A Survey of Adaptive Sorting Algorithms

Vladimir Estivill-Castro
LANIA
Rébsamen 80
Xalapa, Veracruz 91000, México

Derick Wood
Department of Computer Science
University of Western Ontario
London, Ontario N6A 5B7, Canada

The design and analysis of adaptive sorting algorithms has made important contributions to both theory and practice. The main contributions from the theoretical point of view are: the description of the complexity of a sorting algorithm not only in terms of the size of a problem instance but also in terms of the disorder of the given problem instance; the establishment of new relationships among measures of disorder; the introduction of new sorting algorithms that take advantage of the existing order in the input sequence; and, the proofs that several of the new sorting algorithms achieve maximal (optimal) adaptivity with respect to several measures of disorder. The main contributions from the practical point of view are: the demonstration that several algorithms currently in use are adaptive; and, the development of new algorithms, similar to currently used algorithms, that perform competitively on random sequences and are significantly faster on nearly sorted sequences. In this survey, we present the basic notions and concepts of adaptive sorting and the state-of-the-art of adaptive sorting algorithms.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introduction and Survey; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems — *Sorting and Searching*; E.5 [**Data**]: Files — *Sorting/searching*; G.3 [**Mathematics of Computing**]: Probability and Statistics — *Probabilistic algorithms*; E.2 [**Data Storage Representation**]: Composite structures, linked representations.

General Terms: Algorithms, Theory.

Additional Key Words and Phrases: Adaptive sorting algorithms, Comparison trees, Measures of disorder, Nearly sorted sequences, Randomized algorithms.

CONTENTS

```
procedure Straight Insertion Sort(X, n);
X[0] := −∞;
for j := 2 to n do
begin i := j − 1;
        t := X[j];
        while t < X[i] do
        begin X[i + 1] := X[i];
                i := i − 1;
        end;
        X[i + 1] := t;
end;
```

Figure 1: A pseudo-Pascal implementation of *Straight Insertion Sort*.

INTRODUCTION

Sorting is the computational process of rearranging a given sequence of items into ascending or descending order [Knuth 1973]. After a first course in Data Structures or Algorithms, the impression is that, for comparison-based algorithms, we cannot do better than optimal (so called $n \log n$) sorting algorithms. However, when the sorting algorithm takes advantage of existing order in the input, the time taken by the algorithm to sort is a smoothly growing function of the size of the sequence and the disorder in the sequence. In this case, we say that the algorithm is **adaptive** [Mehlhorn 1984]. Adaptive sorting algorithms are attractive because nearly sorted sequences are common in practice [Knuth 1973, page 339; Sedgewick 1980, page 126; Mehlhorn 1984, page 54]; thus, we have the possibility of improving on algorithms that are oblivious to the existing order in the input.

Straight Insertion Sort (see Figure 1) is a classic example of an adaptive sorting algorithm. In *Straight Insertion Sort* we scan the input once, from left to right, repeatedly finding the correct position of the current item, inserting it into an array of previously sorted items. The closer a sequence is to being sorted, the less is the time taken by *Straight Insertion Sort* to sort it. In fact, the performance of *Straight Insertion Sort* can be described in terms of the size of the input and the number of inversions in the input. More formally, let $Inv(X)$ denote the **number of inversions** in a sequence $X = \langle x_1, \ldots, x_n \rangle$, where (i, j) is an **inversion** if $i < j$ and $x_i > x_j$. Intuitively, $Inv(X)$ measures disorder, since its value is minimized when X is sorted and its value depends on only the relative order of the elements in X. For a sequence X with n elements, *Straight Insertion Sort* performs exactly $Inv(X) + n - 1$ comparisons and $Inv(X) + 2n - 1$ data moves, and uses constant extra space. We say that *Straight Insertion Sort* is an adaptive algorithm with respect to Inv. Empirical evidence confirms that *Straight Insertion Sort* is efficient for both small sequences and nearly sorted sequences [Cook and Kim 1980; Knuth 1973].

In this survey we present the basic notions and concepts of adaptive sorting, and the state-of-the-art of adaptive sorting algorithms. The initial motivation for adaptive sorting algorithms is the

high frequency with which nearly sorted inputs occur in practical applications. Sorting a nearly sorted sequence should require less work than sorting a randomly permuted sequence. For example, if we know that, apart from one element in the wrong position, the input sequence is sorted, then the sequence can be sorted by scanning the sequence to find the misplaced element and performing a binary search to find its correct position. In this case, sorting takes linear time without resorting to the full power of a general sorting algorithm. In the general case, we do not know anything in advance about the existing order in the input, but we want the sorting algorithm to "discover" existing order and profit from it to accelerate the sorting.

We focus our attention on comparison-based sorting algorithms for sequential models of computation. The reason for this choice is that most of the work on adaptive sorting falls within this framework. However, in an effort to be comprehensive, we mention briefly in Sections 3 and 4 results that do not fit into this framework.

> **Notational Conventions:** The cardinality of a set S is denoted by $\|S\|$ and the length of a sequence X is denoted by $|X|$. We assume throughout the survey only that the elements to be sorted belong to a total order; however, we always use integer examples. The collection of all finite sequences of distinct nonnegative integers is denoted by $N^{<N}$. The base-2 logarithm is denoted by log.

I.1 Historical Background

As early as 1958, Burge observed that the best algorithm for a sorting problem depends on the order already in the data and he proposed measures of disorder to evaluate the extent to which elements are already sorted [Burge 1958]. Initial research on adaptive sorting algorithms concentrated on internal sorting algorithms and followed three directions. First, during 1977–1980, data structures were designed to represent sorted sequences and provide fast insertions for elements whose position was close to the position of previous insertions. Different adaptive variants of insertion-based sorting algorithms were obtained by using these data structures to maintain the sorted portion of the sequence [Brown and Tarjan 1980; Guibas et al. 1977; Mehlhorn 1979]. Second, Cook and Kim in 1980 and Wainwright in 1985 performed empirical studies [Cook and Kim 1980; Wainwright 1985]. Third, Dijkstra introduced *Smooth Sort* in 1982 [Dijkstra 1982]. The intuitive ideas arising from these efforts were formalized by Mannila who, in 1985, introduced a formal framework for the analysis of adaptive sorting algorithms in the worst case [Mannila 1985b]. This framework consists of two parts: First, the introduction of the concept of a measure of presortedness as a function that evaluates disorder; Second, the concept of optimal adaptivity of an algorithm with respect to a measure of presortedness. Because of the developments of Petersson [Petersson 1990] and Estivill-Castro [Estivill-Castro 1990], adaptive sorting algorithms are now well understood.

We evaluate the disorder in a sequence by a real-valued function that we call a **measure of disorder;** it is a function from $N^{<N}$ to $\Re$. The measure of efficiency of a sorting algorithm is the **number of comparisons** it performs. The number of comparisons provides not only a reasonable estimate of the relative time requirements of all implementations but also enables lower bounds to be obtained under the decision-tree model of computation. Mannila defined a sorting algorithm to

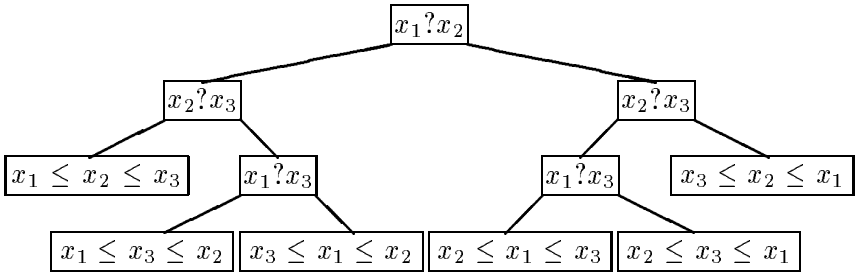

Figure 2: A comparison tree for sorting the sequence x_1, x_2, x_3.

be **optimally adaptive** (or maximally adaptive) with respect to a measure of disorder if it takes a number of comparisons that is within a constant factor of the lower bound.

A general lower bound for sorting with respect to a measure M of disorder is obtained as follows. Let $below(z, n, M)$ denote the set of permutations of n items with no more disorder than z with respect to M; then,

$$below(z, n, M) \quad = \quad \{Y \in N^{<N} \mid |Y| = n \text{ and } M(Y) \le z\}.$$

Consider sorting algorithms having as input not only a sequence X of length n but also an upper bound z on the value of $M(X)$. We analyze the comparison or decision trees for such algorithms [Knuth 1973, page 182; [Mehlhorn 1984, page 68]; we give an example of a comparison tree in Figure 2. These trees are binary and must have at least $\|\text{below}(z, n, M)\|$ leaves, the number of possible inputs; so the height of a tree is at least $\Omega(\log \|\text{below}(z, n, M)\|)$. Therefore, such algorithms take at least $\Omega(\log \|\text{below}(z, n, M)\|)$ time. If an upper bound on $M(X)$ is not available, the algorithm cannot be faster than one that is provided with the best possible bound $z = M(X)$. Finally, since testing for sortedness requires linear time, a sorting algorithm should be allowed at least a linear number of comparisons.

The following definition captures the notion of optimal adaptivity in the worst-case [Mannila 1985b].

Definition I. 1 *Let M be a measure of disorder and S be a sorting algorithm which uses $T_S(X)$ comparisons on input X. We say that S is* **optimal** *with respect to M (or M-**optimal**) if, for some $c > 0$, we have, for all $X \in N^{<N}$,*

$$T_S(X) \quad \le \quad c \cdot \max\{|X|, \log \|below(M(X), |X|, M)\|\}.$$

An example of a sorting algorithm that is optimal with respect to inversions is Mannila's *Local Insertion Sort* [Mannila 1985b]; it is an insertion-based sorting algorithm. On the i-th pass, $x_1, \ldots, x_{i-1}$ have been sorted and x_i is to be inserted into its correct position. Mannila uses a level-linked balanced (a, b)-tree with a finger to represent the initial sorted segment. Let $d_i(X)$ denote the distance from the $(i-1)$-th insertion point to the i-th insertion point; that is,

$d_i(X) = \|\{j \mid 1 \leq j < i \text{ and } \quad (x_{i-1} < x_j < x_i \text{ or } \quad x_i < x_j < x_{i-1})\}\|$. Level-linked balanced (a, b)-trees support the search for x_i's position in $c(1 + \log[d_i(X) + 1])$ amortized time, where c is a constant [Brown and Tarjan 1980]. We use Mannila's argument to show that *Local Insertion Sort* is Inv-optimal. Let $I_i(X)$ denote the number of inversion pairs in X, where x_i is the second element in the inversion pair; that is,

$$I_i(X) = \|\{j \mid 1 \leq j < i \text{ and } x_j > x_i\}\|.$$

Now $\sum_{i=1}^{|X|} I_i(X) = Inv(X)$ and it is not hard to see that

$$d_i(X) \leq \max\{I_i(X), I_{i-1}(X)\}. \tag{1}$$

Thus, $\sum_{i=1}^{|X|} \log[d_i(X) + 1] \leq 2 \sum_{i=1}^{|X|} \log[I_i(X) + 1]$. Since the algorithm inserts $|X|$ elements into an initially empty tree, the amortized bound gives rise to a worst-case upper bound on the time taken by *Local Insertion Sort* to sort X. Using Equation (1), properties of the logarithm, and the fact that the geometric mean is never larger that the arithmetic mean, we obtain the following derivation.

$$\begin{aligned} c\sum_{i=1}^{|X|}(1 + \log[d_i(X) + 1]) &= c|X| + c\log\left[\prod_{i=1}^{n}(d_i(X) + 1)\right] \\ &= c|X| + 2c|X|\log\left(\prod_{i=1}^{|X|}[I_i(X) + 1]\right)^{1/|X|} \\ &\leq c|X|\left(1 + 2\log\left[\frac{Inv(X)}{|X|} + 1\right]\right). \end{aligned}$$

Thus, the time taken by *Local Insertion Sort* is a smooth function of $|X|$ and the disorder of X. The more inversions there are in X, the more work is performed by *Local Insertion Sort*. Guibas and associates obtained the following lower bound for adaptive sorting with respect to inversions [Guibas et al. 1977]:

$$\log \|below(Inv(X), |X|, Inv)\| = \Omega(|X| \log[1 + Inv(X)/|X|]);$$

thus, *Local Insertion Sort* is Inv-optimal.

Note that, if a sorting algorithm is optimal with respect to a measure M, then it is optimal in the worst case; in other words, *optimally adaptive algorithms are optimal.*

I.2 Examples of measures of disorder

Although the number of inversions is an important measure of disorder, it does not capture all types of disorder. For example, the sequence

$$W_0 = \langle \lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n, 1, \ldots, \lfloor n/2 \rfloor \rangle$$

has a quadratic number of inversions although it consists of two ascending runs; therefore, it is nearly sorted in a sense not captured by inversions.

