IEICE TRANS. INF. & SYST., VOL.E94–D, NO.2 FEBRUARY 2011

PAPER Special Section on Foundations of Computer Science — Mathematical Foundations and Applications of Algorithms and Computer Science —

Adaptive Algorithms for Planar Convex Hull Problems*

Hee-Kap AHN[†], *Nonmember and* **Yoshio OKAMOTO**[††a)], *Member*

SUMMARY We study problems in computational geometry from the viewpoint of adaptive algorithms. Adaptive algorithms have been extensively studied for the sorting problem, and in this paper we generalize the framework to geometric problems. To this end, we think of geometric problems as permutation (or rearrangement) problems of arrays, and define the "presortedness" as a distance from the input array to the desired output array. We call an algorithm *adaptive* if it runs faster when a given input array is closer to the desired output, and furthermore it does not make use of any information of the presortedness. As a case study, we look into the planar convex hull problem for which we discover two natural formulations as permutation problems. An interesting phenomenon that we prove is that for one formulation the problem can be solved adaptively, but for the other formulation no adaptive algorithm can be better than an optimal output-sensitive algorithm for the planar convex hull problem. To further pursue the possibility of adaptive computational geometry, we also consider constructing a kd-tree.

key words: *adaptive algorithms, convex hulls, computational geometry*

1. Introduction

One can think of computational geometry as a generalization of numerical problems (namely, 1-dimensional problems) to higher dimensions. A typical example is the 2-dimensional convex hull computation, which can be thought of as a generalization of sorting an array of numbers.

This work is motivated by a thorough treatment for sorting problems to take "presortedness" into account in the analysis of the algorithms. In certain cases one expects sorting algorithms to run faster if a given input is almost sorted. Mehlhorn [1] introduced the term "adaptive sorting algorithms" for those with such a property. A formal framework for the worst-case analysis of adaptive sorting algorithms was introduced by Mannila [2], and the framework is well surveyed by Estivill-Castro and Wood [3].

An adaptive sorting algorithm has several characteristics. First, it runs faster if the presortedness is high. Second, the algorithm does not use any information of the presortedness. That is a reason why it is called "adaptive."

In this paper, we study *adaptive computational geometry*. To apply the adaptiveness framework to geometric problems, we want to think of the problems as permutation problems. That is, we are given an array of objects (points, segments, etc.), and we output a permutation (or a rearrangement) of the objects that represents the desired answer. Naturally, the sorting problem is such a permutation problem, and the planar convex hull problem can be seen as a permutation problem (and actually, a lower bound of the convex hull algorithm is given by a reduction from the sorting problem). Indeed, this work is also motivated by recent studies on in-place geometric algorithms that treat some geometric problems as permutation problems [4]–[6].

Several "presortedness" measures have been proposed [3]. In this work, we use the oldest and the most frequently used measure: the number of inversions. Given two linear orders $\leq_1, \leq_2$ on X, an *inversion* is an ordered pair $(i, j) \in X^2$ such that $i <_1 j$ and $j <_2 i$. We denote the total number of inversions for $\leq_1, \leq_2$ by $\mathsf{inv}(\leq_1, \leq_2)$. Note that $\mathsf{inv}(\leq_1, \leq_2) = 0$ if and only if the order $\leq_1$ conforms to the order $\leq_2$, and therefore we may regard the number of inversions as an appropriate measure of the presortedness.

As a case study, we consider the planar convex hull computation: given a set of points in the plane, we want to compute its convex hull. For this problem, we discover two natural formulations as permutation problems. In both formulations, we require the points on the boundary of the convex hull to be sorted in clockwise order, but they are different in the treatment of the points in the interior. In the first formulation the interior points are required to be sorted (by their x-coordinates) while in the second formulation the interior points are not required so. Interestingly, this makes a huge difference in terms of complexity. We show that in the first formulation the problem can be solved in $O(n(1 + \log(1 + k)))$ time when k is the number of inversions in a given array of n points with respect to the desired output. Since $k \leq \binom{n}{2}$, the running-time bound can be as bad as $O(n \log n)$. Hence, this is still worst-case optimal with respect to n. On the other hand, in the second formulation we give a lower bound of $\Omega(n \log h)$ for computing the convex hull, where h is the complexity of the convex hull. This shows that the second formulation does not allow us to de-

Manuscript received March 28, 2010.
Manuscript revised June 7, 2010.

[†]The author is with the Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH), Korea.
[††]The author is with the Graduate School of Infomation Science and Engineering, Tokyo Institute of Technology, Tokyo, 152–8552 Japan.

*Work by Ahn was supported by the National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development. Work by Okamoto was supported by Global COE Program "Computationism as a Foundation for the Sciences" and Grant-in-Aid for Scientific Research from Ministry of Education, Science and Culture, Japan, and Japan Society for the Promotion of Science.

a) E-mail: okamoto@is.titech.ac.jp

DOI: 10.1587/transinf.E94.D.182

sign any adaptive algorithm. This kind of phenomenon has not been seen for any other problems for which the adaptiveness framework was applied.

