RESULTS OF TESTING A NEW ALGORITHM FOR SORTING JORDAN SEQUENCES

Eduard Sojka, Milan Gvoždík
Department of Computer Science, Technical University of Ostrava,
tr.17.listopadu, 708 33 Ostrava-Poruba, Czech Republic
eduard.sojka@vsb.cz, milan.gvozdik.fei@vsb.cz

Abstract: In this paper, a new simple O(NlogN) worst-case time algorithm for sorting Jordan sequences is tested and compared with other known algorithms that solve the same problem. Namely, the O(N) worst-case time algorithm proposed by Fung, the algorithm based on splay trees, and Quick sort are used for comparison. It is shown that for the Jordan sequences of size from 4 up to at least 10000, the new algorithm is significantly faster than the other algorithms. The source code of the new algorithm is enclosed.

1. Introduction

Let us consider a Jordan curve in the plane. Although the term Jordan curve is usually used to mean a closed curve, which is a homeomorphic image of a circle, we extend this term to mean either an open or a closed curve. An open Jordan curve then is a homeomorphic image of a line segment. The problem this paper deals with can now be formulated as follows:

Problem 1 (*Jordan sorting*). Given N intersection points of a Jordan curve 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.

We use the term *Jordan sequence* for the sequence that can appear on the input of Jordan sorting. In computer graphics, the problem of Jordan sorting arises in clipping a simple polygon against a half-plane, and in clipping a line against a simple polygon. The polygon is supposed to be generally non-convex and represented by a list of the vertices of its boundary. More formally:

Problem 2a (*Clipping a simple polygon against a half-plane*). Given a simple polygon P represented by its boundary and a half-plane σ. Find the simple polygons resulting from the intersection $P \cap \sigma$ and represent them by their boundaries.

Problem 2b (*Clipping a line by a simple polygon*). Given a simple polygon P represented by its boundary and a line l. Find the parts of l that lie inside P. Represent each line segment of the result by its startpoint and endpoint.

For computer graphics, these problems are important. The solution to both these problems can be outlined as follows: (1) Process the edges of the boundary of the polygon one after another and find the intersection points between this boundary and the line. (2) Determine the order in which the points occur along the line. (3) Assemble the boundaries of the resulting polygons (Problem 2a only). Step 1 involves primarily geometric computation. It can be carried out in O(N) time, which follows immediately from the fact that there are O(N) intersection points. Step 2 involves sorting the intersection points along the line. From the point of view of the worst-case time complexity, this step is threatening to be the most difficult step (recall that the best general-purpose sorting algorithms run in θ(NlogN) worst-case time). The problem of sorting the intersection points is linear-time equivalent to

Problem 1, since we can represent the line parametrically, and sort the intersection points according to their value of parameter instead of the x-coordinate. Once the intersection points are sorted, the boundaries of the resulting simple polygons can easily be assembled in O(N) time (Step 3).

In this paper, we present a comparison of the new algorithm that was proposed in [Sojka96] and further analysed in [Sojka97] with other known algorithms for Jordan sorting. Throughout this paper, we will use the name *Algorithm 1* to refer to this new algorithm. We will show that for the Jordan sequences of size from 4 up to at least 10000, Algorithm 1 is significantly faster than other known algorithms that solve the same problem. The paper is organised as follows: In Section 2, the known algorithms for Jordan sorting are summarised. The new algorithm is outlined in Section 3. Section 4 brings the results of testing.

2. Existing Algorithms

One way of sorting Jordan sequences obviously is to use the standard general-purpose sorting algorithms. The disadvantage of this approach however is that the running time is longer than is necessary since the general-purpose sorting algorithms run in $\Omega(N\log N)$ worst-case time. There are at least two reasons why to construct special algorithms for sorting Jordan sequences: (1) Special algorithms may be faster. (2) Special algorithms may be able to detect whether or not the sequence that is to be sorted is a Jordan sequence. In practical applications, the latter property is important from the point of view of their robustness.

