Space-Efficient Geometric Divide-and-Conquer Algorithms

Prosenjit Bose* Anil Maheshwari* Pat Morin* Jason Morrison* Michiel Smid*
Jan Vahrenhold†

July 9, 2004

Abstract

We develop a number of space-efficient tools including an approach to simulate divide-and-conquer space-efficiently, stably selecting and unselecting a subset from a sorted set, and computing the kth smallest element in one dimension from a multi-dimensional set that is sorted in another dimension. We then apply these tools to solve several geometric problems that have solutions using some form of divide-and-conquer. Specifically, we present solutions running in $\mathcal{O}(n \log n)$ time using $\mathcal{O}(1)$ extra memory given inputs of size n for the closest pair problem and the bichromatic closest pair problem. For the orthogonal line segment intersection problem, we solve the problem in $\mathcal{O}(n \log n + k)$ time using $\mathcal{O}(1)$ extra space where n is the number of horizontal and vertical line segments and k is the number of intersections.

1 Introduction

Researchers have studied space-efficient algorithms since the early 1970's. Examples include merging, (multiset) sorting, and partitioning problems; see, for example, references [9, 13, 12]. Brönnimann *et al.* [5] 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 (the number of extreme points). Recently, Chen and Chan [6] addressed the problem of computing all the intersections among a set of n line segments, giving an algorithm that runs in $\mathcal{O}((n + k) \log^2 n)$ time using $\mathcal{O}(\log^2 n)$ extra space where k is the number of intersections and an algorithm that runs in $\mathcal{O}((n + k) \log n)$ time using $\mathcal{O}(1)$ extra space but the initial input is destroyed. Brönnimann, Chan and Chen [4] developed some space efficient data structures and used them to solve a number of geometric problems such as 3-dimensional convex hull, Delaunay triangulation and nearest neighbour queries. Brönnimann and Chan [3] describe $\mathcal{O}(1)$ extra-memory algorithms for computing the convex hull of simple (open or closed) polygonal chains.

In this paper, we develop a number of space-efficient tools outlined in Section 2 that are of interest in their own right. We then apply these tools to several geometric problems that have solutions using some form of divide-and-conquer. Specifically, we address the problems of closest pairs, bichromatic closest pair and orthogonal line segment intersection.

*School of Computer Science, Carleton University, 1125 Colonel By Dr., Ottawa, Ontario, Canada, K1S 5B6. Email: {jit, anil, morin, morrison, michiel}@scs.carleton.ca. Research supported in part by NSERC.

†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 The Model

The goal is to design algorithms that use very little extra space over and above the space used for the input to the algorithm. The input is assumed to be stored in an array of size n, thereby allowing random access. The specifics of the input are outlined with each problem addressed. We assume that a constant size 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. In certain cases, the output may be much larger than the size of the input. For example, given a set of n line segments, if the goal is to output all the intersections, there may be as many as $\Omega(n^2)$. In such cases, we consider the output memory to be write-only space usable for output but cannot be used as extra storage space by the algorithm. This model has been used by Chen and Chan [6] for variable size output, space-efficient algorithms and accurately models algorithms that have output streams with write-only buffer space. Equivalently the algorithm could report the output one item at a time by calling some user-defined subroutine for each item.

The remainder of the paper is organized as follows. In Section 2, we outline the space-efficient tools that will be useful in the solution of the geometric problems addressed. In Section 3, we present an $\mathcal{O}(n \log n)$ time algorithm to solve the closest pair problem using only $\mathcal{O}(1)$ extra memory. In the following subsection, we solve the bichromatic closest pair problem in $\mathcal{O}(n \log n)$ expected time using only $\mathcal{O}(1)$ extra memory. The solution is randomized but the extra memory used is still $\mathcal{O}(1)$ in the worst case. Section 4 presents a solution to the orthogonal line segment intersection problem in $\mathcal{O}(n \log n + k)$ time using $\mathcal{O}(\log n)$ extra space where n is the number of line segments, and k is the number of intersections. Conclusions and open problems are in Section 5. The details of an $O(1)$ extra memory algorithm for orthogonal line segment intersection appear in Appendix B.

2 Space-Efficient Tools

In this section, we outline a number of algorithmic tools that will be useful in the sequel. We begin with a general scheme for implementing divide-and-conquer algorithms and then present tools for selecting subsets and finding the element of a given rank without disturbing the input array.

2.1 Space-Efficient Divide-and-Conquer

In this subsection, we describe a simple scheme for space-efficiently performing divide-and-conquer. In a standard recursive procedure, prior to each recursive call, the current state of the procedure (i.e. the state of the variables, program counter, etc.) that is invoking the recursive call is saved on a stack. The variables making up the current state are often referred to as the *activation frame* and the stack is usually called the *recursion stack*. Thus, in the standard recursive approach, the amount of extra space needed for the recursion stack is directly proportional to the height of the recursion tree times the number of local variables.

A template for recursive divide-and-conquer algorithms that partition into 2 subproblems of equal size is presented as Algorithm 1 (RECURSIVE). It operates on an array $A[0, \ldots, n-1]$. The algorithm makes calls to 4 subroutines: BASE-CODE is used to solve small instances, PRE-CODE is executed before any recursive calls, MID-CODE is executed after the first recursive call but before the second, and POST-CODE is executed after the second recursive call. Normally, the functions PRE-CODE and POST-CODE perform the divide step and the conquer step, respectively. If the functions PRE-CODE, MID-CODE and POST-CODE all run in $O(e - b)$ time then the total running time of a call to RECURSIVE$(A, 0, n)$ is $\mathcal{O}(n \log n)$.

Algorithm 1 RECURSIVE(A, b, e): Standard template recursive divide-and-conquer

```
1: if $e - b \leq 2^{h_0}$ where $s$ is the size of the recursion base. then
2:     BASE-CODE$(A, b, e)$ {Code for solving small instances}
3: else
4:     PRE-CODE$(A, b, e)$ {Computations to setup Subproblem 1 in $A[b, \ldots, \lfloor e/2 \rfloor - 1]$}
5:     RECURSIVE$(A, b, \lfloor e/2 \rfloor)$ {First recursive call}
6:     MID-CODE$(A, b, e)$ {Computations to setup Subproblem 2 in $A[\lfloor e/2 \rfloor, \ldots, e-1]$}
7:     RECURSIVE$(A, \lfloor e/2 \rfloor, e)$ {Second recursive call}
8:     POST-CODE$(A, b, e)$ {Computations to merge Subproblems 1 and 2 in $A[b, \ldots, e-1]$}
9: end if
```

To develop in-place divide-and-conquer algorithms, we must do two things: (1) We must use a stack of size $O(1)$ and (2) We must implement the functions PRE-CODE, MID-CODE and POST-CODE using only $O(1)$ extra space. Note that, if we use Algorithm 1 directly then the stack space required is $\Omega(\log n)$ since the recursion tree has depth $\Theta(\log n)$. In the remainder of this section we show how this stack space can be reduced to $O(1)$.

The main idea is to perform a stackless traversal of the binary recursion tree. Algorithms for stackless traversals of trees date back at least to Knuth [14, Section 2.3.2]. The algorithm for traversing a binary tree without a stack is quite simple; it maintains a pointer to the current node u and a state s that keeps track of whether the algorithm has already visited 0 (pre), 1 (mid) or 2 (post) of u's children. Using 4 simple rules, the algorithm then decides whether the next node visited will be the parent, the left child of u or the right child of u and also decides on the next state.

In our case, the binary tree we are traversing is implicit. For simplicity, we assume that n is a power of 2, so that all leaves of the recursion tree are at the same level. The algorithm, which is described by the pseudocode given in Algorithm 2 uses a $\lceil \log_2 n \rceil$-bit integer u to keep track of the path from the root to the current node. Here, and throughout, we use the notation u_h to denote the hth most significant bit of u. (This bit represents 2^h.) The integer u represents the path as a sequence of 0's and 1's (0 for a left turn and 1 for a right turn). The algorithm also maintains an integer h to keep track of the height (distance to the leaves) of u. In this way, the length of the path from u to the root of the tree is $\log_2 n - h$. With this representation the pair (u, h) represents a subproblem of size 2^h that begins at position u, i.e., the subarray $A[u, \ldots, u+2^h-1]$. It is an easy exercise [8, Exercise 10.4-5] to see that if n is a power of 2 then Algorithm 2 performs exactly the same calls to BASE-CODE, PRE-CODE, MID-CODE, and POST-CODE, and in the same order as Algorithm 1. Furthermore, this transformation does not increase the running time by more than a linear term.

