Space-Efficient Geometric Divide-and-Conquer Algorithms

Prosenjit Bose* Anil Maheshwari† Pat Morin‡
Jason Morrison§ Michiel Smid¶ Jan Vahrenhold‖

Draft as of June 7, 2004

Abstract

We present an approach to simulate divide-and-conquer algorithms in a space-efficient way, and illustrate it by giving space-efficient algorithms for the closest-pair, bichromatic closest-pair, all-nearest-neighbors, and orthogonal line segment intersection problems.

1 Introduction

Researchers have studied space-efficient algorithms since the early 70's. Examples include merging, (multiset) sorting, and partitioning problems; see [6, 12, 11]. Brönnimann *et al.* [3] were the first to consider space-efficient geometric algorithms and showed how to compute the convex hull of a planar set of n points in $\mathcal{O}(n \log h)$ time using $\mathcal{O}(1)$ extra space, where h denotes the size of the output. In this paper, we present a space-efficient scheme for performing divide-and-conquer algorithms without using recursion. We apply this scheme to several problems.

*School of Computer Science, Carleton University, 1125 Colonel By Dr., Ottawa, Ontario, Canada, K1S 5B6. Email: jit@scs.carleton.ca

†School of Computer Science, Carleton University, 1125 Colonel By Dr., Ottawa, Ontario, Canada, K1S 5B6. Email: maheshwa@scs.carleton.ca

‡School of Computer Science, Carleton University, 1125 Colonel By Dr., Ottawa, Ontario, Canada, K1S 5B6. Email: morin@scs.carleton.ca

§School of Computer Science, Carleton University, 1125 Colonel By Dr., Ottawa, Ontario, Canada, K1S 5B6. Email: morrison@scs.carleton.ca

¶School of Computer Science, Carleton University, 1125 Colonel By Dr., Ottawa, Ontario, Canada, K1S 5B6. Email: michiel@scs.carleton.ca

‖Westfälische Wilhelms-Universität, Institut für Informatik, 48149 Münster, Germany. Email: jan@math.uni-muenster.de. Part of this work was done while visiting Carleton University. Supported in part by DAAD grant D/0104616.

1.1 Problem Statement

Definition 1 *An algorithm that needs $\mathcal{O}(1)$ extra working space is called* in-place*, and an algorithm that needs $\mathcal{O}(\log_2 n)$ extra space is called* in situ*.*

Recently, Raman *et al.* [14, 15] considered *succinct* representations of ordered sets that allowed for various dictionary operations.

2 Space-Efficient Divide-and-Conquer

In this section, we describe a simple scheme for space-efficiently performing divide-and-conquer. Using the standard recursive approach requires $\Omega(\log n)$ pointers for maintaining a recursion stack as noted in the context of space-efficiently implementing the Quicksort algorithm [10, 16]. Our technique traverses the recursion tree in the same manner without requiring the extra pointers. In many cases, the same type of result can be obtained using bottom-up merge, however, it is well known that bottom-up merge has poor performance in the presence of caches.

The main idea (which is probably folklore, even though we have not seen it in the literature) is to simulate a post-order traversal of the recursion tree. We assume for simplicity that the data to be processed is stored in an array `A` of size $n = 2^k$ for some positive integer k. The recursion tree corresponding to a divide-and-conquer scheme is a perfectly balanced binary tree, in which each node at depth $0 \leq i < k$ corresponds to a subarray of the form $\mathtt{A}[j \cdot 2^{k-i} \ldots (j+1) \cdot 2^{k-i} - 1]$ for some integer $0 \leq j \leq i$.[1]

Our scheme is presented in Algorithm 1. We maintain two indices b and e that indicate the subarray $\mathtt{A}[b \ldots e-1]$ currently processed. We will use the binary representation of the index e to implicitly store the current status of the post-order traversal, i.e., the node of the simulated recursion tree currently visited.

Algorithm 1 Stackless simulation of post-order traversal.

```
1:  Let b = 0 and e = 1.
2:  while b ≠ 0 or e ≤ n do
3:     {Process all items in a[b ... e − 1].}
4:     Let i be the index of the least significant bit of e (note that the lowest
       index is 0).
5:     for c := 1 to i do
6:        Set b := e − 2^c.
7:        Merge the two subarrays in a[b ... e − 1]
8:     end for
9:     Let e := e + 1.
10: end while
```

[1] If the problem size is not an exact power of two, we imagine the recursion tree to be embedded into a perfectly balanced tree and stop traversing the tree prematurely.

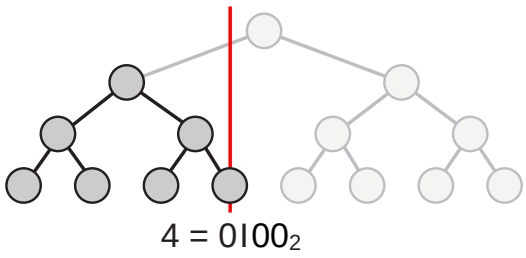

Figure 1: Merging subtrees while traversing left-to-right.

Determining the value of i (Step 3) can be done in $\mathcal{O}(1)$ amortized time without extra space using a straightforward implementation of a binary counter.

The above algorithm basically traverses the "recursion tree" left to right. When processing a leaf v, the algorithm backtracks in geomtrically increasing steps merging all subtrees already traversed completely. This merging is done by traversing a leaf-to-root path starting from v but stopping as soon as the path goes up to the right (Figure 1). The correctness of the algorithm follows from the next lemma.[1]

[1]: JV: Do we need a proof?

Lemma 1 *If the leaves of a complete binary tree are labeled from left to right (starting with* 1 *for the leftmost leaf), the height of the largest subtree containing the leaf* $e > 1$ *as its rightmost leaf is equal to the index* i *of the least significant bit of the number* e*.*

3 Nearest Neighbor Problems

3.1 Closest Pair

Given a set P of n points in the plane stored in an array $\mathtt{A}[0 \ldots n-1]$, a closest pair is a pair of points in P whose Euclidean distance is smallest among all pairs of points. The above scheme can be used to modify an algorithm by Bentley and Shamos [1] to compute the closest pair in a space-efficient manner using only $\mathcal{O}(1)$ extra space.

We first outline Bentley and Shamos' algorithm.

To make this algorithm *in-place*, we require that for each "recursive" call on a subarray $A[b \ldots e], e > b$, the following invariants are fulfilled as a postcondition:

Invariant 1: The first two entries $A[b]$ and $A[b+1]$ of the subarray contain two points p and q that form a closest pair in $A[b \ldots e]$.

Invariant 2: $A[b] <_y A[b+1]$.