Hoffmann et al. [Hoffmann86] proposed an O(N) worst-case time algorithm that solves the Jordan sorting problem and polygon clipping. In their algorithm, they used a sophisticated data structure, the *level-linked search tree*. The drawback is that the algorithm is rather complicated. Later, Fung et al. [Fung90] slightly simplified this algorithm. They replaced the level linked search tree with the *heterogeneous finger tree*. They also reduced the set of operations that must be performed on their data structure. In spite of this simplification, also this algorithm remains rather complicated. Let it be pointed out that the fact that the algorithms are complicated causes not only that they are difficult to implement. It also causes that, for the sequences of sizes that are of practical interest, these algorithms are relatively slow (this assertion will be illustrated in Section 4).

Both in the paper by Hoffmann et al. [Hoffmann86] and in the paper by Fung et al. [Fung90], it is noted that the use of the *splay tree* [Sleator85], a self-adjusting form of search tree, gives a simple algorithm for Jordan sorting that may run in linear time. The linear-time bound for this method, however, remains only a conjecture.

3. Overview of the New Algorithm

The algorithm we propose (Algorithm 1) is based on successively constructing a planar map that corresponds to the given input. The computation starts with an initial map $\mathcal{M}^1$ (Figure 1) containing only the nodes x_o, x_{N+1}, and F. The node F corresponds to the first value from the input sequence. The nodes x_o, x_{N+1} correspond to the values $-\infty$ and ∞, respectively. The algorithm then reads the intersection points from the input one by one, and for each such intersection point, it searches for the segment of the x-axis into which this point falls. Once this segment is found, the map is updated. During the computation, a sequence $\mathcal{M}^1,\mathcal{M}^2,\ldots,\mathcal{M}^N,\mathcal{M}$ of the maps is thus constructed (Figure 1). Once the final map $\mathcal{M}$ is found, the ordered sequence of points can easily be read from this map.

In the algorithm, the map is represented by a doubly linked list. Let X be the intersection point that is just being processed, and let PredX be the point that was processed immediately before X. The step finding the segment into which the new intersection point X falls proceeds as follows (Figure 2): Beginning with the segments adjacent to PredX and

concurrently in both directions along the boundary of the face containing the new arc PredX,X of the curve, all the segments of the x-axis lying on the boundary of this face are tested whether they contain X. In Figure 2, the numbers show the order in which the segments are tested. The source code of Algorithm 1 can be found in the appendix.

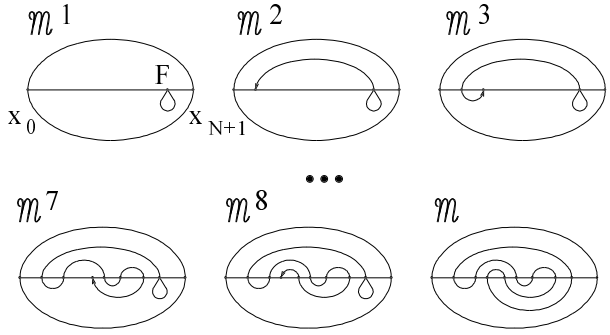

Figure 1. An example of the sequence of the maps that are constructed during the computation.

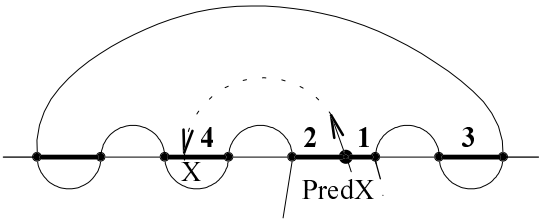

Figure 2. Searching step: The numbers show the order in which the segments are tested whether they contain X.

4. Results of testing

We have implemented and tested Algorithm 1 and compared the results with those we have obtained for other known algorithms solving the same problem. Namely, the algorithm proposed by Fung, the algorithm based on the splay tree, and Quicksort were used for comparison. The original algorithm proposed by Hoffman was not included since it was obviously overcome by its younger simplified version proposed by Fung.