Theorem 1 *For n a power of 2, Algorithm 2 performs the same computations as Algorithm 1 using only $O(1)$ extra space beyond what is used by the functions* BASE-CODE*,* PRE-CODE*,* MID-CODE*, and* POST-CODE*. Furthermore, if the running time of Algorithm 1 is $T(n)$ then the running time of Algorithm 2 is $T(n) + O(n)$.*

Remark: Theorem 1 only addresses the case when n is a power of 2. The general case can be handled by traversing the recursion tree for a problem of size $2^{\lceil \log_2 n \rceil}$ and simply adding checks to Algorithm 2 to make sure that it does not access any non-existent array elements.

We now have the first ingredient required for $\mathcal{O}(1)$ extra-memory recursive algorithms: a stack of size $\mathcal{O}(1)$. In the next few sections we develop some tools that help with the second ingredient required for in-place divide-and-conquer algorithms. In particular, these tools help to implement the functions PRE-CODE, MID-CODE and POST-CODE using $\mathcal{O}(1)$ extra space.

Algorithm 2 Stackless-Recursive(A): Stackless simulation of Recursive($A, 0, n$)

```
$u \leftarrow 0$ {The current node}
$h \leftarrow \log_2 n$ {The height of the current node}
$s \leftarrow$ pre {The current state}
**while** $h \leq \log_2 n$ **do**
  **if** $h = h_0$ {$u$ is a leaf} **then**
    Base-Code($A, u, u + 2^h$)
    {Next we visit the parent of $u$}
    $s \leftarrow$ mid or post (depending on whether $u_h$ is 0 or 1, respectively)
    $u_h \leftarrow 0$
    $h \leftarrow h + 1$
  **else if** $s =$ pre **then**
    Pre-Code($A, u, u + 2^h$)
    {Next we visit left child of $u$}
    $s \leftarrow$ pre
    $h \leftarrow h - 1$
    $u_h \leftarrow 0$
  **else if** $s =$ mid **then**
    Mid-Code($A, u, u + 2^h$)
    {Next we visit right child of $u$}
    $s \leftarrow$ pre
    $h \leftarrow h - 1$
    $u_h \leftarrow 1$
  **else if** $s =$ post **then**
    Post-Code($A, u, u + 2^h$)
    {Next we visit parent of $u$}
    $s \leftarrow$ mid or post (depending on whether $u_h$ is 0 or 1, respectively)
    $u_h = 0$
    $h \leftarrow h + 1$
  **end if**
**end while**
```

2.2 Sorted Subset Selection

In this subsection, we introduce a simple yet surprisingly powerful algorithm, called SubsetSelection(A, b, e, f). The algorithm provides a way of selecting a subset of the elements and moving them to the front of the array without changing their relative order. What makes this algorithm so powerful is that, if the array is initially sorted according to some total order $<$ then it is possible (in fact very easy) to undo the effects of the algorithm to restore the original sorted order. We note that stable partitioning algorithms and stable merging algorithms could achieve the same effect, but these algorithms are an order of magnitude more complicated than the code given here.

The SubsetSelection algorithm, given in pseudocode as Algorithm 3, stably moves to the front of the array all elements in $A[b, \ldots, e-1]$ for which the given $(0, 1)$-valued selection function f returns the value 1.

Algorithm 3 Algorithm SubsetSelection(A, b, e, f) for selecting a subset from a sorted array $A[b, \ldots, e-1]$.

```
Require: Array $A[b, \ldots, e-1]$ is sorted and $f$ is a $(0,1)$-valued function that can be evaluated in constant
    time.
Ensure: $A[b, \ldots, m-1]$ contains all entries for which $f$ is one, and these entries are still sorted.
 1: $i \leftarrow b$, $j \leftarrow b$ and $m \leftarrow b+1$.
 2: while $i < e$ and $j < e$ do
 3:    while $i < e$ and $f(A[i]) = 1$ do
 4:       $i \leftarrow i+1$. {Move index $i$ such that $f(A[i]) = 0$.}
 5:    end while
 6:    $j \leftarrow i+1$;
 7:    while $j < e$ and $f(A[j]) = 0$ do
 8:       $j \leftarrow j+1$. {Move index $j$ such that $f(A[j]) = 1$.}
 9:    end while
10:    if $j < e$ then
11:       swap $A[i] \leftrightarrow A[j]$.
12:    end if
13: end while
14: Return $i$.
```

Algorithm 3 clearly uses only $\mathcal{O}(1)$ extra space and runs in linear time. The correctness of the algorithm follows from the loop invariant that is maintained: $A[b, \ldots, i-1]$ stably contains all selected elements from $A[b, \ldots, e-1]$ whose rank is at most $i-b$. The effects of this algorithm can be reversed by Algorithm 4.

Algorithm 4 Algorithm UndoSubsetSelection(A, b, e, i) for restoring the total order after applying the SubsetSelection-Algorithm 3

```
Require: Array $A[b, \ldots, e-1]$ contains the result of running Algorithm 3 on array $A[b, \ldots, e-1]$ that was
    sorted
Ensure: Array $A[b, \ldots, e-1]$ is sorted
 1: $i \leftarrow i-1$ and $j \leftarrow e-1$
 2: while $i \neq j$ and $i \geq b$ do
 3:    if $A[j] < A[i]$ then
 4:       swap $A[i] \leftrightarrow A[j]$
 5:       $j \leftarrow j-1$
 6:    end if
 7:    $i \leftarrow i-1$
 8: end while
```

There is one important property to note: Algorithm 4 does not require knowledge of the selection function f, but only needs to know the three indices b, e and i. It uses only comparisons to recover the state prior to the invocation of SubsetSelection. This is a key property that is useful particularly if the selection function f is unknown or cannot be easily reconstructed. In the following subsection, we show how useful SubsetSelection and its counterpart UndoSubsetSelection are.

2.3 Selecting the k-th smallest element

In designing in-place geometric algorithms, the ability to undo operations such as selection turns out to be extremely useful. In this section we show how we can select the kth smallest element according to some total order $<_x$ in an array that is sorted according to some other total order $<_y$. Once the kth smallest element is found, the original ($<_y$) sorted order of the array is restored. Our algorithm is a modification of Hoare's Find algorithm [11].

The original Find algorithm selects a random pivot and partitions the array A into the subset of elements that are less than or equal to the pivot and the subset of elements that are greater than the pivot and then searches recursively in one of these two parts. The partitioning and undoing of the partitioning can be accomplished using Algorithms 3 and 4. However, for Algorithm 3 to work it needs to know the sizes of each of the two parts of the array (remembering the pivot element also works). Unfortunately, remembering these sizes for each recursive invocation of the algorithm produces a stack of expected size $\Theta(\log n)$. To get around this problem, we force the algorithm to always partition the current array into two parts of size *exactly* $\lfloor 3n/4 \rfloor$ and $\lceil n/4 \rceil$.

This idea is implemented as Algorithm 5 (Select). Suppose without loss of generality that $k \leq n/2$. Then we find an element p whose rank is between $n/2$ and $3n/4$. Since there are at least $\lfloor n/4 \rfloor$ such elements, with only an expected constant number of random selections, we can find such an element. Computing the rank of a given element takes $\mathcal{O}(n)$ time, thus in $\mathcal{O}(n)$ expected time, we can find p. Note that if we partition the array with p, then at least $n/4$ elements can be eliminated. The key is to eliminate *precisely* $n/4$ elements. This way, when it comes time to undo the operation, we know exactly where the partition boundaries occur. This is where the stack S comes into play. Each time through the loop, we reduce the size of the array to $\lfloor 3n/4 \rfloor$. To undo this operation, we need to record the actual remainder from the division by $3/4$. Two bits are sufficient to record this since the remainder can only be one of $\{0, 1, 2, 3\}$. For example, suppose that the current instance has an array of size 37. Since $3 \times 37/4 = 27 + 3/4$, the next instance has size 27 and we push the remainder 3 on the stack. In the undo step, if the current instance is of size 27, to recover the size of the array at invocation, we note that $(4 \times 27 + 3)/3 = 37$ which is the computation used to recover the indices required for the Undo operation.

