Available online at www.sciencedirect.com

SCIENCE DIRECT®

Information Sciences 169 (2005) 383–408

INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL

www.elsevier.com/locate/ins

Acceleration of sweep-line technique by employing smart quicksort

David Podgorelec [*], Gregor Klajnšek

Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova ulica 17, SI-2000 Maribor, Slovenia

Received 4 September 2003; received in revised form 28 June 2004; accepted 5 July 2004

Abstract

Quicksort is usually the best practical choice for sorting because it is, on average, remarkably efficient. Unfortunately, this popular algorithm has a significant drawback: the slowest performance is obtained in the simplest cases when input data are already initially sorted or only a slight perturbation occurs. In this paper, we propose a combination of quicksort and a new algorithm, which shows excellent time performance in sorting such crucial data arrays, and which is not much slower than quicksort in random cases. Our work was inspired by problems met when sorting polygon vertices in the sweep-line algorithms of computational geometry and, therefore, we have named the new algorithm 'vertex sort'. It splits the input array into three sub-arrays. Two of them are already sorted, and the third one is handled iteratively. A simple test decides whether to continue recursively with vertex sort or to employ quicksort in the second iteration. In this way, we achieve a situation where the worst case time complexity does not exceed the running times of quicksort, but the simplest cases are handled much faster (in linear time) than random cases. We have named the combined algorithm 'smart quicksort'

[*] Corresponding author. Tel.: +386 2 220 74 74; fax: +386 2 251 11 78.
E-mail addresses: david.podgorelec@uni-mb.si (D. Podgorelec), gregor.klajnsek@uni-mb.si (G. Klajnšek).

doi:10.1016/j.ins.2004.07.002

because of this desired property. In the last part of the paper, we prove its efficiency by employing it in a well-known sweep-line-based polygon triangulation algorithm.

Keywords: Computational geometry; Polygon triangulation; Quicksort; Smart quicksort; Sweep-line; Vertex sort

1. Introduction

The efficiency and feasibility of a wide variety of algorithms from various branches of computer science, often directly depends on previously sorted data. In computational geometry, for example, a large family of so-called sweep-line algorithms requires that geometrical elements are previously sorted according to one of the co-ordinates. The sorted data enable such algorithms to perform particular operations on close neighbouring elements only, and to avoid unnecessary calculations on subsets of distant geometric elements that cannot affect the final solution [17]. Computer implementations of these algorithms are expected to run more quickly than programmes initialized with randomly ordered data, but we should not forget that sorting also requires some time. Therefore, the employed sorting algorithm should also be as fast as possible.

Nowadays, the well-known *quicksort* [9] is usually the best practical choice for sorting because it is remarkably efficient on average: its expected running time is $\Theta(n \log n)$, and the constant factors hidden in the $\Theta(n \log n)$ notation are quite small [3]. It is based on the divide-and-conquer paradigm, but the sub-arrays are sorted in place and, therefore, no work is needed to combine them. In spite of its popularity, quicksort suffers from a serious drawback. The $\Theta(n^2)$ running time occurs when the input array is completely sorted initially—a common situation in which even the simplest algorithms like insertion sort run in $O(n)$ time. Similarly, the $\Theta(n^2)$ running time is achieved when the input array is sorted in reverse order and, with some implementations, even in cases where all elements have the same key value [3].

In Section 2, we show that this lack of intelligence is unacceptable in some computational geometry applications. In Section 3, we propose a new sorting algorithm called 'vertex sort'. It is intelligent since it handles the simplest cases in linear time. On the other hand, it needs much more time for randomly ordered data than quicksort. The logical decision is to combine both algorithms and, in particular cases, to benefit from both. The combined algorithm named smart quicksort is described in the Section 4. In Section 5, we analyse the time complexity of smart quicksort, and compare its running times with those of quicksort and vertex sort. We also employ quicksort and smart quicksort in a well-known algorithm for polygon triangulation, and confirm the advantages of smart quicksort. Finally, we introduce some ideas that should improve the algorithm in the near future.

2. Sweep-line-based polygon triangulation

Our work was inspired by studying the running times of a two-step polygon triangulation algorithm. Decomposition of the input polygon into y-monotone pieces is given in [10] and the second step of separate y-monotone polygon triangulation is given in [8]. A polygon is *monotone* with respect to a line l if for any line l' perpendicular to l, the intersection of the polygon with l' is connected. A polygon that is monotone with respect to the y-axis is y-monotone [1]. However, it is much more important to know that the boundary of the y-monotone polygon does not contain any 'split' or 'merge' vertices. In respect of [1], the polygon vertices are classified as start, end, split, merge and regular vertices.

- A convex vertex is *a start vertex* if both neighbours lie below it.
- A convex vertex is *an end vertex* if both neighbours lie above it.
- A concave vertex is *a split vertex* if both neighbours lie below it.
- A concave vertex is *a merge vertex* if both neighbours lie above it.
- All other vertices are *regular vertices*.

Note that a vertex lying to the right of the observed vertex v is considered as being below v since such situations can be eliminated by a slight rotation of the plane in a clockwise direction. Fig. 1 shows a general polygon decomposed into y-monotone pieces. We use five different symbols to differentiate between vertices of particular types. This figure clearly demonstrates that the decomposition into y-monotone pieces is actually performed by drawing diagonals from split and merge vertices. Fig. 2 shows how the monotone pieces from Fig. 1 are triangulated. Indices of triangles represent the order of their formation.

It was proved in [1] that the worst-case time complexity of the first step is $O(n \log n)$ where n is the number of the polygon vertices, and the second step triangulates each y-monotone piece in linear time. In addition, we can state the following theorem.

Theorem 1. *The second step of the algorithm proposed in* [8] *operates in linear time.*

Proof 1. Let $n_1, \ldots, n_k$ represent the numbers of vertices of k monotone pieces obtained by the polygon decomposition, respectively. We have to prove that $n_1 + \cdots + n_k = O(n)$. The proof is based on the fact that a polygon with n

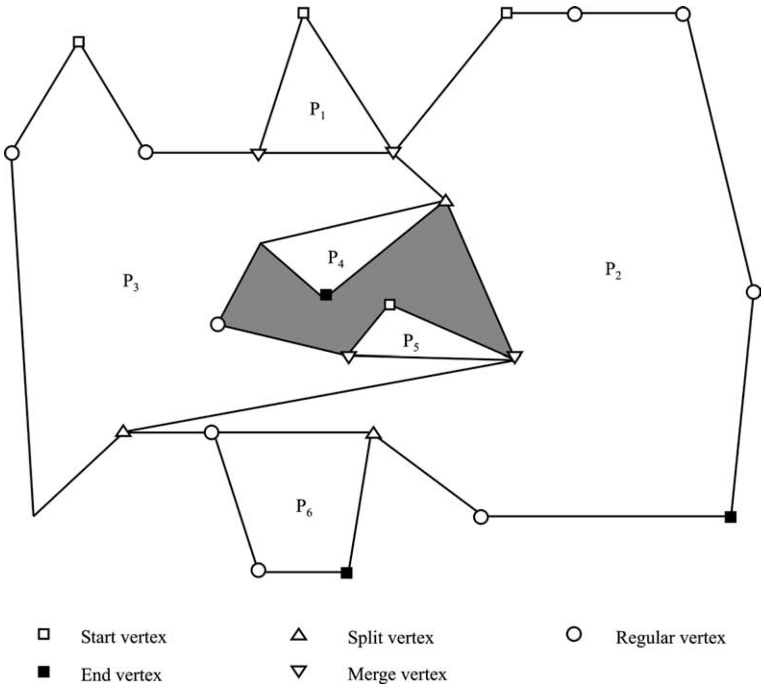


Fig. 1. Classification of vertices and decomposition of a general polygon into y-monotone pieces.

vertices has n edges. Consequently, a set of polygons with the total number of vertices n has n edges. Here, we count edges that belong to two polygons twice, and vertices shared by m polygons m times. We can obviously count edges instead of vertices, and prove that a set of y-monotone pieces has $O(n)$ edges. An edge is either an original edge or a diagonal inserted for each split and each merge vertex. The total number of split and merge vertices cannot exceed the total number of vertices n and therefore we have, at most, n diagonals. Each diagonal separates two monotone pieces, and should be counted twice. In this way, we obtain the upper bound for the number of edges: $n + 2n = 3n = O(n)$. □

