Programming Techniques

R. Rivest Editor

# Multidimensional Divide-and-Conquer

Jon Louis Bentley
Carnegie-Mellon University

**Most results in the field of algorithm design are single algorithms that solve single problems. In this paper we discuss *multidimensional divide-and-conquer*, an algorithmic *paradigm* that can be instantiated in many different ways to yield a number of algorithms and data structures for multidimensional problems. We use this paradigm to give best-known solutions to such problems as the ECDF, maxima, range searching, closest pair, and all nearest neighbor problems. The contributions of the paper are on two levels. On the first level are the particular algorithms and data structures given by applying the paradigm. On the second level is the more novel contribution of this paper: a detailed study of an algorithmic paradigm that is specific enough to be described precisely yet general enough to solve a wide variety of problems.**

**Key Words and Phrases: analysis of algorithms, data structures, computational geometry, multidimensional searching problems, algorithmic paradigms, range searching, maxima problems, empirical cumulative distribution functions, closest-point problems**

**CR Categories: 3.73, 3.74, 5.25, 5.31**

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This research was supported in part by the Office of Naval Research under contract N00014-76-C-0370 and in part by the National Science Foundation under a Graduate Fellowship.

Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

© 1980 ACM 0001-0782/80/0400-0214 $00.75.

## 1. Introduction

The young field of algorithm design and analysis has made many contributions to computer science of both theoretical and practical significance. One can cite a number of properly designed algorithms that save users thousands of dollars per month when compared to naive algorithms for the same task (sorting and Fourier transforms are examples of such tasks). On the more theoretical side, algorithm design has shown us a number of counterintuitive results that are fascinating from a purely mathematical viewpoint (for instance, there is a faster way to multiply two matrices than the standard "high school" algorithm). In one important sense, however, the study of algorithms is rather unsatisfying—the field consists primarily of a scattered collection of results, without much underlying theory.

Recent research has begun laying the groundwork for a theory of algorithm design by identifying certain algorithmic methods (or paradigms) that are used in the solution of a wide variety of problems. Aho et al. [1, ch. 2] describe a few such basic paradigms, and Weide [27] discusses a number of important analysis techniques in algorithm design. Almost all of the algorithmic paradigms discussed to date are at one of two extremes, however: Either they are so general that they cannot be discussed precisely, or they are so specific that they are useful in solving only one or two problems. In this paper we examine a more "middle of the road" paradigm that can be precisely specified and yet can also be used to solve many problems in its domain of applicability. We call this paradigm *multidimensional divide-and-conquer.*

Multidimensional divide-and-conquer is applicable to problems dealing with collections of objects in a multidimensional space. In this paper we concentrate on problems dealing with $N$ points in $k$-dimensional space. In a geometric setting these points might represent $N$ cities in the plane (2-space) or $N$ airplanes in 3-space. Statisticians often view multivariate data with $k$ variables measured on $N$ samples as $N$ points in $k$-space. Yet another interpretation is used by researchers in database systems who view $N$ records each containing $k$ keys as points in a multidimensional space. An alternative formalism views the points as $N$ $k$-vectors; in this paper we use the point formalism, which aids our geometric intuition. The motivating applications for the problems discussed later are phrased in these geometric terms.

Multidimensional divide-and-conquer is a single algorithmic paradigm that can be used to solve many particular problems. It can be described roughly as follows: *to solve a problem of $N$ points in $k$-space, first recursively solve two problems each of $N/2$ points in $k$-space, and then recursively solve one problem of $N$ points in $(k$-1)-dimensional space.* In this paper we study a number of different algorithms and see how each can be viewed as an instance of multidimensional divide-and-conquer. There are three distinct benefits resulting from such a study. First, this coherent presentation enables

descriptions of the algorithms to be communicated more easily. Second, by studying the algorithms as a group, advances made in one algorithm can be transferred to others in the group. Third, once the paradigm is understood, it can be used as a tool with which to attack unsolved research problems. Even another benefit might ultimately result from this study and others like it: a theory of "concrete computational complexity" which explains why (and how) some problems can be solved quickly and why others cannot.

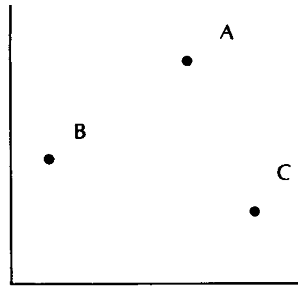
Much previous work has been done on the problems to be discussed. Since most of the work applies to only one problem, we mention that work when discussing the particular problem. Two pieces of work, however, are globally applicable and are therefore mentioned (only) here. Dobkin and Lipton [11] describe a method for multidimensional searching that is radically different from one that we study. Although their method yields search times somewhat faster than those we discuss, the preprocessing and storage costs of their algorithms are prohibitive for practical applications. Shamos [25, 26] has thoroughly investigated a large number of computational problems in plane geometry and has achieved many fascinating results.

The problems discussed in this paper provide an interesting insight into the relation of theory and practice in algorithm design. On the practical side, the previous best-known algorithms for many of the problems we discuss have running time proportional to $N^2$ (where $N$ is the number of points). The algorithms discussed in this paper have running time proportional to $N \lg N$ (at least for low dimensional spaces).[1] To make this abstract difference more concrete we note that if the two algorithms were used to process sets of 1 million points on a 1 million-instruction-per-second computer, then the $N^2$ algorithm would take over 11 days, while the $N \lg N$ algorithm would require only 20 seconds! On the theoretical side many of the algorithms we discuss can be proved to be the best possible for solving their respective problems, and this allows us to specify precisely the computational complexity of those problems. These problems also show us some interesting interactions between theory and practice: Although some of the theoretically elegant algorithms of Sections 2 and 3 are not suitable for implementation, they suggest certain heuristic algorithms which are currently implemented in several software packages.

This paper is more concerned with expressing the important concepts of multidimensional divide-and-conquer than scrupulously examining the details of particular algorithms. For this reason we gloss over many (important) details of the algorithms we discuss; the interested reader is referred to papers containing these details. In Section 2 we examine three problems centered around the concept of point *domination*, and we develop multidimensional divide-and-conquer algorithms for solving those problems. In Section 3 we focus on problems defined by point *closeness*. These two sections constitute the main part of our discussion of multidimensional divide-and-conquer. In Section 4 we survey additional work that has been done, and we then view the paradigm in retrospect in Section 5.

[1] We will use lg as an abbreviation for $\log_2$ and $\lg^k N$ as an abbreviation for $(\lg N)^k$.

Fig. 1. Point *A* dominates point *B*.




## 2. Domination Problems

In this section we investigate three problems defined in terms of point *domination*. We write $A_i$ for the $i$th coordinate of point $A$ and say that point $A$ dominates point $B$ if and only if $A_i > B_i$ for all $i$, $1 \leq i \leq k$. If neither point $A$ dominates point $C$ nor point $C$ dominates point $A$, then $A$ and $C$ are said to be *incomparable*. It is clear from these definitions that the dominance relation defines a partial ordering on any $k$-dimensional point set. These concepts are illustrated for the case $k = 2$ in Figure 1. The point $A$ dominates the point $B$, and both of the pairs $A$, $C$ and $B$, $C$ are incomparable.

In Section 2.1 we investigate the empirical cumulative distribution function, which asks *how many* points a given point dominates. In Section 2.2 we study the related question of *whether* a given point is dominated. In both of these sections we discuss two distinct but related problems. In an *all-points* problem we are asked to calculate something about every point in a set (How many points does it dominate? Is it dominated?). In a *searching* problem we must organize the data into some structure such that future queries (How many points does this point dominate? Is this point dominated?) may be answered quickly. In Section 2.3 we examine a searching problem phrased in terms of domination that has no all-points analog.

### 2.1 Empirical Cumulative Distribution Functions

Given a set $S$ of $N$ points we define the *rank* of point $x$ to be the number of points in $S$ dominated by $x$. Figure 2 shows a point set with the rank of each point written near that point. In statistics the empirical cumulative distribution function (ECDF) for a sample set $S$ of $N$ elements, evaluated at point $x$, is just $\text{rank}(x)/N$. This quantity is the empirical analog of the population cumulative distribution function. Because of the intimate

relation between rank and ECDF, we often write ECDF for rank. With this notation we can state two important computational problems.

(1) *All-Points ECDF.* Given a set $S$ of $N$ points in $k$-space, compute the rank of each point in the set.
(2) *ECDF Searching.* Given a set $S$, organize it into a data structure such that queries of the form "what is the rank of point $x$" can be answered quickly (where $x$ is not necessarily an element of $S$).

The ECDF is often required in statistical applications because it provides a good estimate of an underlying distribution, given only a set of points chosen randomly from that distribution. A common problem in statistics is hypothesis testing of the following form: Given two point sets, were they drawn from the same underlying distribution? Many important multivariate tests require computing the all-points ECDF problem to answer this question; these include the Hoeffding, multivariate Kolmogorov–Smirnov, and multivariate Cramer–Von Mises tests. The solution to the ECDF searching problem is required for certain approaches to density estimation, which asks for an estimate of the underlying probability density function given a sample. These and other applications of ECDF problems in statistics are described by Bentley and Shamos [9], which is the source of the algorithms discussed.

In this section we first devote our attention to the all-points ECDF problem in Section 2.1.1 and then solve the ECDF searching problem by analogy with the all-points solution in Section 2.1.2. Our strategy in both sections is to examine solutions to the problem in increasingly higher dimensions, starting with the one-dimensional, or linear, problem.

**2.1.1 The all-points ECDF problem.** In one dimension the rank of a point $x$ is just the number of points in the set less than $x$, so the all-points ECDF problem can be solved by sorting. After arranging the points in increasing order we assign the first point the rank 0, the second point rank 1, and so on. It is obvious that given the ranks we could produce such a sorted list, so we know that the complexity of the one-dimensional all-points ECDF problem is exactly the same as sorting, which is well known to be $O(N \lg N)$. Thus we have developed an optimal algorithm for the one-dimensional case.[2] The two-dimensional case is not quite as easy, however. To solve this problem we apply the multidimensional divide-and-conquer technique instantiated to two dimensions: To solve a problem of $N$ points in the plane, solve two problems of $N/2$ points each in the plane and then solve one problem of $N$ points on the line.