Invariant 3: If $e > b+1$, then $A[b+2 \ldots e]$ is sorted according to $<_y$.

Algorithm 2 Divide-and-Conquer algorithm for finding a closest pair [1].

Require: All points in the input array A are sorted according to x-coordinate.
Ensure: All points in the array A are sorted according to $<_y$, and two points realizing a closest pair in A are known.

1: If A has a constant number of points, sort the points according to $<_y$, compute a closest pair using a brute-force algorithm, and return.
2: Subdivide the array based upon the median x-coordinate and recurse on both parts.
3: Determine the minimal distance δ given by the two (locally) closest pairs of the subarrays to be merged. Set the closest pair for the current subarray to be the closer of those two points.
4: For both subarrays, extract the points that fall within a strip of width 2δ centered at the median x-coordinate.
5: Simultaneously scan through both sets containing the extracted points (backing up at most a constant number of steps for each point examined) and determine whether there is a pair of points with distance smaller than δ. Update δ and the closest pair as necessary.
6: Merge both subarrays such that all points are sorted according to $<_y$.
7: Return the closest pair.

These invariants can be enforced trivially *in-place* in Step 1 of the above algorithm. After returning from the "recursive" call on $A[0 \dots n-1]$, i.e., at the end of the algorithm, the first two entries $A[0]$ and $A[1]$ contain two points that realize a closest pair.

We can transform the above algorithm into an *in-place* space-efficient variant as follows: The precondition of the algorithm can be met by running any *in-place* sorting algorithm, e.g., *heapsort* [5], to sort the points according to their x-coordinate.

The merge step of the divide-and-conquer scheme can be realized *in-place* as well. By Invariants 1–3, we know that the first two entries of the two subarrays to be merged form a closest pair, and that all other points are sorted w.r.t. the y-coordinate. We first determine the pair realizing the closer of the two pairs, and temporarily use $\mathcal{O}(1)$ extra space for storing these two points in variables p and q and their distance δ (Step 3). Now, we merge the four points realizing the closest pairs into their respective subarrays such that each subarray is sorted according to $<_y$. This merging can be done *in-place* using the algorithm by Geffert *et al.* [6]. We then use a stable *in-place* partitioning algorithm, e.g. [11], to partition each subarray into two sets, one "within" the strip of width 2δ, one "outside" (Step 4). As the partitioning algorithm is stable, all resulting sets are still sorted according to $<_y$. Consequently, we can scan using the standard backtracking algorithm the "inside" set to determine two points that realize a global closest pair (Step 5).

After having determined the closest pair within the strip, we check whether its distance is less than the distance between p and q. If so, we update the

extra space such that p and q contain the new closest pair. We then merge the "inside" and "outside" sets (both of which are still sorted according to $<_y$) using the stable *in-place* algorithm of Geffert *et al.* [6] such as to fulfill Invariant 3 (Step 6). Finally, we stably partition the sorted set such that the two points stored in p and q (realizing the new closest pair) are put at the front of the subarray, hence establishing Invariants 1 and 2.

Each of the sub-algorithms employed works *in-place* and runs in linear time, hence we can conclude:

Theorem 1 *Given a set P of n points in the plan stored in an array $\mathtt{A}[0 \ldots n-1]$, a closest pair in P can be computed in $\mathcal{O}(n \log n)$ time using $\mathcal{O}(\log n)$ extra bits of storage.*

- **ToDo:** What about higher dimensions (using the same algorithm)?

3.2 Bichromatic Closest Pair

In the Bichromatic Closest Pair Problem, we are given a set R of red points and a set B of blue points in the plane. The problem is to return a pair of points, one red and one blue, whose distance is minimum over all red-blue pairs. For simplicity of exposition, we assume that $|R| = n$ and $|B| = m \leq n$ with the red set stored in an array $\mathtt{R}[0 \ldots n-1]$ and blue set in an array $\mathtt{B}[0 \ldots m-1]$.

We first consider solving the problem when the red and blue sets are separated by a vertical line, with red points on the left of the line and blue points on the right of the line. The approach is similar to the merge step in the previous section, except that we no longer have the luxury of the value δ. To circumvent this problem we proceed in the following way.

Algorithm 3 Bichromatic closest pair when R and B are separated by a vertical line.

Require: All points in R and B are sorted by y-coordinate.

Ensure: All points in R and B are sorted by y-coordinate, and the pair ($\mathtt{R}[0]$,$\mathtt{B}[0]$) realizes a bichromatic closest pair.

1: If both R and B store only a constant number of points, sort the points according to $<_y$, compute a bichromatic closest pair using a brute-force algorithm, restore the $<_y$ order for all points, and return.
2: Assume $|R| \geq |B|$, otherwise reverse the roles of R and B.
3: Pick a random element r from R.
4: Find an element $b \in B$ closest to r.
5: Compute the left envelope of disks having radius $\mathrm{dist}(r, b)$ centered at each of the points in B.
6: Remove all elements of R that are outside the envelope and iterate the algorithm.

The above algorithm runs in linear-expected time since each time through the loop, with constant probability, we reduce the size of R or B by a constant fraction just as in the partitioning step of the Quicksort algorithm.

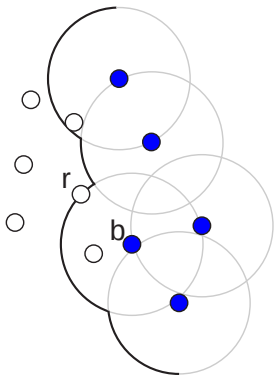

Figure 2: Left envelope of disks centered at blue points (red points are drawn as hollow disks).

To implement this algorithm in-place, we observe that all steps except Steps 5 and 6 are trivial to implement in-place.

Step 5, computing the left-envelope (portions of the disks visible from the point $(-\infty, 0)$), is very similar to the convex hull problem and can be solved in $\mathcal{O}(n)$ time with an algorithm identical to Graham's scan since the points are sorted by $<_y$. The implementation of Graham's scan given by Brönnimann *et al.* [3] does this in-place and results in an array that contains the elements that contribute to the left envelope in the first part of the array and the elements that do not contribute in the second part of the array. Also, we observe that it is not particulary difficult to run Graham's scan "in reverse" to restore the $<_y$ sorted order of the elements in $\mathcal{O}(n)$ time once we are done with the left envelope. To see this, consider the following pseudo-code implementation of Graham's Scan (Algorithm 4):

Algorithm 4 Computing the left convex hull of a point set.

Require: All points in the input array $A[0 \ldots n-1]$ are sorted according to y-coordinate.