Both steps of the algorithm use the sweep-line technique, which requires that vertices are previously sorted in nonascending order according to the y co-ordinate. The first step accepts all types of polygons, and the presortedness of the vertices cannot be predicted. Therefore, quicksort seems a rather good choice for sorting because of its remarkable average speed. Table 1 gives time measurements for the triangulation of convex polygons with 5000, 10,000, 20,000 and 40,000 vertices. Randomized improvement of quicksort, the so-called median-of-3 quicksort [13] is used in our implementation of the polygon triangula-

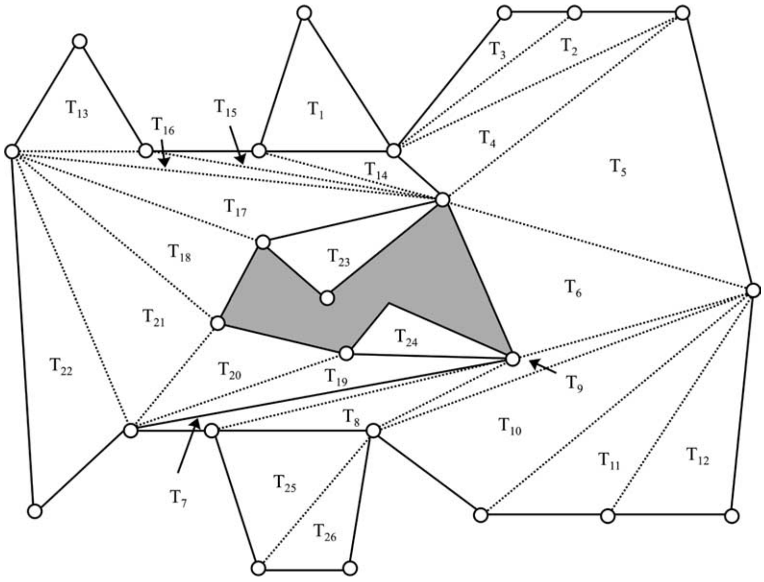


Fig. 2. Separate triangulations of y-monotone pieces from the example introduced in Fig. 1.

Table 1
Convex polygons are not triangulated in O(n log n) as expected

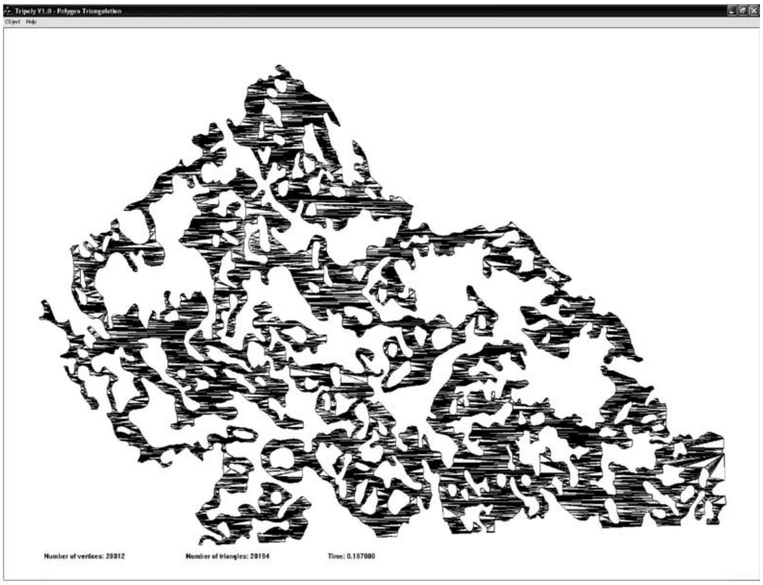
Number of vertices	$n_1 = 5000$	$n_2 = 10{,}000$	$n_3 = 20{,}000$	$n_4 = 40{,}000$
Time [s]	0.0281	0.0690	0.2468	1.7660
Increase [n_i/n_{i1}]		2.46	3.58	7.16
Increase [n_i/n_1]	1	2.46	8.78	62.70

tion algorithm and during all this research. All the measurements in this paper were done on a PC with a 1.67 GHz processor and 1 GB of RAM. All software was written in MS Visual C++ 6.0 and runs under MS Windows XP. Due to the theorem from [1], any polygon should be triangulated, at most, in O(n log n) time but the results in Table 1 report quadratic time growth by increasing the number of vertices. The vertices are initially written down in order as they follow each other along the polygon border. This means that an array presenting a convex polygon consists of a single, two or three sub-arrays of the vertices, which are sorted either in nondescending or nonascending order. The number of monotone sub-arrays in the polygon border depends on the lengths of both boundary chains between the topmost and bottommost vertices, and on selection of the first vertex in the loop. Anyhow, it is well-known that quicksort achieves its worst-case running time $\Theta(n^2)$ if the data are already initially sorted. If the input array consists of a low number of monotone sub-arrays, the results are not much better. Randomization helps a little, but it does not solve the problem in a satisfactory way.

Convex polygons are intuitively the simplest polygons, but quicksort is unable to handle them inside the time bounds of the polygon triangulation. Surprisingly, general polygons are processed much faster. Fig. 3 presents a realistic example of a lake with 91 islands. It consists of 28,012 vertices. The convex polygon with the same number of vertices was triangulated in 0.7 s, but the evidently more complex polygon from Fig. 3 requires 0.185 s only.

Quicksort is obviously not the best choice for sorting polygon vertices. On the other hand, heapsort and merge sort always perform within the time bounds of the polygon triangulation—O($n \log n$), but they are much slower than quicksort in random cases. A time complexity analysis of all three algorithms can be found in [3], as an example. In addition, heapsort and merge sort still do not guarantee that a convex polygon is triangulated faster than a random case.

Let us suppose that we have found an ideal algorithm for sorting polygon vertices. It never spoils the time complexity of the polygon triangulation and, therefore, we can exclude it from the time complexity analysis. This assumption enables us to state two additional theorems addressing the time complexity of the polygon triangulation algorithm. Since their proofs involve a binary search tree (BST), we have to give a short explanation on the role



```
Number of vertices: 28012        Number of triangles: 28194        Time: 0.187000
```

Fig. 3. A triangulated concave polygon with 28,012 vertices and 91 holes.

of this dynamic data structure in the first phase of the algorithm. BST is a balanced binary tree which stores the sweep-line status. More precisely, it stores all edges from left boundary chains (with the polygon interior to the right) intersected by the sweep line. Since BST is balanced, its depth is $\lfloor \log_2 n \rfloor + 1$ where n is the number of edges in BST. The edges in BST are sorted from left to right. Consequently, each element of BST can be found in $O(\log n)$ time [1,3].

Theorem 2. *The algorithm proposed in [8,10] triangulates any y-monotone polygon in O(n).*

Proof 2. Theorem 1 guarantees that the second step is executed in $O(n)$. Therefore, we have to analyse the first step only. Its time complexity has to be calculated as a product of the number of vertices (n) and the maximum depth of BST. A y-monotone polygon contains a single start vertex, a single end vertex, and all other vertices are regular [1]. The start vertex is processed first. The edge connecting it with its left neighbour is inserted into BST. For each regular vertex in the left boundary chain, the edge from the upper neighbour is erased from BST, and the edge to the lower neighbour is added into BST. Finally, the only remaining edge is removed from BST when the end vertex is processed. Obviously, BST contains no more than a single edge all the time if the polygon is y-monotone. All vertices are processed in $O(n \cdot 1) = O(n)$ time. □

Theorem 3. *The time complexity of the polygon decomposition into y-monotone pieces asymptotically increases to O(n log n) with the number of split and merge vertices.*

Proof 3. The time complexity depends on the depth of BST, and this depends on the number of edges in BST. This number is equal to the number of left boundary chains intersected by the sweep-line. Each left (and right) boundary chain begins in a start or a split vertex, and ends in an end or a merge vertex. Obviously, the more split and merge vertices caused, the more edges are present in BST. More edges in BST extends search times and the whole running time of the algorithm. The upper time bound $O(n \log n)$ was proved in [1]. □

3. Vertex sort

Intuitively, we can say that convex and other y-monotone polygons are simpler than polygons containing split and/or merge vertices. Therefore, Theorems 2 and 3 prove that the polygon triangulation algorithm proposed in [8,10] is intelligent because it handles simple cases faster than complex ones. Since these theorems are based on the assumption that sorting does not change the time complexity, an employed sorting algorithm should also be intelligent. We expect it to handle monotone arrays in linear time, its running time to linearly increase with the number of monotone sub-arrays, and that its worst-case time complexity does not exceed the worst-case time complexity of triangulation—$O(n \log n)$. These requirements can be fulfilled only by *an optimal adaptive sorting algorithm with respect to the number of monotone sub-arrays* (NMS). A sorting algorithm is adaptive, with respect to some measure of disorder, if the time taken by the algorithm is a smoothly growing function of the size and disorder of this array [7]. The required optimality originates in the well-known fact that comparison sorting algorithms cannot beat a lower bound of $\Omega(n \log n)$ in the worst-case running time [3]. In continuation, we propose a comparison sorting algorithm, which is much closer to the above requirements than quicksort. Since it was firstly developed to sort polygon vertices, it is simply named *vertex sort*. Before we can describe details, we have to define a monotone sub-array unambiguously.