Fig. 2. With each point is associated its rank.

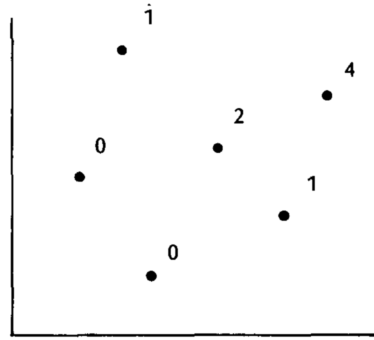
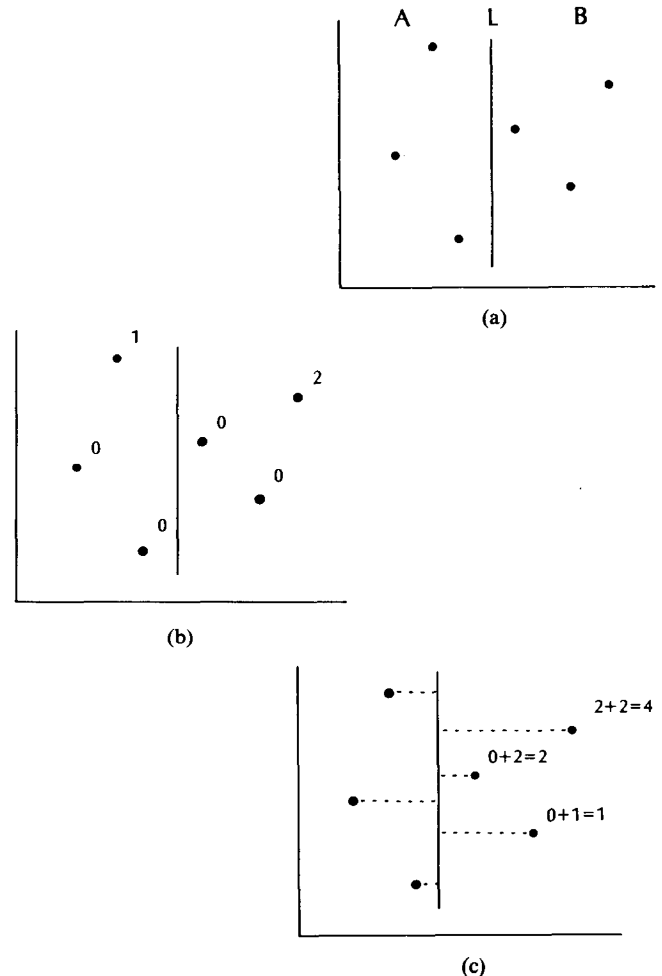



Fig. 3. Operation of Algorithm ECDF2.




Our planar ECDF algorithm operates as follows. The first step is to choose some vertical line $L$ dividing the point set $S$ into two subsets $A$ and $B$, each containing $N/2$ points.[3] This step is illustrated in Figure 3(a). The second step of our algorithm calculates for each point in $A$ its rank among the points in $A$, and likewise the rank of each point in $B$ among the points of $B$. The result of

[2] If we apply the multidimensional divide-and-conquer strategy to the one-dimensional problem, then we achieve a sorting algorithm similar to quicksort (but that always partitions around the median of the set). We do not discuss that algorithm here.

[3] To avoid needless detail we make certain assumptions such as that $N$ is even and that no pair of points share $x$- or $y$-coordinates. To otherwise confront such detail is not particularly illuminating.

this is depicted in Figure 3(b). We now make an important observation that allows us to combine these subsolutions efficiently to form a solution to the original problem. Since every point in $A$ has an $x$-value less than every point in $B$, two facts hold: First, no point in $A$ dominates any point in $B$, and second, a point $b$ in $B$ dominates point $a$ in $A$ iff the $y$-value of $b$ is greater than the $y$-value of $a$. By the first fact we know that the ranks we computed for $A$ are the correct final ranks. We are still faced with the reduced problem of calculating for every point in $B$ the number of points it dominates in $A$ (which we add to the number of $B$'s it dominates to get the final answer). To solve this reduced problem we use the second fact. If we project all the points in $S$ onto the line $L$ (as depicted in Figure 3(c)), then we can solve the reduced problem by scanning up the line $L$, keeping track of how many $A$s we have seen, and add that number to the partial rank of each point in $B$ as we pass that point. This counts the number of points in $A$ with smaller $y$-values, which are exactly the points it dominates. We implement this solution algorithmically by sorting the $A$s and $B$s together and then scanning the sorted list.

Having described the algorithm informally, we are now ready to give it a more precise description as Algorithm ECDF2. Algorithm ECDF2 is a recursive algorithm which is given as input a set $S$ of $N$ points in the plane and returns as its output the rank of each point.

**Algorithm ECDF2**

1. [Division Step.] If $S$ contains just one element then return its rank as 0; otherwise proceed.[4] Choose a cut line $L$ perpendicular to the $x$-axis such that $N/2$ points of $S$ have $x$-value less than $L$'s (call this set of points $A$) and the remainder have greater $x$-value (call this set $B$). Note that $L$ is a median $x$-value of the set.
2. [Recursive Step.] Recursively call ECDF2($A$) and ECDF2($B$). After this step we know the true ECDF of all points in $A$.
3. [Marriage Step.] We must now find for each point in $B$ the number of points in $A$ it dominates (i.e., that have lesser $y$-value) and add this number to its partial ECDF. To do this, pool the points of $A$ and $B$ (remembering their type) and sort them by $y$-value. Scan through this sorted list in increasing $y$-value, keeping track in ACOUNT of the number of $A$s so far observed. Each time a $B$ is observed, add the current value of ACOUNT to its partial ECDF.

That Algorithm ECDF2 correctly computes the rank of each point in $S$ can be established by induction, using the two facts mentioned above. We also use induction to analyze its running time on a random access computer by setting up a recurrence relation describing the running time on $N$ points, say $T(N)$, and then solving that recurrence. To set up the recurrence we must count how many operations each step of the algorithm requires. Step 1 can be solved by a fast median algorithm; we can use the algorithm of Blum et al. [10] to accomplish this step in $O(N)$ operations. Because step 2 solves two problems of size $N/2$, its cost will be $2T(N/2)$, by induction. The sort of step 3 requires $O(N \lg N)$ time, and the scan requires linear time, so the total cost of step 3 is $O(N \lg N)$. Adding the costs of the three steps we find that the total cost of the algorithm is

$$\begin{aligned} T(N) &= O(N) + 2T(N/2) + O(N \lg N) \\ &= 2T(N/2) + O(N \lg N). \end{aligned}$$

This recurrence[5] has solution

$$T(N) = O(N \lg^2 N)$$

so we know that the running time of Algorithm ECDF2 is $O(N \lg^2 N)$.

We can make an observation that will allow us to speed up many multidimensional divide-and-conquer algorithms. In looking carefully at the analysis of Algorithm ECDF2 we see that the running time is dominated by the sort of step 3. To remove this cost we can sort the $N$ points of $S$ once by $y$-coordinate before any invocation of ECDF2, at a once-for-all cost of $O(N \lg N)$. After this we can achieve the effect of sorting (without the cost) by being very careful to maintain "sortedness-by-$y$" when dividing into sets $A$ and $B$ during step 1. After this modification the recurrence describing the modified algorithm becomes

$$T(N) = 2T(N/2) + O(N)$$

which has solution

$$T(N) = O(N \lg N).$$

This technique is known as *presorting* and has very broad applicability; we see that it allows us to remove a factor of $O(\lg N)$ from the running time of many algorithms.

We now turn our attention to developing an algorithm for solving the ECDF problem for $N$ points in 3-space. The multidimensional divide-and-conquer method we use is analogous to the method used by the previous algorithm: To solve a problem of $N$ points in 3-space we solve two problems of $N/2$ points in 3-space and then one problem of $N$ points in 2-space. The first step of our algorithm chooses a cut plane $P$ perpendicular to the $x$-axis dividing $S$ into sets $A$ and $B$ of $N/2$ points each. Figure 4 illustrates this division. The second step then (recursively) counts for each point in $A$ the number of points in $A$ it dominates, and likewise for $B$. By reasoning analogous to that for the planar case we can see that since no point in $A$ dominates any point in $B$, the final ranks of $A$ are exactly the ranks already computed. By the same reasoning we know that a point $b$ in $B$ dominates point $a$ in $A$ iff $b$ dominates $a$ in their projection on $P$, the $(y, z)$ plane. The third step of our algorithm therefore projects all points onto plane $P$ (which merely involves ignoring the $x$-coordinates) and then counts for each $B$-point the number of $A$-points it dominates. This reduced problem, however, is just a

[4] All the algorithms we will see have this test for small input size; we usually omit it for brevity.

[5] To be precise we should also define the "boundary condition" of the recurrence, which is in this case $T(1) = c$, for some constant $c$. Since all the recurrences we will see in this paper have the same boundary, we delete it for brevity. The particular value of the constant does not affect the asymptotic growth rate of the functions.

slightly modified version of the planar ECDF problem, which can be solved in $O(N \lg N)$ time.[6] The recurrence describing our three-dimensional algorithm is then

$$T(N) = 2T(N/2) + O(N \lg N)$$

which, as we saw previously, has solution $T(N) = O(N \lg^2 N)$.

The technique that we just used to solve the two- and three-dimensional problems can be extended to solve the general problem in $k$-space. The algorithm consists of three steps: divide into $A$ and $B$, solve the subproblems recursively, and then patch up the partial answers in $B$ by counting for each point in $B$ the number of $A$s it dominates (a $(k-1)$-dimensional problem). The $(k-1)$-dimensional subproblem can be solved by a "bookkeeping" modification to the $(k-1)$-dimensional ECDF algorithm. We can describe the algorithm formally as Algorithm ECDF$k$.

