In-Place Algorithms for Computing (Layers of) Maxima

Henrik Blunck and Jan Vahrenhold

Westfälische Wilhelms-Universität Münster, Institut für Informatik,
48149 Münster, Germany. {blunck,jan}@math.uni-muenster.de

Abstract. We describe space-efficient algorithms for solving problems related to finding maxima among points in two and three dimensions. Our algorithms run in optimal $\mathcal{O}(n \log n)$ time and occupy only constant extra space in addition to the space needed for representing the input.

1 Introduction

Space-efficient solutions for fundamental algorithmic problems such as merging, sorting, and partitioning have been studied over a long period of time; see [11, 12, 14, 21]. The advent of small-scale, handheld computing devices and an increasing interest in utilizing fast but limited-size memory, e.g., caches, recently led to a renaissance of space-efficient computing with a focus on processing geometric data. Brönnimann *et al.* [7] were the first to consider space-efficient geometric algorithms and showed how to optimally compute $2d$-convex hulls using constant extra space. Subsequently, a number of space-efficient geometric algorithms, e.g., for computing $3d$-convex hulls and its relatives, as well as for solving intersection and proximity problems, have been presented [4–6, 9, 23].

In this paper, we consider the fundamental geometric problems of computing the maxima of point sets in two and three dimensions and of computing the layers of maxima in two dimensions. Given two points p and q, the point p is said to *dominate* the point q iff the coordinates of p are larger than the coordinates of q in all dimensions. A point p is said to be a *maximal point* (or: a *maximum*) of $\mathcal{P}$ iff it is not dominated by any other point in $\mathcal{P}$. The union $\texttt{MAX}(\mathcal{P})$ of all points in $\mathcal{P}$ that are maximal is called the *set of maxima* of $\mathcal{P}$. This notion can be extended in a natural way to compute *layers* of maxima [8]. After $\texttt{MAX}(\mathcal{P})$ has been identified, the computation is repeated for $\mathcal{P} := \mathcal{P} \setminus \texttt{MAX}(\mathcal{P})$, i.e., the next layer of maxima is computed. This process is iterated until $\mathcal{P}$ becomes empty.

Related Work The problem of finding maxima of a set of n points has a variety of applications in statistics, economics, and operations research (as noted by Preparata and Shamos [20]), and thus was among the first problems having been studied in Computational Geometry: In two and three dimensions, the best known algorithm which has been developed by Kung, Luccio, and Preparata [16] identifies the set of maxima in $\mathcal{O}(n \log n)$ time which is optimal since the problem exhibits a sorting lower bound [16, 20]. For constant dimensionality $d \geq 4$, their

divide-and-conquer approach yields an algorithm with $\mathcal{O}(n \log^{d-2} n)$ running time [1, 16], and Matoušek [17] gave an $\mathcal{O}(n^{2.688})$ algorithm for the case $d = n$. The problem has also been studied for dynamically changing point sets in two dimensions [13] and under assumptions about the distribution of the input points in higher dimensions [2, 10]. Buchsbaum and Goodrich [8] presented an $\mathcal{O}(n \log n)$ algorithm for computing the layers of maxima for point sets in three dimensions. Their approach is based on the plane-sweeping paradigm and relies on dynamic fractional cascading to maintain a point-location structure for dynamically changing two-dimensional layers of maxima.

The maxima problem has been actively investigated in the database community following Börzsönyi, Kossmann, and Stocker's [3] definition of the SQL "`skyline`" operator. Such an operator producing the set of maxima[1] is needed in queries that, e.g., ask for hotels that are both close to the beach and have low room rates. A number of results have been presented that use spatial indexes to produce the "skyline", e.g., the set of maxima, practically efficient and/or in a progressive way, that is outputting results while the algorithm is running [15, 19, 22]. For none of these approaches non-trivial upper bounds are known.

The Model The goal of investigating space-efficient algorithms is to design algorithms that use very little extra space in addition to the space used for representing the input. The input is assumed to be stored in an array `A` of size n, thereby allowing random access. We assume that a constant-sized memory can hold a constant number of words. Each word can hold one pointer, or an $\mathcal{O}(\log n)$ bit integer, and a constant number of words can hold one element of the input array. The extra memory used by an algorithm is measured in terms of the number of extra words; an *in-place* algorithm uses $\mathcal{O}(1)$ extra words of memory. It has been shown that some fundamental geometric problems such as computing 2D convex hulls and closest pairs can be solved *in-place* and in optimal time [4, 5, 7]. More involved problems (range searching, line-segment intersection) can be (currently) solved *in-place* only if one is willing to accept near-optimal running time [6, 23], and 3D convex hulls and related problems seem to require both (poly-)logarithmic extra space and time [6].