In the final loop where the steps are *undone*, the only information we require are the three indices $0, D$ and M. We do not need to remember the particular element p that was used in the SubsetSelection step. Comparisons of Y-coordinates are sufficient to recover the original order.

Since at each iteration through the while loop, we eliminate a quarter of the elements, the number of iterations through the loop is $\mathcal{O}(\log n)$. At each iteration, two bits are pushed on the stack, so the total number of bits ever stored in the stack is $\mathcal{O}(\log n)$ which implies that only $\mathcal{O}(1)$ extra space is used. The running time of the algorithm is given by the following recurrence:

$$T(n) = T(3n/4) + \mathcal{O}(n)$$

which resolves to $\mathcal{O}(n)$.

This algorithm can be made to run in $\mathcal{O}(n)$ deterministic time. The only step where randomization is used is in the selection of the partition element p. In the deterministic version, we use techniques from the

Algorithm 5 Algorithm SELECT$(A, 0, e, k)$ Select the kth smallest element in $A[0, \ldots, e-1]$ according to the total order $<_x$.

```
Require: Array $A[0, \ldots, e-1]$ is sorted according to $<_y$
Ensure: The $k$-th smallest element according to $<_x$ is placed at $A[0]$, and $A[1, \ldots, e-1]$ is sorted by $<_y$
 1: $n \leftarrow e$. {$n$ is the current size of the array being processed. }
 2: while $n > 42$ do
 3:   if $k < n/2$ then
 4:     Pick a random element $p$ whose rank is between $n/2$ and $3n/4$
 5:     Use SubsetSelection to move all elements $<_x$ $p$ and a sufficient number $>_x$ $p$ to fill $A[0, \ldots, \lfloor 3n/4 \rfloor - 1]$
 6:     Push $4(3n/4 - \lfloor 3n/4 \rfloor)$ on stack $S$
 7:     $n \leftarrow \lfloor 3n/4 \rfloor$
 8:   else
 9:     Pick a random element $p$ whose rank is between $n/4$ and $n/2$
10:     Use SubsetSelection to move all elements $>_x$ $p$ and a sufficient number $<_x$ $p$ to fill $A[0, \ldots, \lfloor 3n/4 \rfloor - 1]$
11:     Push $4(3n/4 - \lfloor 3n/4 \rfloor)$ on stack $S$
12:     $k \leftarrow k - \lceil n/4 \rceil$
13:     $n \leftarrow \lfloor 3n/4 \rfloor$
14:   end if
15: end while
16: Find $k$-th smallest element $q$ in $A[0, \ldots, n-1]$ by brute force
17: $M \leftarrow n$
18: $R \leftarrow \text{pop}(S)$
19: $D \leftarrow (4M + R)/3$
20: while $D < e$ do
21:   UndoSubsetSelection$(A, 0, D, M)$
22:   $M \leftarrow D$.
23:   $R \leftarrow \text{pop}(S)$
24:   $D \leftarrow (4M + R)/3$
25: end while
26: Find $q$ in $A[0, \ldots, e-1]$ and move it stably to $A[0]$
```

algorithm by Blum *et al.* [2]. Instead of selecting an element at random, we select the partition element by first decomposing the array into $n/5$ groups of 5 elements and computing the median of the medians of the groups. The idea is simple and similar to the above algorithm except some of the details are quite tedious, so we outline the details in Appendix A.

3 Closest Pair Problems

3.1 Closest Pair

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

Algorithm 6 gives pseudocode for a slightly modified version of the Bentley-Shamos divide-and-conquer algorithm for finding the closest pair. The algorithm works by drawing an imaginary vertical dividing line through the median X-coordinate. The algorithm then finds the closest pair with both points to the left of this line, the closest pair with both points to the right of this line and then finds the closest pair with one point on the left and one point on the right of the line. The first two closest pairs are computed with recursive calls while the third closest pair is accomplished by a vertical plane sweep of all the points that are "close to" the dividing line. Details of this plane sweep can be found in the original paper [1].

Note that if we let the work done in Line 2 be denoted by BASE-CODE, the work done in Lines 4 and 5 be denoted by PRE-CODE, the work in Lines 7 and 8 denoted by MID-CODE and the work done in Lines 10–14 by POST-CODE then the algorithm is identical to Algorithm 1. Furthermore, all of these routines use only $O(1)$ extra space. Therefore, by Theorem 1 the entire algorithm can be implemented using only $O(1)$ extra space.

Theorem 2 *Given a set P of n points in the plane stored in an array $A[0, \ldots, n-1]$, a closest pair in P can be computed in $\mathcal{O}(n \log n)$ time using $\mathcal{O}(1)$ extra memory.*

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. We assume that there are a total of $n = r + b$ points, with $r > 0$ red points and $b > 0$ blue points. The input is given to us as two arrays $R[0, \ldots, r-1]$ and $B[0, \ldots, b-1]$.

The algorithm we use, which to the best of our knowledge is new, is similar to the Bentley-Shamos algorithm from the previous section, but has some subtle differences. The algorithm chooses a vertical line, computes the closest bichromatic pair to the left of the line, the closest bichromatic pair to the right of the line and the closest bichromatic pair with one point on the left of the line and one point on the right of the line. The difficulties arise in two places. The first is that, in general there does not exist a vertical line that partitions each of R and B into two sets of equal size. If we choose a line that partitions R into two equal sets then we obtain an uneven partition of B and vice-versa. The second difficulty is that, in order to obtain a running time of $\mathcal{O}(n \log n)$, we need a linear time algorithm for finding the closest pair between two sets separated by a line. We concentrate first on the second problem.

Algorithm 6 CLOSEST-PAIR(A, b, e): Divide-and-Conquer algorithm for finding a closest pair [1].

```
Require: All points in the input array $A$ are sorted according to $Y$-coordinate.
Ensure: The first two points in the array $A[b, \ldots, e-1]$ realize the closest pair and the remaining points
    are sorted by $Y$-coordinate.
 1: if $e - b < 16$ then
 2:     Compute a closest pair using a brute-force algorithm, stably place them at $A[b]$ and $A[b+1]$.
 3: else
 4:     Compute the median $X$-coordinate $x$ of $A[b, \ldots, e-1]$ using Algorithm 5 (SELECT) while maintaining
        the array sorted by $Y$-coordinate.
 5:     Using Algorithm 3 (SUBSETSELECTION), stably select all elements of $A[b, \ldots, e-1]$ with $X$-coordinate
        less than or equal $x$ so that they are stored in $A[b, \ldots, \lfloor e/2 \rfloor - 1]$.
 6:     CLOSEST-PAIR$(A, b, \lfloor e/2 \rfloor)$
 7:     Using Algorithm 4 (UNDOSUBSETSELECTION), restore $A[b, \ldots, e-1]$ so that it is sorted by $Y$-
        coordinate. Stably store the closest pair from the previous step in $A[b]$ and $A[b+1]$.
 8:     Using Algorithm 3 (SUBSETSELECTION), stably select all elements of $A[b, \ldots, e-1]$ with $X$-coordinate
        greater than $x$ so that they are stored in $A[\lfloor e/2 \rfloor, \ldots, e-1]$.
 9:     CLOSEST-PAIR$(A, \lfloor e/2 \rfloor, e)$
10:     Using Algorithm 4 (UNDOSUBSETSELECTION), restore $A[b, \ldots, e-1]$ so that it is sorted by $Y$-
        coordinate. Stably store the closest pair from Step 6 or the closest pair from the previous step
        (whichever is closer) at locations $A[b]$ and $A[b+1]$.
11:     Let $\delta$ be the distance between $A[b]$ and $A[b+1]$.
12:     Using Algorithm 3 (SUBSETSELECTION), extract in $Y$-sorted order the points of $A[b, \ldots, e-1]$ that
        fall within a strip of width $2\delta$ centered at the median $X$-coordinate of $A[b, \ldots, e-1]$.
13:     Scan the points in this strip to determine whether it contains a pair of points with distance smaller
        than $\delta$. Update $\delta$ and the closest pair as necessary.
14:     Using Algorithm 4 (UNDOSUBSETSELECTION), restore $A[b, \ldots, e-1]$ so that it is sorted by $Y$-
        coordinate. Stably store the closest pair at locations $A[b]$ and $A[b+1]$.
15: end if
```

3.2.1 Computing the closest vertically separated pair

In this subsection we give pseudocode in Algorithm 7 (Bichromatic-Closest-Vertically-Separated-Pair) for computing the closest bichromatic pair separated by a given vertical line ℓ. We concentrate on the case in which R and B are separated by ℓ with R to the left of ℓ and B to the right of ℓ. It is clear that if we can solve this case then we can solve the general case using two calls to this algorithm and a constant number of calls to SubsetSelection and UndoSubsetSelection.