**Algorithm ECDF$k$.**

1. Choose a $(k-1)$-dimensional cut plane $P$ dividing $S$ into two subsets $A$ and $B$, each of $N/2$ points.
2. Recursively call ECDF$k(A)$ and ECDF$k(B)$. After this we know the true ECDF of all points in $A$.
3. [For each $B$ find the number of $A$s it dominates.] Project the points of $S$ onto $P$, noting for each whether it was an $A$ or a $B$. Now solve the reduced problem using a modified ECDF$(k-1)$ algorithm and add the calculated values to the partial ranks of $B$.

To analyze the runtime of Algorithm ECDF$k$ we denote its running time on a set of $N$ points in $k$-space by $T(N, k)$. For any fixed value of $k$ greater than 2, step 1 can be accomplished in $O(N)$ time. Step 2 requires $2T(N/2, k)$ time, and the recursive call of step 3 requires $T(N, k-1)$ time. Combining these we have the recurrence

$$T(N, k) = O(N) + 2T(N/2, k) + T(N, k-1).$$

We can use as a basis for induction on $k$ the fact that

$$T(N, 2) = O(N \lg N),$$

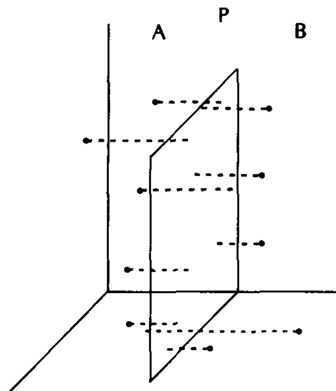
as shown previously, and this establishes that

$$T(N, k) = O(N \lg^{k-1} N).$$[7]

We have therefore exhibited an algorithm that solves the all-points ECDF problem for $N$ points in $k$-space in $O(N \lg^{k-1} N)$ time, for any fixed $k$ greater than 1.

**2.1.2 The ECDF searching problem.** We now turn our attention to the ECDF searching problem. As in the all-points problem, we first investigate the one-dimensional case and then examine successively higher dimensions. There are three costs associated with a search structure: the *preprocessing* time required to build the structure, the *query* time required to search a structure, and the *storage* required to represent the structure in memory. When analyzing a structure containing $N$ points we denote these quantities by $P(N)$, $Q(N)$, and $S(N)$, respectively. We illustrate these quantities as we examine the one-dimensional ECDF searching problem.

Fig. 4. A three-dimensional problem.




In one dimension the ECDF searching problem asks us to organize $N$ points (real numbers) such that when given a new point $x$ (not necessarily in the set), we can quickly determine how many points $x$ dominates. One of the more obvious ways of solving the problem is the "sorted array" data structure. In this scheme the $N$ points are sorted into increasing order and stored in an array. To see how many points a query point dominates we perform a binary search to find the position of that point in the array. This structure has been studied often (see, for example, Knuth [16]) and is known to have properties

$$P(N) = O(N \lg N),$$
$$Q(N) = O(\lg N),$$
$$S(N) = O(N).$$

In the two-dimensional ECDF searching problem we are to preprocess $N$ points in the plane such that we can quickly answer queries asking the rank of a new point, that is, how many points lie below it and to its left. There are many structures that can be used to solve this problem, but we focus on one called the ECDF tree which follows from the multidimensional divide-and-conquer paradigm (others are discussed by Bentley and Shamos [9]). The multidimensional divide-and-conquer method applied to planar search structures represents a structure of $N$ points in 2-space by two substructures of $N/2$ points in 2-space, and one substructure of $N$ points in 1-space. We now describe the top level of an ECDF tree storing the point set $S$. By analogy to the all-points algorithm, we choose a line $L$ dividing $S$ into equal sized sets $A$ and $B$. Instead of solving subproblems $A$ and $B$, however, we now *recursively process them into ECDF trees* representing their respective subsets. Having built these subtrees we are (almost) prepared to answer ECDF queries in set

[6] The following "bookkeeping" operations must be added to ECDF2 to enable it to solve this problem in $O(N \lg N)$ time: Relabel the $A$s and $B$s to be $X$s and $Y$s, respectively. We are now given $N$ points in the plane and asked to count for each $Y$ the number of $X$s it dominates. As in ECDF2, we divide into sets $A$ and $B$ and solve those subproblems recursively. We must now count for each $Y$ in $B$ the number of $X$s in $A$ it dominates; we do this by projecting only the $X$s of $A$ and the $Y$s of $B$ onto $L$ in step 3 of ECDF2.

[7] We use the fact that if $T(N) = 2T(N/2) + O(N \lg^m N)$, then $T(N) = O(N \lg^{m+1} N)$. A more detailed discussion of these recurrences can be found in Monier [21].

Fig. 5. Two cases of planar queries.

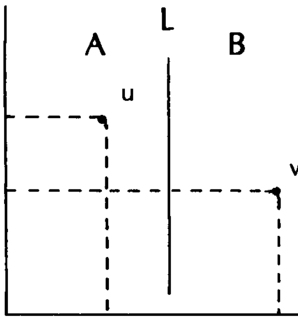
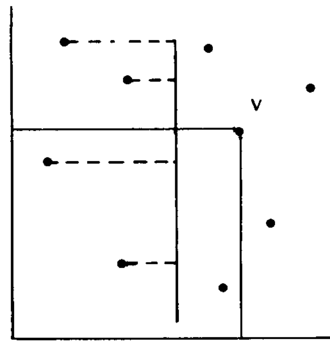



Fig. 6. Calculating $v$'s $y$-rank in $A$.




$S$. The first step of a query algorithm compares the $x$-value of the query point with the line $L$; the two possible outcomes are illustrated in Figure 5 as points $u$ and $v$. If the point lies to the left of $L$ (as $u$ does), then we find its rank in $S$ by recursively searching the substructure representing $A$, for it cannot dominate any point in $B$. If the point lies to the right of $L$ (as $v$ does), then searching $B$ tells how many points in $B$ are dominated by $v$, but we still must find how many points in $A$ are dominated by $v$. To do this we need only calculate $v$'s $y$-rank in $A$; this is illustrated in Figure 6.

We can now describe the planar ECDF tree more precisely. An internal node representing a set of $N$ points will contain an $x$-value (representing the line $L$), a pointer to a left son (representing $A$, the $N/2$ points with lesser $x$-values), a right son representing $B$, and an array of the $N/2$ points of $A$ sorted by $y$-value. To build an ECDF tree recursively one divides the set into $A$ and $B$, builds the subtrees representing each, and then sorts the elements of $A$ by $y$-value (actually by presorting). To search the tree recursively one first compares the $x$-value of the node with the $x$-value of the query point. If the query point is less, then only the left son is searched recursively. If the value is greater, then the right son is searched recursively, a binary search is done in the sorted $y$-sequence representing $A$ to find the query point's $y$-rank in $A$, and the two ranks are added together and returned as the result.

To analyze this search structure we again use recurrences. In counting the preprocessing cost we note that the recurrence describing the algorithm (with presorting) is

$$P(N) = 2P(N/2) + O(N)$$

and the solution is

$$P(N) = O(N \lg N).$$

To store an $N$ element set we must store two $N/2$ element sets plus one sorted list of $N/2$ elements, so the recurrence is

$$S(N) = 2S(N/2) + N/2$$

which has solution

$$S(N) = O(N \lg N).$$

In analyzing the search time our recurrence will depend on whether the point lies in $A$ or $B$, so we assume it lies in $B$ and analyze its worst case. In this case we must make one comparison, perform a binary search in a structure of size $N/2$, and then recursively search a structure of size $N/2$. The cost of this will be

$$Q(N) = Q(N/2) + O(\lg N)$$

so we know that the worst-case cost of searching is

$$Q(N) = O(\lg^2 N).$$

Having analyzed the performance of the planar ECDF tree, we can turn our attention to higher-dimensional ECDF searching problems.

A node representing an $N$-element ECDF tree in 3-space contains two subtress (each representing $N/2$ points in 3-space) and a two-dimensional ECDF tree (representing the projection of the points in $A$ onto the cut plane $P$). This structure is built recursively (analogous to the ECDF3 algorithm). The searching algorithm compares the query point's $x$ value to the value defining the cut plane, and if less, searches only the left substructure. If the query point lies in $B$, then the right substructure is searched, and a search is done in the two-dimensional ECDF tree. The full $k$-dimensional structure is analogous: A node in this structure contains two substructures of $N/2$ points in $k$-space, and one substructure of $N/2$ points in $(k-1)$-space. The recurrences describing the structure containing $N$ points in $k$-space are

$$\begin{aligned} P(N, k) &= 2P(N/2, k) + P(N/2, k-1) + O(N), \\ S(N, k) &= 2S(N/2, k) + S(N/2, k-1) + O(1), \\ Q(N, k) &= Q(N/2, k) + Q(N/2, k-1) + O(1). \end{aligned}$$

We can use the performance of the two-dimensional structure as a basis for induction on $k$, and thus establish (for fixed values of $k$) that

$$\begin{aligned} P(N, k) &= O(N \lg^{k-1} N), \\ S(N, k) &= O(N \lg^{k-1} N), \\ Q(N, k) &= O(\lg^k N). \end{aligned}$$

It is interesting to note how faithfully the actions of the multidimensional divide-and-conquer algorithms are described by the recurrences. Indeed, the recurrences might

provide a suitable definition of multidimensional divide-and-conquer!