1: **for** $i := 0$ to $n-1$ **do**
2: **while** $(h \geq 2)$ and $((\mathtt{A}[h-2], \mathtt{A}[h-1], \mathtt{A}[i])$ does not form a right turn) **do**
3: Set $h := h - 1$.
4: **end while**
5: Swap $\mathtt{A}[i]$ and $\mathtt{A}[h]$.
6: Set $h = h + 1$.
7: **end for**

The effect of this algorithm can be reversed by Algorithm 5:

To perform Step 6 in-place we simultaneously scan the left envelope and the red points from top to bottom and move the discarded points, using swaps to

Algorithm 5 Restoring the $<_y$-order after computing the left envelope.

Require: Array $\mathtt{A}[0 \ldots h]$ contains the result of running Algorithm 4 on (the sorted) array $\mathtt{A}[0 \ldots n-1]$.

```
1:  Set q := n − 1.
2:  while h ≠ q do
3:    if A[q] <_y A[h] then
4:      Swap A[h] and A[q].
5:      Set q := q − 1.
6:      if A[q] <_y A[h − 1] then
7:        Set h := h − 1.
8:      end if
9:      while (A[h] <_y A[h + 1]) and ((A[h − 2], A[h − 1], A[i]) form a right turn)
        do
10:       Set h := h + 1.
11:     end while
12:   else
13:     Set h := h + 1.
14:   end if
15: end while
```

the end of the array.

At this point, we will introduce a simple algorithm, called SortedSubsetSelection$(\mathtt{A}, b, e, f)$, that operates on a given array $\mathtt{A}[b \ldots e-1]$ which is sorted according to some total order, say $<_{\mathtt{A}}$. This algorithm, described below as Algorithm 6, stably moves all elements in $\mathtt{A}[b \ldots e-1]$ for which the given $(0,1)$-valued function f returns the value zero to the front of the array. Moreover, this algorithm has the desirable property that its effect can be reverted easily.

The above algorithm clearly works in-place and runs in linear time. The effects of this algorithm can be reverted by the following algorithm (Algorithm 7):

Again the details are exactly like the implementation of Graham's scan given by Brönnimann *et al.*, and the algorithm can be run in reverse to recover the $<_y$-sorted order of the points.

Theorem 2 *Given sets R and B of $\Theta(n)$ points in the plane, a closest bichromatic pair can be computed in $\mathcal{O}(n \log n)$ expected time using $\mathcal{O}(\log n)$ extra bits of storage.*

3.3 All Nearest Neighbors

In this section, we apply the divide-and-conquer scheme of Section 1 to solve the all-nearest neighbors problem space-efficiently. Again, we present a modification of Bentley and Shamos' algorithm.

We can make this algorithm space-efficient using the framework of Section 1. One detail, however, needs special attention: Bentley and Shamos argue that

Algorithm 6 Algorithm SortedSubsetSelection($\mathtt{A}, b, e, f$) for selecting a subset from a sorted array $\mathtt{A}[b \ldots e-1]$.

Require: Array $\mathtt{A}[b \ldots e-1]$ is sorted according to some total order $<_{\mathtt{A}}$, and the is a constant-size $(0,1)$-valued function f that can be evaluated in constant time.

Ensure: $\mathtt{A}[b \ldots m-1]$ contains all entries for which f is zero, and these entries are still (stably) ordered according to $<_{\mathtt{A}}$.

1: Set $i := b$, $j := b$ and $m := b$.
2: **while** $i < e$ and $j < e$ **do**
3: **while** $i < e$ and $f(\mathtt{A}[i]) = 0$ **do**
4: Set $i := i + 1$. {Move index i such that $f(\mathtt{A}[i]) = 1$.}
5: **end while**
6: Set $j := i + 1$;
7: **while** $(j < e)$ and $(f(\mathtt{A}[j]) = 1)$ **do**
8: Set $j := j + 1$. {Move index j such that $f(\mathtt{A}[j]) = 0$.}
9: **end while**
10: **if** $j < e$ **then**
11: Swap $\mathtt{A}[i]$ and $\mathtt{A}[j]$.
12: Set $m := i + 1$. {$\mathtt{A}[b \ldots m-1]$ contains all 0-valued entries.}
13: **end if**
14: **end while**
15: Return m.

Algorithm 7 Algorithm UndoSubsetSelection($\mathtt{A}, b, e, m$) for restoring the total order after applying the SortedSubsetSelection-Algorithm 6

Require: Array $\mathtt{A}[b \ldots m-1]$ contains the result of running Algorithm 6 on array $\mathtt{A}[b \ldots e-1]$ that was sorted according to $<_{\mathtt{A}}$.

Ensure: Array $\mathtt{A}[b \ldots e-1]$ is sorted according to $<_{\mathtt{A}}$.

1: Set $i := m - 1$ and $j := e - 1$.
2: **while** $i \neq j$ and $i \geq b$ **do**
3: **if** $\mathtt{A}[j] <_{\mathtt{A}} \mathtt{A}[i]$ **then**
4: Swap $\mathtt{A}[i]$ and $\mathtt{A}[j]$.
5: Set $j := j - 1$.
6: **end if**
7: Set $i := i - 1$.
8: **end while**

Algorithm 8 Divide-and-Conquer algorithm for computing all nearest neighbors [1].

Require: All points in the input array A are sorted according to x-coordinate.
Ensure: All points in A are sorted according to $<_y$, and for each point, its nearest neighbor in A is known.

1: If A stores only a constant number of points, sort the points according to $<_y$, compute all nearest neighbors using a brute-force algorithm, and return.
2: Subdivide the array based upon the median x-coordinate $x = \ell$ and recurse on both subarrays A_0 and A_1.
3: Simultaneously scan through both subarrays A_0 and A_1, and for each point $p \in A_i$ check all points in A_{1-i} whose nearest-neighbor ball contains the projection of p onto the line $x = \ell$. Update the nearest neighbor information as necessary.
4: Merge the points according to $<_y$.

for each of the points in the sets to be merged, only "a constant number other points needs be examined" [1, p. 229]. More specifically, "this number is four for points in two dimensions under the L_2 (Euclidean) metric" [1, p. 228]

Note that when processing a point p, the four points above and below p's y-coordinate whose nearest neighbor balls intersect the vertical line may be interspersed (with respect to the $<_y$ order) by linearly many points. In order to provide constant-time access to these points, we proceed as follows. We use $2n \cdot log_2 n$ bits of extra space to maintain the following invariants that have to be fulfilled as a postcondition after each "recursive" call on a subarray $\mathtt{A}[b \ldots e-1]$, $e > b$:

Invariant 1: $A[b \ldots e-1]$ is sorted according to $<_y$.