Algorithm 7 Bichromatic-Closest-Vertically-Separated-Pair(R, B, ℓ): Bichromatic closest pair when R and B are separated by a vertical line ℓ and R is to the left of ℓ.

```
Require: All points in $R[0,\dots,r-1]$ and $B[0,\dots,b-1]$ are sorted by $Y$-coordinate
Ensure: All points in $R$ and $B$ are sorted by $Y$-coordinate, and the bichromatic closest pair is reported
1: $r \leftarrow |R|$ and $b \leftarrow |B|$.
2: while $r > 16$ and $b > 16$ do
3:   Assume $r \geq b$, otherwise reverse the roles of $R$ and $B$
4:   repeat
5:     Pick a random element $p$ from $R[0,\dots,r-1]$
6:     Find the distance $\delta$ from $p$ to the nearest element of $B[0,\dots,b-1]$
7:     Compute the left envelope of disks having radius $\delta$ centered at each of the points in $B[0,\dots,b-1]$.
8:   until at least $\lfloor r/2 \rfloor$ elements of $R[0,\dots,r-1]$ are to the left of the envelope
9:   Move exactly $\lfloor r/2 \rfloor$ elements from $R[0,\dots,r-1]$ that are not to the left of the blue envelope into
     $R[0,\lfloor r/2 \rfloor - 1]$.
10:  Push $2(r/2 - \lfloor r/2 \rfloor)$ on the stack $S$.
11:  Push 1 bit on stack $S$ when $r \geq b$ and 0 bit otherwise.
12:  Undo the envelope computation.
13:  $r \leftarrow \lfloor r/2 \rfloor$.
14: end while
15: Compute and report a bichromatic closest pair using a brute-force algorithm since one of $r$ of $b$ is at
    most 16
16: {Undo operations to restore $Y$-sorted order for both $R$ and $B$}
17: while $r < |R|$ or $b < |B|$ do
18:   $F \leftarrow$ pop($S$)
19:   $X \leftarrow$ pop($S$)
20:   if $F$ is 1 (i.e. $r \geq b$) then
21:     $r_1 \leftarrow 2r + X$
22:     UndoSubsetSelection($R, 0, r_1, r$).
23:     $r \leftarrow r_1$.
24:   else
25:     {$F$ is 0 (i.e. $r < b$)}
26:     $b_1 \leftarrow 2b + X$.
27:     UndoSubsetSelection($B, 0, b_1, b$).
28:     $b \leftarrow b_1$.
29:   end if
30: end while
```

The algorithm is similar to Clarkson and Shor's randomized algorithm for finding the diameter of a (2d or 3d) point set [7]. It selects a random red point and then computes the distance δ from this point to the nearest blue point. It then constructs the left envelope of disks centered at the blue points and having radius δ (see Figure 1). Any red point to the left of this envelope can not take part in the bichromatic closest pair (since there are no blue points within distance δ of it), so it is discarded and the algorithm continues with the undiscarded points.

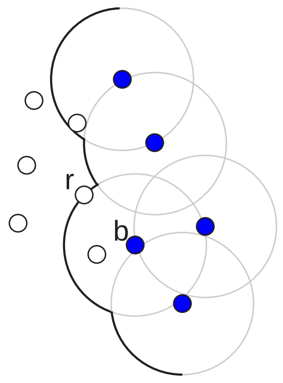

Figure 1: Left envelope of disks centered at blue points (red points are drawn as hollow disks).

Algorithm 7 runs in linear-expected time since each time through the first while loop, with constant probability, we reduce the size of R or B by a factor of 2. This algorithm can be implemented using only a constant amount of extra memory. In the first while loop, there are two steps we elaborate on: (1) how to compute and undo the left-envelope, and (2) how to identify points to the left of the blue envelope. We begin with the former.

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 and $\mathcal{O}(1)$ extra memory using an algorithm identical to Graham's scan since the points are sorted by Y-coordinate. The implementation of Graham's scan given by Brönnimann *et al.* [5] achieves this with the output being an array that contains the elements that contribute to the left envelope in the first part of the array sorted by Y-coordinate and the elements that do not contribute in the second part of the array. This algorithm is similar in several ways to the SUBSETSELECTION algorithm. In particular, it is not difficult to reverse the effects of this algorithm to restore the original $<_y$ sorted order of the elements in $\mathcal{O}(n)$ time once we are done with the left envelope. To see this, consider the pseudo-code implementation of Graham's Scan as given by Brönnimann *et al.* [5] (Algorithm 8). The effects of Algorithm 8 can be reversed by Algorithm 9.

Algorithm 8 Computing the left convex hull of a point set.

```
Require: All points in the input array $A[0 \dots n-1]$ are sorted according to $y$-coordinate.
1: for $i \leftarrow 0$ to $n-1$ do
2:     while $h \geq 2$ and $(A[h-2], A[h-1], A[i])$ does not form a right turn do
3:         $h \leftarrow h-1$.
4:     end while
5:     swap $A[i] \leftrightarrow A[h]$.
6:     $h \leftarrow h+1$.
7: end for
8: return $h$
```

We now address the problem of determining the red points that are to the left of the blue envelope. To determine whether or not a given red point p is to the left of the blue envelope, we compute the intersection of a horizontal line through p with the envelope (see Figure 1). If the intersection point is to the right of p then p is to the left of the envelope. By scanning the red points from top to bottom and simultaneously scanning the blue envelope from top to bottom, similar to merging of two sorted lists, all of these intersections can be computed in linear time.

Algorithm 9 Restoring the $<_y$-order after computing the left envelope.

```
Require: Array $A[0 \ldots h]$ contains the result of running Algorithm 8 on (the sorted) array $A[0 \ldots n-1]$.
 1: Set $q \leftarrow n-1$.
 2: while $h \neq q$ do
 3:   if $A[q] <_y A[h]$ then
 4:     swap $A[h] \leftrightarrow A[q]$.
 5:     Set $q \leftarrow q-1$.
 6:     if $A[q] <_y A[h-1]$ then
 7:       Set $h \leftarrow h-1$.
 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 $h \leftarrow h+1$.
11:     end while
12:   else
13:     Set $h \leftarrow h+1$.
14:   end if
15: end while
```

3.2.2 A divide-and-conquer algorithm for bichromatic closest pairs

The previous section gives an in-place algorithm for implementing the merge step of a divide-and-conquer algorithm for bichromatic closest pairs. All that remains is to show how this can be put into our framework for stackless divide-and-conquer algorithms. The algorithm BICHROMATIC-CLOSEST-PAIR given as pseudocode in Algorithm 10, is very similar to the closest pair algorithm of the previous section and works by divide and conquer on the red array R.

Initially it seems that this algorithm easily fits into the framework of Section 2, with the easily handled exception that it recurses first on the right subproblem and then the left subproblem. However, there is one sticking point. Algorithm 2 (STACKLESS-RECURSIVE) can only call the functions BASE-CODE, PRE-CODE, MID-CODE, and POST-CODE with information about the location of the subproblem in the array R (these are the values b_R and e_R). In particular, after the recursive calls in Lines 7 and 14, the algorithm no longer knows the indices b_B and e_B that mark the current subproblem in the blue array B, nor does the algorithm know the value of k needed to run UNDOSUBSETSELECTION. It must recompute these values.