Algorithm 1 was implemented as shown in the appendix. Fung's algorithm was carefully implemented according to [Fung90] and [Tarjan88]. The splay tree was implemented according to [Sleator85]. Both bottom-up and top-down splaying were implemented and tested. For sorting Jordan sequences, however, the top-down splaying gave better results, therefore, we used this version for comparison.

Quick sort was implemented according to [Wirth86]. Again, we experimented with different variants of this algorithm, namely, with its recursive and non-recursive version [Wirth86]. Let it also be pointed out that although Quick sort is considered to be very useful and efficient, its theoretical worst-case time complexity is $O(N^2)$. Therefore, for completeness, we tried to implement also an O(NlogN) worst-case time version of this algorithm. In each step of this modified version, the problem of size n was always divided into the two subproblems of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Once the solutions to both these subproblems were available, they were merged together thus giving the solution to the original problem of size n. Although in the sense of the asymptotic worst-case time complexity, the modified version is better, the experiments have shown that on a real computer and for the problems of size up to at least 10000, it is always significantly slower than the standard variants of Quicksort. In our experiments, standard recursive Quicksort [Wirth86] was the fastest and, therefore, it was used for comparison.

In the first series of tests, we studied the behaviour of the algorithms for the sequences of size from 10 up to 10000. For testing, we used four types of sequences: (1) the sequences that were hard for Algorithm 1; (2) the sequences that were hard for the splay tree; (3) sequences that were generated randomly; (4) the sequences that had already been sorted.

The sequences (1) were generated by making use of the map similar to that one which was used in Algorithm 1. Let the term *size of face* mean the number of the segments of the x-axis that lie on the boundary of a given face of this map. The generating started with the map $\mathcal{M}^1$ (Figure 1). Then, the curve generating the sequence created two "big" faces in the map

(one in the upper and one in the lower half-plane). The remaining intersection points were generated such that the curve repeatedly split these faces and their descendants. If a face of size s was to be split, then it was split into the faces of size $\lceil (s+1)/2 \rceil$ and $\lfloor (s+1)/2 \rfloor$. The biggest available face was always chosen for splitting. The process of splitting was stopped if no face bigger than a certain value was available. For the sequences of different lengths, this value was found experimentally such that it maximised the time complexity of Algorithm 1. In the paper [Sojka96], it was shown that the process described above leads to the sequences that are close to the worst-case input of Algorithm 1. Figure 3 shows an example of the curve generated in such a way.

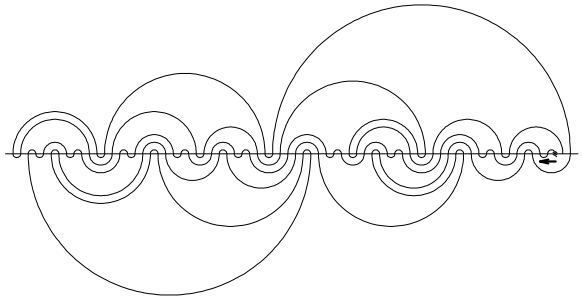

Figure 3. An example of the curve generating the sequence which is hard for Algorithm 1. The arrow shows the startpoint of the curve.

The technique of generating the sequences (2) was similar to the technique used for the sequences (1). The only difference was that the process of splitting continued until no face bigger than 3 was available for splitting. The fact that the sequences arising in this way are hard for the method based on the splay tree was discovered accidentally during the experiments. We did not check theoretically how close these sequences are to the worst-case input for the splay tree.