**2.1.3 Summary of the ECDF problems.** In our study of ECDF problems so far we have concentrated on algorithms for solving the problems without examining lower bounds. We saw that the one-dimensional all-points ECDF problem is equivalent to sorting (that is, one problem can be reduced to the other in linear time), so we know that the ECDF problem has an $\Omega(N \lg N)$ lower bound in the decision tree model of computation. Since the one-dimensional problem can be embedded in any higher-dimensional space (by simply ignoring some coordinates), this immediately gives an $\Omega(N \lg N)$ lower bound for the all-points problem in $k$-space. Lueker [20] has used similar methods to show that $\theta(kN \lg N)$ time is necessary and sufficient for the all-points problem in $k$-space in the decision tree model of computation. Unfortunately, there do not appear to be succinct programs corresponding to the decision trees used in his proof. It therefore appears that a stronger model of computation than decision trees will have to be used to prove lower bounds on these problems; Fredman [12] has recently made progress in this direction. These results show that Algorithm ECDF2 is within a constant factor of optimal; whether Algorithm ECDF$k$ is optimal in some reasonable model of computation remains an open question. Similar methods can be used to show a lower bound on the problem of ECDF searching; it requires at least $\Omega(\lg N)$ time in the worst case.

The analyses that we have seen for the ECDF algorithms have been "rough" in two respects: We have only considered the case that $N$ is a power of 2, and our analyses were given for fixed $k$ as $N$ grows large. Monier [21] has analyzed the algorithms in this section more exactly, overcoming both of the above objections. His analysis shows that the time taken for Algorithm ECDF$k$ is given by

$$T(N, k) = c(N \lg^{k-1} N)/(k-1)! + O(N \lg^{k-2} N)$$

where $c$ is an implementation-dependent constant. It is particularly pleasing to note that the coefficient of the $N \lg^{k-1} N$ term is $1/(k-1)!$; this function goes to zero very rapidly. Monier's analyses also showed that the leading terms of the ECDF searching structures performances have similar coefficients (inverse factorials).

We have now completed our study of the ECDF problems per se, and it is important for us to take a moment to reflect on the things we have learned about multidimensional divide-and-conquer. The paradigm applies directly to all-points problems, and we state it here in its full generality:

> To solve a problem of $N$ points in $k$-space, solve two problems of $N/2$ points each in $k$-space and one problem of (up to) $N$ points in $(k-1)$-space.

Algorithms based on this paradigm have three major parts: the *division*, *recursive*, and *marriage* steps. Because of the recursion on dimension, an important technique in developing these algorithms is to start with low-dimensional problems and then successively move to higher dimensions. Describing an algorithm recursively leads to two advantages: We can describe it succinctly and then analyze its performance by the use of recurrence relations. The recurrence used most often is

$$F(N) = 2F(N/2) + O(N \lg^m N)$$

which has solution $O(N \lg^{m+1} N)$ for $m \geq 0$.

The multidimensional divide-and-conquer paradigm can be applied in the development of data structures as well as all-points algorithms. The data structure strategy can be described as follows:

> To store a structure representing $N$ points in $k$-space, store two structures representing $N/2$ points each in $k$-space and one structure of (up to) $N$ points in $(k-1)$-space.

There are, of course, many similarities between these data structures and the multidimensional algorithms (most importantly, we use such an algorithm to build the structure), so the principles we enumerated above for all-points problems will apply to data structures as well. In addition to the recurrence mentioned above, the recurrence

$$F(N) = F(N/2) + O(\lg^m N)$$

which has solution $F(N) = O(\lg^{m+1} N)$ for $m \geq 0$ arises often in the study of these structures.

*Presorting* is a technique applicable to both multidimensional divide-and-conquer algorithms and data structures. By sorting data once-for-all before a recursive algorithm is initially invoked (and then keeping the data sorted as we divide into subproblems), we can avoid the repetitive cost of sorting. This technique often saves factors of $O(\lg N)$. One might hope that presorting could in some way be used many times to save multiple factors of $O(\lg N)$, but the author doubts that this can be achieved.

Having made these general observations about our primary algorithm design tool, we are ready to apply it to the solution of other problems. Because we have examined the ECDF algorithms in some detail and the algorithms that we will soon examine are so similar, our discussion of those algorithms will not be so precise; their details can be deduced by analogy with the algorithms of this section.

### 2.2 Maxima

In this section we investigate problems dealing with maximal elements, or *maxima*, of point sets. A point is said to be a maximum of a set if there is no other point that dominates it. In Figure 7 we illustrate a planar point set with the maxima of the set circled. We are interested in two types of maxima problems: the all-points problem (given a set, find all the maxima) and the searching problem (preprocess a set to answer queries asking if a new point is a maximum of the set). The problem of computing maxima arises in many diverse applications. Suppose, for example, that we have a set of programs for

Fig. 7. Maxima are circled.

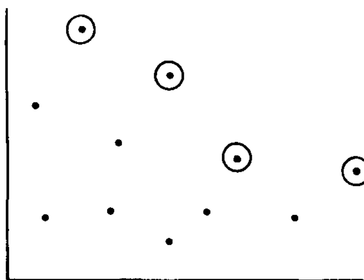
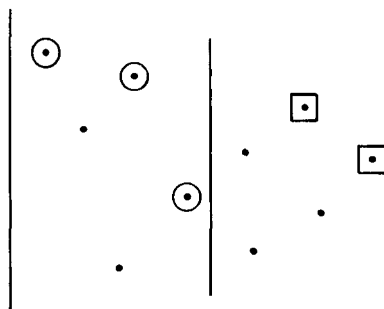


Fig. 8. Maxima of $A$ are circled; $B$'s are squared.



performing the same task rated on the two dimensions of space efficiency and time efficiency. If we plot these measures as points in the $x$–$y$ plane, then a point (program) dominates another only if it is more space efficient and more time efficient. The maximal programs of the set are the only ones we might consider for use, because any other program is dominated by one of the maxima. In general, if we are seeking to maximize some multivariate goodness function (monotone in all variables) over some finite point set, then it suffices to consider only maxima of the set. This observation can significantly decrease the cost of optimization if many optimizations are to be performed. Such computation is common in econometric problems.

Problems about maxima are very similar to problems about ECDFs. If we define the *negation* of point set $A$ (written $-A$) to consist of each of the points of $A$ multiplied by $-1$, then a point is a maximum of $A$ if and only if its rank in $-A$ is zero (for if it is dominated by no points in $A$, then it dominates no points in $-A$). By this observation we can solve the all-points maxima problem in $O(N \lg^{k-1} N)$ time and the maxima searching problem with similar preprocessing time and space and $O(\lg^k N)$ query time, by using the ECDF algorithms of Section 2.1. In this section we investigate a different multidimensional divide-and-conquer algorithm that allows us to reduce those cost functions by a factor of $O(\lg N)$. The all-points maxima algorithm we will see is due to Kung et al. [17] (although our presentation is less complicated than theirs). The searching structure of this section is described here for the first time. Although the algorithms that we will see are similar to the ECDF algorithms of the last section in many respects, they do have some interesting expected-time properties that the ECDF algorithms do not have. Having made these introductory comments, we can now turn our attention to the maxima problems, investigating first the all-points problem and then the searching problem.

The maximum of $N$ points on a line is just the maximum element of the set, which can be found in exactly $N-1$ comparisons. Computing the maxima of $N$ points in the plane is just a bit more difficult. Looking at Figure 7, we notice that the maxima (circled) are increasing upward as the point set is scanned right to left. This suggests an algorithm: Sort the points into increasing $x$-order and then scan that sorted list right to left, observing successive "highest $y$-values so far observed" and marking those as maxima. It is easy to prove that this algorithm gives exactly the maxima, for a point is maximal if and only if all points with greater $x$-values (before it on the list) have lesser $y$-values. The computational cost of the algorithm will be $O(N \lg N)$ for the sort and then $O(N)$ for the scan. (So note that if we have presorted the list, then the total time for finding the maxima is linear.)

We can also develop a multidimensional divide-and-conquer algorithm to solve the planar problem. As before, we divide by $L$ into $A$ and $B$ and solve those subproblems recursively (finding the maxima of each set). This is illustrated in Figure 8, in which the maxima of $A$ are circled and the maxima of $B$ are in boxes. Because no point in $B$ is dominated by any point in $A$, the maxima of $B$ are also maxima of the entire set $S$. Thus the third step (the "marriage" step) of our algorithm must discard points which are maxima of $A$ but not of the whole set, i.e., those maxima of $A$ which are dominated by some point in $B$. Since all points in $B$ $x$-dominate all points in $A$, we need check only for $y$-domination. We therefore project the maxima of $A$ and $B$ onto $L$, then discard $A$-points dominated by $B$-points on the line. This third step can be easily implemented by just comparing the $y$-value of all $A$-maxima with the maximum $y$-value of the $B$-maxima and discarding all $A$'s with lesser $y$-value (we described it otherwise to ease the transition to higher spaces). The running time of this algorithm is described by the recurrence

$$T(N) = 2T(N/2) + O(N)$$

which has solution $O(N \lg N)$.

We can generalize the planar algorithm to yield a maxima algorithm for 3-space. The first step divides into $A$ and $B$, and the second step recursively finds the maxima of each of those sets. Since every maxima of $B$ is a maxima of the whole set, the third step must discard every maxima of $A$ which is dominated by a maxima of $B$. This is accomplished by projecting the respective maxima sets onto the plane and then solving the planar problem. We could modify the two-dimensional maxima algorithm to solve this task, but it will be slightly more efficient to use the "scanning" algorithm. Suppose we

cut into $A$ and $B$ by the $z$-coordinate; we must discard all $A$s dominated by any $B$s in the $x$–$y$ plane. If we have presorted by $x$, then we just scan right to left down the sorted list, discarding $A$s with $y$-values less than the maximum $B$ $y$-value observed to date. This marriage step will have linear time (with presorting), so this algorithm has the same recurrence as the two-dimensional, and its running time is therefore also $O(N \lg N)$.

