In-Place Algorithms for Computing (Layers of) Maxima

Henrik Blunck · Jan Vahrenhold

Received: 8 February 2007 / Accepted: 11 April 2008 / Published online: 2 May 2008
© Springer Science+Business Media, LLC 2008

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.

Keywords In-place algorithms · Pareto-optimal points · Computational geometry

1 Introduction

Space-efficient solutions for fundamental algorithmic problems such as merging, sorting, or partitioning have been studied over a long period of time; see [12, 13, 15, 18, 25, 28]. 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. [6] 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

An extended abstract of this work appeared in *Proceedings of the Tenth Scandinavian Workshop on Algorithm Theory* (2006). Part of this work was done while both authors were at the University of Münster, Germany.

H. Blunck

MADALGO (Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation), Department of Computer Science, University of Århus, 8200 Aarhus, Denmark
e-mail: blunck@madalgo.au.dk

J. Vahrenhold (✉)

Technische Universität Dortmund, Informatik XI, 44221 Dortmund, Germany
e-mail: jan.vahrenhold@cs.uni-dortmund.de

3d-convex hulls and its relatives, as well as for solving intersection and proximity problems, have been presented [1, 2, 8, 9, 27].

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 $\text{MAX}(\mathcal{P})$ 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 [5]. After $\text{MAX}(\mathcal{P})$ has been identified, the computation is repeated for $\mathcal{P} := \mathcal{P} \setminus \text{MAX}(\mathcal{P})$, 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 [23]), and thus was among the first problems studied in Computational Geometry: In two and three dimensions, the best known algorithm which has been developed by Kung, Luccio, and Preparata [17] identifies the set of maxima in $\mathcal{O}(n \log n)$ time which is optimal since the problem exhibits a sorting lower bound [17, 23]. For constant dimensionality $d \geq 4$, their divide-and-conquer approach yields an algorithm with $\mathcal{O}(n \log^{d-2} n)$ running time [3, 17], and Matoušek [20] 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 [16] and under assumptions about the distribution of the input points in higher dimensions [4, 14]. Buchsbaum and Goodrich [5] 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. Using a dynamic convex hull algorithm, an $\mathcal{O}(n \log^2 n)$ algorithm by Overmars and van Leeuwen [22] computes the set of *convex* layers, i.e., the decomposition of a two-dimensional point set into nested convex polygons instead of a decomposition into sets of maxima. This approach was later refined by Chazelle [10] to obtain optimal $\mathcal{O}(n \log n)$ running time.

The maxima problem has been actively investigated in the database community following Börzsönyi, Kossmann, and Stocker's [7] definition of the SQL “skyline” operator. Börzsönyi et al. [7] noted that such an operator producing the set of maxima is needed in queries that, e.g., ask for hotels that are both close to the beach and have low room rates.¹ Following their definition, a number of results have been presented that use spatial index structures to produce the “skyline”, i.e., the set of maxima, practically efficient and/or in a progressive way, that is outputting results while the algorithm is running [19, 24, 26]. It remains open, though, to prove non-trivial upper bounds for the complexity of these approaches.

¹Technically speaking, this example query needs an operator that returns the set of *minima*. To unify the presentation, we do not distinguish between these (and similar) algorithmically equivalent variants of the same problem.

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 [1, 6, 8]. To compute in-place all the h maxima of a planar point set $\mathcal{P}$ in $\mathcal{O}(n \log h)$ time Brönnimann et al. [6] suggested an algorithm which adapts the output-sensitive algorithm of Chan [11] for determining the convex hull of $\mathcal{P}$.

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 [2, 27], and 3D convex hulls and related problems seem to require both (poly-)logarithmic extra space and time [2].

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 make a further step towards a non-affirmative answer to this question. 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 dimensions 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 complete ordering problem that is not amenable to a solution based solely on balanced divide-and-conquer or Graham's scan.

2 Computing the Skyline in Two and Three Dimensions

As mentioned in the introduction, 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. This definition has been transferred by Kung et al. [17] into a plane-sweeping algorithm for the two-dimensional case and a divide-and-conquer approach for the higher-dimensional case. In the following, we will demonstrate that these algorithms can be implemented in-place for the two- and three-dimensional settings. The output of the algorithm will consist of a permutation of the input array A and an index k such that k points constituting the set of maxima are stored sorted by decreasing y -coordinates in $A[0, \dots, k-1]$. Here and throughout the paper, we break ties by applying a standard geometric perturbation (shearing) technique.

2.1 Skyline Computation in Two Dimensions

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`(A, ℓ_b, ℓ_e, π) which processes a sorted (sub)array $A[\ell_b, \dots, \ell_e - 1]$ 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 $A[\ell_b, \dots, \ell_e - 1]$, whereas all elements for which π evaluates to `false` are moved (not necessarily stable) to the back of the array. This algorithm, presented by Bose et al. [8], is relatively simple to implement and 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. [17] 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 that the y -coordinate $p.y$ of the point p is not larger than the y -coordinate $m.y$ of the point m , i.e., $p.y \leq m.y$, thus, it is sufficient to check whether also the x -coordinate $p.x$ is strictly less than the x -coordinate $m.x$ of the point m , i.e. $p.x < m.x$.

The space-efficient implementation of this algorithm thus first presorts A according to the lexicographic order $<_y$ on the y -coordinates using an optimal $\mathcal{O}(n \log n)$ in-place sorting algorithm, e.g., *heapsort* [28]. 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 $A[i]$ is at least as large as the x -coordinate of the point m , i.e., iff $A[i]$ is a maximal point. If so, the algorithm then moves $A[i]$ to the front of the array and updates m to refer to (the new position of) $A[i]$. As all subroutines run in-place, we have the following lemma:

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

2.2 Skyline Computation in Three Dimensions

For the case of a three-dimensional input, Kung et al. [17] suggest a divide-and-conquer algorithm. Their algorithm subdivides the point set $\mathcal{P}$ according to the z -median ζ into two subsets $\mathcal{P}_{|z \leq \zeta}$ and $\mathcal{P}_{|z > \zeta}$ and then recursively computes the three-dimensional set of maxima for each subset.