We now describe ten other measures of disorder that have been used in the study of adaptivity.
1. *Dis*. We may consider that, in terms of the disorder it represents, an inversion pair in which the elements are far apart is more significant than an inversion pair in which the elements are close together. One measure of this kind is *Dis* that is defined as the largest distance determined by an inversion [Estivill-Castro and Wood 1989a]. For example, let

$$W_1 = \langle 6, 2, 4, 7, 3, 1, 9, 5, 10, 8 \rangle;$$

then $(6, 5)$ is an inversion whose elements are farthest apart and $Dis(W_1) = 7$.
2. *Max*. We may also consider that local disorder is not as important as global disorder. For example, in a library, if a book is one slot away from its correct position, we are still able to find it, since the call number will get us close enough to where it is; however, a book that is very far from its correct position is difficult to find. A measure that evaluates this type of disorder is *Max* defined as the largest distance an element must travel to reach its sorted position [Estivill-Castro and Wood 1989a]. In W_1, 1 must travel five positions; thus, $Max(W_1) = 5$.
3. *Exc*. The number of operations required to rearrange a sequence into sorted order may be our primary concern. A simple operation to rearrange the elements in a sequence is an **exchange;** thus, we may define *Exc*as the minimum number of exchanges required to sort a sequence [Mannila 1985b]. It is impossible to sort the example sequence W_1 with fewer than seven exchanges; thus, $Exc(W_1) = 7$.
4. *Rem*. We may consider that disorder is produced by the incorrect insertion of some records, and evaluate disorder by *Rem*, defined as the minimum number of elements that must be removed to obtain a sorted subsequence [Knuth 1973, Section 5.2.1, Exercise 39]. By removing 5 elements from W_1 we obtain the sorted subsequence $\langle 2, 4, 7, 9, 10 \rangle$. Removing fewer than 5 elements cannot give a sorted subsequence; thus, $Rem(W_1) = 5$. We can also define $Rem(X)$ as $|X| - Las(X)$, where $Las(X)$ is the length of a largest ascending subsequence; that is, $Las(X) = \max\{t \mid \exists i(1), i(2), \ldots, i(t) \mid 1 \leq i(1) < i(2) < \ldots < i(t) \leq n \text{ and} x_{i(1)} < \cdots < x_{i(t)}\}$.
5. *Runs*. Since ascending runs represent sorted portions of the input, and a sorted sequence has the minimum number of runs, it is natural to define a measure that is the number of ascending runs. In order to make the function zero for a sequence with no disorder, we define *Runs* as the number of boundaries between runs. These boundaries are called **step-downs** [Knuth 1973, page 161], at which a smaller element follows a larger one. For example, $W_1 = \langle 6 \downarrow 2, 4, 7 \downarrow 3 \downarrow 1, 9 \downarrow 5, 10 \downarrow 8 \rangle$ has $Runs(W_1) = 5$.
6. *SUS*. A natural generalization of *Runs* is the minimum number of ascending subsequences into which we can partition the given sequence. We denote this measure by *SUS* for **Shuffled Up-Sequences** [Levcopoulos and Petersson 1990].
7. *SMS.SUS* can be generalized further by defining *SMS*(X) (for **Shuffled Monotone Subsequence**) as the minimum number of monotone (ascending or descending) subsequences into which we can partition the given sequence [Levcopoulos and Petersson 1990]. For example,

$$W_2 = \langle 6, 5, 8, 7, 10, 9, 12, 11, 4, 3, 2 \rangle$$

has $Runs(W_2) = 7$, whereas

$$\begin{aligned} SUS(W_2) &= \|\{\langle 6,8,10,12\rangle, \langle 5,7,9,11\rangle, \langle 4\rangle, \langle 3\rangle, \langle 2\rangle\}\| \\ &= 5 \end{aligned}$$

and

$$\begin{aligned} SMS(W_2) &= \|\{\langle 6,8,10,12\rangle, \langle 5,7,9,11\rangle, \langle 4,3,2\rangle\}\| \\ &= 3. \end{aligned}$$

8. *Enc*. Several researchers have designed sorting algorithms, determined on which permutations they do well, and then defined new measures as the result of their investigations. Skiena proposed the measure $Enc(X)$, defined as the number of sorted lists constructed by *Melsort* when applied to X [Skiena 1988a].
9. *Osc*. Levcopoulos and Petersson defined the measure *Osc* from a study of *Heapsort* [Levcopoulos and Petersson 1989a]. The measure *Osc* evaluates, in some sense, the "oscillation" of large and small elements in a given sequence.
10. *Reg*. Moffat and Petersson defined a new measure called *Reg*; any *Reg*-optimal sorting algorithm is optimally adaptive with respect to the other measures [Moffat and Petersson 1991; Moffat and Petersson 1992].

Are these measures different? From the algorithmic point of view, if two measures M_1 and M_2 partition the set of permutations into exactly the same classes of *below* sets, then any algorithm that is M_1-optimal is also M_2-optimal and conversely. Therefore, the measures are indistinguishable. We capture these ideas in the following definition.

Definition I. 2 *Let $M_1, M_2 : N^{<N} \rightarrow \Re$ be two measures of disorder. We say that*

1. *M_1 is* **algorithmically finer** *than M_2 (denoted $M_1 \leq_{alg} M_2$) if and only if any M_1-optimal algorithm is also M_2-optimal.*

2. *M_1 and M_2 are* **algorithmically equivalent** *(denoted $M_1 =_{alg} M_2$) if and only if $M_1 \leq_{alg} M_2$ and $M_1 \leq_{alg} M_2$.*

Figure 3 displays the partial ordering, with respect to $\leq_{alg}$, of the measures that we have defined.

Using a complex data structure called a **historical search tree,** Levcopolous and Petersson designed an insertion-based sorting algorithm that makes an optimal number of comparisons with respect to *Reg*, but does not make an optimal number of data moves; therefore, it takes $\Omega(\log Reg(X) + |X| \log\log |X|)$ time and is not *Reg*-optimal Moffat and Petersson 1991; Moffat and Petersson 1992]. Their work constitutes an important theoretical contribution and raises the question of the existence of a universal measure U that is finer than all measures. However, even if such a universal measure U exists, there may be neither a U-optimal algorithm nor a practical implementation [Mannila 1985a].

A sequence is **nearly sorted** if either it requires few operations to sort it or it was created from a sorted sequence with a few perturbations. Each measure that we have defined illustrates

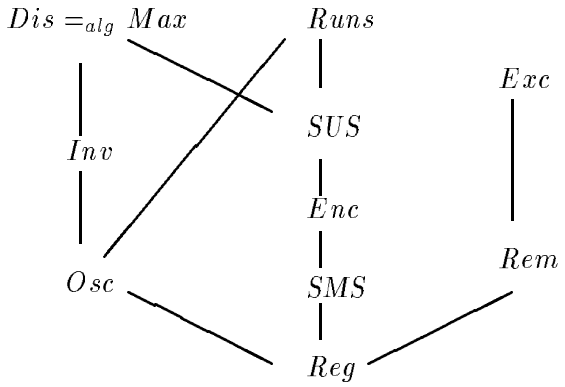

Figure 3: The partial order of the 11 measures of disorder. The ordering with respect to $\leq_{alg}$ is up the page; for example, Inv-optimality implies Dis-optimality.

that disorder can be measured in many different ways, each is important because they formalize the intuitive notion of a nearly sorted sequence. For ease of reference, Table 1 presents tight lower bounds for optimality with respect to 11 different measures of disorder. The lower bound for Inv was established by Guibas and his coworkers [Guibas et al. 1977]; the lower bounds for Rem and $Runs$ were established by Mannila [Mannila 1985b]; the lower bounds for Enc, Exc, Osc, SMS, and SUS were established by Levcopoulos and Petersson [Levcopoulos and Petersson 1989; Levcopoulos and Petersson 1990; Levcopoulos and Petersson 1991b]; the lower bounds for Dis and Max were established by Estivill-Castro and Wood [Estivill-Castro and Wood 1989]; and the lower bound for Reg was established by Moffat and Petersson [Moffat and Petersson 1991; Moffat and Petersson 1992].

Measures of disorder are a fundamental concept, but little can be said about them because of their generality. Moreover, measures of disorder appear in contexts other than adaptive sorting. In Statistics, measures of disarray or right invariant metrics are used to obtain coefficients of correlation for rank correlation methods. These coefficients of correlation are used to test the significance of observed rank correlations. For example, Inv appears in the definition of Kendall's τ, the most popular coefficient of correlation [Kendall 1970]. Right invariant metrics have applications in cryptography where they are used to build tests for random permutations [Sloane 1982]. Other applications of measures of disorder include the study of error-sensitivity of sorting algorithms [Islam and Lakshman 1990]. The study of the relationships between right invariant metrics and measures of disorder has resulted in general algorithms for the pseudo-random generation of nearly sorted sequences with respect to a measure of disorder [Estivill-Castro 1990].

Table 1: The worst-case lower bounds for different measures of disorder.

M	Lower bound: $\log \|below(M(X), \|X\|, M)\|$
Dis	$\Omega(\|X\|\,(1+\log[Dis(X)+1]))$
Exc	$\Omega(\|X\| + Exc(X)\log[Exc(X)+1])$
Enc	$\Omega(\|X\|\,(1+\log[Enc(X)+1]))$
Inv	$\Omega(\|X\|\,\left(1+\log\left\lfloor\frac{Inv(X)}{\|X\|}+1\right\rfloor\right))$
Max	$\Omega(\|X\|\,(1+\log[Max(X)+1]))$
Osc	$\Omega(\|X\|\,\left(1+\log\left\lfloor\frac{Osc(X)}{\|X\|}+1\right\rfloor\right))$
Reg	$\Omega(\|X\| + \log[Reg(X)+1])$
Rem	$\Omega(\|X\| + Rem(X)\log[Rem(X)+1])$
$Runs$	$\Omega(\|X\|\,(1+\log[Runs(X)+1]))$
SMS	$\Omega(\|X\|\,(1+\log[SMS(X)+1]))$
SUS	$\Omega(\|X\|\,(1+\log[SUS(X)+1]))$

I.3 Organization of the paper

Most studies of adaptive sorting have had as their goal a guarantee of optimal worst-case performance. This approach has resulted in a large family of adaptive sorting algorithms and elegant theoretical results. Section 1 studies adaptivity from the point of view of worst-case analysis. Rather than presenting a list of sorting algorithms and their adaptivity properties, we present well-developed design tools and generic sorting algorithms. The adaptivity results are presented in a general form with respect to an abstract measure of disorder. From this general scheme, specific algorithms are obtained as particular applications of the design tools. The adaptivity properties of particular sorting algorithms appear as simple consequences of the general result.

Worst-case analysis requires us to guarantee best possible performance for all inputs even if some cases are unlikely to occur. This guarantee can result in complex sorting algorithms that use sophisticated data structures with large overhead. We suggest an alternative approach: Design algorithms that are adaptive in the expected case. Section 2 studies adaptivity from the point of view of expected-case analysis.

In Section 3, we discuss external sorting algorithms; that is, algorithms that are used to sort sequences that cannot be stored entirely in main memory. Current technology allows the sorting of large files to be performed on disk drives [Salzberg 1988; Salzberg 1989]. Since *Replacement Selection* allows full overlapping of I/O operations during initial run creation for external sorting and it creates runs that are larger than the available memory, *Replacement Selection* is a fundamental practical tool and is almost the only algorithm in use. We present results that demonstrate the adaptive performance of *Replacement Selection*.

Finally, in Section 4, we discuss some open problems and some directions for further research. In particular we survey the efforts that have been made to develop adaptive in-place sorting algorithms (that is, with constant extra space) and adaptive sorting algorithms for sequences with repeated keys. We also mention the efforts that have been made to define adaptivity of parallel sorting algorithms.

1 WORST-CASE (INTERNAL) ADAPTIVE SORTING ALGORITHMS

It is certainly useful to design an adaptive sorting algorithm for a particular measure; however, the diversity of measures of disorder suggests that an algorithm that is adaptive to several measures is much more useful, since in a general setting, the type of disorder in the input is unknown. Initially, many discoveries were made by studying a particular measure, but increasingly researchers have focused their attention on sorting algorithms that are adaptive with respect to several measures. Unfortunately, it appears that we cannot increase the number of measures without adding complex machinery with large overhead that renders the algorithm impractical. The trade-off between the number of measures and a practical implementation is central to the selection of an adaptive sorting method. We illustrate this trade-off by presenting progressively more sophisticated algorithms that are adaptive with respect to more measures, although we favor adaptive algorithms that are practical.

1.1 Generic Sorting Algorithms

We present a generic adaptive sorting algorithm that enables us to focus attention on the combinatorial properties of measures of disorder rather than the combinatorial properties of the algorithm [Estivill-Castro and Wood 1991b; Estivill-Castro and Wood 1992b]. Using it, we obtain a practical adaptive sorting algorithm, optimal with respect to several important measures of disorder and smoothly adaptive for other common measures.

The structure of the generic sorting algorithm, *Generic Sort*, should not be surprising. It uses divide and conquer and balancing to ensure $O(n \log n)$ worst-case time. What is novel, however, is that we can establish adaptability with respect to a measure M of disorder by ensuring that the method of division, the **division protocol,** satisfies three requirements. First, division should take linear time in the worst case; second, the sizes of the sequences that it generates (and that are not sorted) should be almost the same; and, third, it should not introduce too much disorder. We formalize what is meant by too much disorder in Theorem 1.1. The generic sorting algorithm has the form shown in Figure 4. If a sequence cannot be divided into smaller sequences or is so close to sorted order that it can be sorted in linear time by an alternative sorting algorithm, then it is considered to be simple. *Generic Sort* leaves us with two problems: What are reasonable division protocols and what is meant by too much disorder? Three example division protocols are:

- *Straight division.* Divide a sequence $X = \langle x_1, \ldots, x_n \rangle$ into two halves $X_L = X_{1..\lfloor |X|/2 \rfloor}$ and $X_R = X_{1+\lfloor |X|/2 \rfloor..|X|}$.

Generic Sort(X)

- X is sorted. Terminate.
- X is simple. Sort it using an alternative sorting algorithm for simple sequences.
- X is neither sorted nor simple.
  - Apply a division protocol to divide X into at least $s \geq 2$ disjoint sequences.
  - Recursively sort the sequences using *Generic Sort*.
  - Merge the sorted sequences to give X in sorted order.

Figure 4: *Generic Sort*. The first generic sorting algorithm.

- *Odd-Even division*. Divide X into the subsequence X_{even} of elements in even positions and the subsequence X_{odd} of elements in odd positions.
- *Median division*. Divide X into the sequence of all elements smaller than the median of X and the sequence of all elements larger than the median of X (denoted by $X_<$ and $X_>$ respectively).

Observe that each of these division protocols satisfies the time and size requirements. The notion of too much disorder is made precise in the following theorem [Estivill-Castro and Wood 1991b; Estivill-Castro and Wood 1992b].

Theorem 1.1 *Let M be a measure of disorder such that $M(X) = 0$ implies X is simple, $D \in \Re$ and $s \in N$ be constants such that $0 \leq D < 2$ and $s > 1$, and DP be a linear-time division protocol that divides a sequence into s sequences of almost equal sizes.*

1. *Generic Sort is worst-case optimal; it takes $O(|X| \log |X|)$ time in the worst case.*
2. *If there is an $n_0 \in N$ such that, for all sequences X with $|X| > n_0$, DP satisfies*

$$\sum_{j=1}^{s} M(j\text{-}thsequence) \leq D\lfloor s/2 \rfloor M(X),$$

then Generic Sort is adaptive to the measure M; it takes

$$O(|X|(1 + \log[M(X) + 1]))$$

time in the worst case.

```
Straight Merge Sort(X);
if not sorted(X)
then begin
        Straight Merge Sort(X_{1..|X|/2});
        Straight Merge Sort(X_{1+|X|/2..|X|});
        Merge(X_{1..|X|/2},X_{1+|X|/2..|X|});
end;
```

Figure 5: A pseudo-Pascal implementation of *Straight Merge Sort*.