The obvious generalization of this algorithm carries through to $k$-space without difficulty. We solve a problem of $N$ points in $k$-space by solving two problems of $N/2$ points in $k$-space and then solving one problem of (up to) $N$ points in $(k-1)$-space. This reduced problem calls for finding all $A$s in the space dominated by any $B$s, and we can solve this by modifying the maxima algorithm (similar to our modifications of the ECDF algorithm). The resulting algorithm has a recurrence

$$T(N, k) = 2T(N/2, k) + T(N, k-1) + O(N)$$

and we can use the fact that $T(N, 3) = O(N \lg N)$ to establish that

$$T(N, k) = O(N \lg^{k-2} N) \text{ for } k \geq 3.$$

The analysis we just performed, though accurate for the worst case, is terribly pessimistic. It assumes that all $N$ points of the original set will be maxima of their subsets, whereas for many sets there will be relatively few maxima of $A$ and $B$. Results obtained by Bentley et al. [6] show that only a very small number of points usually remain as maxima (for many probability distributions). If only $m$ points remain, then the term $T(N, k-1)$ in the above recurrence is replaced by $T(m, k-1)$, which for small enough $m$ (i.e., $m = O(N^p)$ for some $p < 1$) has running time $O(N)$. If this is true, then the recurrence describing the maxima algorithm is

$$T(N, k) = 2T(N/2, k) + O(N),$$

which has solution $T(N) = O(N \lg N)$. One can formalize the arguments we have just sketched to show the average running time of the above algorithm is $O(N \lg N)$ for a wide class of distributions. The interested reader is referred to [6] in which a linear expected-time maxima algorithm is presented (with poorer worst-case performance than this algorithm); the analysis techniques used therein can be used to prove this result.

We turn our attention now to the maxima searching problem. We start with the planar case, where we must process $N$ points in the plane into a data structure so we can quickly determine if a new point is a maxima (and if not, we must name a point which dominates it). Our structure is a binary tree in which the left son of a given node represents all points with lesser $x$-values ($A$), the right son represents $B$, and an $x$-value represents the line $L$. To answer a query asking if a new point $q$ is a maximum of the set represented by a given node, we compare $q$'s $x$-value to the node's. If the point lies in $B$ (greater $x$-value), then we search the subtree and return the answer. If the point lies in $A$, however, we first search the left subtree; if the point is dominated, we return the dominating point. If it is not dominated by any point in $A$, then we must check to see if it is dominated by any point in $B$. This can be accomplished by storing in each node the maximum $y$-value of any point in $B$. This structure can be built in $O(N \lg N)$ time and requires linear space. Since the worst-case cost of a query satisfies the recurrence

$$T(N) = T(N/2) + O(1),$$

the worst-case search time is $O(\lg N)$.

This search structure can be generalized to $k$-space. In that case a structure representing $N$ points in $k$-space contains two substructures representing $N/2$ points in $k$-space and one substructure representing $N/2$ points in $(k-1)$-space. To test if a new point is a maximum we first determine if it lies in $A$ or $B$. If it is in $B$, then we visit only the right son. If it lies in $A$, we first see if it is dominated by any point in $A$ (visit the left son), and if not then we check to see if it is dominated by any point in $B$ (by searching the $(k-1)$-dimensional structure). The recurrences describing the worst-case performance of this structure are

$$\begin{aligned} P(N, k) &= 2P(N/2, k) + P(N, k-1) + O(N), \\ S(N, k) &= 2S(N/2, k) + S(N/2, k-1) + O(1), \\ Q(N, k) &= Q(N/2, k) + Q(N/2, k-1) + O(1), \end{aligned}$$

which have solutions

$$\begin{aligned} P(N, k) &= O(N \lg^{k-2} N), \\ S(N, k) &= O(N \lg^{k-2} N), \\ Q(N, k) &= O(\lg^{k-1} N). \end{aligned}$$

As in the case of the all-points problem, these times are highly pessimistic, and for many point distributions they can be shown to be much less on the average.

Yao [29] has shown that $\Omega(N \lg N)$ is a lower bound on the decision tree complexity of computing the maxima of $N$ points in 2-space. This result shows that the maxima algorithms we have seen here are optimal for two and three dimensions (by embedding). Lower bounds for the rest of the problems of this section are still open problems, and a model other than the decision tree will have to be used to prove optimal the algorithms that we have seen.

This concludes our study of maxima problems. Clever application of the multidimensional divide-and-conquer strategy allowed us to squeeze a factor of $O(\lg N)$ from the running times of the ECDF algorithms. We also glimpsed how an expected-time analysis might be performed on the computation-cost functions.

### 2.3 Range Searching

In this section we examine the problem of range searching, a searching problem defined by point domination for which there is no corresponding all-points problem. The problem is to build a structure holding $N$ points in $k$-space to facilitate answering queries of the form "report all points which are dominated by point $U$

Fig. 9. Number of points in $R = r(A) - (r(B) + r(D)) + r(C)$.

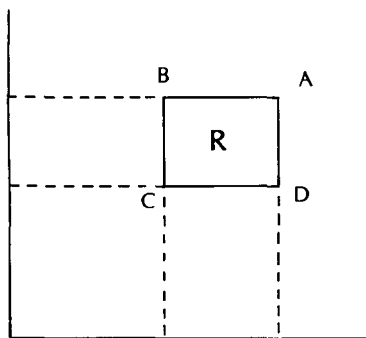
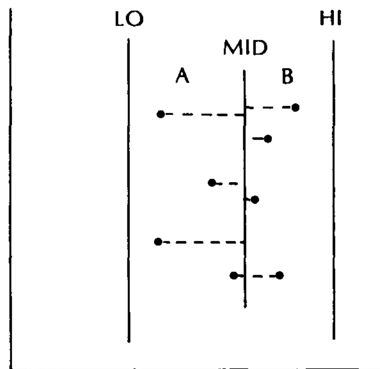



Fig. 10. A node in a planar range tree.




and dominate point $L$." This kind of query is usually called an *orthogonal range query* because we are in fact giving for each dimension $i$ a range $R_i = [l_i, u_i]$ and then asking the search to report all points $x$ such that $x_i$ is in range $R_i$ for all $i$. A geometric interpretation of the query is that we are asking for all points that lie in a given hyper-rectangle. Such a search might be used in querying a geographic database to list all cities with latitude between 37° and 41° N and longitude between 102° and 109° W (this asks for all cities in Colorado). In addition to database problems, range queries are also used in certain statistical applications. These applications and a survey of the different approaches to the problem are discussed in Bentley and Friedman's [5] survey of range searching. The multidimensional divide-and-conquer technique that we will see has also been applied to this problem by Lee and Wong [18], Lueker [20], and Willard [28] who independently achieved structures very similar to the ones we describe.

In certain applications of the range searching problem we are not interested in actually processing each point found in the query rectangle—it suffices rather to know only how many such points there are. (One such example is multivariate density estimation.) Such a problem can be solved by using the ECDF searching algorithm of Section 2.1 and the principle of inclusion and exclusion. Figure 9 illustrates how four planar rank queries can be combined to tell the number of points in rectangle $R$ (we use "r" as an abbreviation for "rank"); in $k$-space $2^k$ range searches are sufficient.

The *sorted array* is one suitable structure for range searching in one-dimensional point sets. The points are organized into increasing order exactly as they were for the ECDF searching problem of Section 2.1. To answer a query we do two binary searches in the array to locate the positions of the low and high ends of the range; this identifies a sequence of points in the array which are the answer to the query, and they can then be reported by a simple procedure. The analysis of this structure for range searching is very similar to our previous analysis: The storage cost is linear and the preprocessing cost is $O(N \lg N)$. The query cost is then $O(\lg N)$ for the binary searches plus $O(F)$, if a total of $F$ points are found to be in the region. Note that any algorithm for range searching must include a term of $O(F)$ in the analysis of query time.

We will now describe *range trees*, a structure introduced by Bentley [4]; as usual, we first examine the planar case. There are six elements in a range tree's node describing set $S$. These values are illustrated in Figure 10. The reals LO and HI give the minimum and maximum $x$-values in the set $S$ (these are accumulated "down" the tree as it is built). The real MID holds the $x$-value defining the line $L$, which divides $S$ into $A$ and $B$, as usual; we then store two pointers to range trees representing the sets $A$ and $B$. The final element stored in the node is a pointer to a sorted array, containing the points of $S$ sorted by $y$-value. A range tree can be built recursively in a manner similar to constructing an ECDF tree. We answer a range query asking for all points with $x$-value in range $X$ and $y$-value in range $Y$ by visiting the root of the tree with the following recursive procedure. When visiting node $N$ we compare the range $X$ to the range [LO, HI]. If [LO, HI] is contained in $X$, then we can do a range search in the sorted array for all points in the range $Y$ (all these points satisfy both the $X$ and $Y$ ranges). If the $X$ range lies wholly to one side of MID, then we search only the appropriate subtree (recursively); otherwise we search both subtrees. If one views this recursive process as happening all at once, we see that we are performing a set of range searches in a set of arrays sorted by $y$. The preprocessing costs of this structure and the storage costs are both $O(N \lg N)$. To analyze the query cost we note that at most two sorted lists are searched at each of the $\lg N$ levels of the tree, and each of those searches cost at most $O(\lg N)$, plus the number of points found during that search. The query cost of this structure is therefore $O(\lg^2 N + F)$, where $F$ (as before) is the number of points found in the desired range.

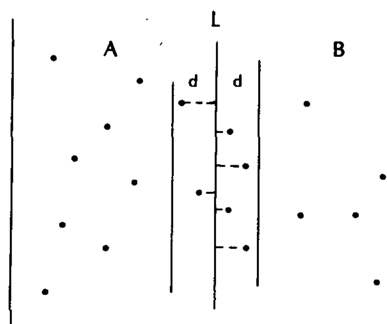
The range tree structure can of course be generalized to $k$-space. Each node in such a range tree contains pointers to two subtrees representing $N/2$ points in $k$-space and one $N$ point subtree in $(k-1)$-space. Analysis of range trees shows that

$$P(N, k) = O(N \lg^{k-1} N),\ S(N, k) = O(N \lg^{k-1} N),$$
$$Q(N, k) = O(\lg^k N + F)$$

where $F$ is the number of points found.