Fact 1 Since all points in $\mathcal{P}_{|z \leq \zeta}$ have smaller z -coordinates than any point in $\mathcal{P}_{|z > \zeta}$, no point in $\mathcal{P}_{|z > \zeta}$ can be dominated by a point in $\mathcal{P}_{|z \leq \zeta}$.

By the above observation, the maxima of $\mathcal{P}_{|z > \zeta}$ are maxima of $\mathcal{P} = \mathcal{P}_{|z \leq \zeta} \cup \mathcal{P}_{|z > \zeta}$ as well. The maxima of $\mathcal{P}_{|z \leq \zeta}$ that are not dominated by any point in (the set of maxima of) $\mathcal{P}_{|z > \zeta}$ are also maxima of $\mathcal{P}$; these points are identified during the “merge” step.

Since the point sets are separated with respect to the z -axis, the merge step only needs to consider the two-dimensional subproblem given by the projection of the points into the (x, y) -plane. This subproblem is solved by simultaneously sweeping $\mathcal{P}_{|z \leq \zeta}$ and $\mathcal{P}_{|z > \zeta}$ in decreasing y -direction and keeping track of the last maximal point m of $\mathcal{P}_{|z > \zeta}$ seen so far. Then, a maximal point p of $\mathcal{P}_{|z \leq \zeta}$ encountered during this sweep is a maximal point of $\mathcal{P}$ iff its x -coordinate is at least as large as the x -coordinate of m (since m has been processed before p and since m is a maximal point of $\mathcal{P}_{|z > \zeta}$, both the y - and the z -coordinate of m are not smaller than the respective coordinates of p).

To implement this algorithm in-place we employ an in-place divide-and-conquer scheme that has been proposed earlier [8]; it uses in-place merging to provide the algorithmic template given below as Algorithm 1.

Algorithm 1 RECURSIVE(A, ℓ_b, ℓ_e): Recursive divide-and-conquer on $A[\ell_b, \dots, \ell_e - 1]$ [8].

```

1: if  $\ell_e - \ell_b \leq s$  where  $s$  is the size of the recursion base then
2:   BASE-CODE( $A, \ell_b, \ell_e$ )                                {Code for solving small instances.}
3: else
4:   PRE-CODE( $A, \ell_b, \ell_e$ )                                {Setup Subproblem 1 in
                                                                 $A[\ell_b, \dots, \lfloor (\ell_b + \ell_e)/2 \rfloor - 1]$ .}
5:   RECURSIVE( $A, \ell_b, \lfloor (\ell_b + \ell_e)/2 \rfloor$ )           {First recursive call.}
6:   MID-CODE( $A, \ell_b, \ell_e$ )                                {Setup Subproblem 2 in
                                                                 $A[\lfloor (\ell_b + \ell_e)/2 \rfloor, \dots, \ell_e - 1]$ .}
7:   RECURSIVE( $A, \lfloor (\ell_b + \ell_e)/2 \rfloor, \ell_e$ )         {Second recursive call.}
8:   POST-CODE( $A, \ell_b, \ell_e$ )                             {Merge Subproblems 1 and 2 in
                                                                 $A[\ell_b, \dots, \ell_e - 1]$ .}
9: end if

```

As a preprocessing step, we sort the input by increasing z -coordinate (breaking ties by y - and x -coordinate), hereby ensuring that in each recursive call no point in the second half of the input is dominated by a point in the first half. While recursing, we maintain the following additional invariant: after having run POST-CODE, the set of maxima is stored in decreasing y -order at the front of the current subarray.

For constant-size subarrays, it is trivial to determine the maxima and to sort them by decreasing y -coordinate (BASE-CODE) and thus we can safely assume that prior to running POST-CODE the subarray $A[\ell_b, \dots, \ell_e - 1]$ containing the current point set $\mathcal{P}$ looks as follows:

...	Maxima of $\mathcal{P}_{ z \leq \zeta}$ (sorted by $<_y$)		Maxima of $\mathcal{P}_{ z > \zeta}$ (sorted by $<_y$)		...
	ℓ_b	ℓ'_b	$\lfloor (\ell_b + \ell_e)/2 \rfloor$	ℓ'_e	ℓ_e

Our in-place divide-and-conquer scheme cannot explicitly keep track of the number of maxima in each subproblem, and thus the indices ℓ'_b and ℓ'_e (denoting the end of each recursively computed set of maxima) as well as the median z -coordinate ζ

have to be recovered algorithmically. As each set of maxima is maintained in decreasing y -order and as we can recover the median z -coordinate ζ by simply finding the maximal z -coordinate in $A[\ell_b, \dots, \lfloor (\ell_b + \ell_e)/2 \rfloor - 1]$, we can recompute these indices during a single scan over $A[\ell_b, \dots, \ell_e - 1]$. We then use linear-time swapping and merging techniques [12, 13, 15] to merge these maxima while maintaining the (decreasing) y -order.

...	Maxima of $\mathcal{P}_{ z \leq \zeta} \cup$ Maxima of $\mathcal{P}_{ z > \zeta}$ (sorted by $<_y$)		...
ℓ_b		ℓ'	ℓ_e

The union of the maxima of both subsets is then processed with an instantiation of the SORTEDSUBSETSELECTION for which the predicate π is based on Fact 1 and given below as Algorithm 2.

Algorithm 2 Predicate ISMAXIMUM($A, j, x_{\text{current}}, \zeta$) for identifying maxima during a sweep in decreasing y -direction (see [23]).