Our Contribution The main issue in designing in-place algorithms is that most powerful algorithmic tools (unbalanced recursion, sweeping, multi-level data structures, fractional cascading) require at least logarithmic extra space, e.g., for the recursion stack or pointer-based structures. This raises the question of whether there exists a time-space tradeoff for geometric algorithms besides range-searching. In this paper, we demonstrate that $\mathcal{O}(1)$ extra space is sufficient to obtain optimal $\mathcal{O}(n \log n)$ algorithms for computing skylines in two and three dimensional and two-dimensional layers of maxima. The solution to the latter problem is of particular interest since it is the first optimal in-place algorithm for a geometric problem that is not amenable to a solution based on balanced divide-and-conquer or Graham's scan.

---

[1] Technically speaking, the operator returns the set of *minima*. To unify the presentation, we do not distinguish between these two variants of the same problem.

2 Computing the Skyline in $\mathbb{R}^2$ and $\mathbb{R}^3$

A point p from a point set $\mathcal{P}$ is said to be *maximal* if no other point from $\mathcal{P}$ has larger coordinates in *all* dimensions; ties are broken using a standard shearing technique. This definition has been transferred by Kung *et al.* [16] into a two-dimensional plane-sweeping algorithm and into a divide-and-conquer approach for the higher-dimensional case. The output of our algorithm consists of a permutation of the input array $\mathtt{A}$ and an index k such that k points constituting the set of maxima are stored sorted by decreasing y-coordinates in $\mathtt{A}[0,\ldots,k-1]$.

The algorithm for computing the skyline, i.e., the set of maxima, in two dimensions is a straightforward selection algorithm. We discuss it in some more detail to introduce an important algorithmic template, called SortedSubsetSelection($\mathtt{A}, \ell_b, \ell_e, \pi$) which processes a sorted (sub)array $\mathtt{A}[\ell_b,\ldots,\ell_e-1]$ from left to right. While doing so, the algorithm evaluates a given predicate π for each of the elements and stably moves all elements for which π evaluates to `true` to the front of $\mathtt{A}[\ell_b,\ldots,\ell_e-1]$. This algorithm, presented by Bose *et al.* [4], runs in linear time provided that π can be evaluated in constant time.

To compute the set of maxima in two dimensions, the algorithm of Kung *et al.* [16] sweeps the point set in *decreasing* y-direction keeping track of the bottommost point m of the skyline seen so far. For each point p encountered during the sweep, the algorithm checks whether p is dominated by m. The sweeping direction ensures $p.y \leq m.y$, thus, it is sufficient to check whether also $p.x < m.x$.

The space-efficient implementation of this algorithm thus first presorts $\mathtt{A}$ using an optimal $\mathcal{O}(n \log n)$ in-place sorting algorithm, e.g., *heapsort* [11]. The algorithm then runs an instantiation of the linear-time SortedSubsetSelection template where the predicate π evaluates to `true` iff the x-coordinate of the current point $\mathtt{A}[i]$ is at least as large as the x-coordinate of the point m, i.e, iff $\mathtt{A}[i]$ is a maximal point. The algorithm then moves $\mathtt{A}[i]$ to the front of the array and updates m to refer to (the new position of) $\mathtt{A}[i]$.

Lemma 1. *The* skyline*, i.e., the set of maxima of a set $\mathcal{P}$ of n points in two dimensions can be computed in-place and in optimal time $\mathcal{O}(n \log n)$. If $\mathcal{P}$ is sorted according to $<_y$, the running time is in $\mathcal{O}(n)$.*

For the case of a three-dimensional input, we implement Kung *et al.*'s [16] divide-and-conquer algorithm using an in-place divide-and-conquer scheme we have proposed earlier [4]; this scheme is based on in-place routines for median-finding, partitioning, and merging. Since we cannot explicitly keep track of the number of maxima in each subproblem, we have to recover them algorithmically during each merging step. The details are given in the full version of this paper.

Theorem 1. *The* skyline*, i.e., the set of maxima of an n-element point set in three dimensions can be computed in-place and in optimal time $\mathcal{O}(n \log n)$.*

3 Computing the Layers of Maxima in Two Dimensions

An obvious way of computing the layers of maxima is to iteratively compute (and remove) the maximal points of the given point set $\mathcal{P}$ using, e.g., the in-

place algorithm described in Section 2. Since a point set may exhibit a linear number of layers, this leads to an $\mathcal{O}(n^2 \log n)$ worst-case running time. In this section, we show that we can simultaneously peel off multiple layers such that the resulting algorithm runs in optimal $\mathcal{O}(n \log n)$ time; its goal is to rearrange the input such that the points are grouped by layers and each layer is sorted by decreasing y-coordinate.

3.1 Computing the Number of Layers of Maxima

As an introductory example for our approach, we present an algorithm for computing the number of layers of maxima for a given point set (the algorithm can be easily modified to compute the skyline in two dimensions). This algorithm builds upon the fact that a layer of points is monotone in both x- and y-direction: a layer $\mathcal{L}_i$ extends vertically to $y = -\infty$ from the point on $\mathcal{L}_i$ that has maximal x-coordinate. This in turn implies that, during the sweep, the x-coordinate of the intersection of $\mathcal{L}_i$ with the sweepline is the x-coordinate of its "tail" point τ_i, i.e., of the last point that has been classified as belonging to $\mathcal{L}_i$ (see Figure 1).