A natural question arises here: since the convex hull of n points in the plane can be computed in linear time once they are sorted along a line or around a point, why do we need another adaptive algorithm other than an optimal adaptive sorting algorithm? An answer to the question is that any existing adaptive sorting method does not reflect the presortedness of a point set on its convex hull: for example, consider an input array A of n points whose i-th element is a point with coordinates $(i, (-1)^{i-1}\sqrt{i})$, for $1 \le i \le n$. Clearly the input points are already sorted along the x-axis, but not along its convex hull. As a result, A has no inversion on sorting along the x-axis, but has $\Omega(n^2)$ inversions on convex hull. As another example, consider an input array A' consisting of the same points in A, but given in a different order: the i-th element is a point with coordinates $(2i-1, \sqrt{2i-1})$ for $1 \le i \le \lceil \frac{n}{2} \rceil$, and $(2(n-i+1), -\sqrt{2(n-i+1)})$ for $\lceil \frac{n}{2} \rceil + 1 \le i \le n$. The points in A' are already given in sequence along their convex hull clockwise, therefore A' has no inversion on convex hull at all. There are, however, $\Omega(n^2)$ inversions in A' on sorting.

We also study an adaptive algorithm to construct a kd-tree for a point set on the plane.

(1) Related work

There are a huge number of articles discussing the adaptive sorting problem. We recommend the readers to consult a survey by Estivill-Castro and Wood [3]. Adaptive sorting algorithms are also discussed in terms of integer sorting [7] and I/O-efficiency (both cache-aware and cache-oblivious) [8].

There are several papers which apply the adaptiveness framework to problems other than sorting. Demaine, López-Ortiz, and Munro [9] considered some set operations on sorted sets, and gave adaptive algorithms with respect to a certain measure of difficulty of the problem. The problem has been motivated from a database application, and this line of research was followed by some subsequent papers [10], [11].

Levcopoulos, Lingas, and Mitchell [12] were the first to study a computational-geometric problem in the adaptive framework. They studied the convex hull computation of a (possibly self-intersecting) piecewise linear chain. Their consideration relies on the fact that the convex hull can be computed in linear time when the chain does not have a self-intersection. Since the x-monotone non-self-intersecting chain can be seen as a sorted sequence, their study is a generalization of the adaptive sorting framework. However, they did not look at the problem as a permutation problem. Besides, Baran and Demaine [13] and Barbay and Chen [14] studied other geometric problems. However, their adaptiveness framework does not look at the presortedness and is completely different from the viewpoint of this work. In this sense, this paper studies the most fundamental counterpart of the adaptive sorting problem in computational geometry.

(2) Notation

An array A of n elements is indexed by $1, \ldots, n$. The i-th element of A is denoted by $A[i]$, $i \in \{1, \ldots, n\}$. The subarray of A consisting of $A[i], \ldots, A[j]$ is denoted by $A[i..j]$. For a subset A' of A, the difference $A \setminus A'$ denotes the array consisting of the elements of $A \setminus A'$ and ordered as in A. The concatenation of two arrays A and B (in this order) is denoted by $A \circ B$. For a set (or an array) P of points, we denote by $\mathrm{conv}(P)$ the convex hull of P, and by $\partial\mathrm{conv}(P)$ the boundary of $\mathrm{conv}(P)$.

(3) Weak orders

For the investigation of geometric problems, it is convenient to extend the framework for linear orders to weak orders. In general, a binary relation $\lesssim$ on a set X is a *weak order* on X if it is reflexive ($x \lesssim x$ for all $x \in X$), transitive ($x \lesssim y$ and $y \lesssim z$ imply $x \lesssim z$ for all $x, y, z \in X$), and total ($x \lesssim y$ or $y \lesssim x$ for all $x, y \in X$). We say $x \sim y$ if $x \lesssim y$ and $y \lesssim x$, and $x < y$ if $x \lesssim y$ and not $x \sim y$. Note that a weak order is a linear order if and only if it is also antisymmetric. In other words, $x \sim y$ does not necessarily imply $x = y$ for a weak order $\lesssim$. Given two weak orders $\lesssim_1, \lesssim_2$ on X, an *inversion* is an ordered pair $(i, j) \in X^2$ such that $i <_1 j$ and $j <_2 i$. We denote the total number of inversions for $\lesssim_1, \lesssim_2$ by $\mathsf{inv}(\lesssim_1, \lesssim_2)$.

2. Planar Convex Hulls

Informally speaking, in the *planar convex hull problem*, we are given a set P of n points in general position (i.e., no three points of P are collinear, and no two points have the same x-coordinate), and want to identify the points on the boundary of the convex hull of P. To cast the problem into the adaptive framework, we introduce the following two formulations.

2.1 First Formulation: The Interior Points Need to be Sorted

We are given P as an array of n points in the plane. The output is a rearrangement Q of the array P in the following way. If h is the number of points on $\partial\mathrm{conv}(P)$, then $Q[1..h]$ should be the array of these points sorted clockwise with $Q[1]$ being the leftmost point in P. Then, $Q[h+1..n]$ should be the array of points lying in the interior of $\mathrm{conv}(P)$, sorted by their x-coordinates. For the example in Fig. 1, the input array is

$$[p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}],$$

and the output array must be

$$[p_1, p_3, p_5, p_8, p_{12}, p_{14}, p_{10}, p_6, p_2, p_4, p_7, p_9, p_{11}, p_{13}].$$

Thus, we obtain two linear orders. The first order $\le_P$ is defined by the input array P as $P[i] \le_P P[j]$ for all $i \le j$. The other order $\le_Q$ is defined by the output array Q as $Q[i] \le_Q Q[j]$ for all $i \le j$. For these we may define the number of inversions. Notice that h is not a part of the input.