Require: ζ is the median z -coordinate that was used for partitioning; x_{current} denotes the x -coordinate of the bottommost maximum of $\mathcal{P}_{|z > \zeta}$ seen so far during the sweep.

```

1: if  $A[j].z \leq \zeta$  then                                     {Case 1:  $A[j] \in \mathcal{P}_{|z \leq \zeta}$ }
2:   if  $A[j].x \geq x_{\text{current}}$ 
3:     Return true                                           { $A[j]$  is not dominated in  $x$ -direction,
                                                             hence a maximum.}
4:   else
5:     Return false                                           { $A[j]$  is dominated, hence no maximum.}
6:   end if
7: else                                                       {Case 2:  $A[j] \in \mathcal{P}_{|z > \zeta}$ }
8:   if  $A[j].x \geq x_{\text{current}}$  then
9:     Set  $x_{\text{current}} := A[j].x$                                {Update the  $x$ -coordinate of the bottommost
                                                             maximum of  $\mathcal{P}_{|z > \zeta}$ .}
10:  end if
11:  Return true                                              {Each  $A[j] \in \mathcal{P}_{|z > \zeta}$  is a maximum.}
12: end if

```

The in-place SORTEDSUBSETSELECTION algorithm works stably, i.e., the y -order of the set of maxima is maintained, and thus the merging step establishes the invariant mentioned above. Since the predicate given in Algorithm 2 can be evaluated in constant time, merging the results of two recursive calls can be done in linear time. Thus we can combine the analysis of Kung et al.'s algorithm with the analysis of the general divide-and-conquer scheme and obtain the following result:

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

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

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 Sect. 2.1. 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 of our approach, we extend the algorithm discussed in Sect. 2.1 to compute the number of layers of maxima for a given point set. 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 Fig. 1).

As usual, we assume that the point set $\mathcal{P}$ to be processed is stored in an array $A[0, \dots, n-1]$. 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 $A[0, \dots, k-1]$.

Invariant (TAILS) is certainly true after having processed the first point $p = A[0]$ 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 $p := A[j]$. 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 $A[0, \dots, k-1]$. If p has a smaller x -coordinate than $\tau_{k-1} = A[k-1]$, p is dominated by this point, and thus p is the first point of a new layer $\mathcal{L}_k$ (see p_{j_1} in Fig. 1). We then swap $p = A[j]$ to $A[k]$ (note that $j \geq k$ trivially holds) and increment k by one. If, on the other hand, p lies right of τ_i but left of τ_{i-1} (for some $i \in \{1, \dots, k-1\}$) then p replaces τ_i as the tail point of $\mathcal{L}_i$ (see p_{j_2} in Fig. 1), that is, we swap $p = A[j]$ to $A[i]$; 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 computed 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.

3.2 Counting the Number of Points on the Topmost κ Layers

The algorithm of Sect. 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 $\mathcal{L}_i$, $i \in \{0, \dots, \kappa - 1\}$. In Sect. 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 $A[0, \dots, \kappa - 1]$ as described above.

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

3.3 Extracting the Topmost κ Layers in Sorted Order

As mentioned above, a naïve iterative approach to computing all layers of maxima leads to an $\mathcal{O}(n^2 \log n)$ worst-case running time for point 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 in a general setting, let us assume that we have already peeled off some layers and stored the result in $A[0, \dots, \ell_b - 1]$. Inductively, we maintain the following invariant which, prior to the first iteration, can be established by sorting A and setting $\ell_b := 0$:

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

...	Points not yet assigned to a layer (sorted)
ℓ_b	$n - 1$

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 $A[\ell_b, \dots, n - 1]$ 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 $A[\ell_b, \dots, \ell_b + \xi - 1]$ while maintaining the sorted y-order in $A[\ell_b + \xi, \dots, n - 1]$:

...	Points on $\mathcal{L}_0 \dots \mathcal{L}_{\kappa-1}$	Points below $\mathcal{L}_{\kappa-1}$ (sorted)
ℓ_b	$\ell_b + \xi$	$n - 1$

This partition is obtained as follows: We run a variant of the algorithm described in Sect. 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 $A[\ell_b, \dots, \ell_b + \kappa - 1]$ and stably moves all points that are *below* $\mathcal{L}_{\kappa-1}$ to the subarray starting at $A[\ell_b + \kappa]$; 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 $\mathcal{L}_i$, $i \in \{0, \dots, \kappa - 1\}$ —as mentioned in Sect. 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 $A[\ell_b, \dots, \ell_b + \kappa - 1]$ can lie below $\mathcal{L}_{\kappa-1}$ (there have to be at least κ points on κ layers).

The algorithm maintains an index ℓ under the invariant that, when processing a point $A[j]$, all points that already have been identified as “below $\mathcal{L}_{\kappa-1}$ ” are stored in decreasing y-order in $A[\ell_b + \kappa, \dots, \ell - 1]$.

...	$\tau_0 \dots \tau_{\kappa-1}$	Points below $\mathcal{L}_{\kappa-1}$ (sorted)	Points on $\mathcal{L}_0 \dots \mathcal{L}_{\kappa-1}$		
ℓ_b	$\ell_b + \kappa$	ℓ	j	$n - 1$	

Just as discussed in Sect. 3.1, the point $A[j]$ either is classified as “below $\mathcal{L}_{\kappa-1}$ ” or replaces the tail τ_i of some layer $\mathcal{L}_i$. In the first situation, $A[j]$ is stably moved directly behind $A[\ell_b + \kappa, \dots, \ell - 1]$, i.e., it is swapped with $A[\ell]$ and ℓ is then incremented by one, and in the second situation, $A[j]$ is swapped with τ_i , i.e., with $A[\ell_b + i]$. When we have reached the end of the array, we inductively see that $A[\ell_b + \kappa, \dots, \ell - 1]$ contains the points below $\mathcal{L}_{\kappa-1}$ in sorted order. Furthermore, by the definition of ξ , we know that the two subarrays $A[\ell_b, \dots, \ell_b + \kappa - 1]$ (containing the tails) and $A[\ell, \dots, n - 1]$ (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) $A[\ell_b + \kappa, \dots, \ell - 1]$ (containing the elements below $\mathcal{L}_{\kappa-1}$) and $A[\ell, \dots, n - 1]$ such that the ξ elements on the layers $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$ (tails and non-tails) are blocked in $A[\ell_b, \dots, \ell_b + \xi - 1]$. 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 $A[\ell_b, \dots, \xi - 1]$. Thus, with $\xi = \sum_{i=0}^{\kappa-1} c_i$, we have an overall $\mathcal{O}(n + \xi \log n)$ algorithm for counting the number of points on the topmost κ layers and for re-establishing Invariant (SORT).