Definition 1. Let $A = (a_0, \ldots, a_{n-1})$ be an array of n numbers that we want to sort. A monotone sub-array is a maximum sequence $(a_i, \ldots, a_j)$, $i \geqslant 0, j < n$, of successive elements of A sorted in intended or reverse order.

The requirement that the monotone sub-array is maximum, means that it does not remain sorted if we extend it by the direct predecessor a_{i-1} or by the direct successor a_{j+1}.

Several adaptive sorting algorithms have been developed to cope with similar goals, but they do not use NMS as a measurement of disorder. They are mostly oriented to handle sorted arrays, arrays sorted in reverse order and nearly sorted arrays. Intuitively, a sub-array is nearly sorted if only a few of its elements are out of order [2]. Cook and Kim have developed an excellent sorting algorithm designed specifically for nearly sorted arrays [2]. It is a combination of insertion sort [3], quickersort [12] and merging. It performs poorly on any other distribution including nearly sorted in reverse [2]. Wainwright was more successful with its Bsort [15] and Qsorte [16]. They can both be characterized as quicksort descendants with an early exit for sorted sub-arrays. The former uses a technique associated with bubble sort to determine if a sub-array is sorted after a particular pass. A similar algorithm Qsorte performs much less interchanges. It marks sub-arrays as sorted or unsorted instead. Qsorte performs just as well as quicksort for random arrays, but it handles sorted and nearly sorted arrays in intended order or in reverse in $O(n)$.

Sorted and nearly sorted arrays in desired order or in reverse all have low NMS. A nearly sorted array with k elements out of order consists of, at most, $2k + 1$ monotone sub-arrays (the proof is left to the reader). If k is small, then

NMS = $2k + 1$ is also small. However, low NMS can be obtained for many other arrays as well. For example, $A = (n/2 + 1, \ldots, n, n/2, n/2 - 1, \ldots, 1)$ has NMS(A) = 2, although the first $n/2$ elements are not in their correct positions. Such array is not nearly sorted with regard to the definition of Cook and Kim [2], but we still expect that it will be sorted much quicker than a random case.

Here, we present a version of vertex sort, which sorts an array $A = A_0$ in nonascending order. Of course, it is trivial to modify it to sort in non-descending order. The idea is simple because it directly originates in requirements for an intelligent sorting algorithm. We need a storage B_1 which would store the whole input array in case it is already initially sorted. We also need a storage B_2 able to revert the input array when it is initially sorted in reverse order. Finally, we need a storage A_1 which would store elements of A_0 that cannot be inserted either at the end of B_1 or at the beginning of B_2. This array (A_1) has to be sorted recursively. Recursion stops when a trivial problem is obtained i.e. the sub-array A_1 is empty or it contains a single element. After the recursion is released, we have to merge all three sorted sub-arrays B_1, B_2 and A_1 into the sorted array A_0. Obviously, the algorithm is based on the *divide-and-conquer* paradigm, although only one of three subproblems at each recursion level has to be solved in the same way as the original problem.

The idea of vertex sort is described by the pseudo code in Fig. 4. The recursion level is explicitly presented by the parameter r for better understanding. The algorithm is called with $r = 0$. It first inserts the greater of the first two elements of A_r into empty $B_{1,r+1}$ and the smaller of them into empty $B_{2,r+1}$ The elements $a_{r,i}$, $i = 2, \ldots, n-1$, are then moved into the sub-arrays A_{r+1}, $B_{1,r+1}$ or $B_{2,r+1}$ due to the criterion given in statements 6–10.

3.1. Spatial requirements

The storage complexity of vertex sort is analysed in this section. We suppose that the algorithm from Fig. 4 requires $R + 1$ recursive calls. Then it addresses the sub-arrays $A_0, \ldots, A_R, B_{1,1}, \ldots, B_{1,R}, B_{2,1}, \ldots, B_{2,R}$. Let us write a partition of the array A_r into three sub-arrays A_{r+1}, $B_{1,r+1}$ and $\mathrm{B}_{2,r+1}$ as a union:

$$A_r = A_{r+1} \cup B_{1,r+1} \cup B_{2,r+1}. \tag{1}$$

If we substitute implicit appearances of A_i, $i > 0$, on the right side with their explicit descriptions, we obtain a relation (2).

$$A_r = A_R \cup B_{1,r+1} \cup \cdots B_{1,R} \cup B_{2,r+1} \cup \cdots B_{2,R} = A_R \cup \left(\bigcup_{i=1,\ j=r+1}^{i=2,\ j=R} B_{i,j} \right). \tag{2}$$

```
Algorithm Vertex_Sort(r, n)
/* r is recursion level; */
/* n is the length of $A_r$ = ($a_{r,0}$, …, $a_{r,n1}$). */

begin
1.  if (n < 2) then return $A_r$; /* trivial problem */
2.  Insert the greater of $a_{r,0}$, $a_{r,1}$ at the end of $B_{1,r+1}$,
    and the smaller of them at the start of $B_{2,r+1}$.
3.  i = 2;
4.  loop /* for elements $a_{r,2}$, …, $a_{r,n1}$ of $A_r$ */
5.    if (i = n) then exit loop;
6.    if ($a_{r,i}$  the last element of $B_{1,r+1}$) then
7.      Insert $a_{r,i}$ at the end of $B_{1,r+1}$.
8.    else if ($a_{r,i}$  the first element of $B_{2,r+1}$) then
9.      Insert $a_{r,i}$ at the beginning of $B_{2,r+1}$.
10.     else Insert $a_{r,i}$ at the end of $A_{r+1}$.
11.   i = i+1;
12. forever;
    /* recursion */
13. $A_{r+1}$ = Vertex_Sort(r+1, index of the last element in $A_{r+1}$);
14. return Merge($A_{r+1}$, $B_{1,r+1}$, $B_{2,r+1}$);
end.
```

Fig. 4. The basic idea of recursive vertex sort.

Since an intersection of any two sub-arrays in a partition is empty, we can directly rearrange the relation (2) into Eq. (3). A special case for $r=0$ is shown in Eq. (4).

$$|A_r| = |A_R| + \left|\bigcup_{i=1,\ j=r+1}^{i=2,\ j=R} B_{i,j}\right|, \tag{3}$$

$$|A_0| = |A_R| + \left|\bigcup_{i=1,\ j=1}^{i=2,\ j=R} B_{i,j}\right| = n. \tag{4}$$

Obviously, we can pack all sub-arrays $B_{i,j}$; $i\in\{1,2\}$, $j=1,\ldots,R$, into a single array B with n elements. If we derive an explicit description of $|A_R|$from Eq. (3), and employ it in Eq. (4), we obtain Eq. (5) which directly reflects the

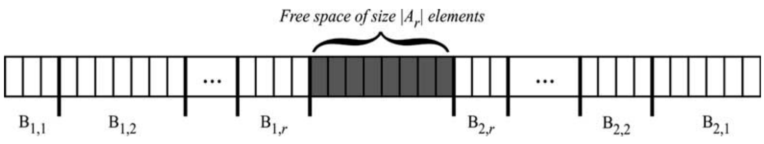


Fig. 5. Contents of array B after the rth iteration of decomposition.

