zero probabilities of existence of the faces having greater sizes. The vectors and the matrix of a size Q thus can be used for the problems of size up to Q. Let $\mu_{s,k}$ denote the mean value of the size of the faces arising after the k-th level of splitting. From the definition of mean value, we have

$$\mu_{s,k} = \sum_{s=2}^{Q+2} s p_k(s). \qquad (3.6)$$

We introduce the vector $\mathbf{s} = (2,3,\ldots,Q+2)^T$. Eq. 3.6 can now be rewritten as follows

$$\mu_{s,k} = \mathbf{s}^T\mathbf{p} = \mathbf{s}^T\Pi^k\mathbf{p}_0. \qquad (3.7)$$

We now evaluate the mean value of the time complexity of the transition in the k-th level of splitting. We introduce the function $\tau(s,t)$ expressing the probability that the time complexity of the transition in a face of size s will be t. From the order in which the segments were numbered (Fig. 3.2a) and from the order in which the segments are tested in the algorithm (Fig. 2.1b), it follows that $\tau(s,1)=\pi(s,1)$, $\tau(s,2)=\pi(s,s)$, $\tau(s,3)=\pi(s,2)$, $\tau(s,4)=\pi(s,s-1)$, etc. Let $q_k(t)$ be the probability of the event that in the k-th level of splitting, the time complexity of the transition is just t. Since the transition having time complexity t can arise only in the face whose size is at least t, we have

$$q_k(t) = \sum_{s=t}^{Q+2} \tau(s,t)p_k(s). \qquad (3.8)$$

We use $\mu_{t,k}$ to denote the mean value of the time complexity of one transition in the face that arose in the k-th level of splitting. The definition of mean value gives

$$\mu_{t,k} = \sum_{t=1}^{Q+1} tq_k(t). \tag{3.9}$$

Note that although we expect the faces of size up to Q+2, the face of size Q+2 is never split. The biggest face that is split is the face of size Q+1. This explains the upper bound in the previous sum. We will introduce the vectors $\mathbf{q}_k=(q_k(1),q_k(2), \ldots , q_k(Q+1))^T$, $\mathbf{t}=(1,2,3, \ldots , Q+1)^T$ and the matrix

$$\mathbf{T} = \begin{bmatrix} \pi(2,1) & \pi(3,1) & \pi(4,1) & \pi(5,1) & \pi(6,1) & \ldots \\ \pi(2,2) & \pi(3,3) & \pi(4,4) & \pi(5,5) & \pi(6,6) & \ldots \\ 0 & \pi(3,2) & \pi(4,2) & \pi(5,2) & \pi(6,2) & \ldots \\ 0 & 0 & \pi(4,3) & \pi(5,4) & \pi(6,5) & \ldots \\ 0 & 0 & 0 & \pi(5,3) & \pi(6,3) & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{bmatrix}. \tag{3.10}$$

Eq. 3.8, 3.9 can now be rewritten in the matrix form

$$\mathbf{q}_k = \mathbf{T}\mathbf{p}_k, \tag{3.11}$$

$$\mu_{t,k} = \mathbf{t}^T\mathbf{T}\mathbf{p}_k = \mathbf{t}^T\mathbf{T}\Pi^k\mathbf{p}_0. \tag{3.12}$$

Furthermore, we introduce

$$\mu_{s,\max} = \max_k\{\mu_{s,k}\}, \qquad \mu_{t,\max} = \max_k\{\mu_{t,k}\}. \tag{3.13}$$

Obviously, if the mean value $\mu_{t,\max}$ is bounded (i.e., for the problems of all dimensions lower than a certain constant), then Algorithm 1 runs in linear expected time that does not exceed $N\mu_{t,\max}$. Note that the sequence $\mu_{t,0}, \mu_{t,1}, \mu_{t,2},\ldots$ is not required to converge. The criterion of convergence is stronger. The sequence can be bounded and still need not converge (for example, it may contain the cycles of values). In the following theorem, we will study a special case.

Theorem 6. Suppose that all the segments lying on the boundary of the face containing the arc arc(γ-Pred(x),x) have equal probabilities that they will contain the new intersection point x, then Algorithm 1 runs in 2N expected time.