For sequences (3), a certain model of generating open Jordan curves and the corresponding sequences was introduced. In details, this model was presented in [Sojka97]. Again, the map similar to that one used in Algorithm 1 was used. The principle of our generator is as follows: Consider a step in which a certain number of intersection points have already been generated. Let X be the point which is to be generated just now, and let PredX be the point that was generated immediately before X (Figure 2). Since the Jordan curve does not intersect itself, the point X must lie inside a segment of the x-axis which is a part of the boundary of a face that has already been generated in the map. The appropriate face was determined when PredX was generated. In our model of generation, the segment containing X was chosen randomly from all the segments lying on the boundary of the face containing the arc PredX,X of the curve. All the segments lying on the boundary of this face had equal probabilities that they would contain X. The generation started with the initial map $\mathcal{M}^1$. Figure 4 shows an example of the curve that was generated randomly.

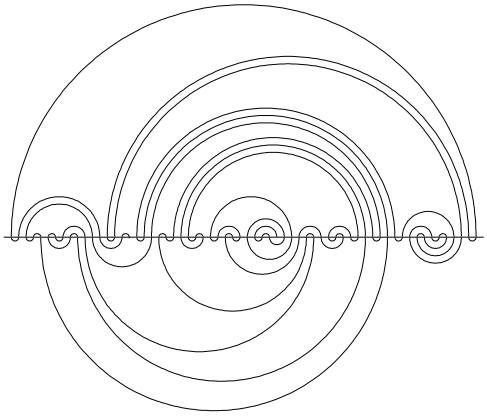

Figure 4. An example of the curve that was generated randomly.

In our experiments, we have evaluated the number of comparisons between the sorted values that were required by the algorithms, and the running time on a real computer. All the algorithms were implemented in Pascal. The running time was measured on an Intel Pentium 75 MHz computer. Borland Pascal 7.0 compiler was used to compile the programs. The sorted values were 4-byte floating-point numbers (single). The programs ran in real mode (therefore, the running time was measured only for the sequences of size to 10000). All the needed memory space was always allocated beforehand. The results of

testing are summarised in Table 1. In Figures 5,6, the main results are depicted also in the form of graphs. Figure 5 shows the maximal running times that were required by particular algorithms. In this figure, the times are related to the running time of Algorithm 1. Figure 6 depicts the running times that were required for the curves that were generated randomly.

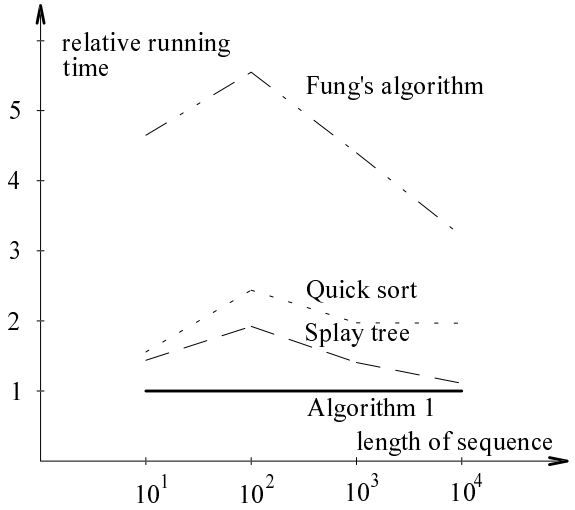

Figure 5. Maximal required running times (relative values, running time of Algorithm 1 is 1.0).

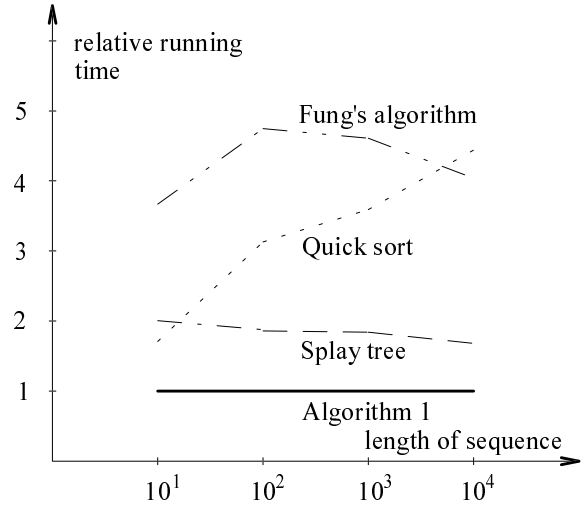

Figure 6. Running times for the sequences that were generated randomly (relative values, running time of Algorithm 1 is 1.0).