Saxe [24] has used the decision tree model of computation to show a lower bound on the range searching problem of approximately $2k \lg N$. Bentley and Maurer [7] have given a range searching data structure that realizes this query time, at the cost of extremely high storage and preprocessing requirements. An interesting open problem is to give bounds on the complexity of this problem in the presence of only limited space (or preprocessing time); Fredman's [12] work is a first step in this direction.

Fig. 11. Fixed-radius near neighbor algorithm.




## 3. Closest-Point Problems

In Section 2 we investigated problems defined by point domination; in this section we discuss a class of problems defined by point *closeness*. We saw that multidimensional divide-and-conquer "works" for domination problems because projection onto a plane preserves point domination. In this section we discuss a number of projections that preserve point closeness.

We investigate three problems dealing with closeness. We use as our "closeness" measure the standard Euclidean distance measure (although the algorithms can be modified to use other measures). The problem we study in Section 3.1 is the easiest of the three problems we discuss because it is defined in terms of "absolute" closeness. The problems of Sections 3.2 and 3.3 are defined in terms of relative distances and are therefore a bit trickier. Throughout this section we describe the algorithms only at a very high level; the interested reader can find the details of these algorithms (as well as a sketch of how they were discovered) in Bentley [3].

### 3.1 Fixed-Radius Near Neighbors

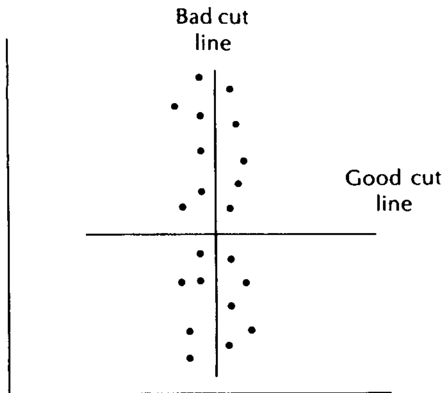
In this section we discuss problems on point sets which deal with absolute closeness of points, that is, pairs of points within some fixed distance $d$ of one another. We concentrate on the all-points problem which asks for all pairs within $d$ to be listed, and then we briefly examine the problem of "fixed-radius near neighbor" searching. Fixed-radius problems arise whenever "multidimensional agents" have the capability of affecting all objects within some fixed radius. Such problems arise in air traffic control, molecular graphics, pattern recognition, and certain military applications. One difficulty in approaching the fixed-radius near neighbors problem, however, is that if points are clustered closely together, then there can be $O(N^2)$ close pairs, and we are therefore precluded from finding a fast algorithm to solve the problem.

We can avoid this difficulty by considering only sparse point sets, that is, sets which are not "clustered." We define sparsity as the condition that no $d$-ball in the space (that is, a sphere of radius $d$) contains more than some constant $c$ points. This condition ensures that there will be no more than $cN$ pairs of close points found. This condition is guaranteed in certain applications from the natural sciences—if an object can affect all objects within a certain radius, then there cannot be too many "affecting" objects. We see that this condition also arises naturally in the solution of other closest-point problems. In this section we investigate the sparse all-points near neighbors problem by examining successively higher dimensions, and then we turn our attention to the searching problem.

In the one-dimensional all-points near neighbor problem we are given $N$ points on a line and constants $c$ and $d$ such that no segment on the line of length $2d$ contains more than $c$ points; our problem is to list all pairs within $d$ of one another. We can accomplish this by sorting the points into a list in ascending order and then scanning down that list. When visiting point $x$ during the scan we check backward and forward on the list a distance of $d$. By the sparsity condition, this involves checking at most $c$ points for "closeness" to $x$. The cost of this procedure is $O(N \lg N)$ for the sorting and then $O(N)$ for the scan, for a total cost of $O(N \lg N)$. Note the very important role sparsity plays in analyzing this algorithm: It guarantees that the cost of the scan is linear in $N$.

Figure 11 shows how we can use multidimensional divide-and-conquer to solve the planar near neighbor problem. The first and second steps of our algorithm are, as usual, to divide the point set by $L$ into $A$ and $B$ and then find all near neighbor pairs in each recursively. At this point we have almost solved our problem—all that remains to be done is to find all pairs within $d$ which have one element in $A$ and one in $B$. Note that the "$A$ point" of such a pair must lie in the slab of $A$ which is within $d$ of $L$, and likewise for $B$. Our third step thus calls for finding all pairs with one element in $A$ and the other in $B$, and to do this we can confine our attention to the slab of width $2d$ centered about line $L$. But this can be transformed into a one-dimensional problem by projecting all points in the slab onto $L$. It is not difficult to show that projection preserves sparsity (details of the proof can be found in Bentley [3]), and it is obvious that projection preserves closeness, for projection only decreases the distance between pairs of points. Our reduced

Fig. 12. Two cut lines.




problem is therefore just the one-dimensional sparse near neighbors problem (though it requires checking both to ensure pairs have one element from $A$ and one from $B$ and to ensure that the pairs were close before projection), and this can be accomplished in $O(N \lg N)$ time, or linear time if presorting is used. The runtime of our algorithm thus obeys the recurrence

$$T(N) = 2T(N/2) + O(N)$$

which has solution $T(N) = O(N \lg N)$. Sparsity played two important roles in this algorithm. Since the original point set was sparse, we could guarantee that both $A$ and $B$ would be sparse after the division step (which in no way alters $A$ or $B$). The sparsity condition was also preserved in the projection of the third step, which allowed us to use the one-dimensional algorithm to solve the resulting subproblem.

The algorithm we just saw can be generalized to three and higher dimensions. In three dimensions we divide the set by a cut plane $P$ into $A$ and $B$ and find all near pairs in those sets recursively. We now need to find all close pairs with one member in $A$ and the other in $B$, and to do this we confine our attention to the "slab" of all points within distance $d$ of $P$. If we project all those points onto the slab (remembering if each was an $A$ or a $B$), then we have a planar near neighbor problem of (up to) $N$ points. Using our previous planar algorithm gives an algorithm for 3-space with $O(N \lg^2 N)$ running time. Extending this to $k$-space gives us an $O(N \lg^{k-1} N)$ algorithm.

Having seen so many $O(N \lg^{k-1} N)$ algorithms in this paper may have lulled the reader into a bleary-eyed state of universal acceptance, but the practicing algorithm designer never sleeps well until he has an algorithm with a matching lower bound. For this problem the best known lower bound is $\Omega(N \lg N)$; so we are encouraged to try to find an $O(N \lg N)$ algorithm. First we consider our planar algorithm in its $O(N \lg^2 N)$ form, temporarily ignoring the speedup available with presorting. If we ask where the extra logarithmic factor comes from, we see that it is due to the fact that in the worst case all $N$ points can lie in the slab of width $2d$; this is illustrated in Figure 12. If the points are configured this way, then we should choose as cut line $L$ a horizontal line dividing the set into halves. It turns out not to be hard to generalize this notion to show that in any sparse point set there is a "good" cut line. By "good" we mean that $L$ possesses the following three properties:

(1) It is possible to locate $L$ in linear time.
(2) The set $S$ is divided approximately in half by $L$.
(3) Only $O(N^{1/2})$ points of $S$ are within $d$ of $L$.

A proof that every sparse point set contains such a cut line can be found in Bentley [3]. We can use the existence of such a cut line to create an $O(N \lg N)$ algorithm. The first step of our algorithm takes linear time (by property 1 of $L$), and the second step is altered (by property 2). The third step is faster because it sorts fewer than $N$ points—only the $O(N^{1/2})$ points within $d$ of $L$, by property 3. Since this can be accomplished in much less than linear time, our algorithm has the recurrence[8]

$$T(N) = 2T(N/2) + O(N)$$

which has solution $O(N \lg N)$. The gain in speed was realized here by solving only a very small problem on the line, so small that it can be solved in much less than linear time. Not unexpectedly, it can be shown that for sparse point sets in $k$-space there will always exist good cut planes, which will have not more than $O(N^{1-1/k})$ points within $d$ of them. These planes imply that the $(k-1)$-dimensional subproblem can be solved in less than linear time, and the full problem thus obeys the recurrence

$$T(N, k) = 2T(N/2, k) + O(N).$$

This establishes that we can solve the general problem in $O(N \lg N)$ time.

The techniques which we have used for the all-points near neighbors problems can also be applied to the near neighbor searching problem. In that problem we are given a sparse set of points to preprocess into a data structure such that we can quickly answer queries asking for all points within $d$ of a query point. If we use the general multidimensional divide-and-conquer strategy, then we achieve a structure very similar to the range tree, with performances

$$P(N) = O(N \lg^{k-1} N),$$
$$S(N) = O(N \lg^{k-1} N),$$
$$Q(N) = O(\lg^k N).$$

If we make use of the good cut planes, however, then we can achieve a structure with performance

$$P(N) = O(N \lg N),$$
$$S(N) = O(N),$$
$$Q(N) = O(\lg N).$$

This modified structure follows immediately from the properties of the cut planes we mentioned above; the

[8] The recurrence actually takes on a slightly different form—details are in Bentley [3].

details are similar to the other multidimensional divide-and-conquer structures we have seen previously.

To show lower bounds on fixed-radius problems in $k$-space we can consider the corresponding problems in 1-space. Fredman and Weide [13] have shown that the problem of reporting all intersecting pairs among a set of segments on the line requires $\Omega(N \lg N)$ time; by embedding, this immediately gives the same lower bound on the all-points fixed-radius near neighbors problem in $k$-space. This shows that our algorithm is optimal (to within a constant factor). Reduction to one dimension can also be used to show that the data structure is optimal.

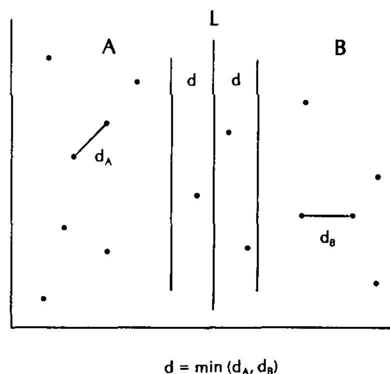
In this section we have seen how multidimensional divide-and-conquer can be applied to closest-point problems, a totally different kind of problem than the domination problems we saw in Section 2. Some of the techniques we have seen in this section will be useful in all other closest-point problems. One such technique is employing the concept of *sparsity*; it was given in the statement of this problem, and we will see how to introduce it into other problems in which it is not given. The second technique that we will use again is projection of all *near* points onto a cut plane. With these tools in hand, we now investigate other closest-point problems.