PROOF. If the probability that the intersection point x will fall into a certain segment is evenly distributed between all possible segments, then the matrices **Π** and **T** are of the form

$$\Pi = \mathbf{T} = \begin{bmatrix} 1/2 & 1/3 & 1/4 & 1/5 & \ldots \\ 1/2 & 1/3 & 1/4 & 1/5 & \ldots \\ 0 & 1/3 & 1/4 & 1/5 & \ldots \\ 0 & 0 & 1/4 & 1/5 & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \end{bmatrix}. \tag{3.14}$$

By making use of Eq. 3.12, $\mu_{t,k}$ is

$$\mu_{t,k} = \frac{2^{k+1}-1}{2^k} = 2 - \frac{1}{2^k}. \tag{3.15}$$

Thus, the sequence $\mu_{t,0}, \mu_{t,1}, \mu_{t,2},\ldots$ is 1, 3/2, 7/4, 15/8, 31/16, 63/32,... . Note that the values in this sequence do not depend on the size of problem. The size of problem is taken into account by considering the appropriate number of terms from this sequence when determining the maximum in Eq. 3.13. It is easily seen that for $k \to \infty$, the value of $\mu_{t,k}$ converges to 2. For completeness, we will also evaluate the mean value $\mu_{s,k}$. By making use of Eq. 3.7, we obtain

$$\mu_{s,k} = 3 - \frac{1}{2^k}. \tag{3.16}$$

The mean value of the face size is never higher than 3, i.e., the probability of the occurrence of big faces is low. This expression elucidates the reason for favourable behaviour of the algorithm and shows the area of practical applicability of the theorem. •

We have implemented and tested the algorithm analysed in this paper. The following table shows the mean value and the variance of the quotient (number of tests)/N for the curves that were generated randomly. The distribution of probability was $\pi(s,u)=1/s$, i.e., constant for every segment lying on the boundary of the face (Theorem 6).

Table 1. *Experimental results for Algorithm 1 and the randomly generated sequences.*

Size of problem (N)	10	100	1000	10000	100000
Number of experiments	10000	10000	1000	1000	500
E{Number of segment tests/N}	1.70	1.97	1.99	2.0	2.0
√Variance of the value above	0.28	0.12	0.04	0.012	0.004

4. Conclusion

In this paper, we have presented and analysed an algorithm for Jordan sorting. Although the algorithm runs in O(N log N) worst-case time, the theorems presented in Section 3 have shown that the worst time is achieved only for a special input. For most inputs, a better running time can be expected. We have also shown that for a certain class of random inputs that may be of practical interest, the algorithm runs even in O(N) expected time. For the Jordan sorting problem, the performance of the algorithm is thus better than the performance of the fast general-purpose sorting algorithms. In comparison with the worst-case time optimal algorithms proposed by Hoffman and by Fung, our algorithm does not use any additional tree data structure and, therefore, it is fairly easy to implement. We believe that due to its simplicity, the algorithm may be useful in a variety of practical applications. The main result presented in this paper is the mathematical analysis of this otherwise rather straightforward algorithm.

References

Fung K.Y. - Nicholl T.M. - Tarjan R.E. - Van Wyk C.J.: Simplified linear-time Jordan sorting and polygon clipping, *Information Processing Letters*, Vol. **35**, (1990), pp 85-92.

Hoffman K. - Mehlhorn K. - Rosenstiehl P. - Tarjan R.E.: Sorting Jordan sequences using level-linked search trees, *Inform. and Control*, Vol. **68** (1986), pp 170-184.

Liang Y.D. - Barsky B.A.: An analysis and algorithm for polygon clipping, *Comm. ACM*, Vol. **26** (1983), pp 868-877.

Shutherland I.E. - Hodgman G.W.: Reentrant polygon clipping, *Comm. ACM,* Vol. **17** (1974), pp 32-42.

Sojka E.: Two simple and efficient algorithms for Jordan sorting and polygon cutting and clipping, In *Compugraphics 96,* Paris 1996, pp 241-252.