3. $D < 2$ is necessary.

1.1.1 Applications of *Generic Sort*

Consider the variant of *Mergesort* displayed in Figure 5. The straight-division protocol takes linear time and $Inv(X_{1..\lfloor |X|/2 \rfloor}) + Inv(X_{1+\lfloor |X|/2 \rfloor..|X|})$ accounts for all the inversions except those inversion pairs with one element in $X_{1..\lfloor |X|/2 \rfloor}$ and the other in $X_{1+\lfloor |X|/2 \rfloor..|X|}$. Thus,

$$Inv(X) \geq Inv(X_{1..\lfloor |X|/2 \rfloor}) + Inv(X_{1+\lfloor |X|/2 \rfloor..|X|}).$$

Similarly,

$$Rem(X) \quad \geq \quad Rem(X_{1..\lfloor |X|/2 \rfloor}) + Rem(X_{1+\lfloor |X|/2 \rfloor..|X|}).$$

Therefore, *Straight Merge Sort* is adaptive with respect to Inv and Rem. Now, $Runs(X_{1..\lfloor |X|/2 \rfloor}) + Runs(X_{1+\lfloor |X|/2 \rfloor..|X|})$ accounts for all the step-downs in X except possibly for a step-down between $X_{1..\lfloor |X|/2 \rfloor}$ and $X_{1+\lfloor |X|/2 \rfloor..|X|}$. Thus,

$$Runs(X) \quad \geq \quad Runs(X_{1..\lfloor |X|/2 \rfloor}) + Runs(X_{1+\lfloor |X|/2 \rfloor..|X|})$$

and Theorem 1.1 and the lower bound imply that *Straight Merge Sort* is optimal with respect to $Runs$.

Now, for the measure Dis, we have

$$Dis(X) \geq Dis(X_{1..|X|/2})$$

and

$$Dis(X) \geq Dis(X_{1+|X|/2..|X|};$$

therefore,

$$2Dis(X) \geq Dis(X_{1..|X|/2}) + Dis(X_{1+|X|/2..|X|}).$$

This bound is tight because, for the sequence $W = \langle 2, 1, 4, 3, \ldots \rangle$, we have $Dis(W) = Dis(W_{1..|W|/2}) = Dis(W_{1+|W|/2..|W|}) = 1$. Assume that Theorem 1.1 holds when $D = 2$. This assumption implies

```
Odd-Even Merge Sort(X);
if not sorted(X)
then begin
        Odd-Even Merge Sort(X_even);
        Odd-Even Merge Sort(X_odd);
        Merge(X_even,X_odd);
end;
```

Figure 6: A pseudo-Pascal implementation of *Odd-Even Merge Sort*.

that *Straight Merge Sort* should take $O(|W|[1+\log 2]) = O(|W|)$ time. But clearly, *Straight Merge Sort* requires $\Omega(|W|\log|W|)$ comparisons to sort W. Thus, $D < 2$ is necessary for Theorem 1.1 to hold and *Straight Merge Sort* is not adaptive with respect to Dis.

To construct an algorithm that is adaptive with respect to Dis, we use the odd-even division protocol. It can be shown that $Dis(X)/2 \geq Dis(X_{odd})$ and $Dis(X)/2 \geq Dis(X_{even})$ and immediately we have

$$Dis(X) \geq Dis(X_{even}) + Dis(X_{odd}).$$

Moreover, it is easy to prove that, for any sequence X,

$$Inv(X) \geq Inv(X_{even}) + Inv(X_{odd}),$$

and

$$Rem(X) \geq Rem(X_{even}) + Rem(X_{odd}).$$

Thus, the modified version of *Mergesort* presented in Figure 6 takes fewer comparisons than the minimum of $6|X|(\log[Inv(X)+1]+1)$, $6|X|(\log[Dis(X)+1]+1)$, $6|X|(\log[Rem(X)+1]+1)$ and $6|X|(\log[Max(X)+1]+1)$. In other words, if *Odd-Even Merge Sort* is given a sequence that is nearly sorted with respect to any of the above measures, it will adapt its time requirements accordingly. In fact, for Dis and Max, it is optimal; see Table 1.

As a final application of Theorem 1.1, we combine the two previous algorithms and obtain an algorithm that is adaptive with respect to Inv, Exc, and Rem and optimal with respect to $Runs$, Dis, and Max. We merely combine the two division protocols to give *Odd-Even Straight Merge Sort*; see Figure 7. With the same design tool we have obtained a sorting algorithm that is

- Simple and leads to implementations that are competitive with traditional methods on random sequences
- Adaptive with respect to several measures; thus, it leads to implementations that are faster on nearly sorted inputs

The reader may ask why we do not use a sorting algorithm that is *Reg*-optimal and covers all the measures we have introduced? As we have pointed out, there is no known sorting algorithm

```
Odd-Even Straight Merge Sort(X);
if not sorted(X)
then begin
        O-E S M Sort(X_{even_L});
        O-E S M Sort(X_{odd_L});
        O-E S M Sort(X_{even_L} );
        O-E S M Sort(X_{odd_R} );
        Merge(X_{even_L},X_{odd_L}, X_{even_R},X_{odd_R});
end;
```

Figure 7: A pseudo-Pascal implementation of *Odd-Even Straight Merge Sort*.

that is *Reg*-optimal. The message we are attempting to communicate is that more machinery is required to obtain adaptivity with respect to more measures and we have to be very careful that the overhead that is needed does not outweigh the savings obtained by the gained adaptivity. We hope to give you a feeling for those methods whose overhead is small that result in adaptivity for many measures.

We now present a division protocol that allows us to achieve Exc-, Inv- and Rem-optimality with little increase in the complexity of the code.

1.2 Cook–Kim division

In their empirical studies Cook and Kim chose the measure *Rem* because it

> '...is an easily computed measure that coincides with our intuitive notion [of nearly sorted].'

They designed a division procedure to make *Quickersort* adaptive with respect to *Rem* [Cook and Kim 1980]. The resulting sorting algorithm is called *CKsort*. Later Levcopolous and Petersson observed that **Cook–Kim division** is very powerful, since we can use it to add *Rem* and Exc to the list of measures for which a sorting algorithm is optimal [Levcopoulos and Petersson 1991b].

Let A be any sorting algorithm that takes $O(n \log n)$ comparisons in the worst case. We describe how Cook–Kim division can be used to obtain a sorting algorithm that is *Rem*- and Exc-optimal and also M-optimal, for any measure M for which algorithm A is M-optimal. Let $X = \langle x_1, \ldots, x_n \rangle$ be the sequence to be sorted. Cook–Kim division divides X into two subsequences such that one of the subsequences is sorted and the other has length at most $2Rem(X)$. We sort the unsorted subsequence with algorithm A and merge the resulting sequences, in linear time, to obtain a sorted sequence. In total, Cook–Kim division adds a linear-time term to the time taken by algorithm A. The division procedure is defined as follows. Initially, x_1 is the only element in the sorted subsequence. Now, examine x_i, for $i = 2, \ldots, |X|$. If x_i is larger than the last element in the sorted

subsequence, then x_i is appended to the sorted subsequence; otherwise, the last element in the sorted sequence and x_i are placed in a second subsequence and the first sorted subsequence shrinks.

For example, Cook–Kim division can be combined with the test for sortedness in *Odd-Even Straight Merge Sort* to obtain an algorithm that is optimal with respect to Dis, Exc, Max, Rem, and $Runs$. Moreover, Levcopoulos and Petersson generalized Cook–Kim division to obtain a division protocol that satisfies the linear-time and equal-size requirements of *Generic Sort* [Levcopoulos and Petersson 1991b]. They demonstrated that the resulting algorithm is Inv- and Rem-optimal. We now present their division protocol.

> *LP division protocol.* Given a sequence $X = \langle x_1, \ldots, x_n \rangle$, initially place x_1 in a sequence S and create two empty sequences X_L and X_G. Now, examine x_i, for $i = 2, \ldots, |X|$. If x_i is larger than the last element in S, append x_i to S; otherwise append x_i to X_L and remove the last element of S and append it to X_G. We obtain a sorted sequence S and two sequences X_G and X_L of the same length.

When we combine *straight division* with *Odd-Even division* and *LP division* in *Generic Sort* we obtain an algorithm which we call *LP Odd-Even Straight Merge Sort* that is Exc-, Dis-, Inv-, Rem-, and $Runs$-optimal. Harris [Harris 1981] has shown that when *Natural Mergesort* is implemented using linked lists, as suggested by Knuth [Knuth 1973, Section 5.2, Ex. 12 and Section 5.2, Ex. 12], it is a competitive (with respect to comparisons and data movements) adaptive sorting algorithm that behaves well on random sequences. In fact, *LP Odd-Even Straight Merge Sort* can be implemented with linked lists without difficulty. With this implementation, *LP Odd-Even Straight Merge Sort* results in a practical and simple utility. Comparisons of CPU time, for nearly sorted sequences (with respect to Dis, Inv, Rem and $Runs$), between *LP Odd-Even Straight Merge Sort* and other algorithms indicate that *LP Odd-Even Straight Merge Sort* is the most effective alternative. Moreover, CPU timings of *LP Odd-Even Straight Merge Sort* show that is is competitive with other sorting algorithms on random data and much faster on nearly sorted data.

1.3 Partition Sort

In Section 1.1, we presented a generic sorting algorithm that divides the input into s parts of almost equal size, where s is a constant. We now describe an alternative generic sorting algorithm in which the number s of parts can vary and depends on the disorder in the input [Estivill-Castro and Wood 1992c]. Moreover, the sizes of the individual parts are not restricted, although the total size of the simple parts must be at least a fixed fraction of the total size of the parts. This approach makes the code for the generic algorithm more complex but, in exchange, we obtain optimal adaptivity with respect to more measures. In particular, the division protocol is more sophisticated and may require more than linear time, since it must ensure that the disorder has been significantly reduced in a constant fraction of the parts. The parts where the disorder is small are considered simple.

Letting X be a sequence, we say that a set $P(X)$ of nonempty sequences $X_1, \ldots, X_p$ is a **partition** of X if every element of X is in one and only one sequence in $P(X)$ and the sequences in $P(X)$ have elements only from X. The number of sequences in $P(X)$ is called the **size** of the partition.

Partition Sort(X)

- If X is sorted, then terminate; otherwise, if X is simple, then sort it using an algorithm for simple sequences.
- Otherwise (X is neither sorted nor simple):
  - create a partition $P(X) = \{X_1, \ldots, X_{\|P(X)\|}\}$
  - For $i = 1, \ldots, \|P(X)\|$, sort X_i recursively
  - Merge the sorted sequences to give X in sorted order

Figure 8: *Partition Sort*. The second generic sorting algorithm.

The structure of the generic sorting algorithm, *Partition Sort* is again based on divide and conquer. Given a sequence X, a partition protocol computes a partition $P(X) = \{X_1, \ldots, X_{\|P(X)\|}\}$. Note that the size of the partition depends on the input X. Each X_i that is simple is sorted using a secondary sorting algorithm that sorts simple sequences. Each part that is not simple is sorted by a recursive call of *Partition Sort*. In the final step, all the sorted parts are merged; see Figure 8.

The merging of $\|P(X)\|$ sorted parts takes $O(|X|(1+\log[\|P(X)\|+1]))$ time (pairing the parts X_{2i-1} and X_{2i}, for $i = 1, \ldots, \lfloor \|P(X)\|/2 \rfloor$ and merging them reduces the number of parts by a half in linear time). *Partition Sort* can take as much time to compute the partition as it does to merge the sequences resulting from the recursive calls. Our goal, however, is to obtain an algorithm that takes $O(|X|(1+\log[\|P(X)\|+1]))$ time. If $\|P(X)\|$ is related to a measure M of disorder, then the algorithm will be adaptive with respect to M. However, we will achieve our goal only if a constant fraction of the recursive calls are simple and the sizes of the partitions obtained during further recursive calls do not increase arbitrarily.

We make these notions precise in the following theorem [Estivill-Castro and Wood 1992c].

Theorem 1.2 *Let $c \in \Re$ be a constant with $0 < c \leq 1$. Let PP be a partition protocol and $d > 0$ be a constant such that, for all sequences $X \in N^{<N}$:*

- *The partition protocol PP creates a partition $P(X) = \{X_1, \ldots, X_{\|P(X)\|}\}$ of X making no more than $d|X|(1+\log[\|P(X)\|+1])$ comparisons*
- *The sum of the lengths of the simple sequences in $P(X)$ is at least $c|X|$*
- *If PP is applied to any X_i in $P(X)$, then no more than $\|P(X)\|$ sequences are obtained*

If there is a constant $k > 0$ and a sorting algorithm S such that, for all simple sequences Y resulting from partitions of X and its parts, algorithm S sorts Y by making no more than $k|Y|\log(\|P(X)\|+1$ comparisons, then Partition Sort makes $O(|X|(1+\log[\|\|P(X)\|\|+1]))$ comparisons. In other words, Partition Sort is adaptive with respect to the size of the initial partition.

Before we present some applications of *Partition Sort* we make two observations. First, *Generic Sort* can be regarded as a variant of *Partition Sort*. The specific form of *Generic Sort*, however, results in a stronger theorem (Theorem 1.1) that is easier to apply. Second, Carlsson and Chen attempted to use *Partition Sort* as a general framework for adaptive sorting algorithms and defined the "...minimal size of a specific type τ of partitions" as a generic measure of disorder; thus the relationship between the size of the partition and the measure would be immediate [Chen and Carlsson 1991].

1.3.1 Applications of *Partition Sort*

The first application is to *Natural Mergesort* [Knuth 1973, page 161]. *Natural Mergesort* partitions the input into ascending runs. A sequence is considered simple if it is sorted and, in fact, in this example all parts of the partition are simple. Clearly, a partition into ascending runs can be obtained in linear time by scanning the input and finding all step-downs. Since the number of ascending runs is directly related to the measure *Runs*, *Natural Mergesort* takes $O(|X|(1+\log[Runs(X)+1]))$ time and is, therefore, *Runs*-optimal; a new proof of a Mannila's known result [Mannila 1985b].