We first observe that recomputing the value of b_B is not necessary. The algorithm always moves the current blue subproblem to the front of the blue array, so b_B never changes. Next we observe that the value of k (in line 8) or $e_B - k$ (in line 15) can be found by scanning B starting at b_B until reaching a point whose X-coordinate is smaller than x (in Line 8) or larger than x (in Line 15).

Our most difficult task is keeping track of the value of e_B. We first note that, unless we are at a node on the righmost path of the recursion tree, the subarray $B[b_B, \ldots, e_B - 1]$ contains only elements of B that are to the left of some dividing line ℓ. This dividing line is defined by the median x-coordinate in some subarray $A[u', \ldots, u' + 2^{h'}]$ of A. When viewed in terms of the recursion tree this subarray corresponds to the last node on the path from the root to the current node at which the path turns left. Furthermore, the values of u' and h' can be determined simply by looking at the binary representation of the current node. If the current node is represented by the pair (u, h) then h' is the smallest value greater than h such that $u_{h'} = 0$ and u' is obtained from u by zeroing $u_{h'}, \ldots, u_0$. Furthermore, line 13 of the algorithm explicitly stores the point with median x-coordinate in $R[u', \ldots, u' + 2^{h'}]$ at location $R[u']$ and the convention of always recursing on the right subproblem first ensures that this value is still stored in $R[u']$ when it is needed. Thus, we can determine the median of $R[u', \ldots, u' + 2^{h'}]$ in constant time once we know the value of u'. Computing u'

Algorithm 10 BICHROMATIC-CLOSEST-PAIR$(R, B, b_R, e_R, b_B, e_B)$: Compute the bichromatic closest pair in $R[b_R, \ldots, e_R - 1]$ and $B[b_B, \ldots, e_B - 1]$.

```
Require: $R$ and $B$ are sorted by $Y$-coordinate
Ensure: The bichromatic closest pair is stored $R[b_R]$ and $B[b_B]$ and the remaining elements of $R$ and $B$ are sorted by $Y$-coordinate
 1: if $e_R - b_R \leq 16$ or $e_B - b_B \leq 16$ then
 2:     Compute the bichromatic closest pair by brute force and store the closest pair in $R[b_R]$ and $B[b_B]$
 3: else
 4:     Using Algorithm 5, compute the median $X$-coordinate $x$ in $R[b_R, \ldots, e_R - 1]$
 5:     Using Algorithm 3 (SUBSETSELECTION), stably store the elements of $R[b_R, \ldots, e_R - 1]$ with $X$-coordinates greater than $x$ at locations $R[\lfloor (e_R - b_R)/2 \rfloor, \ldots, e_R - 1]$
 6:     Using Algorithm 3 (SUBSETSELECTION), stably store the elements of $B[b_B, \ldots, e_B - 1]$ with $X$-coordinates greater than $x$ at locations $B[b_B, \ldots, b_B + k - 1]$
 7:     BICHROMATIC-CLOSEST-PAIR$(R, B, \lfloor (e_R - b_R)/2 \rfloor, e_R, b_B, b_B + k)$
 8:     Compute $b_B$, $e_B$ and $k$ {We now know $b_R$ and $e_R$ but we have forgotten $b_B$, $e_B$ and $k$.}
 9:     Using Algorithm 4 (UNDOSUBSETSELECTION), restore the $Y$-sorted order of $B[b_B, \ldots, e_B - 1]$
10:     Using Algorithm 4 (UNDOSUBSETSELECTION), restore the $Y$-sorted order of $R[b_R, \ldots, e_R - 1]$
11:     Using Algorithm 3 (SUBSETSELECTION), stably store the elements of $R[b_R, \ldots, e_R - 1]$ with $X$-coordinates less than or equal to $x$ at locations $R[b_R, \ldots, \lfloor (e_R - b_R)/2 \rfloor - 1]$
12:     Using Algorithm 3 (SUBSETSELECTION), stably store the elements of $B[b_B, \ldots, e_B - 1]$ with $X$-coordinates less than or equal to $x$ at locations $B[b_B, \ldots, e_B - k - 1]$
13:     Stably store the element with median $x$-coordinate at location $R[b_R]$ and the closest bichromatic pair from step 7 at $R[b_R + \lfloor (e_R - b_R)/2 \rfloor]$ and $B[e_B - k]$
14:     BICHROMATIC-CLOSEST-PAIR$(R, B, b_R, \lfloor (e_R - b_R)/2 \rfloor, b_B, e_B - k)$
15:     Compute $b_B$, $e_B$ and $k$ {We now know $b_R$ and $e_R$ but we have again forgotten $b_B$, $e_B$ and $k$.}
16:     Using Algorithm 4 (UNDOSUBSETSELECTION), restore the $Y$-sorted order of $B[b_B, \ldots, e_B]$
17:     Using Algorithm 4 (UNDOSUBSETSELECTION), restore the $Y$-sorted order of $R[b_R, \ldots, e_R]$
18:     Stably store the bichromatic closest pair computed in Step 7 (saved in Line 13) or Step 14, whichever is closer, at $R[b_R]$ and $B[b_B]$
19:     Using Algorithm 7 (BICHROMATIC-CLOSEST-VERTICALLY-SEPARATED-PAIR) compute the closest bichromatic pair with one point to the left of $x$ and one point to the right of $x$. If this pair is closer together than $R[b_R]$ and $B[b_B]$ then stably store this pair at $R[b_R]$ and $B[b_B]$
20: end if
```

takes $\mathcal{O}(\log n)$ time.

Putting it all together, we obtain a running time recurrence of

$$T(r, b) = T(r/2, k) + T(r/2, b - k) + \mathcal{O}(r + b + \log n) \ ,$$

which solves to $\mathcal{O}(n \log n)$ in an algorithm that uses only $\mathcal{O}(1)$ extra memory.

Theorem 3 *Given sets R and B of n points in the plane, a closest bichromatic pair can be computed in $\mathcal{O}(n \log n)$ expected time using $\mathcal{O}(1)$ extra memory.*

4 Orthogonal line segment intersection

In this section, we present a space-efficient algorithm for the orthogonal line segment intersection problem. The algorithm reports all intersections between a set V of v vertical segments and a set H of h horizontal segments in $O(n \log n + k)$ time using a stack of size $O(\log n)$. Here, and throughout this section, $n = v + h$ and k is the number of intersections between H and V. In Appendix B we describe a version of this algorithm that requires only $\mathcal{O}(1)$ extra space. The main difficulty in achieving the $\mathcal{O}(1)$ extra space result is the same as in the previous section; we perform two recursive calls on pieces of V that are of equal size, but the parts of H that we recurse on are of odd sizes keeping track of these is a technical challenge.

The algorithm is a space-efficient version of the divide-and-conquer algorithm given by Goodrich *et al.* [10], which can be viewed as an implicit traversal of a segment tree [15, Section 1.2.3.1]. The algorithm, which is given as pseudocode in Algorithm 11 (OLSI) works by restricting itself to a vertical slab $[X_{min}, X_{max}]$ that contains all the vertical segments currently being considered. Initially this slab is $[-\infty, +\infty]$. The algorithm selects the median vertical segment from V, which has X-coordinate X_{med}. It then processes all segments of H that completely span $[X_{min}, X_{med}]$ or $[X_{med}, X_{max}]$ to find their intersections with the horizontal segments in $[X_{min}, X_{med}]$ or $[X_{med}, X_{max}]$, respectively. This last step is easily accomplished in linear-time if the horizontal segments are presorted by Y-coordinate and the vertical segments are presorted by the Y-coordinate of their bottom-endpoint (see Goodrich *et al.* [10] for details).

Next the algorithm recurses on the slabs $[X_{min}, X_{med}]$ and $[X_{med}, X_{max}]$ along with the vertical segments that they contain and the horizontal segments that intersect but do not completely span them. This condition on the horizontal segments ensures that each intersection is reported only once and that a horizontal segment appears in at most 2 subproblems at each level of recursion. These two conditions, combined with the fact that there are only $O(\log n)$ levels of recursion ensure a running time of $O(n \log n + k)$.