Sorting the Points by Layer Using the counters c_i computed during the previous step, we now run a variant of counting sort on $A[\ell_b, \dots, \ell_b + \xi - 1]$ 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 $A[\ell_b + \xi, \dots, \ell_b + 2\xi - 1]$ as scratch space that will hold the layers to be constructed (note that we assume $\ell_b + 2\xi \leq n$ and that $\xi = \sum_{i=0}^{\kappa-1} c_i$ holds by definition). We traverse the subarray $A[\ell_b, \dots, \ell_b + \xi - 1]$ maintaining the tails of all layers in $A[\ell_b, \dots, \ell_b + \kappa - 1]$ 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 $A[\ell_b + \xi, \dots, \ell_b + 2\xi - 1]$ 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 $\mathcal{L}_{\kappa-1}$, all active tails are moved to the end of their respective layers and the two subarrays $A[\ell_b, \dots, \ell_b + \xi - 1]$ and $A[\ell_b + \xi, \dots, \ell_b + 2\xi - 1]$ are swapped in-place:

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

To re-establish Invariant (SORT), we finally sort $A[\ell_b + \xi, \dots, \ell_b + 2\xi - 1]$ (note that the points $A[\ell_b + 2\xi, \dots, n - 1]$ have not been touched and thus still are sorted) and update $\ell_b := \ell_b + \xi$.

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

Intermediate Analysis The cost for extracting the topmost κ layers using the above algorithm consists of (1) the cost for computing the number of points on each of the layers, (2) the cost for partitioning the points, (3) the cost for performing the counting-sort-like partitioning of the points, and (4) the cost of re-establishing Invariant (SORT). Since the points are presorted, Steps (1) and (2) both take $\mathcal{O}(n + \xi \log \kappa)$ time (Lemma 3). Sorting ξ points takes $\mathcal{O}(\xi \log \xi)$ time. We account for this by charging $\mathcal{O}(n)$ time to the current iteration of the algorithm and by additionally charging $\mathcal{O}(\log \kappa + \log \xi)$ time to each of the ξ points that have been assigned to the topmost κ layers. As Steps (3) and (4) work on a subarray of size $\Theta(\xi)$, these steps run in $\mathcal{O}(2\xi + \xi \log \xi)$ time, which can be accounted for by again charging $\mathcal{O}(\log \xi)$ time to each point assigned to a layer. We conclude that the overall algorithm for sorted extraction of the ξ points on the topmost κ layers and for re-establishing Invariant (SORT) for $A[\ell_b + \xi, \dots, n - 1]$ runs in $\mathcal{O}(n + \xi \log n)$ time.

3.4 Extracting all Layers in Sorted Order Using $\mathcal{O}(1)$ Extra Space

The exposition of the algorithm presented in the previous section was built on two major assumptions: (1) the algorithm had to have access to κ counters and (2) the

subarray $A[\ell_b, \dots, n - 1]$ 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 $\lceil \log_2 n \rceil$ 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 [21]. 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

If we reserve a block of $\frac{1}{3}n$ elements, we can encode $\frac{1}{6}n$ bits, i.e., $\frac{1}{6}n / \log n$ counters that may be used to represent values less than n . This in turn implies that the maximum number of layers for which we can run the algorithms described in Sects. 3.2 and 3.3 is bounded by $\kappa = \frac{1}{6}n / \log n$. Consequently, to extract *all* layers in-place, we need to *iteratively* execute these algorithms and to extract no more than $\frac{1}{6}n / \log n$ layers in each iteration. However, since our goal is to obtain an $\mathcal{O}(n \log n)$ overall running time, we have to upper bound the number of iterations executed. More precisely, since the analyses at the end of the previous sections gave an $\mathcal{O}(n + \xi \log n)$ bound for each run of the algorithms, we need to make sure to execute no more than $\mathcal{O}(\log n)$ iterations while at the same time being able to efficiently maintain the counters and Invariant (SORT).

3.4.1 The Case $\ell_b < \frac{1}{3}n$

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

Counting the Points on the Topmost κ Layers By Invariant (SORT), $A[\ell_b, \dots, n - 1]$ is sorted by decreasing y -coordinate, so all counters encode the value zero. We set $\kappa := \frac{1}{6}n / \log n$ 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

²We 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.

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 $A[\ell_b, \dots, n-1]$, any point q in $A[\frac{2}{3}n, \dots, n-1]$ 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 Sect. 3.2, we can compute all counters and re-establish Invariant (SORT) in $\mathcal{O}(n + \xi \log n)$ time. The technical details of this algorithm are given in the [appendix](#). 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 $\hat{c}_j := \sum_{i=0}^j c_i$. 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 $\kappa' \leq \kappa$ such that $\ell_b + 2\hat{c}_{\kappa'-1} < \frac{2}{3}n$.

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 $\xi' := \hat{c}_{\kappa'-1}$ points on the κ' topmost layers as described in Sect. 3.3 (again, the necessary modifications are given in the [appendix](#)). Because of the way κ' was chosen, we can guarantee that the scratch space of size $\hat{c}_{\kappa'-1}$ needed for the counting-sort-like partitioning does not interfere with the space $A[\frac{2}{3}n, \dots, n-1]$ reserved for representing the counters. Also, by Invariant (SORT), $A[\frac{2}{3}n, \dots, n-1]$ 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 $A[\frac{2}{3}n, \dots, n-1]$) by scanning through $A[\frac{2}{3}n, \dots, n-1]$ and swapping only adjacent points.