Our second application is to Skiena's *Melsort*, which we now describe [Skiena 1988]. When *Melsort* is applied to an input sequence X, it constructs a partition of the input that consists of a set of sorted lists called the **encroaching lists** of X. Since encroaching lists are sorted, the simple parts are sorted sequences and all parts are simple. In the final step, *Melsort* merges the lists to obtain the elements of X in sorted order. The encroaching set of a sequence $X = \langle x_1, \ldots, x_n \rangle$ is defined by the following procedure: We say that x_i **fits** a double-ended queue D if x_i can be added to either the beginning or the end of D to maintain D in sorted order. We start with x_1 as the only element in the first double-ended queue and, for $i = 2, \ldots, |X|$, we insert x_i into the first double-ended queue in which it fits. We create a new double-ended queue if x_i does not fit any existing double-ended queue. An example should make this process clear. Consider the sequence $W_3 = \langle 4, 6, 5, 2, 9, 1, 3, 8, 0, 7 \rangle$. Initially, D_1 consists of 4. The second element, 6, fits at the end of D_1. The third element, 5, is between 4 and 6, so 5 is added to an empty D_2. The next three elements all fit D_1 and are placed there. The element 3 does not fit D_1 but it fits D_2. Similarly, 8 fits D_2 and 0 fits D_1, but the last element requires a new double-ended queue. The final encroaching set is

$$\{D_1 = [0, 1, 2, 4, 6, 9],\ \ D_2 = [3, 5, 8],\ \ D_3 = [7]\}.$$

The number of lists in the encroaching set is the measure Enc of disorder. Thus, in our example, $Enc(\langle 4, 6, 5, 2, 9, 1, 3, 8, 0, 7 \rangle) = 3$. Since the encroaching set can be constructed in $O(|X|(1+\log[Enc(X)+1]))$ time, *Melsort* takes $O(|X|(1+\log[Enc(X)+1]))$ time; therefore, *Melsort* is Enc-optimal and we have obtained a new proof of Skiena's result [Skiena 1988].

Our third application is to *Slab Sort* [Levcopoulos and Petersson 1990]. *Slab Sort* is a sorting algorithm that achieves optimality with respect to $SMS(X)$ (the minimum number of shuffled

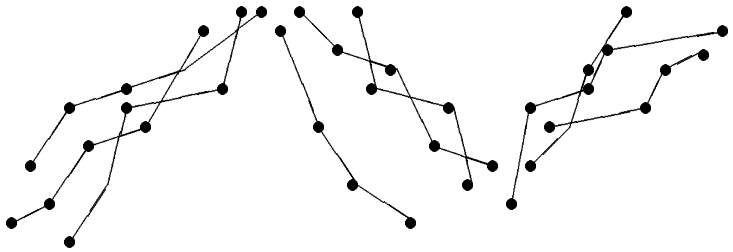

Figure 9: A plot of a zig-zag shuffled sequence.

monotone subsequences of X) and, therefore, optimality with respect to Dis, Max, $Runs$ and SUS. Although *Slab Sort* is an important theoretical breakthrough it has limited practical value because it requires repeated median finding.

Slab Sort sorts sequences X with $SMS(X) \leq z$ using $O(|X|(1 + \log\lceil z + 1\rceil))$ comparisons. The input sequence X is stably partitioned into $p = \lceil z^2/2 \rceil$ parts of almost equal size using the $\lfloor 1 + |X|/p \rfloor$-th, $\lfloor 1 + 2|X|/p \rfloor$-th, ..., $\lfloor 1 + (p-1)|X|/p \rfloor$-th elements as pivots. These elements are found by repeated median finding that make a total of $O(|X| \log\lceil p + 1\rceil)$ comparisons. The partitioning is similar to the partitioning in *Quicksort*, but with $p - 1$ pivots, and it can be carried out in $O(|X|(1 + \log\lceil p + 1\rceil))$ time, as is shown by Frazer and Mckellar for the partitioning in *Samplesort* [Frazer and McKellar 1970].

Before describing *Slab Sort*, we must define *zig-zag shuffled sequences* because, in this application, zig-zag sequences correspond to simple sequences.

Definition 1.3 *Letting $X \in N^{<N}$, we denote a subsequence of X by $\langle x_{i(1)}, \ldots, x_{i(s)} \rangle$, where $i : \{1, \ldots, s\} \rightarrow \{1, \ldots, |X|\}$ is injective and monotonically increasing. We say that a subsequence $\langle x_{i(1)}, \ldots, x_{i(s)} \rangle$ is an* **up-sequence** *if $x_{i(1)} < x_{i(2)} < \ldots < x_{i(s)}$. Similarly, we say that a subsequence $\langle x_{i(1)}, \ldots, x_{i(s)} \rangle$ is a* **down-sequence** *if $x_{i(1)} > x_{i(2)} > \ldots > x_{i(s)}$. A subsequence is* **monotone** *if it is either a down-sequence or an up-sequence. We say that two subsequences $\langle x_{i(1)}, \ldots, x_{i(s)} \rangle$, $\langle x_{j(1)}, \ldots, x_{j(t)} \rangle$* **intersect** *if $\{i(1), i(1)+1, \ldots, i(s)\} \cap \{j(1), j(1)+1, \ldots, j(t)\} \neq \emptyset$.*

For example, if $W_0 = \langle 6, 5, 8, 7, 10, 9, 4, 3, 2, 1 \rangle$, the subsequences $\langle 6, 8, 10 \rangle$ and $\langle 5, 7, 9 \rangle$ intersect but $\langle 6, 8, 10 \rangle$ and $\langle 4, 3, 2, 1 \rangle$ do not.

Definition 1.4 *A sequence X is a* **zig-zag shuffled sequence** *if there is a partition of X into $SMS(X)$ monotone subsequences such that no up-sequence intersects a down-sequence.*

A geometric interpretation of zig-zag shuffled sequences can be obtained as follows. Given a sequence X, for each x_i we plot a point $(i, rank(x_i))$ in the plane. Given the minimum decomposition of X into monotone subsequences, we join the consecutive points of each monotone subsequence with line segments. Then, X is a zig-zag shuffled sequence if no down-sequence curve intersects an up-sequence curve; for example, see Figure 9.

Let X_1, X_2, ..., X_p be the subsequences given by the *Slab Sort* partition of X. Zig-zag shuffled sequences are important because if X_i is a zig-zag shuffled sequence and $SMS(X_i) \leq z$, then

$Enc(X_i) \leq z$ and *Melsort* sorts X in $O(|X|(1+\log[z+1]))$ time; hence, *Melsort* can be used to sort zig-zag shuffled sequences. Moreover, for any X_i, if *Melsort* attempts to construct an encroaching set with more than z sorted lists, then we know that X_i is not simple and should be sorted by a recursive invocation of *Slab Sort*.

Finally, we must ensure that a constant fraction of the elements in X belong to a simple subsequence. Let X be a sequence with $SMS(X) \leq z$. Assume that a minimum partition by monotone sequences has z_1 down-sequences and z_2 up-sequences with $z_1 + z_2 \leq z$. Thus, the number of intersections among up-sequences and down-sequences is bounded by $z_1 z_2 \leq z^2/4$. Since *Slab Sort* stably partitions X into $p = \lceil z^2/2 \rceil$ intervals $X_1, \ldots, X_p$, at least half of these intervals are zig-zag shuffled sequences with $SMS(X_i) \leq z$. Hence, half of the intervals are simple and, since all intervals are almost the same size, about half of the elements in the original sequence belong to simple sequences. We conclude that given a sequence X and an integer z such that $SMS(X) \leq z$, *Slab Sort* sorts X in $O(|X|(1+\log[z+1]))$ time.

1.4 Exponential Search

Let M be a measure of disorder. Suppose that we have a sorting algorithm A that has two inputs, a sequence X and a bound z on the disorder of X, and uses the bound z to sort X in $O(\log \|below(z, |X|, M)\|)$ time. Although algorithm A is adaptive, it requires z as input and thus we do not consider it to be a general sorting algorithm. We describe, however, a design tool that allows us to use algorithm A as a building block for a general sorting algorithm that is M-adaptive.

The idea is to estimate a bound on the disorder in the input and then apply algorithm A [Estivill-Castro and Wood 1989]. We describe a scheme to approximate the disorder based on a variant of **exponential search** [Mehlhorn 1984, page 184].

The sorting algorithm *Search Sort* uses a variant of exponential search to find an upper approximation to $M(X)$. In other words, we find a bound z, such that $M(X) \leq z$ and z is not too far from $M(X)$, and then give this bound to algorithm A [Estivill-Castro and Wood 1992d].

Let k be a constant such that, for all $X \in N^{<N}$ and all z with $M(X) \leq z$, algorithm A on inputs X and z takes no more than $k \times f(|X|, z)$ comparisons, where

$$f(|X|, z) \quad = \quad \max\{|X|, \log \|below(z, |X|, M)\|.$$

The general algorithm *Search Sort* is shown in Figure 10. Note that $f(|X|, z)$ is an nondecreasing function of z. If the updating of the estimate of the bound on the disorder is carefully selected, the time taken by *Search Sort* is dominated by the last invocation of algorithm A and, thus, we obtain an algorithm that requires only the sequence X as input and, up to a constant factor, is as fast as algorithm A. The following theorem formalizes the details.

Theorem 1.5 *Let $k > 0$ be a constant such that, for all $X \in N^{<N}$ and all z with $M(X) \leq z$, algorithm A on inputs X and z takes no more than $k \times f(|X|, z)$ comparisons. If $c > 1$ is a constant such that, for all z,*

$$c \times f(|X|, z^{2^i}) \leq f(|X|, z^{2^{i+1}}),$$

Search Sort(X)

- Set an initial bound $z = c$, where $c > 0$ is a small constant.
- Repeat
  - Call algorithm A with inputs X and z, and count the comparisons A performs.
  - If A makes more than $k \times f(|X|, z)$ comparisons, then interrupt A and update the guess z (usually by squaring z).
  - If A finishes successfully with at most $k \times f(|X|, z)$ comparisons, then terminate.

Figure 10: *Search Sort*. The third generic sorting algorithm.

and Search Sort updates the guess z by squaring it, then Search Sort makes

$$O(f(|X|, M(X))$$

comparisons to sort X; Search Sort is M-optimal.

1.4.1 An application of *Search Sort*

As an application of *Search Sort* we transform *Slab Sort* (see Section 1.3), which requires a bound on the disorder in the input, into a general sorting algorithm that is *SMS*-optimal.

We let the initial guess be $z = 2$ and, rather than counting the comparisons performed by *Slab Sort*, we test if the bound z is adequate by verifying that at least half of the parts in the partition are simple. If at any time we do not obtain this many simple parts, then we square the value of z, and the recursive calls of *Slab Sort* use the new value of z. It is not hard to verify that the new algorithm makes $O(|X|(1 + \log[1 + SMS(X)]))$ comparisons, so it is SMS-optimal. This example illustrates that combining *Partition Sort* and *Search Sort* gives a general sorting algorithm whose division protocol is sophisticated and powerful.

1.5 Adaptive Merging

We have presented three generic sorting algorithms that are based on divide and conquer. Their adaptive behavior is a direct consequence of a carefully designed division protocol, whereas the

merging phase is simple and oblivious to the disorder in the input. However, if the input is nearly sorted, then it is intuitively clear that there should be little disorder among the sorted sequences that result from the recursive calls. There have been two efforts to design sorting algorithms whose merge phase profits from such existing order.

Carlsson, Levcopoulos, and Petersson presented the first adaptive sorting algorithm based on adaptive merging [Carlsson et al. 1990]. Their work extended the notion of adaptivity and optimal adaptivity to merging algorithms; they proposed a new merging algorithm, *Adaptmerge*. Using this merging algorithm to obtain a sorted sequence from the runs in the input, they obtained a variant of *Natural Mergesort* that is *Dis*-, *Exc*-, *Rem*- and *Runs*-optimal. Unfortunately, *Adaptmerge* represents its output as a linked list of sorted segments, which complicates its implementation.

The second sorting algorithm based on adaptive merging was designed by Van Gelder [Van Gelder 1991]. The merging strategy is similar to that of *Adaptmerge* and is based on a simple idea: Let $X = \langle x_1, \ldots, x_n \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ be sorted sequences stored in an array with X before Y. Assume n is known and the goal is to merge X and Y. Their overlap (given by indexes l and r such that $x_l \leq y_1 < x_{l+1}$ and $y_r \leq x_n < y_{r+1}$) is first found. Second, the smaller of $x_l, \ldots, x_n$ and $y_1, \ldots, y_r$ is copied to another array and then merged with the larger sequence in the original array. To obtain a sorting algorithm, Van Gelder uses *straight division* for the divide phase and the above merging strategy to combine sorted sequences. The resulting algorithm has little overhead and is *Dis*- and *Runs*-optimal. It is also adaptive for the measures *Rem* and *Exc*, but not optimal.

2 EXPECTED-CASE (INTERNAL) ADAPTIVE SORTING ALGORITHMS

In contrast to the pessimistic view taken by worst-case analysis, expected-case analysis provides a more practical view, because normally the worst-case instances of a problem are unlikely. We now discuss sorting algorithms that are adaptive in the expected case. There are two approaches for expected-case complexity [Karp 1986; Yao 1977], the **distributional approach** and the **randomized approach.** In the distributional approach a "natural" distribution of the problem instances is assumed and the expected time taken by the algorithm over the different instances is evaluated. This approach may be inaccurate, since the probabilistic assumptions needed to carry out the analysis may be false. We show that, based on the distributional approach, sound definitions of optimality with respect to a measure of disorder can be made; but little new insight is obtained from them. This difficulty is circumvented with randomized sorting algorithms, since their behavior is independent of the distribution of the instances to be solved. (The expected execution time of a randomized algorithm is evaluated on each individual instance.) We demonstrate that adaptive randomized sorting algorithms exist and that they are simple and practical. In contrast, worst-case time sorting algorithms that are adaptive for sophisticated measures like *SMS* use complex data structures or require median finding.