Invariant 2: Any point $p \in A[b \ldots e-1]$ stores an index $i \in [b \ldots e-1]$ such that $A[i]$ is the nearest neighbor of p with respect to $A[b \ldots e-1]$.

If these invariants are maintained throughout the algorithm, each point will store the index of its global nearest neighbor at the end of the algorithm. The main problem in performing the merge step of this *in-place* version of the divide-and-conquer algorithm is that while simultaneously scanning the two sorted arrays of points, i.e. *before* merging them, for each point we need to compute the index of its nearest neighbor with respect to the *merged* array.

This is done in two phases. First, we do a linear scan to compute the index of each element in the merged array and store this index with the element (this is where we use $n \cdot log_2 n$ extra bits). Next, we perform the merge step, and maintain a look-ahead queue of length 8 for each point set. In this queue, we maintain the indices of the next 4 and the last 4 points seen whose nearest neighbor balls intersect the vertical line at the median x-coordinate. It is easy to see that these queues can be maintained space-efficiently while not increasing (asymptotically) the running time.

4 Orthogonal line segment intersection

In this section, we present a space-efficient algorithm for the orthogonal line segment intersection problem. Since this problem has a variable output size $k \in \mathcal{O}(n^2)$ we consider the output memory model to be a write-only space usable only for output and not for temporary space. This model has been used by Chen and Chan [4] for variable size output, space-efficient algorithms and accurately models algorithms that have output streams with write-only buffer space.

Our algorithm can be seen as a variant of the (external-memory) *distribution sweeping* approach taken by Goodrich *et al.* [7]. The (internal memory) non-space efficient version of this algorithm is a top-down divide-and-conquer algorithm, that is, the (algorithmic) work is done prior to recursion. As a precondition, assume that all vertical segments are sorted according to the y-coordinate of their upper endpoints and that all horizontal segments are sorted according to their y-coordinate. All horizontal segments that completely span the "slab" of y-coordinates of the current vertical segments are pushed onto a list. Subsequently a descending sweep of all segments is made and whenever (the lower end-point of) a vertical segment is encountered, the list of spanning horizontal segments is scanned and all intersecting segments are reported. To prepare for recursion, the set of vertical segments is split according to their median x-coordinate, thereby creating two new sub-slabs. Subsequently all horizontal segments (or fragments thereof) that did not completely span the current slab are associated with all of the sub-slabs that they span or partially span. The algorithm recurses on each of the sub-slabs of vertical segments and their associated horizontal segments. The algorithm stops recursing when the size of the slab of vertical segments or its associated horizontal segments is $\mathcal{O}(1)$ and computes all intersections by brute force.

Before presenting the space efficient version of this algorithm we first analyze the time complexity of the above algorithm. To analyze the running time of this algorithm we define n to be the total number of segments and therfore the input sets of vertical segments(`V`) and horizontal segments(`H`) are both of size $\mathcal{O}(n)$.

Since there are $\mathcal{O}(n)$ vertical segments and the recursion is evenly split, the recursion tree has a depth of $\mathcal{O}(\log n)$. Furthermore, all vertical segments appear in the first recursion step and subsequent steps partition the set of vertical segments, therefore each vertical segment appears in $\mathcal{O}(\log n)$ recursion steps.

Each horizontal segment appears in all recursion steps whose slab it partially spans. For each of these $\mathcal{O}(\log n)$ steps the horizontal segment may also appear in at most one child step whose slab it completely spans. This implies that each vertical segment appears in $\mathcal{O}(\log n)$ recursion steps.

The algorithm is easily separated into five stages for analysis:

1. Initial sorting of all segments by y-coordinate,
2. pre-sweep preparation in each recursive step,

3. sweep time in each recursive step and,
4. preparation for each subsequent recursion:
   (a) median x-coordinate of vertical segments
   (b) partitioning of vertical segments and association of horizontal segments

Clearly, step 1 and step 4(a) each requires $O(n \log n)$ time. Step 2 requires that each horizontal segment in each recursion step requires $O(1)$ time. By the previous arguments there are at most $\mathcal{O}(n)$ such segments appearing in $\mathcal{O}(\log n)$ recursion steps thereby requiring $O(n \log n)$ time for all Step 2's combined. Steps 3 and 4(b) requires that every horizontal and vertical segment in each recursion step requires $O(1)$ time plus $O(1)$ for each reported intersection and thus all Step 3's and 4(b)'s combined require $O(n \log n + k)$ time.

Several non-trivial hurdles must be overcome to make the above algorithm space efficient. First and foremost, is the median algorithm in Step 4(a) and the subsequent partitioning of the same set in Step 4(b). By the prerequisites of each recursion step the two partitions must be sorted by y-coordinate; however, space-efficient linear time processes of discovering the x-median destroy the original y-ordering[2]. Secondly the possible appearance of horizontal segments in both children of a recursion step and the need to preserve y-ordering of segments easily leads to implementations with $O(n \log^2 n)$ extra bits of space. Lastly the creation of the extra list of spanning horizontal segments could also lead to implementations requiring $O(n \log n)$ extra bits.

The last item is the most easily solved. If `H` and `V` are originally stored in arrays then each recursive call will simply pass a reference to them both of them with indices of the relevant continuous sub-arrays. This requires $O(\log^2 n)$ bits of extra space. Partitioning the relevant sub-array of `H` into spanning and non-spanning can be achieved just as Graham Scan is by Brönnimann *et al.* [3]. The reversal of this partitioning is achieved, as shown in Algorithm 5, just before Step 4(a). The partitionings of vertical segments and horizontal segments associated with the first slab performed before the first recursion call are possible in linear time if the median operation does not re-order the vertical segments (more on this later). After the recursion call the previous partitioning of the horizontal segments is reversed in linear time. Then the partitioning of horizontal segments associated with the second slab is performed, before the second recursion call. Lastly, the partitioning of horizontal segments is undone.

What remains is to obtain the median in linear time with minimal extra space and while preserving the ordering. Our solution to this problem is to take a randomized approach similar to Quicksort that uses a random element instead of the median to partition. The selection of this element reduces to $O(1)$ while the partitioning steps remain $\mathcal{O}(n)$. The recursion tree is still logarithmic with high probability [9] therefore all of the previous time and space bounds still apply with high probability.

[2]The authors are currently unaware of a linear time, space efficient, deterministic median algorithm that preserves ordering

5 Other Space-Efficient Geometric Algorithms

In this section, we present some simple observations about well-known algorithms for three geometric problems and show how to space-efficiently encode the results.