The counters c_i used during the following counting-sort-like partitioning are initialized with $c_0 := 0$ and $c_i := \hat{c}_{i-1}$ for $0 < i \leq \kappa' - 1$; these counters are stored in $A[\frac{2}{3}n, \dots, n-1]$. Whenever a tail τ_i is updated, the old point p representing τ_i is swapped to $A[\ell_b + \xi' + c_i]$ 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. Thus, we have an overall $\mathcal{O}(n + \xi' \log n)$ time complexity for extracting ξ' points; this also includes the cost for sorting the scratch space $A[\ell_b + \xi', \dots, \ell_b + 2\xi' - 1]$ and re-establishing Invariant (SORT) by merging $A[\ell_b + \xi', \dots, \ell_b + 2\xi' - 1]$ with the (sorted) subarray $A[\ell_b + 2\xi', \dots, n-1]$ (see Sect. 3.3).

Finishing the Iteration If $\kappa' < \kappa$, i.e., if we extract some but not all $\kappa = \frac{1}{6}n/\log n$ layers, we additionally run the linear-time skyline computation algorithm described in Sect. 2.1 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 linear-time skyline computation algorithm on $A[\ell_b, \dots, n-1]$ —note that in this case the topmost layer $\mathcal{L}_0$ contains $c_0 > \frac{1}{2}(\frac{2}{3}n - \ell_b) > \frac{1}{2}(\frac{2}{3}n - \frac{1}{3}n) \in \Theta(n)$ 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 $\kappa = \frac{1}{6}n/\log n$ layers are moved to their final positions in the array. If all ξ points are moved, we know that $\xi \geq \frac{1}{6}n/\log n$, 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 algorithm processes all such iterations in $\mathcal{O}(n \log n)$ time.

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 and re-establishing Invariant (SORT). The definition of κ' guarantees that, after we have performed the skyline computation, we have advanced the index ℓ_b by at least $\frac{1}{2}(\frac{2}{3}n - \ell_b)$ 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 all of them can be processed in $\mathcal{O}(n \log n)$ time.

3.4.2 The Case $\ell_b \geq \frac{1}{3}n$

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

Counter representation		
0	$\frac{1}{3}n$	ℓ_b
		$n - 1$

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 $A[0, \dots, \frac{1}{3}n - 1]$, 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 $\kappa = \frac{1}{6}n / \log n$ and updating the κ counters (which now are represented in $A[0, \dots, \frac{1}{3}n - 1]$). The only difference is that the algorithm's selection process will not touch the space reserved for the counters, and thus, when computing the prefix sums, the algorithm finds the maximal index κ' such that $\ell_b + 2\hat{c}_{\kappa'-1} < n$ (instead of $\ell_b + 2\hat{c}_{\kappa'-1} < \frac{2}{3}n$). Then, 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 $\mathcal{O}(v \log v)$ time where $v := n - \ell_b$. 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

$\kappa = \frac{1}{6}n / \log n$ layers can be moved to their final destination. If this is the case, we know that we have moved $\xi \geq \kappa = \frac{1}{6}n / \log n$ points and can charge $\mathcal{O}(\log n)$ time to each point moved and the remaining $\mathcal{O}(n)$ time to the iteration. Moving $\xi \geq \frac{1}{6}n / \log n$ points also implies that the total number of such iterations is bounded by $\mathcal{O}(\log n)$, and hence we have a *global* $\mathcal{O}(n \log n)$ extra cost resulting from charging $\mathcal{O}(n)$ cost to each such iteration.

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., processing all such iterations can be done in $\mathcal{O}(n \log n)$ time.

Restoring the Layers Stored in $A[0, \dots, \frac{1}{3}n - 1]$ Since the extraction algorithm uses the subarray $A[0, \dots, \frac{1}{3}n - 1]$ to maintain the counters c_i , (some of) the elements in $A[0, \dots, \frac{1}{3}n - 1]$ will be perturbed locally after the last iteration of the algorithm. To obtain *all* points in sorted layer order, we need to restore the y-order of the layers stored in $A[0, \dots, \frac{1}{3}n - 1]$. 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. More precisely, an element $q := A[2j]$ can only swap its position with its “bit-neighbor” $r := A[2j + 1]$ but will always be in the correct order relative to any point in $A[0, \dots, 2j - 1]$ or $A[2j + 1, \dots, n - 1]$. To reestablish the layer order we thus iterate over pairs of “bit-neighbors” in $A[0, \dots, \frac{1}{3}n - 1]$ while maintaining the following invariant:

Invariant (ORDER): When examining the bit-neighbors $A[2j]$ and $A[2j + 1]$, the correct layer order for all points in $A[0, \dots, 2j - 1]$ has been restored.

The details and the correctness of this approach are given as proof of the following lemma.

Lemma 4 *After the extraction phase of the overall algorithm, all local permutations due to counter increments can be undone in-place and in constant time per element, i.e., in linear overall time.*

Proof As mentioned above, the algorithm for restoring the original order of the layers stored in $A[0, \dots, \frac{1}{3}n - 1]$ is based upon the observation that each point is at most one position off its correct location. To prove Lemma 4, we will demonstrate that we can undo each local permutation iteratively by examining at most three points.

More precisely, we will show that for any three points p , q , and r that are stored consecutively in $A[0, \dots, \frac{1}{3}n - 1]$, *each* point of r and q that lies to the right of p belongs to the same layer as p . Analogously, if both q and r lie to the left of p , we will show that (due to the relative y-order of these points) they have to be part of the same layer as well. The immediate consequence will be that either one point

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

Fig. 3 Geometric configurations used for determining the layer order of q and r

dominates the other (and thus the correct layer order is readily inferred) 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 .