2.1 Distributional Analysis

Our goal here is to argue that distributional analysis provides no new insight into the behavior of adaptive sorting algorithms. As we have pointed out, the distributional approach to expected-case complexity analysis assumes a distribution of the problem instances. The standard assumption for distributional analysis of the sorting problem is that all permutations of the keys are equally likely to occur (clearly, an assumption that can hardly be true in practice; however, it is accepted, for example, in the textbooks that analyze *Quicksort*. Moreover, nearly sorted sequences are common in practice, and adaptive sorting algorithms are more efficient [Knuth 1973, page 339; Mehlhorn 1984, page 54]). With this assumption in mind, Katajainen and Mannila defined optimality for adaptive algorithms, in the expected case, using the distributional approach [Katajainen and Mannila 1989]. Although their definition is sound, it is unclear whether it provides any new insights. Katajainen and Mannila have been unable to construct an example of a sorting algorithm that is optimal in the expected case and suboptimal in the worst case [Katajainen and Mannila 1989]. Moreover, the definition does not shed light on the behavior of Cook and Kim's *CKsort*. Recall that *CKsort* applies Cook–Kim division to a sequence X and splits X into two sequences, one of which is sorted. *Quickersort* is used to sort the unsorted sequence and, finally, the two sequences are merged [Cook and Kim 1980]. *CKsort* takes $O(|X|^2)$ time in the worst case and the computer science fraternity has assumed that it is adaptive with respect to *Rem* in the expected case Cook and Kim 1980; Wainwright 1985]. In the second phase of the algorithm, however, the sequences of length at most $2Rem(X)$ that are given to *Quickersort* are not equally likely to occur; therefore, *CKsort* is not expected-case optimal with respect to any nontrivial measure and Katajainen and Mannila's definition of optimality.

Li and Vitanyi show that the **universal distribution** is as reasonable as the uniform distribution and, they are able to explain why nearly sorted sequences appear more frequently in practice [Li and Vitanyi 1989]. Moreover, they show that if an expected-case analysis is carried out for the universal distribution, then the expected-case complexity of an algorithm is of the same order as its worst-case complexity. Their result implies that *Quicksort* requires $\Theta(|X|^2)$ time in the expected case and it shows that distributional-complexity analysis depends heavily on the assumed distribution. More realistic distributions are usually mathematically intractable. In addition, not only are nearly sorted sequences more likely, but also their distribution may change over time. Because of the difficulties posed by the distributional approach, we consider the randomized approach.

2.2 Randomized Algorithms

We use randomization as a tool for the design of sorting algorithms that are both adaptive and practical. With randomization, we can enlarge the set of measures for which a sorting algorithm is adaptive without increasing the difficulty of a practical implementation. The difficulties with the distributional approach are circumvented by randomized algorithms because their behavior is independent of the distribution of the instances to be solved. Janko, and Bentley and coworkers have successfully used randomization for a constrained sorting problem [Janko 1976; Bentley et al. 1981].

Using the terminology introduced by Mehlhorn and Yao for comparison-based algorithms, we

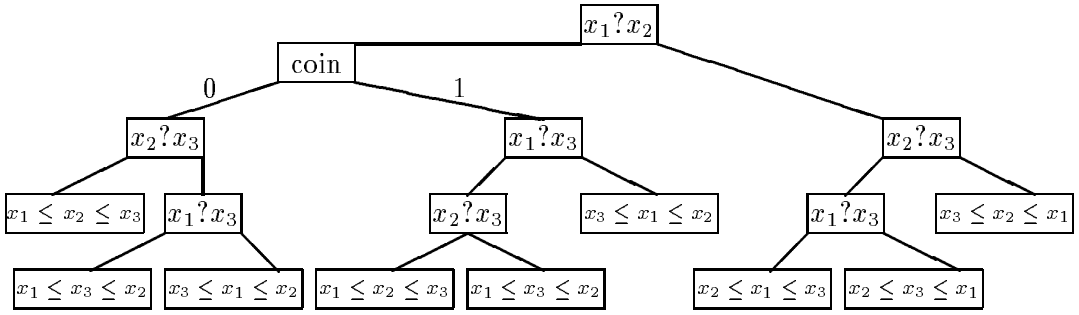

Figure 11: A randomized comparison tree for sorting the sequence x_1, x_2, x_3.

translate the $\Omega(\log \|below(\cdot,\cdot,\cdot)\|)$ lower bounds for the worst case to lower bounds for the distributional expected case, which in turn, result in lower bounds for the randomized expected case [Mehlhorn 1984; Yao 1977]. These implications allow us to capture the notion of optimal adaptivity for randomized sorting algorithms.

Again let M be a measure of disorder and consider sorting algorithms having as input not only the sequence X but also an upper bound z on the value of $M(X)$. Recall that the corresponding decision trees for such algorithms have at least $\|below(z, |X|, M)\|$ leaves, since $below(z, |X|, M)$ is the number of possible inputs. The average depth of a decision tree D with at least $\|below(z, |X|, M)\|$ leaves is denoted by $Average\text{-}depth(D)$ and is

$$\sum_{Y \in below(z,|X|,M)} \frac{depth\ of\ Y\ in\ D}{\|below(z, |X|, M)\|}.$$

This formula implies that there is a constant $c > 0$ such that, for every sorting algorithm D that sorts sequences in $below(z, |X|, M)$,

$$Average\text{-}depth(D) \quad \geq \quad c \times \log \|below(z, |X|, M)\| \tag{2}$$

Now, a randomized sorting algorithm corresponds to a **randomized decision tree** [Mehlhorn 1984]. In a randomized decision tree there are two types of nodes; we give an example of a randomized comparison tree in Figure 11. The first type of node is the ordinary comparison node. The second type is a coin tossing node that has two outgoing edges labeled 0 and 1 which are each taken with a probability of $\frac{1}{2}$. Without loss of generality we can restrict attention to finite randomized decision trees; thus, we assume that the number of coin tosses on inputs of size n is bounded by some function $k(n)$. Which leaf is reached depends not only on the input but also on the sequence $s \in \{0,1\}^{k(n)}$ of outcomes of coin tosses. For any permutation Y and sequence $s \in \{0,1\}^{k(n)}$, let $d^T_{Y,s}$ denote the depth of the leaf reached with the sequence s of coin tosses and input Y. Let T be a randomized decision tree for the sequence X; then, the expected number of

comparisons performed by T on X is denoted by $E[T(X)]$ and is given by

$$E[T(X)] = \sum_{s \in \{0,1\}^{k(n)}} \frac{1}{2^{k(n)}} d^T_{X,s};$$

that is, $E[T(X)]$ is the average number of comparisons of $d^T_{X,s}$ over all random sequences $s \in \{0,1\}^{k(n)}$.

Let $\mathcal{R}$ be the family of all randomized algorithms and let R be a specific randomized algorithm. We denote by $T_R(X)$ the number of comparisons performed by algorithm R on input X; $T_R(X)$ is a random variable. A lower bound on the largest expected value of $T_R(X)$, for X in $below(z, |X|, M)$, is given by the following equation:

$$\max_{X \in below(z,|X|,M)} E[T_R(X)] \quad \geq \quad \min_{T \in \mathcal{R}} \max_{Y \in below(z,|X|,M)} E[T(Y)].$$

Observe that once the sequence $s \in \{0,1\}^{k(n)}$ is fixed, a randomized decision tree becomes an ordinary decision tree D_s that sorts sequences in $below(z, |X|, M)$. since the maximum value of $E[T(Y)]$ is never smaller than the its average value:

$$\max_{X \in below(z,|X|,M)} E[T_R(X)] \geq \min_{T \in \mathcal{R}} \sum_{Y \in below(z,|X|,M)} \frac{E[T(Y)]}{\|below(z, |X|, M)\|}.$$

Now, replacing $E[T(Y)]$ with

$$\sum_{s \in \{0,1\}^{k(n)}} \frac{1}{2^{k(n)}} d^T_{Y,s}$$

reordering the summations, since they are independent, and using Equations (**??**) and (eq:Aver) we obtain

$$\begin{aligned} \max_{X \in below(z,|X|,M)} E[T_R(X)] \quad &\geq \quad \min_{T \in \mathcal{R}} \sum_{s \in \{0,1\}^{k(n)}} \frac{1}{2^{k(n)}} Average\ depth[D_s] \\ &\geq \quad d\ \log \|below(z, |X|, M)\|. \end{aligned}$$

A randomized adaptive sorting algorithm cannot be faster than the fastest randomized sorting algorithm, which uses a bound on the disorder, when it is provided with the best possible bound, namely $z = M(X)$. Intuitively, an adaptive randomized sorting algorithm R is optimal in the expected case with respect to a measure M of disorder if for every sequence X, the expected value of $T_R(X)$ is within a positive constant of the minimum value. Finally, since testing for sortedness requires linear time, a sorting algorithm should be allowed at least a linear number of comparisons. Therefore, we capture the notion of optimal adaptivity for randomized sorting algorithms as follows.

Definition 2.1 *Let M be a measure of disorder, R be a randomized sorting algorithm, and $E[T_R(X)]$ denote the expected number of comparisons performed by R on input X. We say that R is* **optimal** *with respect to M (or* **M-optimal***) if, for some $c > 0$, we have, for all $X \in N^{<N}$,*

$$E[T_R(X)] \quad \leq \quad c \times \max\{|X|, \log \|below(M(X), |X|, M)\|\}.$$

Thus, for randomized algorithms the lower bounds for optimal adaptivity coincide with the lower bounds for optimal adaptivity in the worst case as displayed in Table 1; however, with randomization it is possible to obtain adaptivity for more measures with simpler implementations. Moreover, the relationships $\leq_{alg}$ and $=_{alg}$ between measures are preserved for randomized algorithms.

2.3 Randomized Generic Sort

We now present a randomized generic adaptive sorting algorithm *Randomized Generic Sort* that facilitates the design of an adaptive algorithm by enabling us to focus our attention, once more, on the combinatorial properties of measures of disorder rather than on the combinatorial properties of the algorithm. The randomized generic adaptive algorithm is based on divide-and-conquer. The division phase is performed, however, by a randomized division protocol that, in an expectation sense that we formalize later, does not introduce disorder. The generic algorithm gives rise to randomized sorting algorithms that are adaptive in the expected case; moreover, they are simple and practical.

We denote by $Pr[P(X)]$ the probability that, on input X, the randomized division protocol produces a partition $P(X) = \{X_1, \ldots, X_{||P(X)||}\}$. Let $T_{RDP}(X, P(X))$ be the number of comparisons performed by the randomized division protocol to create $P(X)$ when partitioning X. We say that the protocol is linear in the expected case if there is a constant k such that, for all $X \in N^{<N}$,

$$\begin{aligned} E[T_{RDP}(X)] &= \sum_{P(X)} Pr[P(X)] \cdot T_{RDP}(X, P(X)) \\ &\leq k|X|. \end{aligned}$$

An example of a randomized division protocol is

- *Randomized median division*: Select, with equal probability, an element x from X as a pivot to partition X into $X_{<x}$ (a sequence of elements smaller than x) and $X_{>x}$ (a sequence of elements larger than x). The size of the partition is either two or three (the pivot counts as one element of the partition).

The structure of *Randomized Generic Sort* is presented in Figure 12.

The following theorem formally establishes that *Randomized Generic Sort* results in a sorting algorithm that is adaptive in the expected case with respect to a measure M, when we use a division protocol that reduces the disorder in the expected case. Observe that

$$\sum_{i=1}^{||P(X)||} \frac{|X_i|}{|X|} M(X_i)$$

is the sum of the disorder in the parts of the partition weighted by their corresponding fractions of $|X|$. Observe also that

$$\sum_{P(X)} Pr[P(X)] \sum_{i=1}^{||P(X)||} \frac{|X_i|}{|X|} M(X_i)$$

is the expected disorder in the parts with respect to the set of partitions generated by the division protocol.

Randomized Generic Sort(X)

- X is sorted. Terminate.
- X is simple. Sort X using an alternative sorting algorithm.
- X is neither sorted nor simple.
  - Apply a randomized division protocol to divide X into at least two disjoint sequences.
  - Recursively sort the sequences using *Randomized Generic Sort*.
  - Merge the sorted sequences to give X in sorted order.

Figure 12: *Randomized Generic Sort*. A randomized generic sorting algorithm.

Theorem 2.2 *Let M be a measure of disorder such that $M(X) = 0$ implies X is simple, c be a constant with $0 \leq c < 1$, and RDP be an expected-case linear-time randomized division protocol that randomly partitions a sequence X into a partition $P(X) = \{X_1, \ldots, X_{\|P(X)\|}\}$ with probability $Pr[P(X)]$.*

1. *Randomized Generic Sort is expected-case optimal; on a sequence of length n, it takes $O(n \log n)$ expected time.*
2. *If there is an $n_0 \in N$ and a constant $p \geq 2$ such that, for all sequences X with $|X| > n_0$, RDP satisfies $2 \leq \|P(X)\| \leq p$, and*

$$\sum_{P(X)} Pr[P(X)] \sum_{i=1}^{\|P(X)\|} \frac{|X_i|}{|X|} M(X_i) \quad \leq \quad c \times M(X),$$

then Randomized Generic Sort is adaptive with respect to M in the expected case, it takes

$$O(|X|(1 + \log[M(X) + 1]))$$

time in the expected case.

Since it is the first time that the result has been presented for a generic algorithm, as compared with a similar result for a specific algorithm by Estivill-Castro and Wood [Estivill-Castro and Wood 1992a], we provide the main ideas of its proof.

Sketch of the proof of Theorem 2.2: The proof requires an induction on $|X|$. Let $b > 0$ be a constant such that:

- The expected time taken by *Randomized Generic Sort* to sort all simple sequences of length n is at most bn.
- The expected time taken to discover that a sequence of length n is not simple and, in this case, to partition it and carry out any merging after the recursive calls is at most bn.