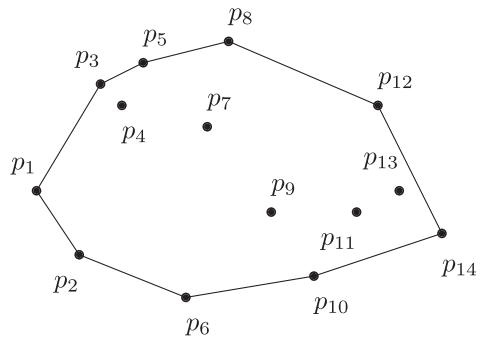

Fig. 1 A two-dimensional convex hull.

2.2 Second Formulation: The Interior Points Need not to be Sorted

In the first formulation, it would be unnatural to require the interior points to be sorted because we are often interested in the points on the boundary of the convex hull only. Therefore, it is natural to think the following variation. We are given P as an array of n points in the plane. The output is a rearrangement Q of the array P in the following way. If h is the number of points on $\partial\mathrm{conv}(P)$, then $Q[1..h]$ should be the array of these points sorted clockwise with $Q[1]$ being the leftmost point in P. Then, $Q[h{+}1..n]$ is any rearrangement of points lying in the interior of $\mathrm{conv}(P)$. So the output array Q is not uniquely determined from P. For the example in Fig. 1, $[p_1, p_3, p_5, p_8, p_{12}, p_{14}, p_{10}, p_6, p_2, p_9, p_4, p_{13}, p_7, p_{11}]$ and $[p_1, p_3, p_5, p_8, p_{12}, p_{14}, p_{10}, p_6, p_2, p_{11}, p_{13}, p_4, p_9, p_7]$ are possible outputs.

Note that this formulation has already been proposed in the literature on in-place algorithms [6].

Then, we define the following two weak orders. The first one $\leq_P$ is the same as the first formulation: $P[i] \leq_P P[j]$ for all integers $0 \leq i \leq j \leq n$. The other order $\lesssim_Q$ is defined from an output array Q as $Q[i] \lesssim_Q Q[j]$ if and only if $0 \leq i \leq j \leq n$ or $h{+}1 \leq j \leq i \leq n$. That means the interior points are indifferent in $\lesssim_Q$. Note that the order $\lesssim_Q$ does not depend on a particular choice of an output Q. This is the spot where we need a weak order since the output is not determined uniquely from the input.

2.3 Results

With these two formulations, we prove the following results.

- For the first formulation, we give an adaptive algorithm running in $\mathrm{O}(n(1 + \log(1 + k)))$ time where $k = \mathsf{inv}(\leq_P, \leq_Q)$. We also give a lower bound $\Omega(n(1 + \log(1 + k/n)))$ for the number of comparisons even if the number h of points on the boundary of the convex hull is constant.
- For the second formulation, we prove that any (fixed-degree) algebraic decision tree solving the problem has height at least $\Omega(n \log h)$. Therefore, with the second framework we cannot beat an optimal output-sensitive algorithm (running in $\mathrm{O}(n \log h)$ time, e.g. [15]) and thus any adaptive algorithm is meaningless.

In the subsequent sections, we will give proofs for our results.

3. An Adaptive Algorithm for the First Formulation

When designing adaptive convex hull algorithms, we may encounter at least the following two difficulties. First, we cannot determine whether $p <_Q q$ just by looking at two points p, q. It depends on how the other points are placed around p, q. Second, related to the first one, if we want to proceed by divide-and-conquer and obtain a subset P' of P that yields the output Q', then it is not generally the case that Q' is a subarray of Q; i.e., subsets do not inherit the linear order. These two issues do not arise in the sorting problem, where any two numbers can be compared just by looking at them, and any smaller subsets inherit the order.

Our algorithm below overcomes these issues and is shown to be adaptive. The call to ConvexHull(P) identifies the upper chain $U(P)$ of $\mathrm{conv}(P)$ in the increasing order of x-coordinates, the lower chain $L(P)$ of $\mathrm{conv}(P)$ in the decreasing order of x-coordinates, and the set $I(P)$ of points in the interior of $\mathrm{conv}(P)$ in the increasing order of x-coordinates. The desired output is the concatenation $U(P) \circ (L(P) \setminus \{\text{the rightmost point, the leftmost point}\}) \circ I(P)$.

Algorithm: ConvexHull(P)

Step 1: If P is arranged as a desired output, then identify $U(P), L(P), I(P)$ and halt.

Step 2: Otherwise, compute a vertical bisector s of P. Let P_A be the set of points in P left to s, and P_B be the set of points in P right to s.

Step 3: Compute the upper tangent u and the lower tangent ℓ both common to $\mathrm{conv}(P_A)$ and $\mathrm{conv}(P_B)$. Let $a_u \in P_A$ and $b_u \in P_B$ be the two points spanning u. Similarly, let $a_\ell \in P_A$ and $b_\ell \in P_B$ be the two points spanning ℓ. Let P_L be the set of points in P that lie left to the line spanned by a_u, a_ℓ, P_R be the set of points in P that lie right to the line spanned by b_u, b_ℓ, and $P_M = P \setminus (P_L \cup P_R)$. Note that $P_L \subseteq P_A$ and $P_R \subseteq P_B$.

Step 4: Call ConvexHull(P_L) and ConvexHull(P_R) to obtain $U(P_L), L(P_L), I(P_L), U(P_R), L(P_R), I(P_R)$. Identify $U(P) = U(P_L) \circ U(P_R)$ and $L(P) = L(P_R) \circ L(P_L)$.

Step 5: Sort the points in P_M by their x-coordinates to obtain the sorted sequence $S(P_M)$, and apply the merge sort for $I(P_L), S(P_M), I(P_R)$. Identify the result as $I(P)$. Halt.