If one of the two bit-neighbors $q (= A[2j])$ and $r (= A[2j + 1])$ currently under inspection 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 top-right and bottom-left part of the situation depicted in the left half of Fig. 2. The situation that no point dominates the other is further detailed in the right part of Fig. 2; in order to bring q and r in their correct order, we need to access the point $p := A[2j - 1]$. If $j = 0$, p does not exist, but then q and r are the first two points of the solution 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 $A[0, \dots, 2j - 1]$, and thus *no* point *left* of p can belong to the same layer.

It remains to discuss the case that no point dominates the other (see the top-left and bottom-right part of the situation depicted in the left half of Fig. 2). W.l.o.g. we assume that r is lexicographically smaller than q , and thus the situation depicted in the right half of Fig. 2 and in Fig. 3 occurs.

By Invariant (ORDER) we know that $p := A[2j - 1]$ already is in its correct and final position, and thus the correct permutation of p , q , r is either pqr or prq . However, we also need to identify whether or not the end of one layer occurs between two elements of this permutation, and thus we have to distinguish between eight potential outcomes:

(1) pqr	(2) $p qr$	(3) $pq r$	(4) $p q r$
(5) prq	(6) $p rq$	(7) $pr q$	(8) $p r q$

The geometry of r and q (Fig. 3(a)) prohibits that r and q appear in the order qr and on the same layer, and this excludes Outcomes (1) and (2). Figure 3(b) demon-

strates why the order rq is forbidden if r is the last point of its containing level, i.e., Outcomes (7) and (8) which contain $r|q$ cannot occur. If r would be the last point of its containing level, the level would extend from r towards $y = -\infty$. Since q lies right of r , it would thus lie *above* the level containing r which would contradict the order rq . Finally, Fig. 3(c) illustrates why q cannot constitute a layer of its own. If this were the case, and if thus $p|q|r$ (Outcome (4)) were the answer (the other possibility, Outcome (8), has already been excluded), the layer induced by q would extend from q towards $x = -\infty$. Since r lies above this extension, it contradicts the outcome that r has to lie below the layer induced by q .

We conclude that the only three possible outcomes are Outcomes (3), (5), and (6), i.e., $pq|r$, prq , and $p|rq$. As we will explain in the following, these configurations can be distinguished solely by the $<_x$ -order of p , r , and q : If the correct outcome is prq , all three points lie on the same layer, and since a layer is monotone in decreasing y -direction and in increasing x -direction, it follows that in this case the x -order of the three points has to be prq .

If the correct outcome is $p|rq$, it follows from the invariant (ORDER) that p is the last point on its containing layer, and using the configuration depicted in Fig. 3(b) with p playing the role of r and either q or r playing the role of q , we conclude that neither r nor q may lie right of p . Combining this with the fact that r lies left of q , it follows that in this case the x -order of the three points has to be rqp .

Finally, let us assume that the correct outcome is $pq|r$. Since p and q are on the same layer, p lies to the left and above of q . By definition, however, r also lies to the left and above of q . This implies that r has to lie to the left of p . If this were not the case, r would lie to the right of p and above q , i.e., above the layer containing p and q and thus precede p in the final order. Since this would contradict the validity of Invariant (ORDER), it follows that r lies to the left of p , and thus in this case the x -order of the three points has to be rpq .

Since the three possible outcomes imply mutually exclusive x -order of the three points p , q , and r , we conclude vice versa that the outcomes can be identified by the relative x -order of the three points in question. Consequently, the algorithm for reconstructing the layer order runs in linear time. $\square$

3.5 Conclusions

Summing up, the proposed algorithm processes all iterations in which $\ell_b < \frac{1}{3}n$ and all iterations in which $\ell_b \geq \frac{1}{3}n$ in $\mathcal{O}(n \log n)$ time. Also, Lemma 4 gives a linear time bound for reconstructing the layer order. 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 $\mathcal{O}(n \log n)$ time such that the points in each layer are sorted by decreasing y -coordinate.*

Appendix: Maintaining Invariant (SORT) in the Presence of Counters

In this appendix, we give a detailed description of the algorithms for counting and for extracting the elements on the topmost κ layers for the case $\ell_b < \frac{1}{3}n$. As discussed in the main text, both algorithms need to be realized such that they run in $\mathcal{O}(n + \xi \log n)$ time where $\xi := \sum_{i=0}^{\kappa-1} c_i$ is the total number of points on the topmost κ (resp. κ') layers. Furthermore, as a postprocessing of the counting algorithm that determines the optimal value $\kappa' \leq \kappa$ for which to continue with the extraction phase, we need to re-establish Invariant (SORT) in the same time complexity, i.e., we need to bring all points in $A[\ell_b, \dots, n-1]$ in decreasing y -order.

The main difficulty in the task of designing such an counting or extraction algorithm space-efficiently is the fact that we need to maintain counters in the same subarray we are working in. Recall that in the situation $\ell_b < \frac{1}{3}n$ the array is implicitly partitioned as follows:

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

We note that Invariant (SORT) implies that $A[\ell_b, \dots, n-1]$ is sorted by decreasing y -coordinate, and that this invariant can be established initially, i.e., for the case $\ell_b = 0$, by sorting the points. Furthermore, we have already seen that all algorithms except for the one we are currently discussing maintain this invariant.

Let us now revisit the extraction algorithm that had been given in Sect. 3.3 for the case that we have $\mathcal{O}(\kappa)$ extra space. This algorithm maintains the invariant that the tail points τ_0 through $\tau_{\kappa-1}$ are stored in $A[\ell_b, \dots, \ell_b + \kappa - 1]$ and stably moves all $v := n - \ell_b - \xi$ points that are below $\mathcal{L}_{\kappa-1}$ to $A[\ell_b + \kappa, \dots, \ell_b + \kappa + v - 1]$; thus, it keeps all points below $\mathcal{L}_{\kappa-1}$ in sorted order. For brevity, we will refer to all points below $\mathcal{L}_{\kappa-1}$ as OUT-points, whereas points on layers $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$ are referred to as IN-points. Since OUT-points are selected stably, they are in sorted order, but IN-points may be in arbitrary order. In our situation, that is while maintaining κ counters in-place, and prior to processing the point $A[j]$, the array is implicitly partitioned as follows:

...	OUT	IN		Counter representation
$\ell_b + \kappa$	ℓ	j	$\frac{2}{3}n$	$n-1$

The point $A[j]$ now either is classified as an OUT-point or replaces the tail τ_h of some layer $\mathcal{L}_h$. In the first situation, $A[j]$ is stably moved to the end of OUT, i.e., it is swapped with $A[\ell]$ and ℓ is then incremented by one, and in the second situation, $A[j]$ is swapped with τ_h , i.e., with $A[\ell_b + h]$.