As usual, we assume that the point set $\mathcal{P}$ to be processed is stored in an array $\mathtt{A}[0, \dots, n-1]$. During the sweep, we ensure that the following invariant holds after having processed a point p:

Invariant (TAILS): Let $k \in \{1, \dots, n\}$ be the number of layers intersected by the sweepline at $y = p.y$ where p is the point that has just been processed. Then the tail points $\tau_0, \dots, \tau_{k-1}$ of the layers $\mathcal{L}_0, \dots, \mathcal{L}_{k-1}$ are stored in decreasing x-order in $\mathtt{A}[0, \dots, k-1]$.

Invariant (TAILS) is certainly true after having processed the first point $p = \mathtt{A}[0]$ encountered during the sweep. This point is the y-maximal point of the point set and thus part of the skyline, i.e., of the topmost layer $\mathcal{L}_0$. We thus inductively assume that the invariant holds prior to processing the next point $p := \mathtt{A}[i]$. To determine which layer p is part of, we perform a binary search for p w.r.t. the x-coordinate among the points in $\mathtt{A}[0, \dots, k-1]$. If p has a smaller x-coordinate than $\tau_{k-1} = \mathtt{A}[k-1]$, p is dominated by this point, and thus p is the first point of a new layer $\mathcal{L}_k$ (see p_{i_1} in Figure 1). We then swap $p = \mathtt{A}[i]$ to $\mathtt{A}[k]$ (note that $i \geq k$ trivially holds) and increment k by one. If, on the other hand, p lies right of τ_j but left of τ_{j-1} then p replaces τ_j as the tail point of $\mathcal{L}_j$ (see p_{i_2} in Figure 1), that is, we swap $p = \mathtt{A}[i]$ to $\mathtt{A}[j]$; similarly, p replaces τ_0 if it lies right of τ_0.

The above in-place algorithm maintains Invariant (TAILS) in $\mathcal{O}(\log n)$ time per point processed. Thus after having processed the last point, the index k gives the total number of layers.

Lemma 2. *The number of layers of maxima exhibited by an n-element point set in two dimensions can be compute in-place and in $\mathcal{O}(n \log n)$ time.*

The above algorithm can also be modified to *output*, i.e., to print to a write-only stream, in $\mathcal{O}(n \log n)$ time each point processed together with the number of its containing layer.

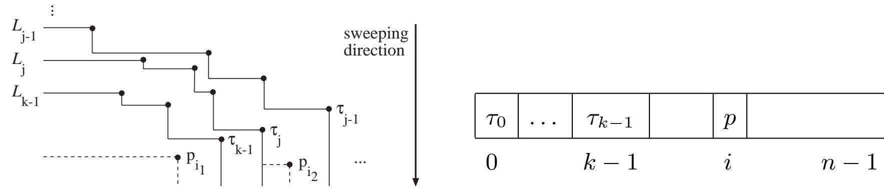


Fig. 1. Classification of a point by binary search (left) and array representation (right).

3.2 Counting the Number of Points on the Topmost κ Layers

The algorithm of Section 3.1 can be modified to *count* the number of points on each of the κ topmost layers. For the simplicity of exposition, we assume that we have access to $\mathcal{O}(\kappa)$ extra space that holds a counter c_i for each layer $\mathcal{L}_i$. In Section 3.4, we will get rid of this assumption, which—in an in-place setting—is prohibitive for non-constant κ.

To compute the number of points on each of the topmost κ layers, we simply increment the counter c_i for layer $\mathcal{L}_i$ whenever we update the value of τ_i. We also stop updating the counter k denoting the number of layers being kept track of at $k = \kappa$. Afterwards, we can determine for each point in $\mathcal{O}(1)$ time whether it lies left of $\tau_{\kappa-1}$ (and thus below $\mathcal{L}_{\kappa-1}$). If so, we simply ignore it, and for all other points, we perform a binary search in $\mathtt{A}[0,\ldots,\kappa-1]$ as described above.

Lemma 3. *The cardinality c_i of each of the topmost κ layers of $\mathtt{A}[0,\ldots,n-1]$ can be computed in $\mathcal{O}(n \log n)$ time. If the points are presorted, the complexity is in $\mathcal{O}(n + \xi \log \kappa)$ where $\xi = \sum_{i=0}^{\kappa-1} c_i$.*

3.3 Extracting the Topmost κ Layers in Sorted Order

As mentioned above, a naive iterative approach to computing all layers of maxima leads to an $\mathcal{O}(n^2 \log n)$ worst-case running time for points sets with a linear number of layers. The algorithm we describe in this section processes several layers at a time to reduce the number of iterations.