Let d be greater that $b/\log[2/(1+c)]$, $T_{RGS}(X)$ denote the number of comparisons performed by *Randomized Generic Sort* on input X, $E[T_{RGS}(X)]$ denote the expected value of this random variable, and $E[T_{RGS}](n,k)$ denote the maximum value of $E[T_{RGS}(X)]$ over all X of length n with $M(X) = k$. We show by induction that

$$E[T_{RGS}](n,k) \leq dn(1 + \log[k+1]),$$

which establishes the theorem. We supply only the induction step since the basis is immediate. Thus, we assume $k \geq 1$ and we let X be a sequence of length n with $M(X) = k$. The induction hypothesis is

$$E[T_{RGS}](n',k) \leq dn'(1 + \log[k+1]),$$

for all $n' \leq n$. Now,

$$E[T_{RGS}(X)] \leq bn + \sum_{P(X)} Pr[P(X)] \left[\sum_{i=1}^{\|P(X)\|} E[T_{RGS}(X_i)] \right],$$

since the right-hand side is the average over all possible partitions of the sum of the cost of sorting each part. By the definition of $E[T_{RGS}(X)]$,

$$E[T_{RGS}(X)] \leq bn + \sum_{P(X)} Pr[P(X)] \sum_{i=1}^{\|P(X)\|} E[T_{RGS}](|X_i|, M(X_i)).$$

The induction hypothesis gives

$$E[T_{RGS}(X)] \leq bn + \sum_{P(X)} Pr[P(X)] \sum_{i=1}^{\|P(X)\|} d|X_i|(1 + \log[1 + M(X_i)]).$$

Algebraic manipulation enables us to show that this inequality implies that

$$E[T_{RGS}(X)] \leq (b+d)n + dn \log \left[1 + \sum_{P(X)} Pr[P(X)] \sum_{i=1}^{\|P(X)\|} \frac{|X_i| M(X_i)}{|X|} \right].$$

Now, by hypothesis 2 and since $d \geq \log[2/(1+c)]$, we have

$$\begin{aligned} E[T_{RGS}(X)] &\leq (b+d)n + dn \log[1 + cM(X)] \\ &\leq bn + dn - dn \log \left[\tfrac{1+M(X)}{1+cM(X)} \right] + dn \log[1 + M(X)] \\ &\leq dn(1 + \log[1 + M(X)]). \end{aligned}$$

as required. □

2.3.1 Applications of *Randomized Generic Sort*

Different variants of *Quicksort* have been obtained by choosing different pivot selection strategies. *Standard Randomized Quicksort* selects the pivot randomly and uniformly from the elements in the sequence [Mehlhorn 1984]. *Standard Randomized Quicksort* can take quadratic time on every sequence; moreover, the number of comparisons performed by *Standard Randomized Quicksort* is a random variable $T_{SRQ}(X)$ such that $T_{SRQ}(X) = \Omega(|X|\log|X|)$ and $E[T_{SRQ}(X)] = \Theta(|X|\log|X|)$, for every sequence X [Knuth 1973]. Thus, *Standard Randomized Quicksort* is oblivious to the order in the input.

As an application of*Randomized Generic Sort*, we modify *Standard Randomized Quicksort* such that while it partitions the input, it checks whether the input is already sorted and, if so, no recursive calls are made. We call this algorithm *Randomized Quicksort*. The number of comparisons performed by the algorithm on an input sequence X is a random variable between $c_1|X|$ and $c_2|X|^2$, for some constants $c_1, c_2 > 0$. Since *Quicksort* uses exchanges to rearrange the elements, we use Exc, the minimum number of exchanges required to sort a sequence, to measure disorder. It turns out that this measure characterizes precisely the adaptive behavior of *Randomized Quicksort* as the following result shows [Estivill-Castro and Wood 1992a].

Theorem 2.3 *The expected number of comparisons performed by Randomized Quicksort on a sequence X is denoted by $E[T_{RQ}(X)]$ and is $\Theta(|X|(1+\log[Exc(X)+1]))$.*

For $k = 2, 3, \ldots, \lfloor n/2 \rfloor$, the sequence

$$W(k) \quad = \quad \langle 2, 1, 4, 3, \ldots, 2\lfloor k/2 \rfloor, 2\lfloor k/2 \rfloor - 1 \rangle$$

has $Exc(W(k)) = k$. We can construct a sequence of length n and measure k by appending the sorted sequence

$$Y_0 = \langle 2\lfloor k/2 \rfloor + 1, 2\lfloor k/2 \rfloor + 2, \ldots, n \rangle$$

to $W(k)$. Since $T_{RQ}(W(k)Y) = \Omega(|n|(1+\log[k+1]))$, we have shown that $E[T_{RQ}(X)]$ is $\Omega(|X|(1+\log[Exc(X)+1]))$. The proof of the upper bound follows from Theorem 2.2. We now show that *Randomized Quicksort* fulfills the hypothesis of Theorem 2.2 [Estivill-Castro and Wood 1992a].

Lemma 2.4 *Let $X = \langle x_1, \ldots, x_n \rangle$, $X_l(s) = \langle x'_1, \ldots, x'_{s-1} \rangle$ be the left subsequence produced by Randomized Quicksort's partition routine when the s-th element is used as the pivot, and $X_r(s) = \langle x'_{s+1}, \ldots, x'_n \rangle$ be the corresponding right subsequence. Then,*

$$\frac{11}{12} Exc(X) \quad \geq \quad \frac{1}{n} \sum_{s=1}^{n} \left[\frac{s-1}{n-1} Exc(X_l(s)) + \frac{n-s}{n-1} Exc(X_r(s)) \right].$$

The proof of this lemma, which uses classic results by Cayley [Cayley 1849], is laborious because there are sequences in which most of the elements, when used as pivots, increase the disorder;

that is, a pivot s can give $Exc(X_l(s)) + Exc(X_r(s)) > Exc(X)$. However, Lemma 2.4 says that, on average, the partition given by a randomly selected pivot does not increase the disorder as measured by Exc.

Cook and Kim [Cook and Kim 1980], Dromey [Dromey 1984], Wainwright [Wainwright 1985], and Wegner [Wegner 1985] have designed adaptive versions of *Quicksort* that are deterministic algorithms with a worst-case complexity of $O(|X|^2)$, but their distributional expected-case analysis seems mathematically intractable. Thus, apart from simulation results, no other description of their adaptive behavior is known. Estivill-Castro and Wood have shown that a worst-case bound of $\Theta(|X|(1+\log[Exc(X)+1]))$ comparisons is achievable if the median is used as the pivot [Estivill-Castro and Wood 1992b]. Observe that, although *Randomized Quicksort* can be coded succinctly and efficiently, it is not Exc-optimal since this hypothesis implies that its running time would be $O(|X| + Exc(X)[1+\log Exc(X)])$.

The second application of *Randomized Generic Sort* is to *Randomized Mergesort*, a variant of *Mergesort* that is *Runs*-optimal in the expected case. It is not optimal in the worst case, with respect to any nontrivial measure, for the same reason that *Randomized Quicksort* is not optimal; that is, $T_{RMS}(X) = \Theta(|X|^2)$ in the worst case, for every unsorted sequence X. *Randomized Mergesort* first determines whether the input is already sorted and, if so, it halts. Otherwise, *Randomized Mergesort* uniformly selects a split position and then proceeds in the same way as the usual *Mergesort*. The time complexity of *Randomized Mergesort* for a sequence X is a random variable independent of the distribution of the problem instances. From Table 1 we know that $\log \|below(X, Runs)\|$ is $\Omega(|X|[1+\log(Runs(X)+1)])$; thus, the following theorem implies that *Randomized Mergesort* is *Runs*-optimal in the expected case.

Theorem 2.5 *If $E[T_{RMS}(X)]$ is the expected number of comparisons performed by Randomized Mergesort on a sequence X, then*

$$E[T_{RMS}(X)] = \Theta(|X|(1+\log[1+Runs(X)])).$$

Theorem 2.5 follows by verifying that *Randomized Mergesort* is an application of *Randomized Generic Sort* that fulfills the hypotheses of Theorem 2.2. Although the uniform selection of the split position is the most practical alternative, *Randomized Mergesort* may use other distributions to select the split position [Estivill-Castro and Wood 1992a].

2.4 Randomized Partition Sort

Naturally we can define a randomized version of *Partition Sort*, which we call *Randomized Partition Sort*. It is also based on divide-and-conquer through a randomized division protocol. We can relax the requirements, however, that the randomized division protocol takes linear time in the expected case and that deeper recursive levels have no more parts than the partition at the first level. For *Randomized Partition Sort*, given a sequence X, the randomized partition protocol obtains a partition $P(X) = \{X_1, \ldots, X_{\|P(X)\|}\}$ with probability $Pr[P(X)]$. Each simple part X_i is sorted using an alternative sorting algorithm for simple sequences, and each part that is not simple is sorted by a recursive call. In the final step, all sorted parts are merged; see Figure 13. The number

Randomized Partition Sort(X)

- X is sorted. Terminate.
- X is simple. Sort it using an algorithm for simple sequences.
- X is neither sorted nor simple.
  - Using a randomized division protocol construct a partition $P(X) = \{X_1, \ldots, X_{\|P(X)\|}\}$.
  - For $i = 1, \ldots, \|P(X)\|$, sort X_i recursively.
  - Merge the sorted sequences to give X in sorted order.

Figure 13: A relaxed, randomized division protocol results in *Randomized Partition Sort*.

of parts in the partition, $\|P(X)\|$, is a random variable; thus, the expected time taken for the merge is no more than

$$\sum_{P(X)} c \times Pr[P(X)]|X|(1 + \log[\|P(X)\| + 1]) \quad \leq \quad c|X|(1 + \log[E[\|P(X)\|] + 1]),$$

where c is a constant and $E[\|P(X)\|]$ is the expected value of $\|P(X)\|$.

Thus, the expected time taken by the partition protocol to obtain a partition of X is bounded from above by $c|X|(1+\log[E[\|P(X)\|]+1])$. If the expected time taken by the recursive calls is also bounded by $c|X|(1+\log[E[\|P(X)\|]+1])$, then we obtain an algorithm that is adaptive with respect to $E[\|P(X)\|]$, and if $E[\|P(X)\|]$ is related to a measure M of disorder, we obtain adaptivity with respect to M.

This approach works however only if the expected length of the simple sequences is a constant fraction of the total length of the sequence to be partitioned as is made precise in the following theorem.

Theorem 2.6 *Let c be a constant with $0 < c \leq 1$, RDP be a randomized division protocol, and d be a constant such that, for all sequences $X \in N^{<N}$, the RDP creates a partition $P(X) = \{X_1, \ldots, X_{\|P(X)\|}\}$ of X in no more than $d|X|(1 + \log[E[\|P(X)\|] + 1])$ comparisons, and*

$$c|X| \quad \geq \quad \sum_{P(X)} Pr[P(X)] \sum_{i \in J(P(X))} |X_i|,$$

where $J(P(X))$ is the set of indices of simple parts.

If there is a constant $k > 0$ and a sorting algorithm S such that for all possible simple sequences X_i in $P(X)$, algorithm S sorts X_i by making no more than $k|X_i|(1 + \log[E[\|P(X)\|] + 1])$ comparisons, then the expected number of comparisons to sort X by Randomized Partition Sort is

$$O(|X|(1 + \log[E[\|P(X)\|] + 1])).$$

Randomized Partition Sort is adaptive with respect to the expected size of the partition.

One application of *Randomized Partition Sort* is to *Randomized Slab Sort*. Recall that *Slab Sort* sorts a sequence X with $SMS(X) \leq z$ using no more than $O(|X|(1 + \log[1 + z]))$ comparisons. It requires, however, $p = \lceil z^2/2 \rceil$ pivots that are found using median finding that makes $O(|X|(1 + \log[1 + p]))$ comparisons. In contrast, *Randomized Slab Sort* uses random selection of a sample of p pivots as in *Samplesort* [Frazer and McKellar 1970]. Estivill-Castro and Wood show that *Randomized Slab Sort* satisfies the requirements of Theorem 2.6 [Estivill-Castro and Wood 1991c]. Thus, for a sequence X with $SMS(X) \leq z$, *Randomized Slab Sort* takes $O(|X|(1 + \log[1 + z]))$ expected time.

2.5 Skip Sort

We have argued that sorting algorithms which are adaptive with respect to several measures take advantage of existing order even when it appears in different forms. Such algorithms are, however, hard to obtain without the use of complex data structures when worst-case adaptivity is desired. In contrast, randomization enables us to achieve expected-case optimality with respect to many measures with simpler data structures. To illustrate this point once more, we recall that Moffat and Petersson designed an insertion-sorting algorithm that, in the worst-case, performs an optimal number of comparisons with respect to all measures of presortedness appearing in the literature [Moffat and Petersson 1991], an important theoretical contribution. The data structure that they use to represent the sorted portion of the input is a historical search tree that provides fast local insertions. A **local insertion** is an insertion at a position in a sorted sequence that is not too far from previous insertions. The additional virtue of Moffat and Petersson's historical search tree is that it implements two types of local insertions efficiently. Those insertions for which distance is measured by space (the number of links followed) in the data structure and those for which distance is measured by time in the sequence of insertions (the number of elements in the sequence between the successor and the element currently being inserted). Unfortunately, the use of a historical search tree implies not only that the time taken by the sorting algorithm is not proportional to the number of comparisons, but also, and more seriously, that the possible applications of the algorithm are limited because of the overhead.

We present *Skip Sort*, a practical sorting algorithm that is optimal with respect to Dis, Exc, Inv, Max, Rem, and $Runs$ in the expected case. *Skip Sort* is an insertion-sorting algorithm that uses a skip list to represent a sorted subsequence. Informally, a **skip list** is a singly linked list with additional forward pointers at different levels; an example skip list is shown in Figure 14. It is a simple and practical probabilistic data structure that has the same asymptotic expected-time bounds as a balanced binary search tree and allows fast local insertions [Pugh 1989]. The price we

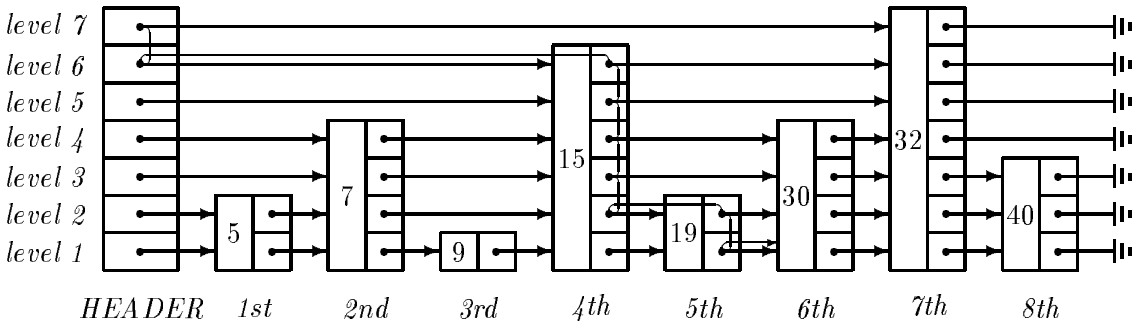

Figure 14: A skip list of 8 elements with the search path when searching for the 6th element. The update vector for the 6th element is [*5th,5th,4th,4th,4th,4th,HEADER*], since these positions define the nodes whose pointers may need to be modified if the height of the node at the 6th position changes.

pay for this performance is that skip lists are no longer worst-case optimal, but only expected-case optimal. Note that the same behavior can be achieved with **randomized treaps**[1] [Aragon and Seidel 1989]; we use skip lists because updates in a treap require rotations and care must be taken to preserve the heap order of the priorities.