Theorem 4 *Given a set of v vertical line segments in an array V and a set of h horizontal line segments in an array H all intersections between horizontal and vertical segments can be reported in $\mathcal{O}(n \log n + k)$ time using $\mathcal{O}(\log n)$ extra space where $n = v + h$ and k is the number of intersections reported.*

As mentioned above, the main difficulty in implementing this algorithm using $O(1)$ extra space is in keeping track of the current subarray of H that is being considered. The details of how this can be done are given in Appendix B.

Algorithm 11 OLSI$(V, H, b_V, e_V, b_H, e_H)$: Orthogonal line segment intersection

```
Require: Segments in $V$ are sorted by $Y$-coordinate of the bottom vertices of the segments and segments
    in $H$ are sorted by $Y$-coordinate.
Ensure: All intersections between a segment in $H[b_H, \ldots, e_H - 1]$ and a segment in $V[b_V, \ldots, e_V - 1]$ are
    reported.
 1: if $e_V - b_V \leq 2$ then
 2:     Use a brute force algorithm to report all intersection between segments in $V[b_V, \ldots, e_V - 1]$ and
        $H[b_H, \ldots, e_H - 1]$
 3: else
 4:     Let $X_{min}$, $X_{med}$, and $X_{max}$ be the minimum, median, and maximum, respectively, $X$-coordinates of
        all segments in $V[b_V, \ldots, e_V]$.
 5:     Using Algorithm 3, (SubsetSelection), select all segments of $H[b_H, \ldots, e_H]$ that completely span
        the slab $[X_{min}, X_{max}]$ and store them stably in $H[b_H, \ldots, m_2 - 1]$
 6:     Scan $V[b_V, \ldots, e_V - 1]$ and $H[b_H, \ldots, m_2 - 1]$ to report all intersections.
 7:     Using Algorithm 4 (UndoSubsetSelection), restore $H[b_H, \ldots, e_H - 1]$
 8:     Using Algorithm 3 (SubsetSelection) select all segments of $V[b_V, \ldots, e_V - 1]$ in the slab
        $[X_{min}, X_{med}]$ and stably store them in $V[b_V, \ldots, m_1 - 1]$.
 9:     Using Algorithm 3 (SubsetSelection), select all horizontal segments that intersect the slab
        $[X_{min}, X_{med}]$ but do not span the slab $[X_{min}, X_{max}]$ and stably store them in $H[b_H, \ldots, m_2 - 1]$.
10:     OLSI$(V, H, b_V, m_1, b_H, m_2)$
11:     Using Algorithm 4 (UndoSubsetSelection), restore $H[b_H, \ldots, e_H - 1]$
12:     Using Algorithm 4 (UndoSubsetSelection), restore $V[b_V, \ldots, e_V - 1]$
13:     Using Algorithm 3 (SubsetSelection), select all segments of $V[b_V, \ldots, e_V - 1]$ in the slab
        $[X_{med}, X_{max}]$ and stably store them in $V[b_V, \ldots, m_1 - 1]$.
14:     Using Algorithm 3 (SubsetSelection), select all segments of $H[b_H, \ldots, e_H - 1]$ that intersect the
        slab $[X_{med}, X_{max}]$ but do not span the slab $[X_{min}, X_{max}]$ and stably store them in $H[b_H, \ldots, m_2 - 1]$.
15:     OLSI$(V, H, b_V, m_1, b_H, m_2)$
16:     Using Algorithm 4 (UndoSubsetSelection), restore $H[b_H, \ldots, e_H - 1]$
17:     Using Algorithm 4 (UndoSubsetSelection), restore $V[b_V, \ldots, e_V - 1]$
18: end if
```

5 Conclusions

We have given a number techniques for developing space-efficient algorithms. In particular, we presented a scheme for transforming a recursive function in standard form to one in space-efficient form reducing the amount of extra memory used to traverse the recursion tree. We provided a simple way to stably select a set of elements within an ordered array, and *undoing* the selection to restore the array to its original order. Using this, we have also given simple algorithm for selecting in linear time with constant extra space, the k-th smallest element in one dimension from an array of points in 2 or more dimensions when the points are sorted in another dimension, without disturbing the sorted order. All of these tools are applied to solve several geometric problems that have solutions using some form of divide-and-conquer. Specifically, we address the problem of nearest neighbor, bichromatic nearest neighbor and orthogonal line segment intersection.

Some of the techniques we use to achieve $\mathcal{O}(1)$ extra memory may be a little too complicated for use in practice. In particular, the algorithms for bichromatic closest pair and orthogonal line segment intersection do some extra work in order to keep track of subproblems in the blue, respectively horizontal, arrays. We note, however, if one is willing to use $\mathcal{O}(\log n)$ extra space then all of our algorithms can be implemented recursively so that they use $\mathcal{O}(\log n)$ extra space and are very implementable.

We conclude with the following open problem: Can one devise a deterministic counterpart to the randomized algorithm presented for computing the bichromatic closest pair for points separated by a vertical line? This would yield a deterministic $\mathcal{O}(1)$ extra memory algorithm for the bichromatic closest pair problem.

References

[1] Jon Louis Bentley and Michael Ian 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] Manuel Blum, Robert W. Floyd, Vaughan Ronald Pratt, Ronald Linn Rivest, and Robert Endre Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, August 1973.

[3] H. Brönnimann and T. Chan. Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. In *Proceedings of Latin American Theoretical Informatics (LATIN'04)*, volume 2976 of *Lecture Notes in Computer Science*, pages 162–171. Springer-Verlag, 2004.

[4] Hervé Brönnimann, Timothy Chan, and Eric Chen. Towards in-place geometric algorithms and data structures. In *Proceedings of ACM Symp. on Comp. Geom.*, pages 239–246, 2004.

[5] Hervé Brönnimann, John Iacono, Jyrki Katajainen, Pat Morin, Jason Morrison, and Godfried T. Toussaint. Optimal in-place planar convex hull algorithms. In Sergio Rajsbaum, editor, *Proceedings of Latin American Theoretical Informatics (LATIN 2002)*, volume 2286 of *Lecture Notes in Computer Science*, pages 494–507, Berlin, 2002. Springer.

[6] Eric Y. Chen and Timothy Moon-Yew Chan. A space-efficent algorithm for line segment intersection. In *Proceedings of the 15th Canadian Conference on Computational Geometry*, pages 68–71, 2003.

[7] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry II. *Discrete & Computational Geometry*, 4:387–421, 1989.

[8] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms.* McGraw Hill, second edition, 2001.

[9] Viliam Geffert, Jyrki Katajainen, and Tomi Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, 237(1–2):159–181, April 2000.

[10] Michael T. Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[11] C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7):321–322, 1961.

[12] Jyrki Katajainen and Tomi Pasanen. Stable minimum space partitioning in linear time. *BIT*, 32:580–585, 1992.

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

[14] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, 2nd edition, 1997.

[15] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry. An Introduction.* Springer, Berlin, second edition, 1988.

A Selecting the k-th smallest element

In this appendix, 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 $A[0 \dots n-1]$ which is sorted according to some total order $<_y$. The goal of the algorithm is, given an integer $k \in [0 \dots n-1]$, to select the k-th smallest element in A *according to some other total order* $<_x$. This algorithm will run in linear time and will require only $\mathcal{O}(1)$ extra space. Additionally, the algorithm returns the array $A[0 \dots n-1]$ in the same order as it was presented, namely sorted according to $<_y$.

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

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

The invariant is to enforce trivially for any constant-sized input. Algorithm 13 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 a stack of two-bit entries that indicate whether the current "invocation" took place from line 5, 26, or 38. This stack has a worst-case depth of $\mathcal{O}(\log n)$ and thus will (in an asymptotic sense) not increase the extra space required by this algorithm. Also, the stack S used in lines 4 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 2: The array $A[b \dots e_1 - 1]$ contains the medians in sorted $<_y$-order that had been selected from $A[b \dots e_2 - 1]$, and the remaining i elements are still untouched, hence also sorted according to $<_y$.