Extracting the points on the topmost κ layers Our algorithm imitates counting-sort, i.e., prior to actually partitioning the points into layers, it first computes the number of points for each of the layers. To illustrate the algorithm, let us assume that we have already peeled off some layers and stored the result in $\mathtt{A}[0,\ldots,\ell_b-1]$. Inductively, we maintain the following invariant which, prior to the first iteration, can be established by sorting $\mathtt{A}$ and setting $\ell_b := 0$:

Invariant (SORT): The points in $\mathtt{A}[0,\ldots,n-1]$ that have not yet been assigned to a layer are stored in $\mathtt{A}[\ell_b,\ldots,n-1]$ and are sorted by decreasing y-coordinate.

Let us further assume that the total number of points on the topmost κ layers $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$ of the *remaining* points stored in $\mathtt{A}[\ell_b, \ldots, n-1]$ is ξ and that $\ell_b + 2\xi \leq n$. The first step of the algorithm is to stably extract the ξ points on the topmost κ layers and move them to $\mathtt{A}[\ell_b, \ldots, \ell_b + \xi - 1]$ while maintaining the sorted y-order in $\mathtt{A}[\ell_b + \xi, \ldots, n-1]$:

...	Points on $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$	Points below $\mathcal{L}_{\kappa-1}$
	ℓ_b	$\ell_b + \xi$ $\quad$ $n-1$

This partition is obtained as follows: We run a variant of the algorithm described in Section 3.2 that is combined with SortedSubsetSelection. This algorithm maintains the invariant that the tail points τ_0 through $\tau_{\kappa-1}$ are stored in $\mathtt{A}[\ell_b, \ldots, \ell_b + \kappa - 1]$ and stably moves all points that are *below* $\mathcal{L}_{\kappa-1}$ to the subarray starting at $\mathtt{A}[\ell_b + \kappa]$; thus, it keeps all points below $\mathcal{L}_{\kappa-1}$ in sorted order. At the same time, we also maintain a counter c_i for the size of each layer $\mathcal{L}_i$—as mentioned in Section 3.2, we will later discuss how to do this in-place. The correctness of this algorithm follows from the observation that none of the first κ points in $\mathtt{A}[\ell_b, \ldots, \ell_b + \kappa - 1]$ can lie below $\mathcal{L}_{\kappa-1}$ (there have to be at least κ points on κ layers).

The algorithm maintains the invariant that, when processing point $\mathtt{A}[j]$, all points that already have been identified as "below $\mathcal{L}_{\kappa-1}$" are stored in decreasing y-order in $\mathtt{A}[\ell_b + \kappa, \ldots, i-1]$.

...	$\tau_0 \ldots \tau_{\kappa-1}$	Points below $\mathcal{L}_{\kappa-1}$	Points on $\mathcal{L}_0 \ldots \mathcal{L}_{\kappa-1}$		
	ℓ_b	$\ell_b + \kappa$	i	j	$n-1$

The point $\mathtt{A}[j]$ now either is classified as "below $\mathcal{L}_{\kappa-1}$" or replaces the tail τ_h of some layer $\mathcal{L}_h$. In the first situation, $\mathtt{A}[j]$ is stably moved directly behind $\mathtt{A}[\ell_b + \kappa, \ldots, i-1]$, i.e., it is swapped with $\mathtt{A}[i]$ and i is then incremented by one, and in the second situation, $\mathtt{A}[j]$ is swapped with τ_h, i.e., with $\mathtt{A}[\ell_b + h]$. When we have reached the end of the array, we inductively see that $\mathtt{A}[\ell_b + \kappa, \ldots, i-1]$ contains the points below $\mathcal{L}_{\kappa-1}$ in sorted order. Furthermore, by the definition of ξ, we know that the two subarrays $\mathtt{A}[\ell_b, \ldots, \ell_b + \kappa - 1]$ (containing the tails) and $\mathtt{A}[i, \ldots, n-1]$ (containing the remaining points on the layers $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$) together consist of exactly ξ points. We then swap (in linear time) $\mathtt{A}[\ell_b + \kappa, \ldots, i-1]$ (containing the elements below $\mathcal{L}_{\kappa-1}$) and $\mathtt{A}[i, \ldots, n-1]$ such that the ξ elements on the layers $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$ (tails and non-tails) are blocked in $\mathtt{A}[\ell_b, \ldots, \ell_b + \xi - 1]$. To re-establish the y-order of these ξ points, we sort them in $\mathcal{O}(\xi \log \xi) \subset \mathcal{O}(\xi \log n)$ time, that is, we establish Invariant (SORT) for $\mathtt{A}[\ell_b, \ldots, \xi - 1]$. Thus, the overall running time for counting the number of points on the topmost κ layers and for re-establishing Invariant (SORT) is in $\mathcal{O}(n + \xi \log n)$ where $\xi = \sum_{i=0}^{\kappa-1} c_i$.