Since *Skip Sort* is an insertion-sorting algorithm, the elements are considered one at a time, and each new element is inserted at its appropriate position relative to the previously inserted elements. The standard skip-list insertion of x_i [Pugh 1989], begins at the header; however, for *Skip Sort*, the insertion of x_i into the skip list begins at the position of x_{i-1}, the previously inserted element. During the insertion of each element into (and the deletion of an element from) the skip list, Pugh [Pugh 1989] maintains what he calls the **update vector.** It satisfies the invariant:

> **update**[j] points to the rightmost node of level j or higher that is to the left of the position of x_{i-1} in the skip list; see Figure 14.

Since level-1 links are the standard list links, *update*[1] points to the node just before x_{i-1}. The search for the insertion position of x_i begins by comparing x_i and x_{i-1}. If $x_i < x_{i-1}$, then we search to the left; otherwise, we search to the right. If we must search to the left, then we repeatedly increase the level and compare x_i with the nodes pointed to by *update*[1], *update*[2], ..., until we find a level j such that the element of the node pointed to by *update*[j] is smaller than x_i. At this stage, x_i can be inserted after the node given by *update*[j] using the skip-list insertion algorithm. If we ever reach the top level of the skip list, we insert x_i by the skip-list insertion algorithm beginning at the header. For example, suppose we insert 18 immediately after 30 has been inserted into the skip list of Figure 14. We compare 18 with 30 and determine that we must go to the left. After comparing 18 with *update*[1] = 19 and *update*[2] = 19, we find that *update*[3] = 15 is smaller than

[1] Treaps are search trees with keys and priorities stored in their nodes. The keys are arranged in inorder and the priorities in heap order.

18. Thus, skip-list insertion proceeds from the fourth element and down from level 3. A search to the right is performed similarly.

To insert an element into a skip list that is d positions from the last insertion position, takes $O(1 + \log d)$ expected time [Papadakis et al. 1990; Pugh 1989]. Let $C_I(d)$ denote the number of comparisons performed by an insertion d positions away from the last position and $T_{SS}(X)$ denote the number of comparisons performed by *Skip Sort* on input $X = \langle x_1, \ldots, x_n \rangle$. Since $E[T_{SS}(X)]$ is the sum of the expected times of the $|X|$ insertions,

$$\begin{aligned} E[T_{SS}(X)] &= \sum_{i=1}^{|X|} E[C_I(d_i(X))] \\ &= O(\sum_{i=1}^{|X|} 1 + \log[1 + d_i(X)]), \end{aligned}$$

where $d_i(X) = \|\{j \mid 1 \leq j < i \text{ and } (x_{i-1} < x_j < x_i \text{ or } x_i < x_j < x_{i-1})\}\|$. From this point onwards, the proofs of worst-case optimality [Estivill-Castro and Wood 1989a; Levcopoulos and Petersson 1991b; Mannila 1985b] can be modified to obtain the following theorem.

Theorem 2.7 *Skip Sort is expected-case optimal with respect to Inv, Runs, Rem, Exc, Max, and Dis.*

Sketch of the proof: To illustrate the technique, we give the proof for *Runs*. We analyze $\sum_{i=1}^{|X|} 1 + \log[1 + d_i(X)]$ as we did in the proof for the worst-case time of *Local Insertion Sort* and use

$$E[T_{SS}(X)] = O(\sum_{i=1}^{|X|} 1 + \log[1 + d_i(X)]) \tag{3}$$

to bound the expected value of $T_{SS}(X)$ by a function of $|X|$ and the disorder in X. Since $\log \|below(Runx(X), |X|, Runs)\| = \Omega(|X|[1+\log(Runs(X)+1)])$, to show that *Skip Sort* is *Runs*-optimal in the expected case we must show that, for all X, $E[T_{SS}(X)] = O(|X|[1+\log(Runs(X)+1)])$. Let $R = Runs(X)+1$ and $t(r)$ be the set of indices of the r-th run. Thus, $\sum_{r=1}^{R} \|t(r)\| = |X|$. Moreover, for each run r, $\sum_{j \in t(r)} d_j(X) \leq 2|X|$. Now,

$$\sum_{i=1}^{|X|} 1 + \log[1 + d_i(X)] = |X| + \sum_{r=1}^{R} \sum_{j \in t(r)} \log(1 + d_j(X)).$$

For integers $d_j \geq 0$ such that $\sum_{j \in t(r)} d_j \leq 2|X|$, the sum $\sum_{j \in t(r)} \log(1 + d_j)$ is maximized when $d_j = 2|X|/\|t(r)\|$. Therefore,

$$\sum_{i=1}^{|X|} 1 + \log[1 + d_i(X)] \leq |X| + \sum_{r=1}^{R} \|t(r)\| \log(1 + 2|X|/\|t(r)\|).$$

Table 2: Comparing *Local Insertion Sort* and *Skip Sort* on *Rem* nearly sorted sequences. The CPU time is measured in milliseconds.

Algorithm	$\|X\|$							
	64	128	256	512	1024	2048	4096	8192
L I Sort	57.8	120.2	247.0	517.1	1086.56	2191.28	4618.76	9224.77
Skip Sort	25.2	51.2	105.6	224.4	483.11	979.32	2094.97	4181.15

Table 3: Comparing *Local Insertion Sort* and *Skip Sort* on *Max* nearly sorted sequences. The CPU time is measured in milliseconds.

Algorithm	$\|X\|$							
	64	128	256	512	1024	2048	4096	8192
L I Sort	63.5	127.4	245.6	497.9	1052.76	2163.80	3115.64	9552.80
Skip Sort	27.1	54.2	109.5	234.4	507.61	1080.16	2370.37	4881.20

The right-hand side

$$\begin{aligned} |X| + \sum_{r=1}^{R} \|t(r)\| \log(1 + 2|X|/\|t(r)\|) &= |X| + |X| \sum_{r=1}^{R} [\log(1 + 2|X|/\|t(r)\|)]^{\|t(r)\|/|X|} \\ &= |X| + |X| \log \prod_{r=1}^{R} [1 + 2|X|/\|t(r)\|]^{\|t(r)\|/|X|}. \end{aligned}$$

Since the geometric mean is no larger than the arithmetic mean, we obtain.

$$\begin{aligned} |X| + |X| \log \prod_{r=1}^{R} [1 + 2|X|/\|t(r)\|]^{\|t(r)\|/|X|} &\leq |X| + |X| \log \sum_{r=1}^{R} [2 + \|t(r)\|/|X|] \\ &= |X|(1 + \log[2R + 1]). \end{aligned}$$

This inequality and Equation (3) give $E[T_{SS}(X)] = O(|X|(\log[1 + Runs(X)]))$ as required □

Furthermore, because of the simplicity of skip lists, *Skip Sort* is a practical alternative to *Local Insertion Sort*. Tables 2 and 3 presents simulation results that imply that *Skip Sort* is twice as fast as *Local Insertion Sort*. Both algorithms were coded in C and simulations were performed on a VAX-8650 running UNIX^{TM} 4.3BSD and measured using `gprof` [Graham et al. 1982]. Each algorithm sorted the same set of 100 nearly sorted sequences for each input length. The *Rem* nearly sorted sequences were generated using Cook and Kim's method [Cook and Kim

1980; Wainwright 1985] with $Rem(X)/|X| \leq 0.2$. The percentage of disorder in Table 3 is given by $Max(X)/|X| \leq 0.2$ and the permutations were generated on the basis that Max is a normal measure of disorder [Estivill-Castro 1991].

3 EXTERNAL SORTING

Currently, the sorting of sequences of records that are too large to be held in main memory is performed on disk drives [Salzberg 1988; Salzberg 1989]. Initial runs are created during the first phase of external sorting and, during a second phase, the runs are merged. With more than one disk drive, *Replacement Selection* allows full overlapping of I/O operations and the calculations performed in main memory to create initial runs that are usually larger than the available main memory. If only one disk drive is available, *Heapsort* also allows full overlapping, but the length of the created runs is the size of available main memory [Salzberg 1988]. During the second phase of external sorting the initial runs are combined by *Multiway Merge* and the number of runs merged in each pass (the order of the merge) is chosen to optimize the seek time and the number of passes [Salzberg 1988; Salzberg 1989]. Indeed it is possible to achieve a two-pass sort, one pass to create the initial runs and one pass to sort the runs with *Multiway Merge* [Zheng and Larson 1992].

Sorting a very large sequence implies that the records must be read from disk at least once; thus, full overlapping of I/O with the operations performed in main memory produces initial runs for free. *Replacement Selection* is of central importance because longer initial runs result in fewer initial runs and, therefore, fewer passes over the sequence in the second phase. Thus, larger initial runs reduce the overall sorting time.

The classic result on the performance of *Replacement Selection* establishes that, when all input sequences are assumed to be equally likely, the asymptotic expected length of the produced runs is twice the size of available main memory [Knuth 1973, page 254]. Similar results concerning the length of the first, second, and third initial run as well as the last and penultimate initial run have been obtained by Knuth [Knuth 1973, page 262]. Other researchers have modified *Replacement Selection* such that, asymptotically, the expected length of the initial runs is more than twice the size of available memory [Dinsmore 1965; Dobosiewicz 1984; Frazer and Wong 1972; Ting and Wang 1977]. These methods have received limited acceptance because they require more sophisticated I/O operations and prohibit full overlapping; thus, the possible benefits hardly justify the added complexity of the method. Similarly, attempts to design new external sorting methods that profit from the existing order in the input face inefficient overlapping of I/O operations.

Several researchers have observed, however, that the lengths of the runs created by *Replacement Selection* increase as the disorder in the input sequence decreases. For fixed input length, as the lengths of the initial runs increases, the number of passes to merge these runs decreases. Thus, external sorting using *Replacement Selection* is sensitive to the disorder in the input when two or more disk drives are available. Let P be the maximum number of records that can be stored in main memory. In the worst case, *Replacement Selection* produces runs no larger than P. *Replacement Selection* facilitates the merging phase when it produces runs of more than P elements. Based on this idea, Estivill-Castro and Wood describe mathematically the worst-case performance of *Replacement Selection* on nearly sorted sequences [Estivill-Castro and Wood 1991a].

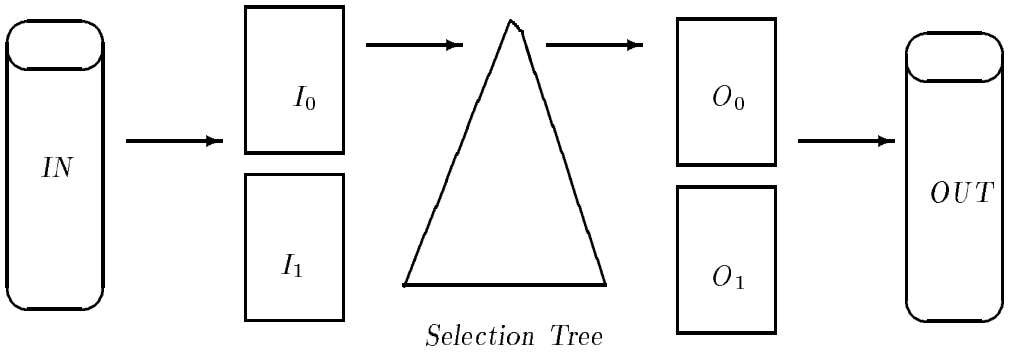

Figure 15: Environment for *Replacement Selection*.

They introduce two measures to assess the performance of *Replacement Selection*. These measures are minimized when *Replacement Selection* does not create runs with more than P elements and are maximized when *Replacement Selection* creates a sorted sequence. Their results show how the performance of*Replacement Selection* smoothly decreases as the disorder in the input sequence increases. In their analysis, they show that, as is common in practice, it is essential to consider input sequences that are much larger than the size of main memory; otherwise, the disorder in the input sequence has little effect. The reason is that, for short nearly sorted sequences, *Replacement Selection* does not produce long runs because there are simply not enough elements. We now summarize these results.

3.1 Replacement Selection

The environment for *Replacement Selection* consists of two disk drives; see Figure 15. The input sequence is read sequentially from the *IN* disk drive using double buffering. While one of the input buffers, say I_i is being filled, $i \in \{0,1\}$, *Replacement Selection* reads elements from the other input buffer I_{1-i}. The roles of I_i and I_{1-i} are interchanged when one buffer is full and the other one is empty. The output sequence is written sequentially to the second disk drive using a similar buffering technique.

For the purposes of our discussion we concentrate on *Replacement Selection* based on **selection trees** [Knuth 1973, page 254]. We assume that there is room in main memory for P elements, together with buffers, program variables, and code. *Replacement Selection* organizes the P elements in main memory into a selection tree with P nodes.

> **Notational convention:** We use X_I to denote the input sequence and X_O to denote the corresponding output sequence produced by *Replacement Selection* with P nodes. P is omitted throughout—we should really write X_O^P.

Replacement Selection operates as follows. Initially, the selection tree is filled with the first P elements of X_I; the smallest element is at the root. The element at the the root is the smallest

element in the selection tree that belongs to the current run. Repeatedly, the root element is removed from the tree, appended to X_O, and replaced by the next element from X_I. In this way P elements are kept in main memory, except during initialization and the termination. To make certain that the elements that enter and leave the tree do so in proper order, we number the runs in X_O. After an element has been deleted from the selection tree and is replaced by a new element x, we discover the run number that x belongs to by the following reasoning: If x is smaller than the element that was just added to the current run, then x must be in the next run; otherwise, x belongs to the current run.