5.1 Convex Hull of a Simple Polygon

Using an space-efficient stack, e.g., along the lines of Brönnimann *et al.*´s [3] description of Graham´s Scan [8], we can implement an optimal space-efficient algorithm for computing the convex hull of a simple polygon. To do so, we only need to observe that the only operations performed in the algorithm described by Preparata and Shamos [13, Chap. 4.1.4] are push and pop operations.

Lemma 2 *The convex hull of a simple polygon with n vertices can be computed* in-place *in $\mathcal{O}(n)$ time.*

5.2 Diameter of a Point Set

Using the space-efficient algorithm of Brönnimann *et al.* [3], we immediately obtain an optimal space-efficient algorithm for computing the diameter of a point set. This is due to the fact that the algorithm for enumerating antipodal pairs as described by Preparata and Shamos [13, Chap. 4.2.3] already is an *in-place* traversal of the boundary of the convex hull. To encode the output without any extra space, we use an *in-place* partitioning algorithm, e.g. [11], to move an antipodal pair determining the diameter to the start of the array containing the points.

Lemma 3 *The diameter of a set of n points in the plane can be computed* in place *in $\mathcal{O}(n \log_2 n)$ time. The encoding of the result does not need any extra space.*

5.3 Minimum Enclosing Circle

A close look at Welzl´s incremental algorithm [17] shows that this algorithm already works *in-place*. We can modify the output of this algorithm such that the array containing the points is reorganized in the following way: Either the first three points determine the minimum enclosing circle or the first two points determine the diameter of the circle. Which of these cases applies, can be checked by first assuming that the first two points determine the diameter and then checking whether the third point is contained within this circle. As we can always arrange the three points that determine the minimum enclosing circle in such a way that the circle given by first two points does not contain the third point (take the endpoints of the shortest edge in the triangle formed by these points), this shows that the result can be encoded without any extra space.

Lemma 4 *The Minimum Enclosing Circle of a set of n points in the plane can be computed* in-place *in $\mathcal{O}(n)$ expected time. The encoding of the result does not need any extra space.*

6 Conclusions

References

[1] J. L. Bentley and M. I. Shamos. Divide-and-Conquer in multidimensional space. In *Proceedings of the Eighth Annual ACM Symposium on the Theory of Computation*, pages 220–230. Association for Computing Machinery, 1976.

[2] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, Aug. 1973.

[3] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. Optimal in-place planar convex hull algorithms. In S. Rajsbaum, editor, *Proceedings of Latin American Theoretical Informatics (LATIN 2002)*, volume 2286 of *Lecture Notes in Computer Science*, pages 494–507, Berlin, 2002. Springer.

[4] E. Y. Chen and T. M.-Y. Chan. A space-efficent algorithm for line segment intersection. In *Proceedings of the 15th Canadian Conference on Computational Geometry*, pages 68–71, 2003.

[5] R. W. Floyd. Algorithm 245: Treesort. *Communications of the ACM*, 7(12):701, Dec. 1964.

[6] V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, 237(1–2):159–181, Apr. 2000.

[7] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[8] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[9] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, July 1961.

[10] B.-C. Huang and D. E. Knuth. A one-way, stackless quicksort algorithm. *BIT*, 26:127–130, 1986.

[11] J. Katajainen and T. Pasanen. Stable minimum space partitioning in linear time. *BIT*, 32:580–585, 1992.

[12] J. Katajainen and T. Pasanen. Sorting multisets stably in minimum space. *Acta Informatica*, 31(4):301–313, 1994.

[13] F. P. Preparata and M. I. Shamos. *Computational Geometry. An Introduction*. Springer, Berlin, second edition, 1988.

[14] R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In F. Dehne, J.-R. Sack, and R. Tamassia, editors, *Algorithms and Data Structures, 7th International Workshop, WADS 2001*, volume 2125 of *Lecture Notes in Computer Science*, pages 426–437, Berlin, 2001. Springer.

[15] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encodung k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242. Association for Computing Machinery, 2002.

[16] L. M. Wegner. A generalized, stackless quicksort algorithm. *BIT*, 27:44–48, 1987.

[17] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes in Computer Science*, pages 359–370. Springer, Berlin, 1991.

A Selecting the k-th smallest element

In this section, we describe a space-efficient variant of the well-known median-find algorithm by Blum *et al.* [2]. Our algorithm assumes that the input is given in the form of an array $\mathtt{A}[0 \ldots n-1]$ which is sorted according to some total order $<_{\mathtt{A}}$. The goal of the algorithm is, given an integer $k \in [0 \ldots n-1]$, to select the k-th smallest element in $\mathtt{A}$ *according to some other total order* $<_{\mathtt{B}}$. This algorithm will run in linear time and will require only $\mathcal{O}(\log n)$ extra bits of storage. Additionally, the algorithm shall return the array $\mathtt{A}$ in the same order as it was presented, namely sorted according to $<_{\mathtt{A}}$.

The correctness of the algorithm described below will follow form the following invariant:

Invariant: Assume the algorithm is called to select the k-th smallest element from a $<_{\mathtt{A}}$-sorted (sub-)array $\mathtt{A}[b \ldots e-1]$, where b, e, and k, are three global variables. Then, upon returning from this call, b, e, and k will be restored to the values they had when the algorithm was called. Additionally, $\mathtt{A}[b \ldots e-1]$ will be sorted according to $<_{\mathtt{A}}$.

The invariant is trivially to enforce for any constant-sized input. We now are ready to describe the algorithm.

Note that the above algorithm is described in a recursive way to facilitate the analysis and the proof of correctness. We can convert this algorithm into a non-recursive variant by simply maintaining all stack of two-bit entries that

Algorithm 9 The algorithm RESTORINGSELECT($\mathtt{A}, b, e, k,$ mode) for selecting the k-th smallest element in $\mathtt{A}[b \ldots e-1]$ (if mode = SELECT) or the median element (if mode = MEDIAN).

Require: $\mathtt{A}[b \ldots e-1]$ is sorted according to $<_{\mathtt{A}}$.

Ensure: $\mathtt{A}[b \ldots e-1]$ is sorted according to $<_{\mathtt{A}}$. The variables b, e, and k are reset to their original values.