situation after the rth iteration of decomposition. Fig. 5 shows the organization of sorted arrays $B_{i,j}$; $i \in \{1, 2\}, j = 1, \ldots, r$ in the array B.

$$|A_r| - \left| \bigcup_{i=1,\, j=r+1}^{i=2,\, j=R} B_{i,j} \right| + \left| \bigcup_{i=1,\, j=1}^{i=2,\, j=R} B_{i,j} \right| = |A_r| + \left| \bigcup_{i=1,\, j=1}^{i=2,\, j=r} B_{i,j} \right| = n. \tag{5}$$

We have to describe some details about merging before we can analyse storage requirements for sub-arrays $A_0, \ldots, A_R$. At recursion level r of the algorithm from Fig. 4, sub-arrays A_{r+1}, $B_{1,r+1}$ and $B_{2,r+1}$ are merged by calling the *Merge* procedure in statement 14, and the result is returned to the higher ($r-1$) level, where it is assigned to A_r in statement 13. Obviously, we do not have to remember a sub-array A_r after partition since it is reproduced in sorted order by merging just before it is needed again. This means that partitioning may write A_{r+1} in the place of A_r. After the partitioning is completed, we obtain all sub-arrays $B_{i,j}$; $i \in \{1, 2\}, j = 1, \ldots, R$, in the array B, and eventually the sub-array A_R with a single element at the beginning of the array A. For analogy, we need the sub-arrays $B_{i,j}$; $i \in \{1, 2\}$, $j = 1, \ldots, r$, in the array B, and the sub-array A_r at the beginning of the array A before merging at higher recursion levels. We have inherited a simple idea from merge sort [3], which requires additional memory for merging. This simple merging is employed twice at each recursion level. Let us show that the required additional storage can be found in the already introduced array B and in the original array A. First, we merge $B_{1,r}$ and $B_{2,r}$. Let us label the result A'_r. It can be written just behind A_r in the array A. Eq. (5) reports that we have more than enough space for each $r > 1$, and exactly enough space for $r = 1$. Then, we can forget $B_{1,r}$ and $B_{2,r}$. In this way, we extend the free space in the middle of the array B to $|A_r| + |B_{1,r}| + |B_{2,r}|$elements (Fig. 5). This space can then be used for merging A_r and A'_r into A_{r-1} The three arrays are merged now, but the result was obtained at a wrong location. It is placed somewhere in the array B, but is expected to be at the beginning of the array A. The latter still contains sub-arrays A_r and A'_r, but these may be rewritten since all their elements are contained in A_{r-1} as well. Therefore A_{r-1} is copied into the array A at the end of the merging procedure.

We have proved that vertex sort requires two arrays with the total number of elements $2n = O(n)$. Since the copying of array A_{r-1} is extremely slow, it is better to use three arrays A, B and C. $3n$ elements still belong to $O(n)$. Merging is performed as follows.

1. Sub-arrays $B_{1,r}$ and $B_{2,r}$ are merged into A'_r placed in the array A just behind A_r.
2. Then we merge A_r and A'_r into A_{r-1} placed at the beginning of array C.
3. We substitute the addresses of the arrays A and C to 'move' A_{r-1} to the required position.

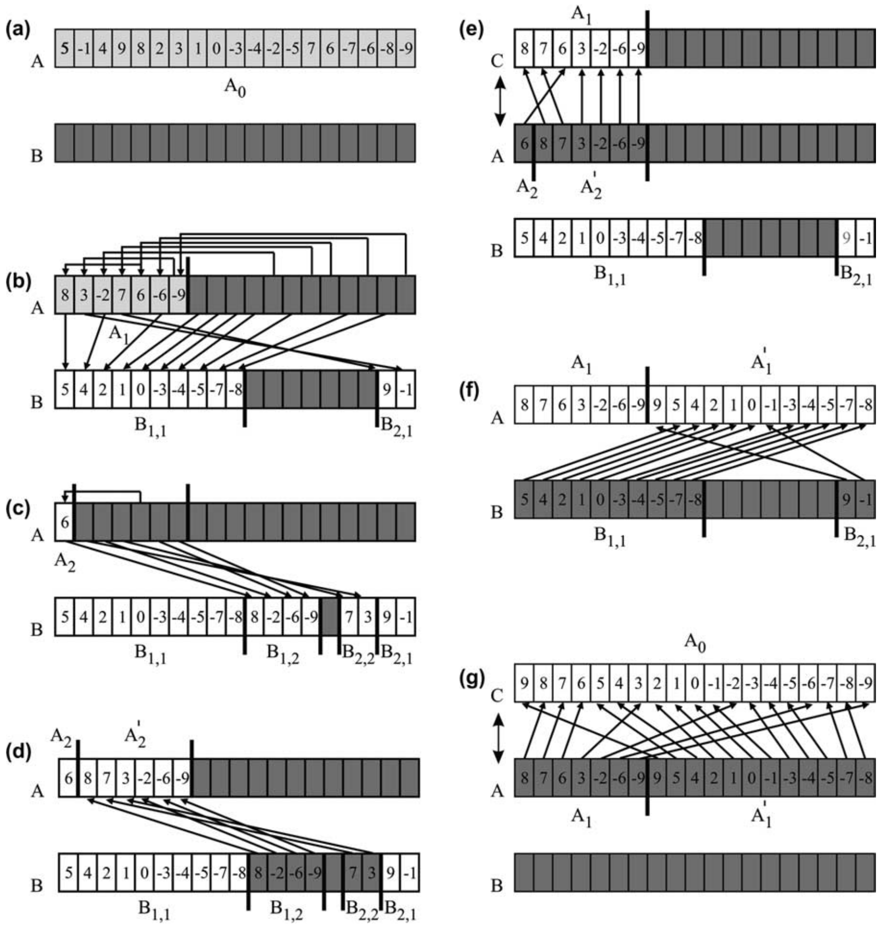


Fig. 6. An example of partitioning and merging for $R = 2$.

Let us support all this theory by a practical example from Fig. 6. Employed colours have the following meaning: the white colour presents those sub-arrays already sorted, the light grey is employed for sub-arrays not yet sorted, and the dark grey is used for free space. At the beginning, the array A_0 with n = 19 elements is stored in the array A (Fig. 6a). Fig. 6b and c present the partitioning. The arrows should be followed from left to right as they originate in the array A. After the second partitioning, we obtain a single element in A_2, and the recursion ends. The partitioning is followed by merging. First, the sub-arrays $B_{1,2}$ and $B_{2,2}$ located in the array B are merged into the sub-array A_2' in the array A (Fig. 6d). After this, the sub-arrays A_2 and A_2' from the array A are merged into the sub-array A_1 in the array C, and the addresses of the arrays A and C are substituted (Fig. 6e). Then the algorithm jumps to the higher recursion level where the process is repeated with the sub-arrays $B_{1,1}$, $B_{2,1}$ (Fig. 6f) and A_1 (Fig. 6g). After the substitution of the addresses of A and C, the sorted array A_0 is obtained in the array A i.e. at the position where the input array was passed to the algorithm. Finally, the auxiliary arrays B and C can be deleted.

In the implementation of vertex sort, we do not have to address recursion level r and sub-arrays A_r, $B_{i,r}$ and A'_r. We cope with three arrays A, B and C only. All mentioned sub-arrays are maintained inside these arrays by a proper combination of indices. Let us introduce two important accelerations before we give the complete pseudo code suitable for direct implementation.

3.2. Merging from both ends

Merging is a simple task and it should run in linear time if it is implemented correctly. We perform it from both ends. First, the starting elements of both two sub-arrays are compared, and the greater one is inserted at the beginning of the merged array. After this, the last elements of the arrays are compared, and the smaller is inserted at the end of the merged array. The corresponding index denoting the beginning or the end of the array is incremented or decremented by one. After one of the sub-arrays is completely inserted into the resulting array, the remaining elements of the other sub-array are simply copied into the result. The detailed pseudo code is given in Fig. 7.