### 3.2 Closest Pair

In this section we examine the closest-pair problem, an all-points problem with no searching analog. We are given $N$ points in $k$-space and must find the closest pair in the set. Notice that this problem is based on *relative*, not absolute, distances. Although the distance separating the closest pair could be used as a rotation-invariant "signature" of a point set, its primary interest to us is not as an applications problem but rather in its status as an "easiest" closest-point problem. We call it easiest because there are a number of other geometric problems (such as nearest neighbors and minimal spanning trees) that find the closest pair as part of their solution. For a long time researchers felt that there might be a quadratic lower bound on the complexity of the closest-pair problem, which would have implied a quadratic lower bound on all the other problems. In this section we will see an $O(N \lg N)$ closest-pair algorithm, which gives us hope for the existence of fast algorithms for the other problems. (The $O(N \lg N)$ planar algorithm we will see was first described by Shamos [26], who attributes to H.R. Strong the idea of using divide-and-conquer to solve this problem.)

The one-dimensional closest-pair problem can be solved in $O(N \lg N)$ time by sorting. After performing the sort we scan through the list, checking the distance between adjacent elements. In two dimensions we can use multidimensional divide-and-conquer to solve the problem. The first step divides $S$ by line $L$ into sets $A$ and $B$, and the second step finds the closest pairs in $A$ and $B$, the distances between which we denote by $d_A$ and $d_B$, respectively. This is illustrated in Figure 13. Note that we have now introduced a sparsity condition into both $A$ and $B$. Because the closest pair in $A$ is $d_A$ apart, no $d_A$-ball in $A$ can contain more than seven points. This follows from the fact that at most six unit circles can be made to touch some fixed unit circle in the plane without overlapping; details of the proof are in Bentley [3]. Likewise we can show that $B$ is sparse in the sense that no $d_B$-ball in $B$ contains more than seven points. If we let $d$ be the minimum of $d_A$ and $d_B$, notice that the whole space is sparse in the sense that no $d$-ball contains more than 14 points. This observation of "induced" sparsity will make the third step of our algorithm much easier, which is to make sure that the closest pair in the space is actually that corresponding to $d_A$ or to $d_B$. We could just run a sparse fixed-radius near neighbor algorithm at this point to find any pairs within $d$ of one another, but there is a more elegant approach. Note that any close pair must have one element in $A$ and one element in $B$, so all we have to do is consider the slab of all points within $d$ of $L$, and the third step of this algorithm becomes exactly the third step of the near neighbor algorithm. If we do not use presorting, this gives an $O(N \lg^2 N)$ algorithm.

Fig. 13. A planar closest-pair algorithm.




The generalization to 3-space is obvious: We choose a plane $P$ defining $A$ and $B$ and solve the subproblems for those sets. After this we have introduced sparsity into both $A$ and $B$ (relative to $d_A$ and $d_B$), and we can ensure that our answer is correct by solving a planar fixed-radius subproblem. In $k$-space we solve two closest-pair problems of $N/2$ points each in $k$-space and one fixed-radius problem of (up to) $N$ points in $k-1$ dimensions. If we use the $O(N \lg N)$ algorithm for near neighbors, then our recurrence is

$$T(N) = 2T(N/2) + O(N \lg N)$$

which has solution $T(N) = O(N \lg^2 N)$. Although we will not go into the details of the proof here, Bentley [3] has shown how the good cut planes we saw for the fixed-radius problem can be applied to this problem. If they are used appropriately, then the running time of the closest-pair algorithm in $k$-space can be reduced to $O(N \lg N)$. Shamos [26] has shown an $\Omega(N \lg N)$ lower bound on this problem in 1-space by reduction to the "element uniqueness" problem; this algorithm is therefore optimal to within a constant factor.

### 3.3 Nearest Neighbors

The final closest-point problem we investigate deals with nearest neighbors. In the all-points form we ask that for each point $x$ the nearest point to $x$ be identified (ties may be broken arbitrarily). In the searching form we give a new point $x$ and ask which of the points in the set is nearest to $x$. The all-points problem has applications in cluster analysis and multivariate hypothesis testing; the searching problem arises in density estimation and classification. As usual, we begin our discussion of this problem by examining the planar case of the all-points problem.

It is not hard to see how multidimensional divide-and-conquer can be used to solve the planar problem. The first step divides $S$ into $A$ and $B$ and the second step finds for each point in $A$ its nearest neighbor in $A$ (and likewise for each point in $B$). The third step must "patch up" by finding if any point in $A$ actually has its true nearest neighbor in $B$, and similarly for points in $B$. To aid in this step we observe that we have established a particular kind of sparsity condition. We define the NN-ball (for nearest neighbor ball) of point $x$ to be the circle centered at $x$ which has radius equal to the distance from $x$ to $x$'s nearest neighbor. It can be shown (see Bentley [3]) that with this definition *no point in the plane is contained in more than seven NN-balls of points in A*. We will now discuss one-half of the third step, namely, the process of ensuring for each point in $A$ that its nearest neighbor in $A$ is actually its nearest neighbor in $S$. In this process we need consider only those points in $A$ with NN-balls intersecting the line $L$ (for if their NN-ball did not intersect $L$, then their nearest neighbor in $A$ is closer than any point in $B$). The final step of our algorithm projects all such points of $A$ onto $L$ and then projects every point of $B$ onto $L$. It is then possible to determine during a linear-time scan of the resulting list if any point $x$ in $A$ has a point in $B$ nearer to $x$ than $x$'s nearest neighbor in $A$. This results in an $O(N \lg N)$ algorithm if presorting is used. Shamos [26] has shown that it is within a constant factor of optimal.

The extension of the algorithm to $k$-space yields $O(N \lg^{k-1} N)$ performance. It is not clear that there is a search structure corresponding to this algorithm. Shamos [26] and Lipton and Tarjan [19] have given nearest neighbor search structures for points in the plane that are analogous to this algorithm. Whether there exists a fast $k$-dimensional nearest neighbor search structure is still an open question; this approach is certainly one promising point of attack for that problem.

## 4. Additional Work

In Sections 2 and 3 we saw many aspects of the multidimensional divide-and-conquer paradigm, but there are many other aspects that can only be briefly mentioned. The paradigm has been used to create fast algorithms for other multidimensional problems, such as the all-points problem of finding the minimal-perimeter triangle determined by $N$ points and the searching problem of determining if a query point lies in any of a set of $N$ rectangles. Another aspect of this paradigm is the work of Bentley [3] on heuristics that algorithm designers can use when applying this paradigm to their problems. These heuristics were enumerated after the paradigm had been used to solve the closest-point problems of Section 3 and were then used in developing the algorithms of Section 2, among others. A final aspect of this paradigm is the precise mathematical analysis of the resulting algorithms; Monier [21] has used beautiful combinatorial techniques to analyze all of the algorithms we have seen in this paper.

We now briefly examine two paradigms of algorithm design closely related to multidimensional divide-and-conquer. The first such paradigm, *planar divide-and-conquer*, is really just the specialization of the general paradigm to the planar case. Shamos [25, 26] has used this technique to solve many computational problems in plane geometry. Among these problems are constructing the convex hulls of planar point sets, constructing Voronoi diagrams (a planar structure which can be used to solve various problems), and two-variable linear programming. It is often easier to apply the paradigm in planar problems than in $k$-dimensional problems, because the third (marriage) step of the algorithm is one-dimensional, and there are many techniques for solving such problems. Lipton and Tarjan [19] have given a very powerful "planar separator theorem" that often aids in applying the planar divide-and-conquer paradigm.[9]

The second related paradigm is what we might call *recursive partitioning*. This technique is usually applied to searching problems, but it can then be used to solve all-points problems by repeated searching. The idea underlying this technique can be phrased as follows: To store $N$ points in $k$-space, store two substructures each of $N/2$ points in $k$-space. Searches in such structures must then occasionally visit both subtrees of a given node to answer some queries, but with "proper" choice of cut planes this can be made to happen very infrequently. Bentley [2] described a search structure based on this idea which he called the *multidimensional binary search tree*, abbreviated as a $k$-d tree when used in $k$-space. That structure has been used to facilitate fast nearest neighbor searching, range searching, fixed-radius near neighbor searching, and for a database problem called "partial match" searching. Reddy and Rubin [23] use recursive partitioning in algorithms and data structures

[9] The author cannot resist pointing out that the planar divide-and-conquer paradigm is also used by police officers. Murray [22] offers the following advice in a hypothetical situation: "A crowd of rioters far outnumbers the police assigned to disperse it. If you were in command, the best action to take would be to split the crowd into two or more parts and disperse the parts separately." An interesting open problem is to apply other algorithmic paradigms to problems in police work, thus establishing a discipline of "computational criminology."

for representing objects in computer graphics systems. Friedman [14, 15] has used the idea of recursive partitioning to solve many problems in multivariate data analysis such as classification and regression. In addition to their theoretical interest, these structures are quite easy and efficient to implement; their use has reduced the costs of certain computations by factors of a hundred to a thousand (examples of such savings can be found in the above references).

All of the data structures described in this paper have been *static* in the sense that once they are built, additional elements cannot be inserted into them. Many applications, however, require a *dynamic* structure into which additional elements can be inserted. Techniques described by Bentley [4] can be applied to all of the data structures that we have seen in this paper to transform them from static to dynamic. The cost of this transformation is to add an extra factor of $O(\lg N)$ to both query and preprocessing times ($P(N)$ now denotes the time required to insert $N$ elements into an initially empty structure), while leaving the storage requirements unchanged. The details of this transformation can be found in Bentley [4]. Recent work by Lueker [20] and Willard [28] can be applied to all of the data structures in this paper to convert them to dynamic at the cost of an $O(\lg N)$ increase in $P(N)$, leaving both $Q(N)$ and $S(N)$ unchanged. Additionally, their method facilitates *deletion* of the elements.