The algorithm is based on the same idea as the divide-and-conquer algorithm by Kirkpatrick and Seidel [15]. In their algorithm the upper hull and the lower hull are computed separately, but we cannot do so here since we may lose the adaptiveness. Rather, we compute the upper and lower hulls simultaneously. The correctness of the algorithm is straightforward.

Now we estimate the running time. From now on, denote by n the number of input points in P, and by k the number of inversions between $\leq_P$ and $\leq_Q$.

Step 1 can be executed in $O(n)$ time as follows. First we find the leftmost point p and the rightmost point q of P in $O(n)$ time. In the desired output, p should come first ($p = P[1]$) and q should come somewhere, say at the h'-th position ($q = P[h']$). Then we check whether the subarray $P[1..h']$ is a concave chain C_u by looking at turns at all three consecutive points. This can be done in $O(n)$ time. Next, we compute the second leftmost point p' on the lower hull of P in $O(n)$ time, and determine $h \geq h'$ such that $p' = P[h]$. Then, we check whether the subarray $P[h'..h]$ with p forms a convex chain C_ℓ in $O(n)$ time. Now the points $P[h+1..n]$ must lie in the interior of $\mathrm{conv}(P)$, and be sorted by their x-coordinates. First we check if they are sorted in $O(n)$ time. Then, for each point $r \in P[h+1..n]$ from left to right, we check if r lies between the concave chain C_u and the convex chain C_ℓ. This can be done in $O(n)$ time since they are all sorted.[†] Finally, we identify $U(P) = P[1..h']$, $L(P) = P[h'..h] \circ P[1]$, $I(P) = P[h+1..n]$. Thus, we can execute Step 1 in $O(n)$ time.

Step 2 reduces to the median finding problem, which can be solved in $O(n)$ time [16].

In Step 3, computing the upper and the lower tangents reduces to 2-dimensional linear programming (as in Kirkpatrick and Seidel [15]), which can be solved in $O(n)$ time [17]. Also it is straightforward to find P_L and P_R in $O(n)$ time. Note that $|P_L|, |P_R| \leq n/2$.

Step 4 involves recursive calls. A crucial observation is that a point of P on $\partial\mathrm{conv}(P)$ lies on $\partial\mathrm{conv}(P_L)$ or $\partial\mathrm{conv}(P_R)$, and a point of P_L (and P_R) on $\partial\mathrm{conv}(P_L)$ (and $\partial\mathrm{conv}(P_R)$ respectively) lies on $\partial\mathrm{conv}(P)$. Therefore, the desired output Q_L for ConvexHull(P_L) and the desired output Q_R for ConvexHull(P_R) are subsequences of Q. This way, we succeed in overcoming the second difficulty described before. If we denote by $t(n,k)$ the worst-case running time of ConvexHull(P) over all P with $|P| = n$ and $\mathsf{inv}(\leq_P, \leq_Q) = k$ (when Q is the desired output), Step 4 takes at most $t(|P_L|, k_L) + t(|P_R|, k_R)$ time, where k_L, k_R denote the number of inversions for P_L, P_R and (the restriction to P_L, P_R of) Q, respectively. Since P_L, P_R, P_M are disjoint subsequences of P, we have the following lemma.

Lemma 1. *Denote by k_L, k_R, k_M the number of inversions for P_L, P_R, P_M and (the restriction to P_L, P_R, P_M of) Q, respectively. Then, it holds that $k_L + k_R + k_M \leq k$.*

In Step 5, we sort the points in P_M. If we apply an adaptive sorting algorithm, say by Estivill-Castro and Wood [18], we can sort P_M in $O(|P_M|(1+\log(1+k_M)))$ time. Further, the merging can be done in $O(n)$ time in a standard way since $I(P_L), I(P_R), S(P_M)$ are all sorted.

Now we estimate the overall running time summarizing the discussion above. Consider all linear-time processing in Steps 1, 2, 3, 5 takes an time for some constant a and for all sufficiently large n, and the adaptive sorting in Step 5 takes $b|P_M|(1 + \log_2(1 + k_M))$ time for some constant b and for all sufficiently large n. If we denote the number of points in P_L, P_R, P_M by n_L, n_R, n_M, respectively (note that $n_L + n_R + n_M = n$), then we obtain the following recurrence:

$$\begin{aligned} t(n,k) \leq\ & an + t(n_L, k_L) + t(n_R, k_R) \\ & + bn_M(1 + \log_2(1 + k_M)) \end{aligned}$$

for sufficiently large n and $k \geq 1$. Note that for small n it holds $t(n,k) = O(1)$ and when $k \leq 0$ it holds that $t(n,k) = O(n)$. We now derive that $t(n,k) \leq cn(1 + \log_2(1 + k))$ for some constant c and for all sufficiently large n.