As follows from the figures, for the sequences of all tested sizes, Algorithm 1 is faster than all the remaining algorithms. Especially for the sequences that were generated randomly, the results are very optimistic. Here, however, it should be pointed out that it is unclear to what extent our generator of random Jordan sequences reflects particular practical cases and it is even unclear what could be considered to be a "practical case". Therefore, we have carried out one another series of experiments.

In the second series of tests, we generated and sorted all the existing Jordan sequences of a certain size and evaluated the average running time, the maximal and the average number of comparisons that were needed to sort one input sequence. Since there are c^N Jordan sequences (c is a constant, it is known that $2<c<4$), it was possible to carry out these tests only for short Jordan sequences. In our experiments, the sequences of size from 4 to 18 were tested. However, we believe that this experiment was useful. In computer graphics practice, just the short sequences are very common.

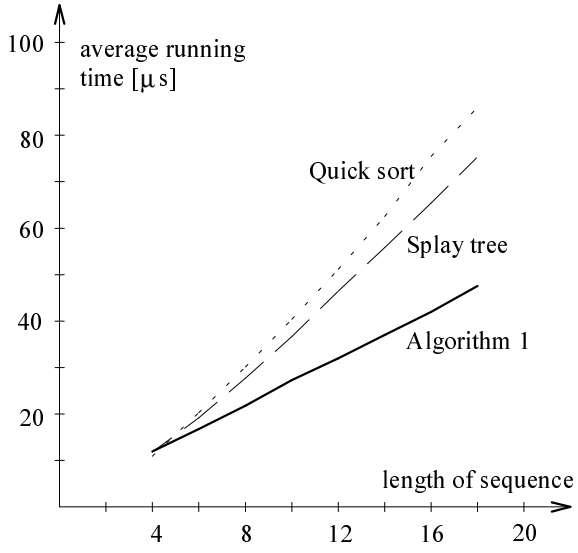

Figure 7. Sorting short Jordan sequences: Average running times (in microseconds per a sequence) on an Intel Pentium 75 MHz computer.

The results of the second series of tests are summarised in Tables 2,3,4. Figure 7 depicts the average running time (in microseconds) that was needed to sort one sequence. Again, Algorithm 1 was faster than all the remaining algorithms. Fung's algorithm was excluded from this series of tests. As the first series of

tests shown, this algorithm is very slow. Therefore, especially in the case of short sequences, it cannot be a serious competitor for the remaining algorithms.

Finally, let it be noted that we have checked the running times also on an Intel 386SX computer (without cache memory, but with mathematic coprocessor), on an AMD K5 computer, and in Windows NT operating system. The results were similar.

Table 1. The running times (in milliseconds) on a Pentium 75 computer, and the numbers of comparisons between sorted values that were required by the algorithms.

Type of sequence	Size	Running times in ms				Number of comparisons			
		Alg. 1	Quick	Fung	Splay	Alg. 1	Quick	Fung	Splay
Hard for Algorithm 1 "Hb"	10^1	0.0286	0.0446	0.133	0.0412	27	40	44	24
	10^2	0.331	0.807	2.03	0.533	459	1045	542	378
	10^3	4.91	9.67	20.7	5.62	6414	12382	4936	3890
	10^4	67.3	132.4	202	57.4	90187	194708	44026	36825
Hard for Splay tree "Hc"	10^1	0.0286	0.0445	0.133	0.0412	27	40	44	24
	10^2	0.321	0.833	1.94	0.637	427	1051	611	499
	10^3	3.72	10.2	20.3	6.90	4949	13467	6063	5401
	10^4	50.6	130.7	199	74.7	58585	181242	57423	57423
Generated randomly "Ra"	10^1	0.0261	0.0446	0.0959	0.0374	22	41	30	20
	10^2	0.236	0.740	1.12	0.440	253	904	393	286
	10^3	2.58	9.27	11.9	4.76	2591	12223	4033	2978
	10^4	29.4	130.6	120	49.5	26245	186651	41013	30470
Already sorted "St"	10^1	0.0237	0.0324	0.111	0.0283	18	31	24	9
	10^2	0.204	0.457	1.42	0.288	198	606	294	99
	10^3	2.23	5.62	15.3	3.14	1998	9009	2994	999
	10^4	26.04	74.0	154	32.8	19998	125439	29994	9999