This survey of additional work is not completed by having mentioned only what *has* been done; there is much more eagerly waiting *to be* done. Perhaps the single most obvious open problem is that of developing methods to reduce the times of our algorithm from $O(N \lg^k N)$ to $O(N \lg N)$. We saw how presorting could be used to remove one logarithmic factor (in all the algorithms) and certain other techniques that actually achieved $O(N \lg N)$ time, such as the expected analysis of Section 2.2 and the "good" cut planes of Sections 3.1 and 3.2. One might hope for similar techniques of broader applicability to increase the speed of our algorithms even more. Another area which we just barely scratched (in Section 2.2) was the *expected* analysis of these algorithms—experience indicates that our worst-case analyses are terribly pessimistic. A more specific open problem is to use this method to solve the nearest neighbor searching problem in $k$-space; one might also hope to use the method to give a fast algorithm for constructing minimal spanning trees. Although it has already been used to solve a number of research problems, much work remains to be done before we can "write the final chapter" on multidimensional divide-and-conquer.

## 5. Conclusions

In this section we summarize the contributions contained in this paper, but before we do so we will briefly review the results of Sections 2 and 3. Those sections dealt with two basic classes of problems: all-points problems and searching problems. For five all-points problems of $N$ points in $k$-space we saw algorithms with running time of $O(N \lg^{k-1} N)$; for certain of these problems we saw how to reduce their running time even more. For four searching problems we developed data structures that could be built on $O(N \lg^{k-1} N)$ time, used $O(N \lg^{k-1} N)$ space, and could be searched in $O(\lg^k N)$ time. Both the all-points algorithms and the searching data structures were constructed by using one paradigm: multidimensional divide-and-conquer. All of the all-points problems that we saw have $\Omega(N \lg N)$ lower bounds and all of the searching problems have $\Omega(\lg N)$ lower bounds; the algorithms and data structures that we have seen are therefore within a constant factor of optimal (some for only small $k$, others for any $k$).

The contributions of this paper can be described at two levels. At the first level we have seen a number of particular results of both theoretical and practical interest. The algorithms of Sections 2 and 3 are currently the best algorithms known for their respective problems in terms of asymptotic running time. They (or their variants) can also be implemented efficiently for problems of "practical" size, and several are currently in use in software packages. At a second level this paper contains contributions in the study of a particular *algorithmic paradigm*, multidimensional divide-and-conquer. This paradigm is essentially a general algorithmic *schema*, which we *instantiated* to yield a number of particular algorithms and data structures. The study of this paradigm as a paradigm has three distinct advantages. First, we have been able to present a large number of results rather succinctly. Second, advances made in one problem can be applied to other problems (indeed, once one search structure was discovered, all the rest came about quite rapidly). Third, this paradigm has been used to discover new algorithms and data structures. The author has (on occasion) consciously applied this paradigm in the attempt to solve research problems. Although the paradigm often did not yield fruit, the ECDF, range searching, and nearest neighbor problems were solved in exactly this way.

In this paper the author has tried to communicate some of the flavor of the *process* of algorithm design and analysis, in addition to the nicely packaged results. It is his hope that the reader takes away from this study not only a set of algorithms and data structures, but also a feeling for how these objects came into being.

*Acknowledgments.* The presentation of this paper has been greatly improved by the careful comments of J. McDermott, J. Traub, B. Weide, and an anonymous referee. I am also happy to acknowledge assistance received as I was working on the algorithms described in this paper. D. Stanat was an excellent thesis advisor, and M. Shamos was a constant source of problems and algorithmic insight.

Received 9/78; revised 6/79; accepted 1/80

**References**

1. Aho, AV., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. Bentley, J.L. Multidimensional binary search trees used for associative searching. *Comm. ACM 18,* 9 (Sept. 1975), 509–517.
3. Bentley, J.L. Divide and conquer algorithms for closest point problems in multidimensional space. Unpublished Ph.D. dissertation, Univ. of North Carolina, Chapel Hill, N.C., 1976.
4. Bentley, J.L. Decomposable searching problems. *Inform. Proc. Letters 8,* 5 (June 1979), 244–251.
5. Bentley, J.L., and Friedman, J.H. Algorithms and data structures for range searching. *Comptng. Surv. 11,* 4 (Dec. 1979), 397–409.
6. Bentley, J.L., Kung, H.T., Schkolnick, M., and Thompson, C.D. On the average number of maxima in a set of vectors and applications. *J. ACM 25,* 4 (Oct. 1978), 536–543.
7. Bentley, J.L., and Maurer, H.A. Efficient worst-case data structures for range searching. To appear in *Acta Informatica* (1980).
8. Bentley, J.L., and Shamos, M.I. Divide and conquer in multidimensional space. In Proc. ACM Symp. Theory of Comptng., May 1976, pp. 220–230.
9. Bentley, J.L., and Shamos, M.I. A problem in multivariate statistics: Algorithm, data structure, and applications. In Proc. 15th Allerton Conf. Communication, Control, and Comptng., Sept. 1977, pp. 193–201.
10. Blum, M., et al. Time bounds for selection. *J. Comptr. Syst. Sci. 7,* 4 (Aug. 1972), 448–461.
11. Dobkin, D., and Lipton, R.J. Multidimensional search problems. *SIAM J. Comptng. 5,* 2 (June 1976), 181–186.
12. Fredman, M. A near optimal data structure for a type of range query problem. In Proc. 11th ACM Symp. Theory of Comptng., April 1979, pp. 62–66.
13. Fredman, M., and Weide, B.W. On the complexity of computing the measure of $\cup [a_i, b_i]$. *Comm. ACM 21,* 7 (July 1978), 540–544.
14. Friedman, J.H. A recursive partitioning decision rule for nonparametric classification. *IEEE Trans. Comptrs. C-26,* 4 (April 1977), 404–408.
15. Friedman, J. H. A nested partitioning algorithm for numerical multiple integration. Rep. SLAC-PUB-2006, Stanford Linear Accelerator Ctr., 1978.
16. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
17. Kung, H.T., Luccio, F., and Preparata, F.P. On finding the maxima of a set of vectors. *J. ACM 22,* 4 (Oct. 1975), 469–476.
18. Lee, D.T., and Wong, C.K. Qintary trees: A file structure for multidimensional database systems. To appear in *ACM Trans. Database Syst.*
19. Lipton, R., and Tarjan, R.E. Applications of a planar separator theorem. In Proc. 18th Symp. Foundations of Comptr. Sci., Oct. 1977, pp. 162–170.
20. Lueker, G. A data structure for orthogonal range queries. In Proc. 19th Symp. Foundations of Comptr. Sci., Oct. 1978, pp. 28–34.
21. Monier, L. Combinatorial solutions of multidimensional divide-and-conquer recurrences. To appear in the *J. of Algorithms.*
22. Murray, J.A. *Lieutenant, Police Department—The Complete Study Guide for Scoring High* (4th ed.). Arco, New York, 1966, p. 184, question 3.
23. Reddy, D.R., and Rubin, S. Representation of three-dimensional objects. Carnegie-Mellon Comptr. Sci. Rep. CMU-CS-78-113, Carnegie-Mellon Univ., Pittsburgh, Pa., 1978.
24. Saxe, J.B. On the number of range queries in $k$-space. *Discrete Appl. Math. 1,* 3 (Nov. 1979), 217–225.
25. Shamos, M.I. Computational geometry. Unpublished Ph.D. dissertation, Yale Univ., New Haven, Conn., 1978.
26. Shamos, M.I. Geometric complexity. In Proc. 7th ACM Symp. Theory of Comptng., May 1975, pp. 224–233.
27. Weide, B. A survey of analysis techniques for discrete algorithms. *Comptng. Surv. 9,* 4 (Dec. 1977), 291–313.
28. Willard, D.E. New data structures for orthogonal queries. Harvard Aiken Comptr. Lab. Rep., Cambridge, Mass., 1978.
29. Yao, F.F. On finding the maximal elements in a set of planar vectors. Rep. UIUCDCS-R-74-667, Comptr. Sci. Dept., Univ. of Illinois, Urbana, July 1974.

Programming Techniques — R. Rivest, Editor

# A Unifying Look at Data Structures

Jean Vuillemin
University of Paris–South

**Examples of fruitful interaction between geometrical combinatorics and the design and analysis of algorithms are presented. A demonstration is given of the way in which a simple geometrical construction yields new and efficient algorithms for various searching and list manipulation problems.**

**Key Words and Phrases: data structures, dictionaries, linear list, search, merge, permutations, analysis of algorithms**

**CR Categories: 4.34, 5.24, 5.25, 5.32, 8.1**

## 1. Introduction

Whenever two combinatorial structures are counted by the same number, there exist bijections (one–one mappings) between the two structures. One goal of geometrical combinatorics (see, for example, Foata and Schutzenberger [7]) is to explicitly construct such bijections. This is bringing the field very close to computer science: One can regard combinatorial representations of remarkable numbers as equivalent data structures; explicit bijections between such representations provide coding and decoding algorithms between the structures. Earlier investigations along these lines are reported in Françon et al. [10] and Flajolet et al. [6].

This paper should be regarded as an introduction to using methods of geometrical combinatorics in the field of algorithm design and analysis. For this purpose, we consider representation of $n!$ as a running example and demonstrate how we are led to discovering new and efficient data structures and algorithms for solving various data manipulation problems.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This work was supported by the National Center for Scientific Research (CNRS), Paris, under Grant 3941.

Author's address: J. Vuillemin, Laboratory for Information Research, Building 490, University of Paris–South, 91405 Orsay, France.

© 1980 ACM 0001-0782/80/0400-0229 $00.75.