The important fact to keep in mind is that, even though the elements in $A[\frac{2}{3}n, \dots, n-1]$ are used to represent counters and thus will be touched, the relative order of the elements will change only locally. More precisely, since the elements in two consecutive positions are used to represent a bit (and thus may be swapped), an element cannot be more than one position off its correct location in sorted order—as long as the pointer j is outside the range $[\frac{2}{3}n, \dots, n-1]$.

As soon as the pointer j advances beyond $\frac{2}{3}n$, i.e., as soon as it interferes with the space reserved for representing the counters, we need to keep track of constant sized extra information: First, we monitor (the index of) the counter in the representation of which the index j (and eventually ℓ as well) is in. Recall that inside a counter's representation, the elements in two successive locations may be swapped. Thus, when trying to process the next point $A[j]$, we need to also check its “bit-neighbor”, i.e., the neighbor $A[j - 1]$ or $A[j + 1]$ together with which $A[j]$ encodes a single bit, and access the proper element in decreasing y -order. This check adds only constant extra time and space.

Let us assume for simplicity of exposition, that the proper point to be processed actually is the point $p := A[j]$ and that the bit-neighbor of $A[j]$ is $A[j + 1]$. The point p will be swapped with either $A[\ell]$ or with some τ_h , and we refer to this element as s_j . By swapping p and s_j , the relative order of $A[j]$ and its bit-neighbor $A[j + 1]$ may change, and in order to retain the value of the counter whose representation contains $A[j]$, we may need to swap the bit-neighbor of $A[j]$ and s_j . This swap may affect the position of the next element $A[j + 1] := q$ to be processed, but we can also keep track of this using constant extra space and time (any element not yet processed may be only one position off its proper location, so we can recover the correct order online).

If, on the other hand, the bit-neighbor of $A[j]$ is $A[j - 1]$, the situation is even easier to handle, since swapping s_j and the bit-neighbor will not affect the next point $A[j + 1]$ (resp. its bit-neighbor $A[j + 2]$) to be processed.

Similar care has to be taken of the counter the representation of which contains the entry $A[\ell]$, i.e., the first element of the IN-block. Whenever the current point $p := A[j]$ is classified as an OUT-point, it will be swapped to $s_j := A[\ell]$. This swap may corrupt the bit encoding that $A[\ell]$ is part of, and thus (the new value of) $A[\ell]$ may need to be swapped with its bit-neighbor. If the bit-neighbor of $A[\ell]$ is $A[\ell - 1]$, we know that $A[\ell - 1]$ contains an OUT-point $o := A[\ell - 1]$, and thus swapping o and p only affects their relative order but not the end of the OUT-block.

If, on the other hand, the bit-neighbor of $A[\ell]$ is $A[\ell + 1]$, swapping p and the entry $m := A[\ell + 1]$ will cause the boundary between IN and OUT to blur.

If the boundary was affected, we know that the next swap of some point p' to the OUT-block has to replace m , thus reconstructing a clear delineation between IN- and OUT-points. Again, we can keep track of such a situation and perform all necessary operations with $\mathcal{O}(1)$ space and constant extra time per point processed. This also allows us to reconstruct the boundary in the case that p was the last point to be classified as an OUT-point.

Inductively, we see that after having processed the last entry $A[n - 1]$, the array implicitly is partitioned as follows.

If needed, we now can at this point read off all counters and use this information to compute the optimal value for κ' as described in the main text for the case of *counting* the points on the topmost κ layers.

In any case, we need to re-establish Invariant (SORT). The key observations that allow for doing this in the desired $\mathcal{O}(n + \xi \log n)$ time complexity are that the OUT-points are not too unsorted (and thus can be sorted in linear time) and that we are allowed to sort $\Theta(\xi)$ points.

Recall that all points classified as OUT-points are selected stably and that a point used for representing a counter can be at most one position off. This allows us to traverse the subarray $A[\ell_b + \kappa, \dots, \ell - 1]$ in linear time and restore the sorted order

by looking ahead a constant number of entries. After we have done this, we swap $A[\ell_b + \kappa, \dots, \ell - 1]$ to the end of the array.

...	$\tau_0 \dots \tau_{\kappa-1}$	IN	OUT (sorted)
	ℓ_b	$\ell_b + \kappa$	$\ell_b + \xi$ $n - 1$

The union of the tails stored in $A[\ell_b, \dots, \ell_b + \kappa - 1]$ and the points classified as IN-point consists of exactly ξ points (this is how ξ was defined). We thus simply sort $A[\ell_b, \dots, \ell_b + \xi - 1]$ in $\mathcal{O}(\xi \log \xi) \subset \mathcal{O}(\xi \log n)$ time to re-establish their sorted order.

...	IN (sorted)	OUT (sorted)
	ℓ_b	$\ell_b + \xi$ $n - 1$

At this point we distinguish whether the algorithm is meant to simply *count* or to *extract* the points on the topmost κ (resp. κ') layers. If the goal of the algorithm was to extract the ξ points on the topmost κ layers, we are already done since these points are stored in $A[\ell_b, \dots, \ell_b + \xi - 1]$ and since both this subarray and $A[\ell_b + \xi, \dots, n - 1]$ are sorted. If, on the other hand, the goal was to count the points (in order to determine the proper value of κ'), we need to re-establish Invariant (SORT) for $A[\ell_b, \dots, n - 1]$. To do so, we invoke a linear-time, in-place algorithm [12, 13, 15] to merge the two sorted subarrays $A[\ell_b, \dots, \ell_b + \xi - 1]$ and $A[\ell_b + \xi, \dots, n - 1]$ into the desired decreasing y -order.