By induction, we obtain

$$\begin{aligned} t(n,k) \leq\ & an + cn_L(1 + \log_2(1 + k_L)) \\ & + cn_R(1 + \log_2(1 + k_R)) \\ & + bn_M(1 + \log_2(1 + k_M)). \end{aligned}$$

We choose c so that it satisfies $2b \leq c$. Let $n_L = \alpha n$ and $n_R = \beta n$. Then we have $0 \leq \alpha \leq 1/2$, $0 \leq \beta \leq 1/2$, and $n_M = (1 - \alpha - \beta)n$. Note that α and β are parameters that cannot be freely chosen but depend on the input. The recurrence becomes

$$\begin{aligned} t(n,k) &\leq an + c\alpha n(1 + \log_2(1 + k_L)) \\ &\quad + c\beta n(1 + \log_2(1 + k_R)) \\ &\quad + \frac{c}{2}(1 - \alpha - \beta)n(1 + \log_2(1 + k_M)) \\ &= an + c\left(\alpha + \beta + \frac{1 - \alpha - \beta}{2}\right)n \\ &\quad + cn\log_2(1 + k_L)^\alpha(1 + k_R)^\beta(1 + k_M)^{(1-\alpha-\beta)/2} \\ &\leq an + cn \\ &\quad + cn\log_2(1 + k_L)^\alpha(1 + k_R)^\beta(1 + k_M)^{(1-\alpha-\beta)/2}. \end{aligned}$$

Here, we want to know when the argument of the last logarithm $(1 + k_L)^\alpha(1 + k_R)^\beta(1 + k_M)^{(1-\alpha-\beta)/2}$ is maximized. Taking the logarithm further, we reduce this maximization to the following linear program with two variables α, β:

$$\begin{aligned} \text{maximize} \quad & \alpha\log_2(1 + k_L) + \beta\log_2(1 + k_R) \\ & + \frac{1 - \alpha - \beta}{2}\log_2(1 + k_M) \\ \text{subject to} \quad & 0 \leq \alpha, \beta \leq 1/2. \end{aligned}$$

This problem can be directly solved. We have four cases. Let $A = \log_2(1 + k_L) - \frac{1}{2}\log_2(1 + k_M)$ (that is the coefficient of α), and $B = \log_2(1 + k_R) - \frac{1}{2}\log_2(1 + k_M)$ (that is the coefficient of β).

Case 1: when $A \geq 0$ and $B \geq 0$. Then, the optimum is attained at $\alpha = \beta = 1/2$, and the optimal value is $(\log_2(1 + k_L) + \log_2(1 + k_R))/2$.

Case 2: when $A \geq 0$ and $B < 0$. Then, the optimum is attained at $\alpha = 1/2, \beta = 0$, and the optimal value is $(2\log_2(1 + k_L) + \log_2(1 + k_M))/4$.

[†]This is why we require the points in the interior to be sorted by their x-coordinates. If they are not, for each point in $P[h+1..n]$ we would need to search back and forth around C_u and C_ℓ so that we cannot achieve the linear running time.

Case 3: $A < 0$ and $B \geq 0$. Then, the optimum is attained at $\alpha = 0, \beta = 1/2$, and the optimal value is $(2\log_2(1 + k_R) + \log_2(1 + k_M))/4$.

Case 4: $A < 0$ and $B < 0$. Then, the optimum is attained at $\alpha = \beta = 0$, and the optimal value is $(\log_2(1 + k_M))/2$.

For each of these four cases, we proceed with the estimation of $t(n, k)$. When Case 1 occurs, we obtain

$$\begin{aligned} t(n,k) &\leq (a+c)n + cn\log_2(1+k_L)^{1/2}(1+k_R)^{1/2} \\ &\leq (a+c)n + cn\log_2\frac{(1+k_L)+(1+k_R)}{2} \\ &\leq (a+c)n + cn\log_2\frac{2+k}{2} \\ &\leq (a+c)n + cn\log_2\left(\frac{3}{4}(1+k)\right) \\ &= (a+c)n + \left(\log_2\frac{3}{4}\right)cn + cn\log_2(1+k), \end{aligned}$$

where we use a relation of arithmetic means and geometric means in the second inequality, Lemma 1 in the third inequality, and $\frac{2+k}{2} \leq \frac{3}{4}(1+k)$ for $k \geq 1$ in the second to last inequality. Thus, if we choose c so that it satisfies $a + (1 + \log_2\frac{3}{4})c \leq c$, then we obtain $t(n, k) \leq cn(1 + \log_2(1 + k))$ as desired. Note that $\log_2\frac{3}{4} \approx -0.415037$.

When Case 2 occurs, we obtain

$$\begin{aligned} t(n,k) &\leq (a+c)n + cn\log_2(1+k_L)^{1/2}(1+k_M)^{1/4} \\ &\leq (a+c)n + cn\log_2(1+k_L)^{1/2}(1+k_M)^{1/2}, \end{aligned}$$

and we then go along the same line as Case 1. Case 3 is also similar. When Case 4 occurs, we obtain

$$\begin{aligned} t(n,k) &\leq an + cn + cn\log_2(1+k_M)^{1/2} \\ &\leq an + cn + cn\log_2(1+k_M/2) \end{aligned}$$

and we proceed with the same argument. This way, we conclude $t(n, k) \leq cn(1 + \log_2(1 + k))$ for all of the four cases. We summarize the discussion in the following theorem.

Theorem 2. *The algorithm* ConvexHull *above computes the convex hull of a given (non-degenerate) point set in the plane in* $O(n(1 + \log(1 + k)))$ *time, where n is the number of input points and k is the number of inversions as defined in the first formulation, namely* $k = \mathsf{inv}(\leq_P, \leq_Q)$.

As for the lower bound it is easy to observe the following.