Sorting the points by layer Using the counters c_i computed during the previous step, we now run a variant of counting sort to extract the layers $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$

in sorted y-order. To do this in-place, we use the subarray $\mathtt{A}[\ell_b+\xi,\dots,\ell_b+2\xi-1]$ as scratch space that will hold the layers to be constructed (note that we assume $\ell_b+2\xi \le n$ and that $\xi = \sum_{i=0}^{\kappa-1} c_i$ holds by definition). We traverse the subarray $\mathtt{A}[\ell_b,\dots,\ell_b+\xi-1]$ maintaining the tails of all layers in $\mathtt{A}[\ell_b,\dots,\ell_b+\kappa-1]$ as before, but whenever we update the tail τ_i of a layer $\mathcal{L}_i$, we swap the old tail to the next available position in the subarray of $\mathtt{A}[\ell_b+\xi,\dots,\ell_b+2\xi-1]$ that is reserved to hold $\mathcal{L}_i$.

...	τ_0	...	$\tau_{\kappa-1}$		$\mathcal{L}_0$	...	$\mathcal{L}_{\kappa-1}$	...
	ℓ_b				$\ell_b+\xi$			$\ell_b+2\xi$

After having constructed a sorted representation of the layers $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$, the two subarrays $\mathtt{A}[\ell_b,\dots,\ell_b+\xi-1]$ and $\mathtt{A}[\ell_b+\xi,\dots,\ell_b+2\xi-1]$ are swapped in-place:

...	$\mathcal{L}_0$	...	$\mathcal{L}_{\kappa-1}$	Points below $\mathcal{L}_{\kappa-1}$	...
	ℓ_b			$\ell_b+\xi$	$\ell_b+2\xi$

To re-establish Invariant (SORT), we finally sort $\mathtt{A}[\ell_b+\xi,\dots,\ell_b+2\xi-1]$ (note that the points $\mathtt{A}[\ell_b+2\xi,\dots,n-1]$ have not been touched and thus still are sorted) and update $\ell_b := \ell_b+\xi$. The running time for sorted extraction of the ξ points on the topmost κ layers and for re-establishing Invariant (SORT) for $\mathtt{A}[\ell_b+\xi,\dots,n-1]$ is in $\mathcal{O}(n+\xi\log n)$.

3.4 Extracting all Layers in Sorted Order

The exposition of the algorithm presented in the previous section was build on two major assumptions: (1) the algorithm had to have access to κ counters and (2) the subarray $\mathtt{A}[\ell_b,\dots,n-1]$ had to be large enough to accommodate two subarrays of size ξ. In this section, we demonstrate how to maintain both assumptions in an in-place setting.

The first issue to be resolved is how to maintain a non-constant number κ of counters without using $\Theta(\kappa)$ extra space. Each such counter c_i is required to represent values up to n, and thus has to consist of $\log_2 n$ bits. We resort to a standard technique in the design of space-efficient algorithms, namely to encode a single bit by a permutation of two distinct (but comparable) elements q and r: assuming $q<r$, the permutation rq encodes a binary zero, and the permutation qr encodes a binary one [18]. As the elements in our case are two-dimensional points, we use the (lexicographical) y-order for deciding whether two points encode a binary zero or a binary one.[2]

[2] We point out that a *set* of elements cannot contain duplicates; hence the relative order of two points is unique. Furthermore, the set of maxima of a *multiset* M consists of the same points as the set of maxima of the *set* that is obtained by removing the duplicates from M. Duplicate removal can be done in-place and in $\mathcal{O}(n\log n)$ time by first sorting M according to $<_y$ and then stably selecting exactly one occurrence of each point.

If we reserve a block of $\frac{1}{3}n$ elements, we can encode $\frac{1}{6}n$ bits, i.e., $\frac{1}{6}n/\log_2 n$ counters that may be used to represent values less than n, and this in turn implies that the maximum number of layers for which we can run the algorithms described in Sections 3.2 and 3.3 is bounded by $\kappa = \frac{1}{6}n/\log_2 n$. The analyses at the end of the respective sections gave an $\mathcal{O}(n + \xi \log n)$ bound for each run of the algorithms, and thus we have to make sure that maintaining the counters and the second invariant does not interfere with keeping the overall number of iterations in $\mathcal{O}(\log n)$.

The case $\boldsymbol{\ell_b < \frac{1}{3}n}$ If, prior to the current iteration, $\ell_b < \frac{1}{3}n$ holds, we maintain the counters in $\mathtt{A}[\frac{2}{3}n, \ldots, n-1]$.