Algorithm 12 The algorithm RestoringSelect($A, b, e, k,$ mode) for selecting the k-th smallest element in $A[b \dots e-1]$ (if mode = Select) or the median element (if mode = Median).

```
Require: $A[b \dots e-1]$ is sorted according to $<_y$.
Ensure: $A[b \dots e-1]$ is sorted according to $<_y$. The variables $b, e$, and $k$ are reset to their original values.
 1: Subdivide $A[b \dots e-1]$ into $\lfloor (e-b)/5 \rfloor$ groups of 5 elements and (possibly) one group of size $\leq 4$.
 2: Move the medians of the first $\lfloor (e-b)/5 \rfloor$ groups to $A[b \dots b + \lfloor (e-b)/5 \rfloor - 1]$ using algorithm Sorted-
    SubsetSelection.
 3: $i \leftarrow (e-b) - \lfloor (e-b)/5 \rfloor \cdot 5$
 4: Push$(S, i)$
 5: $i_{\text{med}} \leftarrow$ RestoringSelect$(A, b, b + \lfloor (e-b)/5 \rfloor, k,$ Median$)$
 6: UndoSubsetSelection$(A, b, b + \lfloor (e-b)/5 \rfloor \cdot 5, \lfloor (e-b)/5 \rfloor)$
 7: $i \leftarrow$ Pop$(S)$
 8: $e \leftarrow b + \lfloor (e-b)/5 \rfloor \cdot 5 + i$
 9: if mode = Median then
10:     $l \leftarrow \lfloor (e-b)/2 \rfloor$
11: else
12:     $l \leftarrow k$
13: end if
14: $x \leftarrow A[i_{\text{med}}]$
15: $k_< \leftarrow |\{a \in A[b \dots e-1] \mid a < x\}|$
16: $k_= \leftarrow |\{a \in A[b \dots e-1] \mid a = x\}|$
17: $k_> \leftarrow |\{a \in A[b \dots e-1] \mid a > x\}|$
18: if $l \notin [k_< + 1, k_< + k_=]$ then
19:     if $l \leq k_<$ then
20:         if $l = k_<$ then
21:             Set $i_{\text{med}}$ to point to the largest element in $A[b \dots e-1]$ less than $x$ (according to $<_x$).
22:         else
23:             Move $x$ to $A[b]$.
24:             Using SortedSubsetSelection, move all elements in $A[b+1 \dots e-1]$ less than $x$ to $A[b+
                1 \dots b + k_<]$.
25:             Move the largest element less than $x$ (according to $<_x$) to $A[e-1]$ .
26:             $i_{\text{med}} \leftarrow$ RestoringSelect$(A, b+1, b+k_<, k,$ Select$)$
27:             Starting at $A[b+k_<]$, scan $A$ to find $e-1$ (the index of the first element $y$ for which $y <_x x := A[b]$).
28:             Move $A[e-1]$ to its proper position in $A[b+1 \dots b+k_<]$.
29:             UndoSubsetSelection$(A, b+1, e, b+k_< + 1)$.
30:             Reinsert (according to $<_y$) $x := A[b]$ into $A[b \dots e-1]$ maintaining $i_{\text{med}}$.
31:         end if
32:     else
33:         (...)

45:     end if
46: end if
47: Return $i_{\text{med}}$.
```

Algorithm 12 Algorithm RESTORINGSELECT$(A, b, e, k, \text{mode})$ (contd.)

```
18: if $l \notin [k_< + 1, k_< + k_=]$ then
19:   if $l \leq k_<$ then
20:     $(\dots)$

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

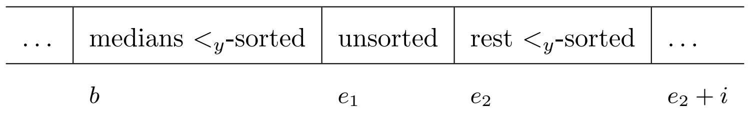

Figure 2: Restoring the $<_y$-order after having computed the median of the $\lfloor (b-e)/5 \rfloor$ medians. Here $e_1 := b + \lfloor (b-e)/5 \rfloor$ and $e_2 := b + \lfloor (b-e)/5 \rfloor \cdot 5$.

As a consequence, we can first undo the effects of SORTEDSUBSETSELECTION on $A[b \dots e_2 - 1]$, hence restoring it to sorted $<_y$-order and finally adjust e to point to $e_2 + i$. This implies that b and e are known and $A[b \dots e-1]$ is sorted according to $<_y$. 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 26 and 38), we do not know how many elements are passed to the "recursive" call. 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 to the recursive call. Let us consider the situation where the k-th element to be selected is larger (according to $<_x$) than the median-of-medians x (the other situation is handled analogously). Prior to the "recursive" call in line 38 we have moved all elements larger than x to the front of the array, more specifically to the subarray $A[b+1 \dots b+k_>]$. Then we find the largest element larger than x (using a single scan) and move it to the end of the current array $A[b \dots e-1]$. 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 A starting from $A[b+k_>]$ (Figure 3).

By the invariant, we know that after the "recursive call" to RESTORINGSUBSETSELECTION$(A, b+1, b+k_>, k-(k_< + k_=)$, the subarray $A[b+1 \dots b+k_> - 1]$ will be sorted according to $<_y$, 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 $A[b]$

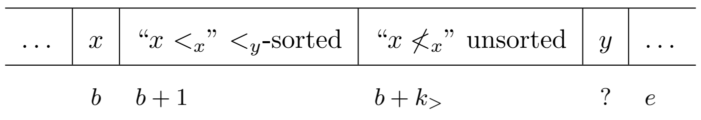

Figure 3: Restoring the $<_y$-order after having selected the $k-(k_< + k_=)$-th element from $A[b+1 \ldots b+k_> -1]$. Here y is the largest element in $A[b \ldots e-1]$ for which $x <_x y$.

and (starting from $A[b+k_>]$) to scan for the first element larger than x. After we have found this element at position $e-1$, we have restored to original value of e, and a single scan over $A[b \ldots e-1]$ 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 $<_x$ while maintaining the $<_y$-order in which the elements had been given. The space requirement of this algorithm is $\mathcal{O}(\log n)$ bits, because besides 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 theorem:

Theorem 5 *The k-th smallest element in an array of n element can be selected in linear time using $\mathcal{O}(1)$ extra space. Furthermore, 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

In this appendix, we describe an algorithm that solves the orthogonal line segment intersection problem in $\mathcal{O}(n \log n + k)$ time using $\mathcal{O}(1)$ extra space, where n is the total number of line segments and k is the total number of intersections. We assume that the input is given in the form of two arrays $H[0 \ldots n_h - 1]$ and $V[0 \ldots n_v - 1]$ of horizontal and vertical line segments, respectively, where $n = n_h + n_v$.

We will describe the algorithm as a recursive algorithm. Since it follows the general framework of Section 2, however, we can make it space-efficient so that it uses only $\mathcal{O}(1)$ extra space.

Consider a subarray $V[0 \ldots e_v - 1]$, and let $m := 2^{\lfloor \log_2((e_v)/2) \rfloor}$. Let i be the index such that the X-coordinate of $V[i]$ is the m-th smallest among all X-coordinates of the line segments in this subarray. Our algorithm will use $V[i]$ to first partition the subarray into a subarray of length m (we call this "partitioning the left") and then into a subarray of length $e_v - m$ (we call this "partitioning the right"). These partitioning algorithms are given as Algorithms 13 and 15. Both of them can be undone, see Algorithms 14 and 16.

The observation that shows the correctness of the formula for restoring the value of e_v (Line 3 in Algorithm 16) 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. The number of leaves in the left subtree, and hence the number of elements on which the first recursive call took place, is then $2^{\lceil \log_2 |V| \rceil - (d+1)}$, which is also the index of the split point.

After a subarray $V[0 \ldots e_v - 1]$ has been "partitioned to the left", the vertical slab spanned by $V[0 \ldots e_v - 1]$ (this is the *current slab*) has been partitioned into a *left sub-slab* and a *right sub-slab*. At this moment, we use these sub-slabs to partition the corresponding subarray $H[0 \ldots e_h - 1]$ of horizontal line segments. To be more precise, in "partitioning the left" (see Algorithm 17), the initial part of the subarray of H contains

Algorithm 13 PreVerticalPartition(V,e_v) Partition the vertical segments before the first recursive call (Partitioning the left)

```
Require: $V[0 \dots e_v - 1]$ is sorted according to $<_{y.\text{upper}}$, i.e., $Y$-coordinates of the upper endpoints.
Ensure: That $e_v := m$ and the resulting array $V[0 \dots e_v - 1]$ contains all vertical segments not to the right
    of the $X$-median, and is sorted according to $<_{y.\text{upper}}$.
1: $m \leftarrow 2^{\lfloor \log_2((e_v)/2) \rfloor}$
2: $i \leftarrow$ RestoringSelect($V, 0, e_v, m$, Select).
3: Using SortedSubsetSelection, move all elements less than or equal to $V[i]$ to $V[0 \dots m - 1]$.
4: Push($S$, Left).
5: $e_v \leftarrow m$
6: Return $e_v$
```

Algorithm 14 UndoPreVerticalPartition(V,e_v) Undo the pre-partitioning of the vertical segments after returning from the first recursive call.