Theorem 3. *Any algorithm to solve the planar convex hull problem in the first formulation needs at least* $\Omega(n(1+\log(1+k/n)))$ *operations in the worst case even if the number of points on the boundary of the convex hull is constant (three). Here, n is the number of input points and k is the number of inversions as defined in the first formulation.*

Proof. We reduce the adaptive sorting problem to our problem. In the sorting problem we are given n numbers in an array, and need to sort them in increasing order. Guibas, McCreight, Plass, and Roberts [19] showed that any sorting algorithm needs at least $\Omega(n(1 + \log(1 + k/n)))$ comparisons in the worst case, where n is the size of an input array and k is the number of inversions between the positions (or indices) in the input array and the increasing order of numbers themselves.

Let A be an input array of size n for the sorting problem. Then, we construct an array P of $n + 3$ planar points as follows. For each number $A[i]$ in the input array, we set $P[i+3] = (A[i], \varepsilon^i)$ for sufficiently small ε. This determines $P[4..n+3]$. The other three points are determined as follows. Let ℓ be the smallest number in A, and u be the largest number in A. Then, we set $P[1] = (\ell - 1, 1), P[2] = (u+2, 1), P[3] = (u+1, -1)$. This completes the construction of the point set P, and we consider the planar convex hull problem when the input array is P. Let Q be the output array. Then we can see that $P[1], P[2]$ and $P[3]$ are the points on the boundary of the convex hull, and $k = \mathsf{inv}(\leq_P, \leq_Q)$. Furthermore, from Q we can extract the sorted sequence in the increasing order as the x-coordinates of $Q[4..n+3]$. Since ℓ and u can be found in linear time, this finishes the whole reduction. □

4. Lower Bound for the Second Formulation

To obtain a lower bound for the second formulation, we consider the following No Inversion Problem: Given an array P of (non-degenerate) point set in the plane, we want to determine whether $\mathsf{inv}(\leq_P, \leq_Q) = 0$. The following theorems show that this problem is as hard as the planar convex hull problem itself.

Theorem 4. *Any (fixed-degree) algebraic decision tree solving the* No Inversion Problem *has height at least* $\Omega((n-h)\log h)$*, where n is the number of input points and h is the number of points on the boundary of the convex hull.*

Proof. We construct a linear-time reduction to No Inversion Problem from the following CHIR Problem:[†] Given a regular convex h-gon R with its smallest circumscribing disk D, and $n-h$ points in D, determine whether all of these $n-h$ points lies in R. See Fig. 2 for an illustration. Kapoor and Ramanan [20] proved that any (fixed-degree) algebraic decision tree solving the CHIR Problem has height at least $\Omega((n-h)\log h)$.[††]

For the reduction, we are given a regular convex h-gon R with its smallest circumscribing disk D, and a set X of $n-h$ points in D. Then, we construct an array P of points that is supposed to be an instance of the No Inversion Problem as follows. At $P[1..h]$ we place the vertices of R in the clockwise order in such a way that $P[1]$ will be the leftmost one. Then, at $P[h+1..n]$ we place the points of X arbitrarily. We

[†]This is the abbreviation of "Convex hull inclusion with restriction" [20].

[††]Actually the CHIR problem by Kapoor and Ramanan [20] is a bit different from ours, but the lower bound proof of them can be easily adapted to our variation.

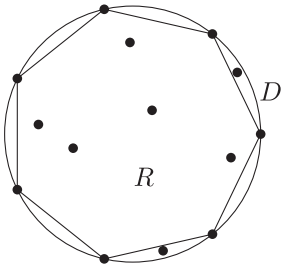

Fig. 2 The CHIR problem.

can see that P can be constructed in linear time. We can also see that $\mathsf{inv}(\leq_P, \lesssim_Q) = 0$ if and only if all points of X lie in R. Thus, the reduction is completed. □

Theorem 5. *Any (fixed-degree) algebraic decision tree solving the* No Inversion Problem *has height at least* $\Omega(n \log n)$*, where n is the number of input points.*

Proof. Follow the proof of Theorem 4, but this time we set $h = n/2$ in the CHIR Problem. Then, the same argument gives a desired lower bound. □

The following corollary is straightforward from the theorems above.

Corollary 6. *Any (fixed-degree) algebraic decision tree solving the* No Inversion Problem *has height at least* $\Omega(n \log h)$*, where n is the number of input points and h is the number of points on the boundary of the convex hull.*

Proof. When $h < n/2$, we have an $\Omega(n \log h)$ lower bound from Theorem 4. When $h \geq n/2$, we have an $\Omega(n \log h)$ lower bound from Theorem 5. □

As shown in the following theorem, the lower bound for the No Inversion Problem can be translated to the lower bound for the planar convex hull problem in the second formulation.

Theorem 7. *For the (fixed-degree) algebraic-computation-tree model, any algorithm to solve the planar convex hull problem in the second formulation requires at least* $\Omega(n \log h)$ *time, where n is the number of input points and h is the number of points on the boundary of the convex hull.*

Proof. Let A be an algorithm to solve the planar convex hull problem in the second formulation, and let Q be an output array from A when we input P into A. From Q, we can determine h in $O(n)$ time as Step 1 of the algorithm ConvexHull in the previous section. Therefore, by looking through $P[1..h]$ and $Q[1..h]$, we can determine whether $\mathsf{inv}(\leq_P, \lesssim_Q) = 0$ in $O(h)$ time. In this way, we can solve No Inversion Problem, and so A needs at least $\Omega(n \log h)$ time by Corollary 6. □

5. Constructing a *k*d-Tree Adaptively

To further pursue the possibility of adaptive computational geometry, we consider constructing a *k*d-tree. A *kd-tree* [21] can be thought of as a permutation of the points [5]. Namely, the root comes first and then the left subtree comes recursively, and finally the right subtree comes also recursively. We assume canonical orders in which the left side and the top side come as left subtrees, the right side and the bottom side come as right subtrees, and left subtrees always contain no more points than right subtrees. Figure 3 shows an example, in which the desired output is the array $[p_7, p_1, p_5, p_6, p_3, p_2, p_4, p_{11}, p_9, p_{12}, p_{10}, p_8, p_{13}]$. Note that a *k*d-tree is always balanced.