Another important distinction between counting and extracting is that during the *counting* process the counters c_i not only have to be preserved, but are also updated throughout the algorithm. The local permutations of the involved bit-neighbors potentially could affect the boundary of the IN- and OUT-block, but, fortunately, such problems can be avoided by requiring that counters should be incremented *before* moving any newly found tail to its new position. The special case, in which $A[j]$ will become a new tail τ_h and, together with its bit-neighbor, is affected by a counter update (and thus swapped) can be dealt with easily using constant extra space and time.

Summing up, we conclude that we can both count and extract the points on the topmost κ (resp. κ') layers in the presence of implicitly coded counters in-place and in time $\mathcal{O}(n + \xi \log n)$ which includes the in-place re-establishing of the Invariant (SORT).

References

1. Brönnimann, H., Chan, T.M.-Y.: Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. *Comput. Geom. Theory Appl.* **34**(2), 75–82 (2006)
2. Brönnimann, H., Chan, T.M.-Y., Chen, E.Y.: Towards in-place geometric algorithms. In: *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, pp. 239–246. ACM Press, New York (2004)
3. Bentley, J.L.: Multidimensional divide-and-conquer. *Commun. ACM* **23**(4), 214–229 (1980)
4. Bentley, J.L., Clarkson, K.L., Levine, D.B.: Fast linear expected-time algorithms for computing maxima and convex hulls. *Algorithmica* **9**(2), 168–183 (1993)

5. Buchsbaum, A.L., Goodrich, M.T.: Three-dimensional layers of maxima. *Algorithmica* **39**(4), 275–286 (2004)
6. Brönnimann, H., Iacono, J., Katajainen, J., Morin, P., Morrison, J., Toussaint, G.T.: Space-efficient planar convex hull algorithms. *Theor. Comput. Sci.* **321**(1), 25–40 (2004)
7. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proceedings of the 17th International Conference on Data Engineering, pp. 421–430. IEEE Computer Society, Los Alamitos (2001)
8. Bose, P., Maheshwari, A., Morin, P., Morrison, J., Smid, M., Vahrenhold, J.: Space-efficient geometric divide-and-conquer algorithms. *Comput. Geom. Theory Appl.* **37**(3), 209–227 (2007)
9. Chen, E.Y., Chan, T.M.-Y.: A space-efficient algorithm for line segment intersection. In Proceedings of the 15th Canadian Conference on Computational Geometry, pp. 68–71, 2003
10. Chazelle, B.M.: On the convex layers of a planar set. *IEEE Trans. Inf. Theory* **IT-31**(4), 509–517 (1985)
11. Chan, T.M.Y.: Optimal output-sensitive convex hull algorithms in two and three dimensions. *Comput. Geom. Theory Appl.* **16**(14), 361–368 (1996)
12. Chen, J.-C.: Optimizing stable in-place merging. *Theor. Comput. Sci.* **302**(1–3), 191–210 (2003)
13. Chen, J.-C.: A simple algorithm for in-place merging. *Inf. Process. Lett.* **98**(1), 34–40 (2006)
14. Dai, H.K., Zhang, X.W.: Improved linear expected-time algorithms for computing maxima. In: Farach-Colton, M. (ed.) Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN 2004). Lecture Notes in Computer Science, vol. 2976, pp. 181–192. Springer, Berlin (2004)
15. Geffert, V., Katajainen, J., Pasanen, T.: Asymptotically efficient in-place merging. *Theor. Comput. Sci.* **237**(1–2), 159–181 (2000)
16. Kapoor, S.: Dynamic maintenance of maxima of 2-D point sets. *SIAM J. Comput.* **29**(6), 1858–1877 (2000)
17. Kung, H.T., Luccio, F., Preparata, F.P.: On finding the maxima of a set of vectors. *J. ACM* **22**(4), 469–476 (1975)
18. Katajainen, J., Pasanen, T.: Stable minimum space partitioning in linear time. *BIT: Comput. Sci.* **32**, 580–585 (1992)
19. Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: An online algorithm for skyline queries. In: Proceedings of the 28th International Conference on Very Large Data Bases, pp. 275–286. Morgan Kaufmann, San Mateo (2002)
20. Matoušek, J.: Computing dominances in E^n . *Inf. Process. Lett.* **38**(5), 277–278 (1991)
21. Munro, J.I.: An implicit data structure supporting insertion, deletion, and search in $o(\log^2 n)$ time. *J. Comput. Syst. Sci.* **33**(1), 66–74 (1986)
22. Overmars, M.H., van Leeuwen, J.: Maintenance of configurations in the plane. *J. Comput. Syst. Sci.* **23**(2), 166–204 (1981)
23. Preparata, F.P., Shamos, M.I.: *Computational Geometry. An Introduction*, 2nd edn. Springer, Berlin (1988)
24. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. *ACM Trans. Database Syst.* **30**(1), 41–82 (2005)
25. Salowe, J.S., Steiger, W.L.: Stable unmerging in linear time and constant space. *Inf. Process. Lett.* **25**(3), 285–294 (1987)
26. Tan, K.-L., Eng, P.-K., Ooi, B.C.: Efficient progressive skyline computation. In: Apers, P.M.G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K., Snodgrass, R.T. (eds.) Proceedings of the 27th International Conference on Very Large Data Bases, pp. 301–310. Morgan Kaufmann, San Mateo (2001)
27. Vahrenhold, J.: Line-segment intersection made in-place. *Comput. Geom. Theory Appl.* **38**(3), 213–230 (2007)
28. Williams, J.W.J.: Algorithm 232: Heapsort. *Communications of the ACM* **7**(6), 347–348 (1964)