1: Subdivide $\mathtt{A}[b \ldots e-1]$ into $\lfloor (e-b)/5 \rfloor$ groups.
2: PUSH(S, i).
3: Move the medians of the groups to $\mathtt{A}[b \ldots b + \lfloor (e-b)/5 \rfloor - 1]$ using the algorithm SORTEDSUBSETSELECTION.
4: Let $i := (e-b) - \lfloor (e-b)/5 \rfloor \cdot 5$.
5: $i_{\text{med}} :=$ RESTORINGSELECT$(\mathtt{A}, b, b + \lfloor (e-b)/5 \rfloor, k,$ MEDIAN$)$.
6: UNDOSUBSETSELECTION$(\mathtt{A}, b, b + \lfloor (e-b)/5 \rfloor \cdot 5, \lfloor (e-b)/5 \rfloor)$.
7: Let $i :=$ POP(S).
8: Let $e := b + \lfloor (e-b)/5 \rfloor \cdot 5 + i$.
9: **if** mode = MEDIAN **then**
10: Let $l := \lfloor (e-b)/2 \rfloor$
11: **else**
12: Let $l := k$
13: **end if**
14: Let $x := \mathtt{A}[i_{\text{med}}]$.
15: Let $k_< := |\{a \in \mathtt{A}[b \ldots e-1] \mid a < x\}|$, $k_= := |\{a \in \mathtt{A}[b \ldots e-1] \mid a = x\}|$, and $k_> := |\{a \in \mathtt{A}[b \ldots e-1] \mid a > x\}|$.
16: **if** $l \notin [k_< + 1, k_< + k_=]$ **then**
17: **if** $l \leq k_<$ **then**
18: **if** $l = k_<$ **then**
19: Set i_{med} to point to the largest element in $\mathtt{A}[b \ldots e-1]$ less than x (according to $<_{\mathtt{B}}$).
20: **else**
21: Move x to $\mathtt{A}[b]$.
22: Using SORTEDSUBSETSELECTION, move all elements in $\mathtt{A}[b+1 \ldots e-1]$ less than x to $\mathtt{A}[b+1 \ldots b+k_<]$.
23: Move the largest element less than x (according to $<_{\mathtt{B}}$) to $\mathtt{A}[e-1]$.
24: $i_{\text{med}} :=$ RESTORINGSELECT$(\mathtt{A}, b+1, b+k_<, k,$ SELECT$)$.
25: Starting at $\mathtt{A}[b+k_<]$, scan $\mathtt{A}$ to find $e-1$ (the index of the first element y for which $y <_{\mathtt{B}} x := A[b]$).
26: Move $\mathtt{A}[e-1]$ to its proper position in $\mathtt{A}[b+1 \ldots b+k_<]$.
27: UNDOSUBSETSELECTION$(\mathtt{A}, b+1, e, b+k_<+1)$.
28: Reinsert (according to $<_{\mathtt{A}}$) $x := \mathtt{A}[b]$ into $\mathtt{A}[b \ldots e-1]$ maintaining i_{med}.
29: **end if**
30: **else**
31: $(\ldots)$
43: **end if**
44: **end if**
45: Return i_{med}.

Algorithm 9 Algorithm RestoringSelect($\mathtt{A}, b, e, k,$ mode) (contd.)

16: **if** $l \notin [k_< + 1, k_< + k_=]$ **then**
17: Move x to $\mathtt{A}[b]$.
18: **if** $l \leq k_<$ **then**
19: $(\dots)$

30: **else**
31: **if** $l = b - e$ **then**
32: Set i_{med} to point to the largest element in $\mathtt{A}[b \dots e-1]$ larger than x (according to $<_{\mathtt{B}}$).
33: **else**
34: Using SortedSubsetSelection, move all elements in $\mathtt{A}[b+1 \dots e-1]$ larger than x to $\mathtt{A}[b+1 \dots b+k_>]$.
35: Move the smallest element larger than x (according to $<_{\mathtt{B}}$) to $\mathtt{A}[e-1]$.
36: $i_{\text{med}} :=$ RestoringSelect($\mathtt{A}, b+1, b+k_>, k-(k_< + k_=)$, Select).
37: Starting at $\mathtt{A}[b+k_>]$, scan $\mathtt{A}$ to find $e-1$ (the index of the first element y for which $\mathtt{A}[b] =: x <_{\mathtt{B}} y$).
38: Move $\mathtt{A}[e-1]$ to its proper position in $\mathtt{A}[b+1 \dots b+k_>]$.
39: UndoSubsetSelection($\mathtt{A}, b+1, e, b+k_>+1$).
40: Reinsert (according to $<_{\mathtt{A}}$) $x := A[b]$ into $\mathtt{A}[b \dots e-1]$ maintaining i_{med}.
41: Recompute $k_<$ and $k_=$ (as above), set $k := (k-(k_< + k_=)) + k_< + k_=$.
42: **end if**
43: **end if**
44: **end if**
45: Return i_{med}.

indicate whether the current "invocation" took place from line 5, 24, or 36. This stack has a worst-case depth of $\mathcal{O}(\log n)$ and thus will (in an asymptotic sence) not increase the extra space required by this algorithm. Also, the stack S used in Steps 2 and 7 contains only integers in the range $[0 \dots 4]$, so its overall size is bounded by $\mathcal{O}(\log n)$ bits, too.

Assuming that the above invariant is fulfilled for any constant-size input, we can inductively assume that the invariant holds after the "recursive" call in line 5. This implies, that for the successive call to UndoSubsetSelection the parameters b, $\lfloor (e-b)/5 \rfloor$, and hence also $e_1 := b + \lfloor (e-b)/5 \rfloor$ and $e_2 := b + \lfloor (e-b)/5 \rfloor \cdot 5$ are known. The situation prior to the call to UndoSubsetSelection is depicted in Figure 3: The array $\mathtt{A}[b \dots e_1 - 1]$ contains the medians in sorted $<_{\mathtt{A}}$-order that had been selected from $\mathtt{A}[b \dots e_2 - 1]$, and the remaining i elements are still untouched, hence also sorted according to $<_{\mathtt{A}}$.

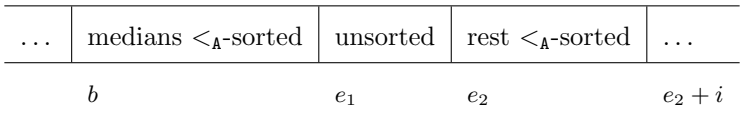

Figure 3: Restoring the $<_{\mathtt{A}}$-order after having computed the median of the $\lfloor (b-e)/5 \rfloor$ medians. Here $e_1 := b + \lfloor (b-e)/5 \rfloor$ and $e_2 := b + \lfloor (b-e)/5 \rfloor \cdot 5$.

As a consequence, we can first undo the effects of SortedSubsetSelection on $\mathtt{A}[b \dots e_2 - 1]$, hence restoring it to sorted $<_{\mathtt{A}}$-order and finally adjust e to point to $e_2 + i$. This implies that b and e are known and $\mathtt{A}[b \dots e - 1]$ is sorted according to $<_{\mathtt{A}}$. As the original value of k had been passed to the "recursive" call to RestoringSelection, by the invariant, we still know its value.