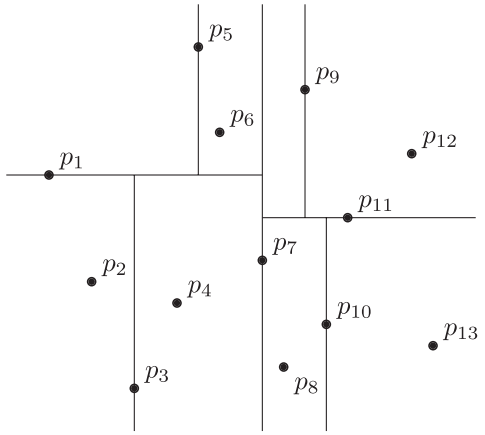

Fig. 3 A *k*d-tree.

Given an array of n (non-degenerate) points in the plane, we can determine whether it represents the *k*d-tree in $O(n)$ time in the following bottom-up fashion. First from the input array P of n points we construct a binary tree T recursively as follows. If P contains only one point, then T consists of a single node. Otherwise, $P[1]$ is the root of T, the binary tree for $P[2..\lceil \frac{n-1}{2} \rceil + 1]$ comes as the left subtree, and the binary tree for $P[\lceil \frac{n-1}{2} \rceil + 2..n]$ comes as the right subtree. This can be done in $O(n)$ time simply by scanning the input array. Then, for each subtree T' of T, we compute the four points on the boundary of the bounding box of the points contained in T'. This can be done in $O(n)$ time in total by traversing the tree in postorder and computing the bounding box of the points in each subtree. Then, for each node p of T, we check whether all points contained in the left subtree of p lie on the left of p (or above p depending on the depth of p in T), and similarly whether all points contained in the right subtree of p lie on the right of p (or below p). If it is the case for every node of T, then we can conclude that T is the *k*d-tree for P, and consequently P represents the *k*d-tree; Otherwise P does not represent the *k*d-tree. This check can also be done in $O(n)$ time with help of four extreme points associated with each subtree that we have already computed. Thus, the whole computation can be done in linear time.

Combining this procedure with a usual recursive construction of a *k*d-tree gives the following theorem.

Theorem 8. *We can compute the kd-tree of a given (non-*

degenerate) point set in the plane in $O(n(1+\log(1+k)))$ *time, where n is the number of input points and k is the number of inversions for the input and the desired output.*

Proof. The algorithm below does the job. It has an auxiliary argument "subdiv," which describes the direction for subdivision. Namely, when subdiv = v, we subdivide the point set by a vertical line (left and right); when subdiv = h, we subdivide the points by a horizontal line. Given an input array P, we execute KdTree(P, v).

Algorithm: KdTree(P, subdiv)

Step 1: If P is arranged as a desired output, then output P itself and halt. (Note: this case only deal when P contains at most one point.)

Step 2: Otherwise, we have two cases depending on subdiv.

Step 2-1: If subdiv = v, then we identify a point p in P with the median x-coordinate. Let P_L be the array of points in P that are left to p, and P_R be the array of points in P that are right to p. Then, output $\{p\} \circ$ KdTree(P_L, h) $\circ$ KdTree(P_R, h). Halt.

Step 2-2: If subdiv = h, then we identify a point p in P with the median y-coordinate. Let P_A be the array of points in P that are above p, and P_B be the array of points in P that are below p. Then, output $\{p\} \circ$ KdTree(P_A, v) $\circ$ KdTree(P_B, v). Halt.

The correctness of the algorithm immediately follows from the definition of kd-trees. To bound the running time, we just need to observe the following lemma which can be seen as easily as Lemma 1.

Lemma 9. *Denote by* k', k'' *the number of inversions for* P_L, P_R *(or* P_A, P_B*) and their rearrangements appearing exactly as subsequences of the desired output array. Then, it holds that* $k' + k'' \leq k$.

Then, the analysis similar to one in the proof of Theorem 2 goes. This concludes the proof of Theorem 8. □

A usual argument of the lower bound for the kd-tree construction gives an adaptive lower bound similar to Theorem 3.

Theorem 10. *Any algorithm to solve the kd-tree construction problem in the plane needs at least* $\Omega(n(1+\log(1+k/n)))$ *operations in the worst case. Here, n is the number of input points and k is the number of inversions between the input array and the desired output array.*

Proof. To sort an array A of n numbers, we just need to set up an array P of n planar points by $P[i] = (A[i], A[i]^2)$ for all $i \in \{1, \ldots, n\}$, construct a kd-tree for P, and traverse the tree in an appropriate order. Since the procedures except for the kd-tree construction can be done in $O(n)$ time, the lower bound by Guibas, McCreight, Plass, and Roberts [19] proves the theorem. □

6. Concluding Remarks

For the sorting problem, several algorithms running in $O(n(1+\log(1+k/n)))$ time [1], [2], [22]–[25] have been presented, and there is a tight lower bound [19]. This lower bound also applies to the first formulation, and there is a gap between this lower bound and the running time of our algorithm. It is desirable to find an optimal algorithm.