We could perform merging from a single end as well, but the presented solution is usually quicker and undoubtedly more 'intelligent'. When the input array is sorted in reverse order, the second element a_1 is inserted into the first sub-array (B_1), and all other elements $a_{n-1}, \ldots, a_2, a_0$ into the second one (B_2). Since the only element of $B_1(a_1)$ is smaller than all the elements of B_2, except the last one, we would need $n-2$ comparisons until a_1 is inserted into the resulting array. Alternatively, merging from both ends requires only two comparisons. First, a_1 is compared to a_{n-1} and the latter is inserted at the beginning of A. After this, a_1 is compared to a_0 and the former is inserted at the end of A.

3.3. Prediction of order of testing sub-arrays

The second acceleration refers to partitioning. Let A be a nondescending array again. The algorithm from Fig. 4 inserts a_0 into B_2, and a_1 into B_1. After this, all elements $a_2, \ldots, a_{n-1}$ are compared with the last element of $B_1(a_1)$, and then with the first element of B_2. Altogether, $1 + (n-2) + (n-2) = 2n-3$

```
Algorithm Merge(A, startA, endA, B, startB, endB, C, startC, endC)
/* Merge $a_{startA}$, …, $a_{endA}$ and $b_{startB}$, …, $b_{endB}$ into
   $c_{startC}$, …, $c_{endC}$. A, B and C are the addresses of the input
   arrays and of the result respectively. */

begin
   loop
     if ($a_{startA}$ > $b_{startB}$) then begin /* merge from left ends */
       $c_{startC}$ = $a_{startA}$;      startA = startA+1;      startC = startC+1;
       if (startA > endA) then exit loop;
     end
     else begin
       $c_{startC}$ = $b_{startB}$;  startB = startB+1;  startC = startC+1;
       if (startB > endB) then exit loop;
     end;
     if ($a_{endA}$ < $b_{endB}$) then begin  /* merge from right ends */
       $c_{endC}$ = $a_{endA}$;      endA = endA1;        endC = endC1;
       if (startA > endA) then exit loop;
     end
     else begin
       $c_{endC}$ = $b_{endB}$;      endB = endB1;         endC = endC1;
       if (startB > endB) then exit loop;
     end;
   forever;
   while (startA  endA) do begin
   /* Copy the remaining elements of A into C. */
     $c_{startC}$ = $a_{startA}$;    startA = startA+1;    startC = startC+1;
   end;
   while (startB  endB) do begin
   /* Copy the remaining elements of B into C. */
     $c_{startC}$ = $b_{startB}$;    startB = startB+1;    startC = startC+1;
   end;
end.
```

Fig. 7. Merging two sub-arrays from both ends.

comparisons are needed. Since $a_2, \ldots, a_{n-1}$ are all inserted into B_2, it would be nice to avoid testing these elements against the last element of B_1. On the other hand, only $n-1$ comparisons are needed for a nonascending array, where $a_2, \ldots, a_{n-1}$ are all inserted into B_1, and do not require testing against the starting element of B_2.

Here, we present a simple data-dependent criterion, which decides whether to test an element a_i with either B_1 or B_2 first. The criterion is based on the assumption that the majority of a long input array's elements should belong to long monotone sub-arrays. Of course, it is easy to generate an array where this assumption does not hold, but our work mostly concerns arrays with low NMS. If such an array is long then its monotone sub-arrays cannot all be short. At least one of them contains a rather high number of elements. This implies that the probability of an observed element a_i belonging to the same monotone sub-array as its predecessor a_{i-1} is rather high. Before we proceed, let us prove the following theorem.

Theorem 4. *Two different elements of the same nondescending monotone sub-array cannot be placed into B_1 but on the other hand, two elements of the same nonascending monotone sub-array cannot be placed into B_2.*

Proof 4. Let us have two different elements of the same nondescending sub-array: $a_i < a_j$, $i < j$. Let us suppose that we manage to insert a_i at the end of B_1. Since $j > i$, a_j is handled after a_i. It cannot be inserted into B_1 because only the elements that are smaller than the last element of B_1 can be inserted later into this sub-array. The first part of the theorem is proved, and the proof for the second part is also evident. □

Since we expect that monotone sub-arrays of A are rather long, it is worthwhile to employ the following two rules based on Theorem 4:

1. If a_{i-1} was read from a nondescending monotone sub-array of A, then try to insert a_i into B_2 first. Only if this insertion fails, a_i should be tested against the end of B_1.
2. If a_{i-1} was read from a nonascending monotone sub-array of A, then try to insert a_i into B_1 first. Only if this insertion fails, a_i should be tested against the beginning of B_2.

In this way, we realize that a sorted input array (with a single monotone sub-array) requires only $n - 1$ comparisons no matter whether it is nonascending or nondescending. Obviously, the prediction of the testing order of sub-arrays B_1 or B_2 does not improve the time complexity of the algorithm, since $2n - 3$ and $n - 1$ comparisons both present $O(n)$. The running time however, is mostly improved by some 10%. Even when we sort integers (4 bytes), almost 40% of CPU time is saved when an input array is sorted in nondescending order, and more than 20% on average. If we have to compare more complex types (double precision floating point numbers or structures with several fields), this percentage increases importantly.

3.4. Implementation

In this section, we list a complete pseudo code of vertex sort. In comparison to the basic idea given in Fig. 4, the following modifications are shown in Fig. 8:

- Only two arrays A and B are employed instead of $A_0, \ldots, A_r$, $B_{i,j}$; $i \in \{1, 2\}$, $j = r + 1, \ldots, R$.

```
Algorithm Vertex_Sort2(endA, endB1, startB2)
/*     endA: input array is A[0..endA];
   endB1, startB2: the first and the last free index in B.
   The algorithm is called by Vertex_Sort2(n-1, 0, n-1). */

begin
  if (endA < 1) then return A; /* trivial problem */
  new_endA = endA-2;  new_endB1 = endB1;    new_startB2 = startB2;
  if ($a_0 \geq a_1$) then begin
    $b_{new\_endB1}$ = $a_0$;  $b_{new\_startB2}$ = $a_1$;  test_B1_first = true;
  end
  else begin
    $b_{new\_startB2}$ = $a_0$;  $b_{new\_endB1}$ = $a_1$;  test_B1_first = false;
  end;
  first_free = 0;
  i = 2;
  loop /* for elements $a_2$, …, $a_{endA}$ of A */
    if (i > endA) then exit loop;
    if test_B1_first then begin
      if ($a_i \leq b_{new\_endB1}$) then go to handle_B1;
      test_B1_first = false;
      if ($a_i \geq b_{new\_startB2}$) then go to handle_B2;
      go to handle_A;
    end
    else if ($a_i \geq b_{new\_startB2}$) then go to handle_B2;
    test_B1_first = true;
    if ($a_i \leq b_{new\_endB1}$) then go to handle_B1;
handle_A: /* Insert $a_i$ at the end of A. */
    $a_{first\_free}$ = $a_i$;  first_free = first_free+1;
    go to next_iteration;
handle_B1: /* Insert $a_i$ at the end of $B_1$. */
    new_endB1 = new_endB1+1; new_endA = new_endA-1; $b_{new\_endB1}$ = $a_i$;
    go to next_iteration;
handle_B2: /* Insert $a_i$ at the beginning of $B_2$. */
    new_startB2 = new_startB2-1;  new_endA = new_endA-1;
    $b_{new\_startB2}$ = $a_i$;
next_iteration: i = i+1;
  forever;
  /* recursion */
  Vertex_Sort2(new_endA, new_endB1-1, new_startB2+1);
  /* merging */
  endA2 = endA+1+endB1-startB1+endB2-startB2+1;
  Merge(B, startB1, endB1, B, startB2, endB2, A, endA+1, endA2);
  Merge(A, 0, endA, A, endA+1, endA2, C, 0, endA2);
  Tmp = A; A = C; C = A; /* Substitute the addresses A and C. */
end.
```

Fig. 8. The detailed pseudo code of vertex sort.

- Recursion level r is replaced by two parameters *endB1* and *startB2* representing borders of free space in B. The algorithm accepts the array $A_r = (a_0, \ldots, a_{endA})$, partitions it into $A_{r+1} = (a_0, \ldots, a_{new\_endA})$, $B_{1,r+1} = (b_{endB1}, \ldots, b_{new\_endB1-1})$ and $B_{2,r+} = (b_{new\_startB2+1}, \ldots, b_{startB2})$, recursively handles A_{r+1} and merges sorted sub-arrays A_{r+1} $B_{1,r+1}$ and $B_{2,r+1}$ after the recursion.