For the second and third situtation in which a "recursive" call to RestoringSelection may happen (lines 24 and 36), we do not not exactly how many elements are passed to the "recursive" call. So, in order to recover the original value of e after the call, we move the median-of-medians x to the front of the array and use a distinguished element y to denote the *end* of the subarray that is not passed the recursive call. Let us consider the situation where the k-th element to be selected is larger (according to $<_{\mathtt{B}}$) than the median-of-medians x (the other situation is handled analogously). Prior to the "recursive" call in line 36 we have moved all elements larger than x to the front of the array, more specifically to the subarray $\mathtt{A}[b+1 \dots b+k_<]$. Then we find the largest element larger than x (using a single scan) and move it to the end the current array $\mathtt{A}[b \dots e-1]$. This element, being larger than x, is also larger than any element not passed to the "recursive" call and will be the first element larger than x encountered when scanning the array $\mathtt{A}$ starting from $\mathtt{A}[b + k_>]$ (Figure 4).

By the invariants, we know that after the "recursive call" to RestoringSubsetSelection$(\mathtt{A}, b+1, b+k_>, k-(k_<+k_=)$, the subarray $\mathtt{A}[b+1 \dots b+k_<-1]$ will be sorted accordint to $<_{\mathtt{A}}$, and we will know the values of $b+1$, $b+k_>$, and $k-(k_<+k_=)$. This enables us to retrieve the median-of-medians x from $\mathtt{A}[b]$ and (starting from $\mathtt{A}[b+k_>]$) to scan for the first element larger than x. After

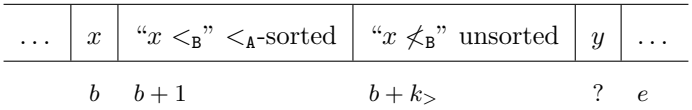

Figure 4: Restoring the $<_{\mathtt{A}}$-order after having selected the $k-(k_<+k_=)$-th element from $\mathtt{A}[b+1\ldots b+k_<-1]$. Here y is the largest element in $\mathtt{A}[b\ldots e-1]$ for which $x <_{\mathtt{B}} y$.

we have found this element at position $e-1$, we have restored to original value of e, and a single scan over $\mathtt{A}[b\ldots e-1]$ allows us to compute the values $k_<$ and $k_=$, which are needed to restore k to its original value.

Inductively, we see that the invariant holds for the initial call to the algorithm, and this implies that the algorithm selects the k-th smallest element according to $<_{\mathtt{B}}$ while maintaining the $<_{\mathtt{A}}$-order in which the elements had been given. The space requirement of this algorithm is $\mathcal{O}(\log n)$ bits, as except for a constant number of indices, only two stacks of size $\mathcal{O}(\log n)$ bits are needed. Using the analysis of the original algorithm by Blum *et al.* [2], the running time can be shown to be $\mathcal{O}(n)$, and we conclude with the following lemma:

Lemma 5 *The k-th smallest element in an array of n element can be selected in linear time spending only $\mathcal{O}(\log n)$ extra bits. Futhermore, if the set is given sorted according to some total order, this order can be restored in the same time and space complexity.*

B Orthogonal Line-Segment Intersection

We observe that if we require the divide-and-conquer algorithm to be an almost complete binary tree, the left subtree of any node that has a right subtree as well will be a complete binary tree. Consequently, during the recursion an interval $\mathtt{V}[b_v \dots e_v - 1]$ an interval processed if of length $e_v - b_v = 2^k$ for some $k \in \mathbb{N}$ if and only if $e_v \neq |V|$.

Algorithm 10 Partition the vertical segments when recursing "to the left".

Require: $\mathtt{V}[b_v \dots e_v - 1]$ is sorted according to $<_{y.\text{upper}}$.

Ensure: $\mathtt{V}[b_v \dots b_v + m]$ contains all vertical segments left of the median and is sorted according to $<_{y.\text{upper}}$.

1: Let $m := 2^{\lfloor \log_2((e_v - b_v)/2) \rfloor}$ $\{m = (e_v - b_v)/2 \iff e_v \neq |V|\}$
2: Let $i :=$ RestoringSelect$(\mathtt{V}, b_v, e_v, m,$ Select$)$.
3: Using StableSubsetSelection, move all elements less than $\mathtt{V}[i]$ to $\mathtt{V}[b_v \dots b_v + m - 1]$.
4: Push$(S,$ Left$)$.

Algorithm 11 Undo partitioning of the vertical segments after returning from a recursion "to the left".

Require: The last recursion "to the left" has ended, and we are given an array $\mathtt{V}[b_v \dots b_v + m - 1]$ that is sorted according to $<_{y.\text{upper}}$.

Ensure: $\mathtt{V}[b_v \dots e_v - 1]$ is sorted according to $<_{y.\text{upper}}$. The variables b_v and e_v are reset to their original values.

1: Pop(S).
2: **if** $b_v + 2m \leq |V|$ **then**
3: Let $e_v := b_v + 2m$.
4: **else**
5: Let $e_v := |V|$.
6: **end if**
7: UndoSubsetSelection$(\mathtt{V}, b_v, e_v, m)$

The observation that shows the correctness of the formula for restoring the value of e_v (Line 3) is that the left subtree of the current node is a complete binary tree. The height of the tree is the height of the recursion tree (which is $\lceil \log_2 |V| \rceil$) minus the depth of the current node, that is minus the depth of the "recursion" at present. The number of leaves in the left subtree, and hence the number of elements on which the recursion "to the left" took place, is then $2^{\lceil \log_2 |V| \rceil - (d+1)}$ which the number to be added to b_v in order to re-obtain the index of the split point.

The element selected and moved to $\mathtt{H}[b_h + m]$ in Line 13 is needed to distinguish horizontal segments crossing the left slab from horizontal segments crossing a slab corresponding to a recursive call higher up in the recursion tree. These segment might be stored in the cells $\mathtt{H}[e_h]$ and higher and might make

Algorithm 12 Partition the vertical segments when recursing "to the right".

Require: $\mathtt{V}[b_v \dots e_v - 1]$ is sorted according to $<_{y.\text{upper}}$.

Ensure: $\mathtt{V}[b_v \dots b_v + (e_v - b_v - m)]$ contains all vertical segments not left of the median and is sorted according to $<_{y.\text{upper}}$.