...		Counter representation
	ℓ_b	$\frac{2}{3}n$ $\qquad$ $n-1$

Counting the points on the topmost κ layers By Invariant (SORT), $\mathtt{A}[\ell_b, \ldots, n-1]$ is sorted by decreasing y-coordinate, so all counters encode the value zero. We set $\kappa := \frac{1}{6}n/\log_2 n$ and run the algorithm for counting the elements on each of the topmost κ layers. We maintain each of the counters c_i in its fixed-size representation by exchanging adjacent elements as needed to implement changing a binary digit, and using the standard analysis for incrementing a binary counter, we observe that all counters can be maintained in $\mathcal{O}(\xi)$ time where $\xi := \sum_{i=0}^{\kappa-1} c_i$. Note that, since the algorithm processes *all* points in $\mathtt{A}[\ell_b, \ldots, n-1]$, any point q in $\mathtt{A}[\frac{2}{3}n, \ldots, n-1]$ may be swapped to the front of the array since it may become the tail τ_i of some layer $\mathcal{L}_i$. Using a more careful implementation of the approach given in Section 3.2, we can compute all counters and re-establish Invariant (SORT) in $\mathcal{O}(n + \xi \log n)$ time. After we have computed the values of all counters c_i—but prior to re-establishing Invariant (SORT)—we compute the prefix sums of c_0 through $c_{\kappa-1}$, i.e., we replace c_j by $\hat{c}_j := \sum_{i=0}^{j} c_j$. This can be done in-place spending $\mathcal{O}(\log n)$ time per counter, and thus in $\mathcal{O}(n)$ overall time. While doing so, we maintain the maximal index κ' such that $\ell_b + 2\hat{c}_{\kappa'} < \frac{2}{3}n$.

Extracting and sorting the points on the topmost κ layers If the index κ' described above exists, we run (a slightly modified implementation of) the algorithm for extracting the $\xi' := \hat{c}_{\kappa'}$ points on the κ' topmost layers as described in Section 3.3. Because of the way κ' was chosen, we can guarantee that the scratch space of size $\hat{c}_{\kappa'}$ needed for the counting-sort-like partitioning does not interfere with the space $\mathtt{A}[\frac{2}{3}n, \ldots, n-1]$ reserved for representing the counters. Also, by Invariant (SORT), $\mathtt{A}[\frac{2}{3}n, \ldots, n-1]$ is guaranteed to be sorted by decreasing y-coordinate. Both conditions imply that, once we have extracted all ξ' points, we can "reset" the counters in $\mathcal{O}(n)$ time (thus re-establishing Invariant (SORT) for $\mathtt{A}[\frac{2}{3}n, \ldots, n-1]$) by scanning through $\mathtt{A}[\frac{2}{3}n, \ldots, n-1]$ and swapping only adjacent points.

The counters used during the following counting-sort-like partitioning are initialized with $c_0 := \ell_b + \xi'$ and $c_j := \sum_{i=0}^{j-1} c_i$ for $j > 0$. Whenever a tail τ_i is

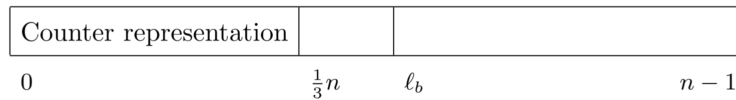
updated, the old point p representing τ_i is swapped to $\mathtt{A}[\ell_b + \xi' + c_i]$ and then c_i is incremented by one. Decoding and incrementing c_i can be done in $\mathcal{O}(\log n)$ time, and this cost is charged to the point p moved to its containing layer. The total cost for extracting ξ' points is in $\mathcal{O}(n + \xi' \log n)$; this also includes the cost for sorting the scratch space $\mathtt{A}[\ell_b + \xi', \ldots, \ell_b + 2\xi' - 1]$ and re-establishing Invariant (SORT) by merging $\mathtt{A}[\ell_b+\xi', \ldots, \ell_b+2\xi'-1]$ with the (sorted) subarray $\mathtt{A}[\ell_b + 2\xi', \ldots, n-1]$ (see Section 3.3).

If $\kappa' < \kappa$, i.e., if we extract some but not all $\kappa = \frac{1}{6}n/\log_2 n$ layers, we additionally run the $\mathcal{O}(n \log n)$ skyline computation algorithm described in Section 2 to extract the points on the next topmost layer, regardless of its size. Similarly, if the index κ' does not exist at all, we extract the topmost layer $\mathcal{L}_0$ using the $\mathcal{O}(n \log n)$ skyline computation algorithm on $\mathtt{A}[\ell_b, \ldots, n-1]$—note that in this case the topmost layer $\mathcal{L}_0$ contains $c_0 > \frac{1}{2}\left(\frac{2}{3}n - \ell_b\right) > \frac{1}{2}\left(\frac{2}{3}n - \frac{1}{3}n\right) \in \Theta(n)$ points. In any case, we spend another $\mathcal{O}(n \log n)$ time to re-establish Invariant (SORT) by sorting.

Analysis Our analysis classifies each iteration according to whether or not all ξ points on the topmost $\kappa = \frac{1}{6}n/\log_2 n$ layers are moved to their final position in the array. If all ξ points are moved, we know that $\xi \geq \frac{1}{6}n/\log_2 n$, and thus only a logarithmic number of such iterations can exist. Also, we can distribute the $\mathcal{O}(n + \xi \log n)$ time spent per iteration such that each iteration gets charged $\mathcal{O}(n)$ time and that each of the ξ points moved to its final position gets charged $\mathcal{O}(\log n)$ time, so the overall cost for all such iterations is in $\mathcal{O}(n \log n)$.