For further investigation, we can think of other presortedness measures, and other geometric problems that can be thought of as permutation problems. A lot of questions remain unsolved, and we hope that this is a stimulating line of research.

Acknowledgments

The main part of this work was done while the second author visited Postech in May 2008 and January 2009. He thanks the kind support of Postech. Our thanks also go to Tetsuo Asano, Michael Hoffmann, Yun-Ho Hwang, Sang-Sub Kim, Iris Reinbacher, and Wan-Bin Son for fruitful discussion.

References

[1] K. Mehlhorn, Sorting and Searching, Data Structures and Algorithms, vol.1, Springer-Verlag, Berlin Heidelberg, 1984.

[2] H. Mannila, "Measures of presortedness and optimal sorting algorithms," IEEE Trans. Comput., vol.C-34, no.4, pp.318–325, 1985.

[3] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," ACM Comput. Surv., vol.24, pp.441–476, 1992.

[4] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold, "Space-efficient geometric divide-and-conquer algorithms," Computational Geometry: Theory and Applications, vol.37, pp.209–227, 2007.

[5] H. Brönnimann, T. Chan, and E. Chen, "Towards in-place geometric algorithms and data structures," Proc. 20th Annual Symposium on Computational Geometry (SoCG), pp.239–246, 2004.

[6] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint, "Space-efficient planar convex hull algorithms," Theoretical Computer Science, vol.321, pp.25–40, 2004.

[7] A. Pagh, R. Pagh, and M. Thorup, "On adaptive integer sorting," Proc. 12th Annual European Symposium on Algorithms (ESA), pp.556–579, 2004.

[8] G. Brodal, R. Fagerberg, and G. Moruz, "Cache-aware and cache-oblivious adaptive sorting," Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP), pp.576–588, 2005.

[9] E. Demaine, A. López-Ortiz, and J. Munro, "Adaptive set intersections, unions, and differences," Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp.743–752, 2000.

[10] J. Barbay and C. Kenyon, "Adaptive intersection and t-threshold problems," Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp.390–399, 2002.

[11] J. Barbay, A. Golynski, J. Munro, and S. Rao, "Adaptive searching in succinctly encoded binary relations and tree-structured documents," Theor. Comput. Sci., vol.387, pp.284–297, 2007.

[12] C. Levcopoulos, A. Lingas, and J. Mitchell, "Adaptive algorithms for constructing convex hulls and triangulations of polygonal chains," Proc. 8th Scandinavian Workshop on Algorithm Theory (SWAT), pp.80–89, 2002.

[13] I. Baran and E. Demaine, "Optimal adaptive algorithms for finding

the nearest and farthest point on a parametric black-box curve," International Journal of Computational Geometry and Applications, vol.15, pp.327–350, 2005.
[14] J. Barbay and E. Chen, "Convex hull of the union of convex objects in the plane: An adaptive analysis," Proc. 20th Canadian Conference on Computational Geometry (CCCG), pp.47–51, 2008.
[15] D. Kirkpatrick and R. Seidel, "The ultimate planar convex hull algorithm?," SIAM J. Comput., vol.15, pp.287–299, 1986.
[16] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan, "Time bounds for selection," J. Comput. Syst. Sci., vol.7, pp.448–461, 1972.
[17] N. Megiddo, "Linear programming in linear time when the dimension is fixed," J. ACM, vol.31, pp.114–127, 1984.
[18] V. Estivill-Castro and D. Wood, "Practical adaptive sorting," Proc. International Conference on Computing and Information (ICCI), pp.47–54, 1991.
[19] L. Guibas, E. McCreight, M. Plass, and J. Roberts, "A new representation of linear lists," Proc. 9th ACM Symposium on Theory of Computing (STOC), pp.49–60, 1977.
[20] S. Kapoor and P. Ramanan, "Lower bounds for maximal and convex layers problems," Algorithmica, vol.4, pp.447–459, 1989.
[21] J. Bentley, "Multidimensional binary search trees used for associative searching," Commun. ACM, vol.18, pp.509–517, 1975.
[22] A. Elmasry, "Priority queues, pairing, and adaptive sorting," Proc. 29th International Colloquium on Automata, Languages and Programming (ICALP), pp.183–194, 2002.
[23] A. Elmasry and M. Fredman, "Adaptive sorting: An information theoretic perspective," Acta Informatica, vol.45, pp.33–42, 2008.
[24] C. Levcopoulos and O. Petersson, "Splitsort—An adaptive sorting algorithm," Inf. Process. Lett., vol.39, pp.205–211, 1991.
[25] C. Levcopoulos and O. Petersson, "Adaptive heapsort," J. Algorithms, vol.14, pp.395–413, 1993.

Hee-Kap Ahn was born in 1973. He obtained his Ph.D. from Utrecht University in 2001, and was a scientific researcher at Korea Institute of Science and Technology from 2001 to 2004, a research professor at Korea Advanced Institute of Science and Technology from 2004 to 2005, an assistant professor at Sejong university from 2006 to 2007. Since 2007, he has been an assistant professor at Pohang University of Science and Technology. His research interests include geometric shape matching and shape approximation, robust algorithm design under uncertainty, and data structures.

Yoshio Okamoto was born in 1976. He obtained his Ph.D. from ETH Zurich in 2005, and was an assistant professor at Toyohashi University of Technology from 2005 to 2007. Since 2007, he has been an associate professor at Tokyo Insitute of Technology. His research focuses on discrete mathematics and theory of computing, especially problems on graphs and discrete geometry.