- The idea of predicting a sub-array ($B_{1,r+1}$ or $B_{2,r+1}$) which should contain an observed element a_i is incorporated. The Boolean variable *test_B1_first* is employed for this purpose. Of course, unpopular *goto* statements can be avoided by doubling some code sequences.

The body of the employed function *Merge* is given in Fig. 7. The first call merges the sub-arrays $B_1 = (b_{startB1}, \ldots, b_{endB1})$ and $B_2 = (b_{startB2}, \ldots, b_{endB2})$ into the sub-array A_2, stored in the array A just behind A_1. The second call merges the sub-arrays A_1 and A_2. Since the result is stored in the array C, the addresses A and C are substituted at the end.

3.5. Time complexity—weakness of vertex sort

In this section, the running times of quicksort and vertex sort are compared, and the time complexity of the latter is analysed theoretically. We have tested input arrays with 50,000, 100,000 and 200,000 elements and with various numbers of monotone sub-arrays. Since two-directional merging and the prediction of sorted sub-arrays' testing order are employed, vertex sort achieves practically the same speeds for nonascending and nondescending sub-arrays. The results for quicksort do not depend a lot on this fact either. Therefore, we do not have to consider this difference anymore. The results are given in Table 2. Each measurement was obtained as the average time of 400 measurements. Sequences of nondescending monotone sub-arrays, nonascending monotone sub-arrays, alternating and random sequences of nonascending and nondescending sub-arrays of equal and randomly distributed lengths have all been

Table 2
Comparison of CPU times (in seconds) of quicksort and vertex sort

NMS	$n = 50{,}000$		$n = 100{,}000$		$n = 200{,}000$	
	Quicksort	Vertex sort	Quicksort	Vertex sort	Quicksort	Vertex sort
1	0.006	0.003	0.014	0.007	0.031	0.015
2	0.062	0.005	0.170	0.012	0.822	0.020
3	0.014	0.006	0.023	0.015	0.051	0.026
5	0.013	0.008	0.022	0.018	0.048	0.040
8	0.015	0.012	0.034	0.026	0.084	0.057
9	**0.014**	**0.013**	**0.027**	**0.030**	**0.052**	**0.063**
10	**0.015**	**0.014**	**0.034**	**0.033**	**0.077**	**0.066**
11	**0.014**	**0.015**	**0.024**	**0.036**	**0.053**	**0.071**
12	**0.015**	**0.015**	**0.030**	**0.041**	**0.076**	**0.076**
13	0.014	0.017	0.025	0.043	0.054	0.082
14	0.015	0.019	0.029	0.044	0.067	0.094
50	0.015	0.056	0.027	0.139	0.067	0.282
Random	0.017	0.305	0.032	1.014	0.069	2.834
$n/3$	0.015	14.844	0.031	?????	0.053	?????

generated and tested. No particular result has differed from the average by more than 20%. The following conclusions can be made:

1. Vertex sort has linear time complexity if the input array is already sorted.
2. Running time of vertex sort increases linearly with NMS.
3. Vertex sort is faster than quicksort when the input array has low NMS.
4. In random cases with large NMS, vertex sort is much slower than quicksort.

Conclusions 1 and 2 prove that vertex sort is adaptive with respect to NMS, but it is not optimal. Quicksort runs in $\Omega(n\log n)$ in random cases, but vertex sort obviously does not reach this speed. Theorems 5 and 6 will help us to estimate its time complexity theoretically. The term that an element of the input array is *eliminated* means that it is inserted either into B_1 or into B_2 and, therefore, it is absent in the input for subsequent iterations.

Theorem 5. *Each iteration of vertex sort reduces NMS at least by one*: $NMS(A_{r+1}) < NMS(A_r)$.

Proof 5. Let $A' = (a_0, \ldots, a_s)$, $s < n$, represent the first monotone sub-array in A. If A' is nonascending then $a_0, a_2, \ldots, a_s$ are all inserted into B_1, and a_1 is inserted into B_2. Otherwise $a_0, a_2, \ldots, a_s$ are all inserted into B_2, and a_1 is inserted into B_1. □

Theorem 6. *Each iteration of vertex sort eliminates at least three elements of the input array.*

Proof 6. The proof is trivial. The first three elements a_0, a_1 and a_2 are certainly eliminated. The element $\max(a_0, a_1)$ is inserted into B_1, and $\min(a_0, a_1)$ is inserted into B_2. If $a_2 \leqslant \max(a_0, a_1)$ then a_2 is inserted into B_1, otherwise it is inserted into B_2. □

Of course, the last iteration may be an exception if A_{k-1} contains two elements only. Theorem 6 has two important corollaries. The proofs are trivial and left to the reader.

1. If n_i represents the number of elements in the input array in the ith iteration then $n_i \leqslant n - 3i$.
2. Vertex sort requires, at most, $\lceil n/3 \rceil$ iterations. More precisely, $n/3$ iterations are needed if $n = 3m$ (n is a multiple of 3), $(n-1)/3$ iterations if $n = 3m+1$, or $(n+1)/3$ iterations if $n = 3m-1$.

Let us suppose that the input array in the ith iteration contains n_i elements. Then, at most, $1 + 2(n_i - 2) = 2n_i - 3$ comparisons are required in the decomposition phase and, at most, $\max((|B_{i+1,1}| + |B_{i+1,2}|-1) + |B_{i+1,1}| + |B_{i+1,2}| + |A_{i+1}| - 1) = (n_i - 1 - 1) + n_i - 1 = 2n_i - 3$ comparisons in the merging phase. The total number of comparisons in k iterations is:

$$\sum_{i=0}^{k-1} 2(2n_{i-3}) = 4\sum_{i=0}^{k-1} n_i - 6k \leqslant 4\sum_{i=0}^{k-1}(n-3i) - 6k$$
$$= 4\left(nk - 3\frac{(k-1)k}{2}\right) - 6k = 2k(2n-3k). \qquad (6)$$

We take into consideration $1 \leqslant k \leqslant \lceil n/3 \rceil$, and obtain: $2n-3 \geqslant 2n-3k \geqslant n-1$. This implies $2n-3k = \mathrm{O}(n)$, and $2k(2n-3k) = \mathrm{O}(kn)$. In a similar way, we can establish that the number of rewritings belongs to $\mathrm{O}(kn)$ as well. If the array is already sorted, the algorithm requires a single iteration ($k = 1$) and runs in $\mathrm{O}(n)$. The worst case appears when the number of iterations is $\lceil n/3 \rceil$. $A = (100, 99, -99, 98, 97, -97, 96, 95, -95, \ldots)$ represents the worst-case example where each iteration (except eventually the last one) eliminates exactly three elements. The last row of Table 2 shows time measurements for longer arrays constructed in the same way. Quicksort handles them in similar times as random cases are handled, but vertex sort is extremely slow. Our computer was even unable to handle examples with 100,000 or 200,000 elements (since the total number of input parameters and local variables is rather high, we had filled up the internal stack). Eq. (7) gives a rather precise upper bound for the worst-case time complexity:

$$2k(2n-3k) = \begin{cases} 2\frac{n-1}{3}\left(2n - 3\frac{n-1}{3}\right) = \frac{2}{3}(n^2-1) = \mathrm{O}(n^2); \\ \quad n = 2m+1, \\ 2\frac{2}{3}\left(2n - 3\frac{n}{3}\right) = \frac{2}{3}n^2 = \mathrm{O}(n^2); \\ \quad n = 2m, \\ 2\frac{n+1}{3}\left(2n - 3 + 1\frac{n+1}{3}\right) = \frac{2}{3}(n^2-1) = \mathrm{O}(n^2-1); \\ \quad n = 2m-1. \end{cases} \qquad (7)$$

4. Smart quicksort—combination of vertex sort and quicksort

Since the time performances of vertex sort seem to be just the opposite as those obtained by quicksort, a logical decision is to combine both algorithms and to benefit from both in particular cases. Here, we present such a combination named *smart quicksort*. The idea is rather simple. The speed of vertex sort decreases by increasing NMS. On the other hand, quicksort achieves excellent time performances when the data are random i.e. when NMS is rather high. A certain number of monotone sub-arrays must exist where the speeds of both algorithms are nearly the same. If NMS is lower than this *threshold*, it is better to use vertex sort, otherwise quicksort is more suitable. In Table 2, the measurements, when both algorithms achieve similar speeds, are written boldly. This happens when NMS is between 9 and 12. All these values are acceptable for the threshold. 12 seems the most suitable from our experience.