If less than κ layers can be processed in the iteration in question (this also includes the case that κ' does not exist), the $\mathcal{O}(n + \xi \log n)$ cost for counting the ξ points on the topmost κ layers and the $\mathcal{O}(n + \xi' \log n)$ cost for extracting ξ' points on the topmost κ' layers is dominated by the $\mathcal{O}(n \log n)$ cost for the successive skyline computation. The definition of κ' guarantees that, after we have performed the skyline computation, we have advanced the index ℓ_b by at least $\frac{1}{2}\left(\frac{2}{3}n - \ell_b\right)$ steps. Combining this with the fact that $\ell_b < \frac{1}{3}n$, we see that there may exist only a constant number of such iterations, and hence their overall cost is in $\mathcal{O}(n \log n)$.

The case $\boldsymbol{\ell_b \geq \frac{1}{3}n}$ If, prior to the current iteration, $\ell_b \geq \frac{1}{3}n$ holds, we maintain the counters in $\mathtt{A}[0, \ldots, \frac{1}{3}n - 1]$:



Note that this subarray contains (part of) the layers that have been computed already. Since maintaining a counter involves swapping some of the elements in $\mathtt{A}[0, \ldots, \frac{1}{3}n - 1]$, this disturbs the y-order of (some of) the layers already computed, and we have to make sure that we can reconstruct the layer order. We will discuss this at the end of this section.

Counting the points on the topmost κ layers The algorithm for counting the points on the topmost κ layer proceeds exactly as described above, i.e., starting with $\kappa := \frac{1}{6}n/\log_2 n$ and updating the κ counters (which now are represented in $\mathtt{A}[0,\ldots,\frac{1}{3}n-1]$). The only difference is that when computing the prefix sums, the algorithm finds the maximal index κ' such that $\ell_b + 2\hat{c}_{\kappa'} < n$, i.e., the algorithm does not need to avoid the space reserved for the counters. This implies that we can run (a simplified version of) the algorithm we used for the case $\ell_b < \frac{1}{3}n$. Thus we spend $\mathcal{O}(n + \xi \log n)$ time per iteration including the cost for re-establishing Invariant (SORT).

Extracting and sorting the points on the topmost κ layers As for the case $\ell_b < \frac{1}{3}n$ we either extract all ξ points on the topmost κ layers in $\mathcal{O}(n + \xi \log n)$ time or extract less than κ layers followed by a skyline computation in $\mathcal{O}(\nu \log \nu)$ time where $\nu := n - \ell_b$. In both cases, the complexity given also includes the cost for re-establishing Invariant (SORT).

Analysis To estimate the overall running time for the case $\ell_b > \frac{1}{3}n$, we again classify the iterations according to whether or not all ξ points on the topmost $\kappa := \frac{1}{6}n/\log_2 n$ layers can be moved to their final destination. If this is the case, we know that we have moved $\xi \geq \kappa = \frac{1}{6}n/\log_2 n$ points and can charge $\mathcal{O}(\log n)$ time to each point moved and the remaining $\mathcal{O}(n)$ time to the iteration. Moving $\xi \geq \frac{1}{6}n/\log_2 n$ points also implies that the total number of such iterations is bounded by $\mathcal{O}(\log n)$, and hence the *global* cost incurred by assigning $\mathcal{O}(n)$ cost to such an iteration is in $\mathcal{O}(n \log n)$.

If we move $\kappa' < \kappa$ layers, the next step of the algorithm is a skyline computation (or the algorithm terminates), and we analyze these steps together. After we have performed these steps, we know (by the definition of κ') that we have advanced ℓ_b by at least $\frac{1}{2}(n - \ell_b)$. Thus, the next time, this situation occurs, it will occur for a subarray of at most half the size. This geometrically decreasing series implies that the cost for all iterations in which $\kappa' < \kappa$ layers are moved is dominated by the cost of the first such iteration (if any), i.e., it is in $\mathcal{O}(n \log n)$.

Restoring the layers stored in $\mathtt{A}[0,\ldots,\frac{1}{3}n-1]$ After the last iteration of the algorithm, we need to restore the y-order of the layers stored in $\mathtt{A}[0,\ldots,\frac{1}{3}n-1]$. The main problem when doing this is that we have no memory of the size of each layer, and thus we cannot simply sort the points. However, we know that each point (having been used for bit encoding) can only be one position off its correct location. Our algorithm for reconstructing the layers exploits this and iterates over pairs of "bit-neighbors" in $\mathtt{A}[0,\ldots,\frac{1}{3}n-1]$ while maintaining the invariant that, when processing $q := \mathtt{A}[2i]$ and $r := \mathtt{A}[2i+1]$, all points in $\mathtt{A}[0,\ldots,2i-1]$ have be restored to their correct order.