```
Require: The first recursion has ended, and we are given an array $V[0 \dots e_v - 1]$ that is sorted according
    to $<_{y.\text{upper}}$.
Ensure: The variable $e_v$ has been reset to its original value and the resulting array $V[0 \dots e_v - 1]$ is sorted
    according to $<_{y.\text{upper}}$.
1: Pop($S$).
2: if $2e_v \leq |V|$ then
3:     $e_v^{orig} \leftarrow 2e_v$.
4: else
5:     $e_v^{orig} \leftarrow |V|$.
6: end if
7: UndoSubsetSelection($V, 0, e_v^{orig}, e_v$)
8: $e_v \leftarrow e_v^{orig}$
9: Return $e_v$
```

Algorithm 15 MidVerticalPartition(V,e_v) Partition the vertical segments before the second recursive call.

```
Require: $V[0 \dots e_v - 1]$ is sorted according to $<_{y.\text{upper}}$.
Ensure: That $e_v := e_v - m$ and that the resulting array $V[0 \dots e_v]$ contains all vertical segments to the
    right of the $X$-median and is sorted according to $<_{y.\text{upper}}$.
1: $m \leftarrow 2^{\lfloor \log_2(e_v/2) \rfloor}$
2: $i \leftarrow$ RestoringSelect($V, 0, e_v, m$, Select).
3: Using SortedSubsetSelection, move all elements larger than $V[i]$ to $V[0 \dots e_v - m - 1]$.
4: Push($S$, Right).
5: $e_v \leftarrow e_v - m$
6: Return $e_v$
```

Algorithm 16 UndoMidVerticalPartition(V,e_v) Undo the partitioning of the vertical segments after returning from the second recursive call

```
Require: The last recursive call has ended, and we are given an array $V[0 \dots e_v]$ that is sorted according to $<_{y.\text{upper}}$.
Ensure: The variable $e_v$ has been reset to its original value and the resulting array $V[0 \dots e_v - 1]$ is sorted according to $<_{y.\text{upper}}$.
 1: Pop($S$).
 2: Let $d$ be the number of elements on the stack $S$.
 3: $e_v^{orig} \leftarrow 2^{\lceil \log_2 |V| \rceil - (d+1)} + e_v$.
 4: UndoSubsetSelection($V, 0, e_v^{orig}, e_v$)
 5: $e_v \leftarrow e_v^{orig}$
 6: Return $e_v$
```

Algorithm 17 PreHorizontalPartition(H,e_h) Partition the horizontal segments before the first recursive call.

```
Require: $H[0 \dots e_h - 1]$ is sorted according to $<_y$. The current slab boundaries as well as the median for splitting the slab are known.
Ensure: That $e_h := m$ and the resulting array $H[0 \dots e_h - 1]$ contains all horizontal segments to be passed to the first recursion sorted according to $<_y$.
 1: Let $m$ be the number of elements in $H[0 \dots e_h - 1]$ that intersect the left sub-slab and do not cross the current slab.
 2: if $m < e_h$ then
 3:     Push($T, 1$). {At least one segment will not move.}
 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 $R =$ Right 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 $H[m]$. {Don't do anything.}
12:     end if
13:     Move $h$ to $H[m]$.
14:     Using SortedSubsetSelection, move all elements except for those that (a) avoid the left sub-slab or (b) cross the current slab to $H[0 \dots m - 1]$.
15: else
16:     Push($T, 0$).
17: end if
18: $e_h \leftarrow m$
19: Return $e_h$
```

Algorithm 18 UndoPreHorizontalPartition(H,e_h) Undo the partitioning of the horizontal segments after returning from the first recursive call.

Require: The first recursive call has ended, and we are given an array $H[0 \dots e_h - 1]$ that is sorted according to $<_y$. The current slab boundaries as well as the median for splitting the slab is known.

Ensure: The variable e_h have been reset to its original value and the resulting array $H[0 \dots e_h - 1]$ is sorted according to $<_y$.

```
1: $i \leftarrow$ Pop($T$).
2: if $i = 0$ then
3:   {No partitioning needs to be reverted.}
4: else
5:   $h \leftarrow H[e_h]$.
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 $R =$ Right then
9:       Starting at $H[e_h + 1]$, 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 $H[e_h + 1]$, 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 $H[e_h + 1]$, 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 $e_h^{orig}$ be the index of the element just found.
17:  UndoSubsetSelection($H, 0, e_h^{orig}, e_h$)
18:  Move $h$ to its proper position.
19:  $e_h \leftarrow e_h^{orig}$
20:  Return $e_h$
21: end if
```

all m horizontal line segments in the subarray that intersect the left sub-slab and do not cross the current slab. A problem arises when we want to undo this partitioning (in Algorithm 18): At that moment, we do not know the original value of e_h. The solution is to store a "special" horizontal segment (*viz.* the segment h in Line 13 of Algorithm 17) in $H[m]$.

This segment is used to distinguish horizontal segments crossing the left sub-slab from horizontal segments crossing a slab corresponding to a recursive call higher up in the recursion tree. These segments may be stored in cells $H[m]$ and higher and may make it impossible to re-obtain the original index e_h that is needed in the restoration process. If there is no segment crossing the current slab, but at least one segment did not move, we can easily re-obtain the original index e_h by searching for the first segment that is either to the right of the current slab or completely crosses the current slab.

Because of the special role of the horizontal line segment h, we use a variant of SortedSubsetSelection in Algorithm 17 and a variant of UndoSubsetSelection in Algorithm 18. These variants skip over the line segment h.

We have only described how to partition the subarray $H[0 \ldots e_h - 1]$ "to the left". In a completely symmetric way, this subarray can be partitioned "to the right".

Having these subroutines at hand, the algorithm solving the orthogonal line segment intersection problem is given as Algorithm 19.

Algorithm 19 ImprovedOLSI(V,e_v,H,e_h) Solving the Orthogonal Line Segment Intersection Problem.

```
1: Scan $V[0 \ldots e_v - 1]$ to compute the boundaries of the current slab (min/max values of $X$-coordinates).
2: Using SortedSubsetSelection, move all horizontal segments spanning the current slab to $H[0 \ldots \ell - 1]$.
3: Perform a top-down sweep over $H[0 \ldots \ell - 1]$ and $V[0 \ldots e_v - 1]$ to find all intersections.
4: UndoSubsetSelection on $H[0 \ldots e_h - 1]$.
5: $e_v \leftarrow$ PreVerticalPartition($V$,$e_v$)
6: $e_h \leftarrow$ PreHorizontalPartition($H$,$e_h$)
7: ImprovedOLSI($V$,$e_v$,$H$,$e_h$)
8: $e_v \leftarrow$ UndoPreVerticalPartition($V$,$e_v$)
9: UndoPreHorizontalPartition($H$,$e_h$,$m_h$)
10: $e_v \leftarrow$ MidVerticalPartition($V$,$e_v$)
11: $e_h \leftarrow$ MidHorizontalPartition($H$,$e_h$)
12: ImprovedOLSI($V$,$e_v$,$H$,$e_h$)
13: $e_v \leftarrow$ UndoMidVerticalPartition($V$,$e_v$)
14: $e_h \leftarrow$ UndoMidHorizontalPartition($H$,$e_h$)
```

Theorem 6 *Given a set of n horizontal and vertical line segments, all k intersections among them can be reported in $\mathcal{O}(n \log n + k)$ time using $\mathcal{O}(1)$ extra space.*