How to count monotone sub-arrays in the input array? Counting certainly requires some time. We have to compare all pairs of neighbouring elements. In addition, we have to remember whether the array was descending or ascending just before the current comparison. We use variable *ordering* with four possible states for this task: NOTHING_YET, CHANGED, ASCENDING and DESCENDING. All possible transitions between these states are shown in Fig. 9. Whenever *ordering* is set to CHANGED, the beginning of a new monotone sub-array is encountered, and we increment the counter *NMS*. This task can obviously be performed in linear time, but it still slows down the algorithm. For this reason, we have decided on a slightly more bit modified and, certainly, more elegant solution. Two accelerations are introduced:

1. Instead of counting monotone sub-arrays in the input array, we run the first iteration of vertex sort and count monotone sub-arrays in the sub-array A_1 only. Then we use the threshold to decide whether to sort A_1 recursively with vertex sort or to employ quicksort. This variation results in at least two advantages:
   - Since all the operations concerning NMS are performed only when an element is inserted into A_1, we do not spend additional time when the input array is already sorted.

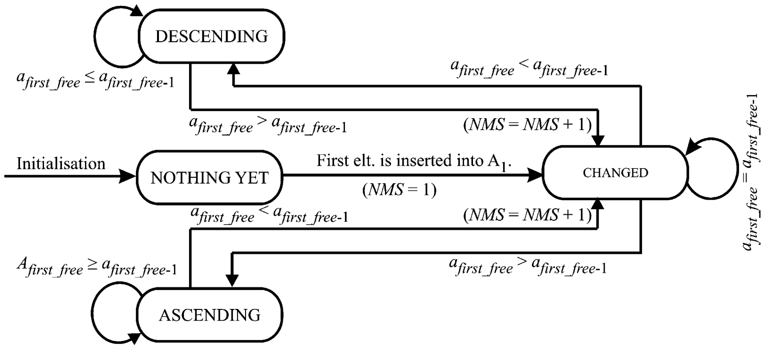

DESCENDING
$a_{first\_free} \le a_{first\_free-1}$
$a_{first\_free} < a_{first\_free-1}$
$a_{first\_free} > a_{first\_free-1}$
$(NMS = NMS + 1)$
Initialisation
NOTHING YET
First elt. is inserted into A_1.
$(NMS = 1)$
CHANGED
$a_{first\_free} = a_{first\_free-1}$
$a_{first\_free} < a_{first\_free-1}$
$(NMS = NMS + 1)$
$A_{first\_free} \ge a_{first\_free-1}$
$a_{first\_free} > a_{first\_free-1}$
ASCENDING

Fig. 9. Counting monotone sub-arrays in the sub-array A_1.

- A_1 is usually much shorter than the whole input array and, therefore, the code that handles NMS is executed less times.

2. After the counter *monotone* exceeds the threshold, NMS need not be calculated any more.

5. Time measurements and analysis of the time and storage complexities

We tested smart quicksort using the same testing data sets as for quicksort and vertex sort (Table 2). The measurements are given in Table 3. Each of them almost matches the best of both results (by quicksort and vertex sort) from Table 2. Slight differences were obtained particularly when smart quicksort runs quicksort because this is always pre-processed by the first iteration of vertex sort. Besides this, smart quicksort needs some time for counting monotone sub-arrays.

Table 4 gives time measurements for the triangulation of convex polygons with 5,000, 10,000, 20,000 and 40,000 vertices. Table 1 identifies the quadratic growth of time by increasing the number of vertices. This problem is now

Table 3
Time measurements (in seconds) confirm decision for smart quicksort

NMS	$n = 50{,}000$	$n = 100{,}000$	$n = 200{,}000$
1 (sorted array)	0.004	0.007	0.016
2	0.006	0.012	0.022
3	0.007	0.015	0.028
5	0.009	0.019	0.043
8	0.012	0.028	0.059
9	0.014	0.031	0.067
10	0.015	0.034	0.069
11	0.016	0.037	0.075
12	0.015	0.041	0.081
13	0.015	0.026	0.085
14	0.015	0.029	0.065
50	0.015	0.030	0.070
Random data	0.016	0.031	0.082

Table 4
Convex polygons are triangulated in linear time

Number of vertices	$n_1 = 5000$	$n_2 = 10{,}000$	$n_3 = 20{,}000$	$n_4 = 40{,}000$
Time [s]	0.0234	0.0470	0.0959	0.1914
Increase [n_i/n_{i1}]		2.01	2.04	2.00
Increase [n_i/n_1]	1	2.01	4.10	8.18

avoided by employing smart quicksort instead of quicksort. Obviously, double number of polygon vertices implies double running time. This confirms the linear time complexity, as stated in [1]. We have also tested the realistic example from Fig. 3 and the convex polygon with the same number of vertices. The latter was triangulated in 0.133 s. This is much less than the 0.7 s spent when quicksort was employed. The running time for the example from Fig. 3 remains practically identical (0.188 instead of 0.185 s).

The theoretical analysis reveals some weaknesses in the algorithm. The time complexity should be evaluated as a sum of the time complexity $T_1(n) = O(n)$ of the first iteration, and the time complexities of recursive calls. Both situations, when vertex sort or quicksort are employed, should be observed separately. Let us label their time complexities $T_v(n)$ and $T_q(n)$ respectively. We know that vertex sort is executed in $O(kn)$ where k is the number of recursive calls. Since smart quicksort executes vertex sort only if NMS(A_1) is smaller than the threshold, and since each iteration of vertex sort reduces NMS at least by one (Theorem 5), we can conclude that the number of recursive calls is smaller than the threshold. Since the threshold 12 is rather low, we obtain $T_v(n) = T_1(n)+O(kn) \leqslant O(n)+O(11n) = O(n)$.

By employing vertex sort for handling nearly sorted arrays, we avoid many situations where quicksort would achieve quadratic time complexity. The question arises as to whether all the remaining cases are random enough to be handled by quicksort in $\Theta(n \log n)$. Unfortunately, the answer is negative. In continuation, we will construct some input arrays which require enormous running times for quicksort.

Let A consist of a rather large number k of monotone sub-arrays. The first sub-array contains only three elements $a_0 = n$, $a_1 = n-1$, $a_2 = -n$. All the other elements are between $-n$ and $n-1$. This guarantees that the first iteration eliminates only the elements a_0, a_1 and a_2 Since k is a large number, the remaining sub-array A_1 will be handled by quicksort. Let A_1 consist of $k-1$

Table 5
The random is lost by increasing number of monotone subarrays

NMS	n=50,000	n=100,000	n=200,000
Random	0.016	0.031	0.082
n/1000	0.015	0.038	0.101
n/500	0.019	0.054	0.213
n/250	0.029	0.100	0.364
n/100	0.058	0.221	0.831
n/50	0.109	0.435	1.578
n/10	0.531	2.254	9.015
n/5	1.225	4.953	30.453
n/3	2.110	12.555	???
n/21	6.281	45.844	???

equal monotone sub-arrays $(1, 2, \ldots, \lfloor (n-3)/(k-1) \rfloor)$. The last monotone sub-array may be longer if $k-1$ is not a divisor of $n-3$. An example of such input array is (17, 16, −17, 1, ..., 4, 1, ... ,4, 1, ..., 6), where $n = 17$, $k = 4$. Table 5 shows time measurements for arrays with 50,000, 100,000 and 200,000 elements with various numbers of monotone sub-arrays k depending on n. For $k = n/1000$, the results are still close to those obtained in the random case, although they become slightly worse by increasing n. In all other cases where $k(n) > n/1000$, the results significantly differ from the random case. The time complexity obviously does not belong to $\Theta(n \log n)$.

A common ingredient of arrays constructed in the described way is that all the keys a_i, $i > 2$, are repeated several times. This is however, not the only reason for catastrophic results. Table 6 presents time measurements for a polygon triangulation which employs smart quicksort. Some of the employed polygons are shown in Fig. 10. They have 28,012 vertices each. Besides the convex and concave vertices visible in the figure, each example consists of a large number of regular vertices. The presented polygons vary in number of split and merge vertices and, consequently, in NMS. In all examples, each vertex appears exactly once and, therefore, none of the keys is multiple. However, the measured time for the polygon with 100 split and merge vertices (and $202 = n/137$ monotone sub-arrays) is comparable with the random case from Fig. 3 (0.188 s), but the results for higher numbers of monotone sub-arrays are catastrophic.