Table 2. Maximal number of comparisons between sorted values.

Size of sequence	4	6	8	10	12	14	16	18
Algorithm 1	10	18	26	34	42	50	58	66
Splay tree	6	12	20	30	40	50	61	73
Quick sort	12	27	45	68	95	126	161	200

Table 3. Average number of comparisons between sorted values.

Size of sequence	4	6	8	10	12	14	16	18
Algorithm 1	7.3	12.8	18.5	24.3	30.2	36.2	42.2	48.3
Splay tree	4.6	9.4	14.9	21	27.4	34.1	41.1	48.2
Quick sort	9.1	18.9	29.6	41.6	54.4	67.8	81.8	96.4

Table 4. Average running times (in microseconds) on an Intel Pentium 75 MHz computer.

Size of sequence	4	6	8	10	12	14	16	18
Algorithm 1	11.9	16.8	21.8	27.3	32.0	37.0	41.95	47.55
Splay tree	11.8	19.2	27.7	36.7	46.5	55.8	65.4	75.3
Quick sort	10.9	20.3	30.1	40.4	51.1	62.5	75.5	86.0

5. Conclusion

In this paper, the results of testing a new algorithm for sorting Jordan sequences were presented. The source code of the new algorithm is enclosed. The performance of the algorithm was compared with the performance of other known algorithms that solve the same problem. Namely, the algorithm proposed by Fung, the algorithm based on splay trees, and Quicksort were used in the comparison.

The experiments have shown that the new algorithm is useful. Although its asymptotic worst-case time complexity is O(NlogN) and although it did not always required the lowest number of comparisons between the sorted values, for the sequences of size from 4 up to 10000, it always beat all the remaining algorithms in running time on a real computer. The reason is straightforward: The new algorithm is very simple. Except comparing sorted values, it performs only a low number of other instructions. Moreover, the worst running time is achieved only if the input sequence is generated according to special rules. Any deviation from these rules causes that the time complexity decreases. Thus, the input leading to the worst running time is highly improbable. The situation resembles the situation with Quick sort: The worst-case time complexity of Quick sort is $O(N^2)$, but since the probability of occurrence of the input that requires this running time is low, it has commonly been accepted, that it is more appropriate to judge the performance of Quick sort by its average running time, which is O(NlogN).

The Fung's algorithm is obviously too complicated, therefore, although linear in time, it is slow. In our experiments, for the sequences of size, at least up to 10000 it was even slower than Quick sort. The predominance of simple algorithms for Jordan sorting was confirmed also by the method based on the splay tree. In our experiments, the running times required by this relatively simple method were not too higher than the running times required by Algorithm 1. Moreover, the difference decreased with the increasing size of the problem.

One more general conclusion follows from our experiments: Our experience confirmed the known fact that the asymptotic worst-case time complexity need not necessarily be a suitable criterion of the practical usefulness of algorithms. We add that sometimes it may even be a very bad criterion. This criterion may prefer the algorithms whose practical usefulness is limited and may even stimulate the development of such algorithms. Therefore, whenever it is possible, a mean value of running time should be used as a much more exact criterion of usefulness.

References

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

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

Sleator,D.D., Tarjan,R.E. (1985), Self-Adjusting Binary Search Trees, *Journal of the Association for Computing Machinery*, Vol. **32** (3), pp 652-686.