If one of the two points q and r dominates the other point, the two points cannot be part of the same layer. Thus the point with larger y-coordinate is the last point of the current layer, and the other point is the first point of the next layer—see the left part of Figure 2. The situation that no point dominates the other is further detailed in the right part of Figure 2; in order to bring q and r

in their correct order, we need to access the point $p := \mathtt{A}[2i-1]$. If $i = 0$, p does not exist, but then q and r are the first two points and thus their y-order gives their correct and final position. If p exists, it is (by the invariant) the right- and bottommost point of its containing layer in $\mathtt{A}[0, \ldots, 2i-1]$, and thus *no* point *left* of p can belong to the same layer. A careful analysis (omitted due to space constraints) shows that *each* point of r and q that is right of p belongs to the same layer as p. Also, if both r and q are left of p, we can show that (due to their relative y-order) they have to be part of the same layer. We conclude that either one point dominates the other or that a simple, constant-time test, namely comparing the relative x-order of p, q, and r is sufficient to reconstruct the correct layer order of q and r. Consequently, the algorithm for reconstructing the layer order runs in linear time.

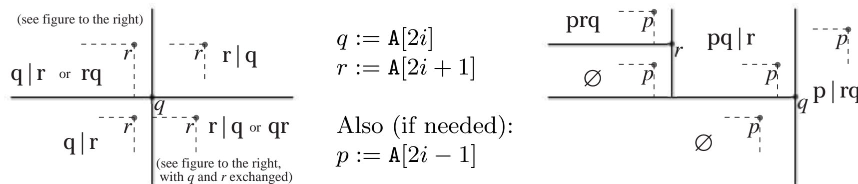


Fig. 2. Restoring the layer order for q and r. A "|" represents a break between layers.

Conclusions Summing up, the cost for all iterations in which $\ell_b < \frac{1}{3}n$ and for all iterations in which $\ell_b \geq \frac{1}{3}n$ is $\mathcal{O}(n \log n)$. Combining this with the fact that each point gets charged $\mathcal{O}(\log n)$ cost for the iteration in which it is moved to its final location, we obtain our main result:

Theorem 2. *All layers of maxima of an n-element point set in two dimensions can be computed in-place and in optimal time $\mathcal{O}(n \log n)$ such that the points in each layer are sorted by decreasing y-coordinate.*

References

1. J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.
2. J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. *Algorithmica*, 9(2):168–183, 1993.
3. S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430. 2001.
4. P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry: Theory & Applications*, 2006. To appear, accepted Nov. 2004.

5. H. Brönnimann and T. M.-Y. Chan. Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. *Computational Geometry: Theory & Applications*, 34(2):75–82, 2006.
6. H. Brönnimann, T. M.-Y. Chan, and E. Y. Chen. Towards in-place geometric algorithms. In *Proceedings of the 20th Annual Symposium on Computational Geometry*, pages 239–246. 2004.
7. H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. Space-efficient planar convex hull algorithms. *Theoretical Computer Science*, 321(1):25–40, 2004.
8. A. L. Buchsbaum and M. T. Goodrich. Three-dimensional layers of maxima. *Algorithmica*, 39(4):275–286, 2004.
9. E. Y. Chen and T. M.-Y. Chan. A space-efficient algorithm for line segment intersection. In *Proceedings of the 15th Canadian Conference on Computational Geometry*, pages 68–71, 2003.
10. H. K. Dai and X. W. Zhang. Improved linear expected-time algorithms for computing maxima. In *Proceedings of the Latin American Theoretical Informatics Symposium*, Lecture Notes in Computer Science 2976, pages 181–192. Springer, 2004.
11. R. W Floyd. Algorithm 245: Treesort. *Communications of the ACM*, 7(12):701, 1964.
12. V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, 237(1–2):159–181, 2000.
13. S. Kapoor. Dynamic maintenance of maxima of 2-D point sets. *SIAM Journal on Computing*, 29(6):1858–1877, 2000.
14. J. Katajainen and T. Pasanen. Stable minimum space partitioning in linear time. *BIT*, 32:580–585, 1992.
15. D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 275–286. 2002.
16. H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, 1975.
17. J. Matoušek. Computing dominances in E^n. *Information Processing Letters*, 38(5):277–278, 1991.
18. J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33(1):66–74, 1986.
19. D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems*, 30(1):41–82, 2005.
20. F. P. Preparata and M. I. Shamos. *Computational Geometry. An Introduction*. Springer, 1988.
21. J. S. Salowe and W. L. Steiger. Stable unmerging in linear time and constant space. *Information Processing Letters*, 25(3):285–294, 1987.
22. K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 301–310. 2001.
23. J. Vahrenhold. Line-segment intersection made in-place. In *Proceedings of the 9th International Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science 3608, pages 146–157. Springer, 2005.