Finally, let us analyse the storage complexity. In Section 3.1, we have established that vertex sort requires three arrays with n elements each. If quicksort is employed, it sorts A_1 in place. After this, A_1, B_1 and B_2 are merged in the same way as with vertex sort. Obviously, the storage complexity is $O(n)$ no matter whether quicksort or vertex sort is employed.

5.1. Future work

Our future work is oriented towards confirming of smart quicksort as a general sorting algorithm. The following tasks should be solved to achieve this goal.

- The storage complexity should be reduced to two arrays instead of three. We can merge B_1 and B_2 into A'_r before proceeding to the next decomposition iteration. They can be merged into the free space in array A between

Table 6
Triangulation of polygons with 28,012 vertices and with different numbers of split and merge vertices

Number of split and merge vertices	0	5	20	100	1,000	14,006
Time [s]	0.133	0.145	0.158	0.205	0.672	28.922

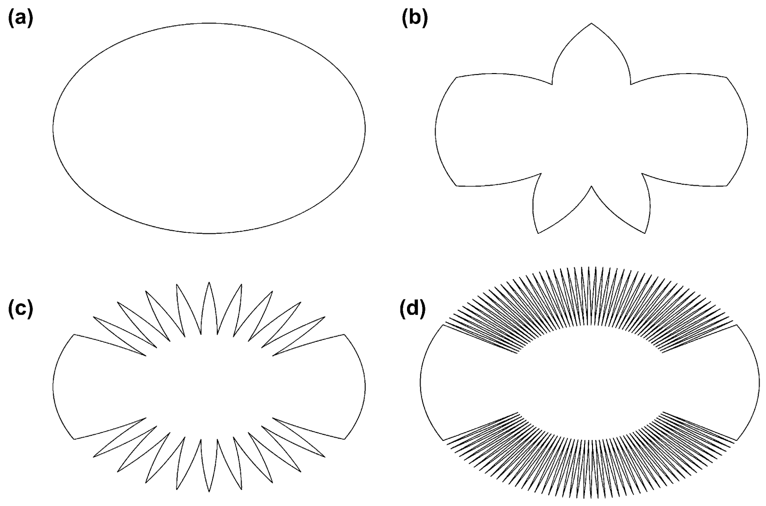


Fig. 10. Polygons with 0, 5, 20, and 100 split and merge vertices of 28,012.

sub-arrays A_r and A'_{r-1} obtained by merging in the previous iteration. After the decomposition is terminated, we merge A_R and A'_R into A_{R-1} at the beginning of the array B, then A_{R-1} and A'_{R-1} into A_{R-2} at the beginning of the array A, then A_{R-2} and A'_{R-2} into A_{R-3} at the beginning of the array B again, and so on until the whole sorted array A_0 is obtained. If it is located in array B, we have to switch the addresses of A and B before deleting array B.

- Recursion should be removed. The above idea facilitates this task. Sorted sub-arrays should be merged instantaneously before proceeding to the next decomposition iteration.
- The threshold should be estimated analytically. We believe that the empirically obtained value 12 is affected too much by slow recursion performance. Therefore, we have to implement and analyse a nonrecursive version first.
- It is possible that some other descendants of quicksort give better results in crucial situations with large NMS than the employed median-of-3 quicksort. Quickersort [12], qsort [6], meansort [11], Bsort [15], Qsorte [16], Bentley's and McIlroy's combination of standard quicksort, median-of-3 quicksort and pseudomedian-of-9 quicksort [5] will all be tested.
- We may also introduce an additional higher threshold, and employ some asymptotically optimal comparison sort above this threshold. Heapsort and merge sort [3] are good candidates for this task.
- The above two ideas may be joined if this turns out reasonable.
- Final confirmation of a general sorting algorithm can only be achieved by proposing a parallel [4] and an external version [14], as well. The uni-proc-

essor (serial) internal version presented here is suitable for sorting a small array which fits the main memory. On the other hand, external sorting assumes large data arrays stored in files [14]. Since the idea of smart quicksort and, especially, the idea of vertex sort are based on merging independently sorted sub-arrays, they offer a good groundwork for the development of parallel internal, serial external and parallel external sorting algorithms.

6. Conclusion

This paper presents an original internal sorting algorithm. It is simply called 'vertex sort' because it was firstly developed to sort out polygon vertices in applications of computational geometry. The idea of the algorithm is rather simple. The input array is split into three sub-arrays, where the last two are already sorted, and the first one has to be sorted recursively. The algorithm is extremely efficient when the input array consists of a low number of monotone sub-arrays (NMS). Namely in such cases, quicksort achieves its worst time complexity $\Theta(n^2)$, but vertex sort runs in linear time. On the other hand, it is much less efficient when the data are completely random i.e. NMS is rather high. We propose a combination of vertex sort and quicksort, for this reason. During the first vertex sort iteration, the algorithm calculates NMS for the sub-array, which has to be handled recursively. If this number is smaller than the threshold (experimentally, the most suitable value 12 was determined), the sub-array is sorted by vertex sort, otherwise quicksort is employed. We call this combination of two algorithms 'smart quicksort'. Smart quicksort is more efficient than quicksort since it beats the latter in nearly sorted examples, and since both algorithms achieve practically the same times in the random case. We could say it is 'intelligent' because it solves the easiest problems in the shortest time.

In future, we plan some improvements to smart quicksort. First of all, we would like to avoid as many worst case situations as possible. Various descendants of quicksort and the idea of combining them by asymptotically optimal comparison sorting algorithms will be studied. In addition, we already have ideas on how to reduce spatial requirements from three to two arrays only, and how to remove recursion.

Acknowledgement

We are grateful to the Ministry of Education, Science and Sport of Republic of Slovenia for supporting this research under the project P2-0041 Computer systems, methodologies and intelligent services.

References

[1] M. Berg, M. Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry, Algorithms and Applications, Springer Verlag, Berlin, 1997.
[2] C.R. Cook, D.J. Kim, Best sorting algorithm for nearly sorted lists, Communications of the ACM 11 (1980) 620–624.
[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, Massachusetts, London, England, 1990.
[4] N. Deo, D. Sarkar, Parallel algorithms for merging and sorting, Information Sciences 56 (1991) 151–161.
[5] M. Durand, Asymptotic analysis of an optimized quicksort algorithm, Information Processing Letters 2 (2003) 73–77.
[6] M.H. van Emden, Increasing the efficiency of quicksort, Communications of the ACM 9 (1970) 563–567.
[7] V. Estivill-Castro, D. Wood, A survey of adaptive sorting algorithms, ACM Computing Surveys 4 (1992) 441–476.
[8] M.R. Garey, D.S. Johnson, F.P. Preparata, R.E. Tarjan, Triangulating a simple polygon, Information Processing Letters 7 (1978) 175–179.
[9] C.A.R. Hoare, Quicksort, Computer Journal 1 (1962) 10–15.
[10] D.T. Lee, F.P. Preparata, Location of a point in a planar subdivision and its applications, SIAM Journal of Computing 6 (1977) 594–606.
[11] D. Motzkin, Meansort, Communications of the ACM 4 (1983) 250–251.
[12] R.S. Scowen, Algorithm 271: Quickersort, Communications of the ACM 11 (1965) 669–670.
[13] R.C. Singleton, Algorithm 347: An efficient algorithm for sorting with minimal storage, Communications of the ACM 3 (1969) 186–187.
[14] D. Taniar, J.W. Rahayu, Parallel database sorting, Information Sciences 146 (2002) 171–219.
[15] R.L. Wainwright, A class of sorting algorithms based on quicksort, Communications of the ACM 4 (1985) 396–402.
[16] R. L. Wainwright, Quicksort algorithms with an early exit for sorted subfiles, in: Proceedings of the 15th Annual Conference on Computer Science, ACM Press, New York, 1987, pp. 669–670.
[17] B. Žalik, Merging a set of polygons, Computer and Graphics 1 (2001) 77–88.