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

Sojka,E. (1997) A simple and efficient algorithm for sorting the intersection points between a Jordan curve and a line, in: *Proc. The Fifth International Conference in Central Europe on Computer Graphics and Visualisation WSCG '97*, Plzen, pp 524-533.

Tarjan,R.E., Van Wyk,C.J. (1988), An O(n log log n) time algorithm for triangulating a simple polygon, *SIAM J. Comput.*, Vol **17**, pp 143-178; Erratum: *SIAM J. Comput*, Vol **17**, pp 1061.

Wirth,N. (1986), Algorithms and data structure, *Prentice-Hall, Englewood Cliffs*, New Jersey.

Appendix A: Algorithm 1

This appendix contains the source code of our implementation of Algorithm 1.

```
{The following record is used to represent each node of the map.}
PMapNode = ^TMapNode;
TMapNode = record               {For each intersection point, the map contains:}
   Pred, Succ: PMapNode;        {Neighbours along the x-axis}
   Upper, Lower: PMapNode;      {Neighbours via upper and lover arcs}
   X: TFloat;                   {Coordinate of intersection point}
end;

procedure JordanSorting;
{ This procedure sorts Jordan sequences. ValX always contains the value that is just
being sorted. PredX always points towards the node containing the value that was
processed immediately before ValX. L,R point towards the left and the right endpoint,
respectively, of the segment containing the sorted value.}
begin
  ValX:= Read the first item from the input sequence;
  InitialiseMap( ValX, L,PredX,R );  Xo:= L;
  while  input is not empty  do begin
    ValX:= Read the next item from the input sequence;
    if ValX > PredX^.X
      then UpperRightSearch( ValX,PredX,L,R )
      else UpperLeftSearch ( ValX,PredX,L,R );
    PredX:= UpperUpdate( ValX,PredX,L,R );
    If any intersection left in the input, perform the same sequence of symmetric
    operations also in the lower half-plane Lx.
  end;
  Beginning with Xo^.Succ read the sorted sequence from the map;
end;

procedure UpperLeftSearch( ValX:TFloat; PredX:PMapNode; var L,R:PMapNode );
{ In the upper half-plane and beginning with the segments adjacent to PredX, this
procedure searches for the segment containing ValX. In L,R, the procedure returns the
pointers to its left and right node, respectively. It is supposed that ValX < PredX^.X.
The procedures UpperRightSearch, LowerLeftSearch and LowerRightSearch are symmetrical.}
begin
  repeat
    if ( ValX > L^.X ) then begin R:= L^.Succ; Exit; end;
    L:= L^.Upper^.Pred; R:= R^.Upper^.Succ;
  until ( R^.X < PredX^.X );
  repeat
    if ( ValX < R^.X ) then begin L:= R^.Pred; Exit; end;
    if ( ValX > L^.X ) then begin R:= L^.Succ; Exit; end;
    L:= L^.Upper^.Pred; R:= R^.Upper^.Succ;
  until false;
end;

function UpperUpdate( ValX: TFloat; PredX,L,R: PMapNode ): PMapNode;
{ This function updates the map. It returns the pointer to the new node containing ValX.
The function LowerUpdate is symmetrical.}
var X: PMapNode;
begin
  New( X ); X^.Pred:= L; X^.Succ:= R; X^.Upper:= PredX; X^.X:= ValX;
  R^.Pred:= X; L^.Succ:= X; PredX^.Upper:= X; UpperUpdate:= X;
end;

procedure InitialiseMap( FirstValX: TFloat; var L,F,R: PMapNode );
{ This procedure initialises the map.}
begin
  New( L ); New( R ); New( F );
  L^.Pred:= Nil; L^.Succ:= F; L^.Upper:= R; L^.Lower:= R; L^.X:= MinFloat;
  R^.Pred:= F; R^.Succ:= Nil; R^.Upper:= L; R^.Lower:= L; R^.X:= MaxFloat;
  F^.Pred:= L; F^.Succ:= R; F^.Upper:= F; F^.Lower:= F; F^.X:= FirstValX;
end;
```