1: Let $m := 2^{\lfloor \log_2((e_v - b_v)/2) \rfloor}$ $\{m = (e_v - b_v)/2 \iff e_v \neq |V|\}$
2: Let $i :=$ RestoringSelect$(\mathtt{V}, b_v, e_v, m,$ Select$)$.
3: Using StableSubsetSelection, move all elements not less than $\mathtt{V}[i]$ to $\mathtt{V}[b_v \dots b_v + (e_v - b_v - m)]$.
4: Push$(S,$ Right$)$.

Algorithm 13 Undo the partitioning of the vertical segments after returning from a recursion "to the right".

Require: The last recursion "to the right" has ended, and we are given an array $\mathtt{V}[b_v \dots b_v + (e_v - m - b_v)]$ that is sorted according to $<_{y.\text{upper}}$.

Ensure: $\mathtt{V}[b_v \dots e_v - 1]$ is sorted according to $<_{y.\text{upper}}$. The variables b_v and e_v are reset to their original values.

1: Pop(S).
2: Let d be the number of elements on the stack S.
3: Let $e_v := b_v + 2^{\lceil \log_2 |V| \rceil - (d+1)} + (e_v - m - b_v)$.
4: UndoSubsetSelection$(\mathtt{V}, b_v, e_v, (e_v - m))$

it impossible to re-obtain the index $e_h - 1$ needed in the restoration process. If there is no segment crossing the whole slab but at least one segment not moved, we can easily re-obtain the index e_h by looking for the first segment that either is right of the current slab or completely crosses the current slab.

[2]: JV: See `StopElement1.pdf`

[3]: JV: See `StopElement2.pdf`

> At this point, maybe even earlier, we need to say that we actually are using a variant of stable subset selection (and its inverse) that skips over one particular entry (in this case $\mathtt{H}[b_h + m]$).

The case for partitioning in preparation for a recursion "to the right" is completely symmetric and hence will not be discussed here in detail.

Having these subroutines at hand, we can describe the overall algorithm for computing all intersections between the horizontal and vertical segments.

Algorithm 14 Partition the horizontal segments when recursing "to the left".

Require: $\texttt{H}[b_h \ldots e_h - 1]$ is sorted according to $<_y$. The current slab boundaries as well as the median for splitting the slab is known.

Ensure: $\texttt{H}[b_h \ldots b_h + m - 1]$ contains all horizontal segments to be passed to the recursion "to the left" sorted according to $<_y$.

1: Let m be the number of elements in $\texttt{H}[b_h \ldots e_h - 1]$ that (a) avoid the left sub-slab or (b) cross the current slab.
2: **if** $b_v + m < e_v$ **then**
3: PUSH$(T, 1)$. {At least one segment will not moved.}
4: Synchronously go back in stack T and stack S and find the most recent recursion (except for the current) where at least one segment was not moved. Let R be the type of this recursion.
5: **if** $R =$ LEFT **then**
6: Let h be the segment of those crossing the current slab with the leftmost left endpoint.
7: **else**
8: Let h be the segment of those crossing the current slab with the rightmost right endpoint.
9: **end if**
10: **if** h is undefined **then**
11: Let h be $\texttt{H}[b_h + m]$. {Don't do anything.}
12: **end if**
13: Move h to $\texttt{H}[b_h + m]$.
14: Using STABLESUBSETSELECTION, move all elements except for those that (a) avoid the left sub-slab or (b) cross the current slab to $\texttt{H}[b_h \ldots b_h + m - 1]$.
15: **else**
16: PUSH$(T, 0)$.
17: **end if**

Algorithm 15 Undo the partitioning of the horizontal segments after returning from a recursion "to the left".

Require: The last recursion "to the left" has ended, and we are given an array $\texttt{H}[b_h \ldots b_h + m - 1]$ that is sorted according to $<_y$. The current slab boundaries as well as the median for splitting the slab is known.

Ensure: $\texttt{H}[b_h \ldots e_h - 1]$ is sorted according to $<_y$. The variables b_h and e_h are reset to their original values.

1: $i := \textsc{Pop}(T)$.
2: **if** $i = 0$ **then**
3: {No partitioning needs to be reverted.}
4: **else**
5: Let $h := \texttt{H}[b_h + m]$.
6: **if** h crosses the current slab **then**
7: Synchronously go back in stack T and stack S and find the most recent recursion (except for the current) where at least one segment was not moved. Let R be the type of this recursion.
8: **if** $R = \textsc{Left}$ **then**
9: Starting at $\texttt{H}[b_h + m + 1]$, scan to find the first element that either is right of the current slab or which crosses the current slab and whose left endpoint is left of h's left endpoint.
10: **else**
11: Starting at $\texttt{H}[b_h + m + 1]$, scan to find the first element that either is right of the current slab or which crosses the current slab and whose right endpoint is right of h's right endpoint.
12: **end if**
13: **else**
14: Starting at $\texttt{H}[b_h + m + 1]$, scan to find the first element whose right endpoint is right of the right slab boundary or the first element which crosses the current slab.
15: **end if**
16: Let e_v be the index of the element just found.
17: $\textsc{UndoSubsetSelection}(\texttt{V}, b_v, e_v, b_v + m)$
18: Move h to it proper position.
19: **end if**

Algorithm 16 Solving the Orthogonal Line Segment Intersection Problem.

1: Scan $\texttt{V}[b_v \ldots e_v - 1]$ to compute the boundaries of the current slab (min/max values of x-coordinates).
2: Using StableSubsetSelection, move all segments spanning the current slab to the front of $\texttt{V}[b_v \ldots e_v - 1]$.
3: Perform a top-down sweep over $\texttt{H}[b_h \ldots e_h - 1]$ and the front of $\texttt{V}[b_v \ldots e_v - 1]$ to find all intersections.
4: Undo StableSubsetSelection on $\texttt{V}[b_v \ldots e_v - 1]$.
5: Partition "to the left" on $\texttt{V}[b_v \ldots e_v - 1]$.
6: Partition "to the left" on $\texttt{H}[b_h \ldots e_h - 1]$.
7: {"Recurse"}
8: Undo the partitioning "to the left" on $\texttt{V}[b_v \ldots e_v - 1]$.
9: Undo the partitioning "to the left" on $\texttt{H}[b_h \ldots e_h - 1]$.
10: Partition "to the right" on $\texttt{V}[b_v \ldots e_v - 1]$.
11: Partition "to the right" on $\texttt{H}[b_h \ldots e_h - 1]$.
12: {"Recurse"}
13: Undo the partitioning "to the right" on $\texttt{V}[b_v \ldots e_v - 1]$.
14: Undo the partitioning "to the right" on $\texttt{H}[b_h \ldots e_h - 1]$.