Consider the length of the input to be fixed and the disorder in the input to be variable. It is intuitively clear that *Replacement Selection* creates initial runs with more than P elements when X_I is nearly sorted. However, it is unclear how many initial runs will be larger than P. It is even less clear, for example, how sorted should X_I be to guarantee that at least one in four of the initial runs is larger than P. This observation leads us to a fundamental question: Letting c be a positive integer, how sorted should X_I be to guarantee that, on average, at least one in c of the initial runs is larger than P? The first measure of the performance of *Replacement Selection* is *Runs-Ratio*(X_O), the fraction of the runs that are larger than P; it is defined as follows:

$$\textit{Runs-Ratio}(X_O) = \frac{\text{number of runs in } X_O \text{ thatare larger than } P}{\text{number of runs in } X_O}.$$

This measure is naive but we are able to analyze it. The second measure of the performance of *Replacement Selection* is the**average run length** of X_O. It is denoted by $ARL(X_O)$ and is defined as the average of the lengths of the runs in X_O:

$$\begin{aligned} ARL(X_O) &= \frac{\text{sum of the lengths of the runs in } X_O}{\text{number of runs in } X_O} \\ &= \frac{|X_O|}{\text{number of runs in } X_O}. \end{aligned}$$

In fact, the time taken by the second phase of external sorting is a function of the average run length of X_O. The number of passes required by a multiway merge is

$$1 + \lfloor \log_\omega(\text{number of runs in } X_O) \rfloor \quad = \quad 1 + \lfloor \log_\omega(\text{length of } X_O / ARL(X_O) \rfloor, \tag{4}$$

where ω is the order of the merge.

The measure *Runs-Ratio*(X_O) can be as small as 0. We expect that if $M(X_I)$ is small with respect to P, for some measure M of disorder, then *Runs-Ratio*(X_O) is at least positive. Some consideration must be given, however, to the length of X with respect to P because if X_I is shorter than P, then the value *Runs-Ratio*(X_O) is 0 independently of how sorted X_I is. Similarly, we expect that if $M(X_I)$ is small, then $ARL(X_O)$ is somewhat larger than P; however, this property does not hold for all measures of disorder. Fortunately, disorder can be evaluated in many ways. Intuitively, *Replacement Selection* is oblivious to local disorder, since it can place elements that are no more than P positions away from their correct position. *Replacement Selection* is sensitive, however, to global disorder. Recall that one measure that evaluates this type of disorder is Max, defined as the

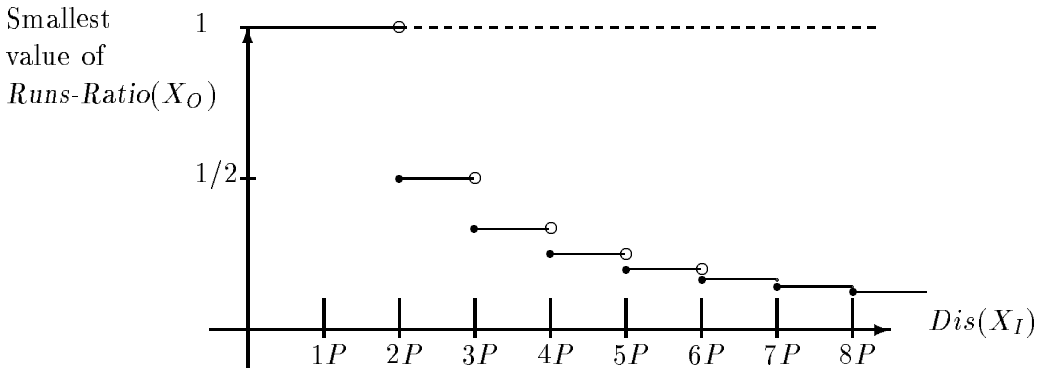

Figure 16: As the disorder in X_I grows, the performance of *Replacement Selection*, quantified by the smallest value of *Runs-Ratio*,decreases.

largest distance an element must travel to reach its sorted position, and another is Dis, defined as the largest distance determined by an inversion. In fact, Dis and Max are equivalent since, for all unsorted$X$, $Max(X) \leq Dis(X) \leq 2Max(X)$ [Estivill-Castro and Wood 1989].

Theorem 3.1 characterizes the *Runs-Ratio* with respect to Dis. We present the result in graphical form in Figure 16, which illustrates how the performance of *Replacement Selection* decreases as the disorder increases.

Theorem 3.1 *Let $c > 0$ be a constant and let X_I be such that $|X_I| > (\lfloor c \rfloor + 1)P$. If $Dis(X_I) \leq cP$, then*

1. *Runs-Ratio$(X_O) = 1$ when $c < 1$; thus, when the disorder according to Dis is less than P, Replacement Selection maximizes Runs-Ratio.*
2. *Runs-Ratio$(X_O) \geq \frac{1}{\lfloor c \rfloor}$ when $c \geq 1$; thus, Runs-Ratio is at least $P/Dis(X)$.*

Moreover, these bounds are tight.

It is not surprising that if the distance determined by an inversion pair that is farthest apart is less than P, then $X_O$is sorted and *Runs-Ratio*(X_O) is maximized. However, it is not immediate that $Dis(X_I) < 2P$implies that X_O is sorted. As the bound on the disorder grows, the performance of *Replacement Selection* decreases. Now we can determine how sorted should X_I be to guarantee that at least one in c runs in X_O are larger than P. For example, if we want to make sure that one in two runs produced by *Replacement Selection* is larger than P, then $Dis(X_I)$ should be less than $3P$. What was unclear, now seems simple.

As we have mentioned, *Runs-Ratio* is a simple measure. We have used it as a first approach to the study of the average run length. Recall (see Equation (4)) that the overall cost of external sorting is a function of the the average run length.

As promised, Theorem 3.2 gives the corresponding bounds on the average run length of X_O when X_I is nearly sorted with respect to Dis; Figure 17 is the corresponding graphical interpretation of

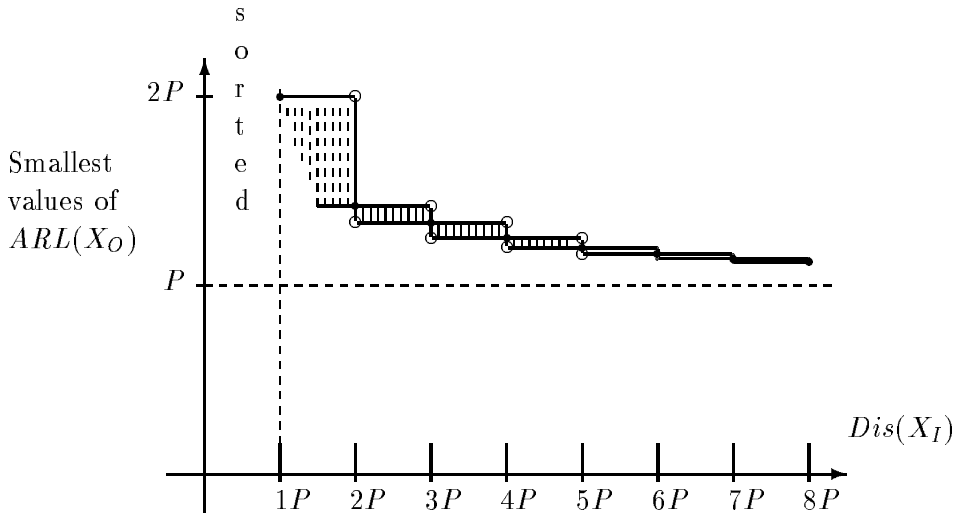

Figure 17: The smallest values of $ARL(X_O)$ define a decreasing function of the disorder. These values, which lie in the shaded area, show how the performance of *Replacement Selection* decreases as the disorder increases.

the theorem. This result confirms that the performance of *Replacement Selection*,measured by the average run length of X_O, decreases smoothly as the disorder in X_I (measured by Dis) increases.

Theorem 3.2 *Let X_I be such that $|X_I| > (\lfloor c \rfloor + 2)P$, $r = P/|X_I|$, and $Dis(X_I) \leq cP$. Then,*

$$ARL(X_O) \geq \begin{cases} \frac{(3-c)P}{1+(2-c)/r} & when\ 1 \leq c \leq 3/2 \\ \frac{(3/2)P}{1+1/2r} & when\ 3/2 \leq c \leq 2 \\ \frac{(1+\frac{1}{\lfloor c \rfloor})P}{1+\lfloor c \rfloor/r} & when\ 2 \leq c\ and\ c < \lfloor c \rfloor + 1 - \frac{1}{\lfloor c \rfloor}. \end{cases}$$

Moreover, for all integers $c \geq 2$, these bounds are tight.

Theorems 3.1 and 3.2 provide bounds on the performance of *Replacement Selection* and formally establish that if the input is nearly sorted, then the runs that are produced must be longer. Figures 16 and 17 are graphical representations of Theorems 3.1 and 3.2, respectively, that show that the performance of *Replacement Selection* decreases slowly as a function of the disorder measured by Dis.

4 FINAL REMARKS

The objectives of this survey are primarily:

- to present generic adaptive sorting algorithms
- to describe the performance of the generic algorithms
- to illustrate how the generic algorithms can be used to obtain adaptivity with respect to different measures of disorder

We believe that our survey provides insight into the theory of adaptive sorting algorithms and brings important points to the attention of practitioners. Research on adaptive sorting algorithms is still active; moreover, adaptive sorting offers several directions for further research. There have been and continue to be attempts to modify well-known sorting algorithms to achieve adaptivity. Moffat studied *Insertion Sort* using a **splay tree** [Moffat 1990]; however, the characterization of the adaptive behavior of this algorithm is open. Sophisticated variants of *Natural Mergesort* that increase the number of measures to which the algorithm is adaptive have also been proposed [Moffat 1991; Moffat et al. 1992].

Other researchers have studied measures of disorder and their mathematical structure [Estivill-Castro et al. 1989; Chen and Carlsson 1989]. In addition, the discoveries that have been made for sequential models have prompted other researchers to design parallel sorting algorithms with respect to *Dis* [Igarashi and Altman 1988] and other measures [Levcopoulos and Petersson 1988; Levcopoulos and Petersson 1989b].

We now discuss, in some detail, some other directions for further research. First, we believe that researchers should develop universal methods for large classes of measures of disorder that lead to practical implementations. If $X_1, X_2, \ldots, X_s$ is a partition of X into disjoint subsequences, then

$$\sum_{j=1}^{s} M(X_j) \leq sM(X)$$

normally holds, for a measure M of disorder; that is, all division protocols result in a division with $D \leq 2$. With *Generic Sort*, we showed that it is necessary and sufficient for a protocol to satisfy $D < 2$ to obtain an adaptive algorithm with respect to M. We conjecture that it is always possible to design a linear-time division protocol that partitions a sequence X into subsequences $X_1, X_2, \ldots, X_s$ of almost the same length such that $D < 2$. If this conjecture holds, then an adaptive sorting algorithm that makes $O(|X|(1 + \log[1 + M(X)]))$ comparisons, in the worst case, could always be obtained.

Second, Estivill-Castro and Wood obtained adaptivity results using the median-division protocol [Estivill-Castro and Wood 1991b; Estivill-Castro and Wood 1992b]. These results suggest a generalization of Theorem 1.1 in which the bound on the disorder introduced at the division phase is a function of the input size rather than constant, but is still bounded above by 2. In other words, the results suggest that the condition in Theorem 1.1 can be weakened to:

> Let $D(n)$ be a function such that, for all n, $D(n) < 2$ and, for all sequences X of size n, the division protocol satisfies
>
> $$\sum_{j=1}^{s} M(\text{s-th subsequence}) \quad \leq \quad D(n)\lfloor s/2 \rfloor M(X)$$

If this generalization holds, then more sorting algorithms could be shown to have the performance claimed in Theorem 1.1.

Third, *Shell Sort* is a very efficient sorting method in practice. There are many ways of choosing the sequence of increments for *Shell Sort* [Gonnet 1984; Knuth 1973]; however, *Shell Sort* requires $\Omega(n \log n)$ comparisons to sort every sequence of length n. *Shell Sort* shows some adaptivity, but it has not been shown to be optimal with respect to any measure of disorder. It would be useful if we could describe the behavior of *Shell Sort* as a function of the input size and the disorder in the input. Furthermore, a variant of *Shell Sort* that adapts the sequence of increments to the existing order in the input would be even more interesting.

Fourth, adaptive variants of *Heapsort* have been proposed [Dijkstra 1982; Leong 1989; Levcopoulos and Petersson 1989a; Petersson 1991] but either the in-place property of *Heapsort* is lost or their adaptivity is insignificant. A natural question is: Are there any in-place sorting algorithms that are optimally adaptive with respect to important measures? Levcopoulos and Petersson have modified Cook–Kim division to obtain an in-place *Rem*-optimal algorithm [Levcopoulos and Petersson 1991b]. Huang and Langston have developed an algorithm to merge two sorted sequences in-place [Huang and Langston 1988]. Their linear-time in-place merging algorithm can be used with *Generic Sort* to obtain *Runs*-, *Dis*-, *Max*-, *Exc*- and *Rem*-optimal sorting algorithms that sort in-place such that the number of data moves moves is proportional to the number of comparisons. The resulting sorting algorithms are bottom up rather than recursive. Finally, using linear-time in-place merging, linear-time in-place selection, and encoding pointers by swapping elements, Levcopoulos and Petersson have recently designed an in-place sorting algorithm that is *Inv*- and *Rem*-optimal [Levcopoulos et al. 1991a].

Fifth, Carlsson and Chen consider adaptivity with respect to the number of distinct keys in a sequence rather than to its disorder [Chen and Carlsson 1991]. Their results are not new [Munro and Spira 1976; Wegner 1985], they suggest, however, a new direction for research; namely, design sorting algorithms that are adaptive with respect to both the disorder and the number of distinct keys.

Finally, we mention adaptive parallel sorting. There are several models of parallel computers and many different sorting algorithms have been designed for different architectures [Akl 1985; Quinn 1987]. An SIMD computer consists of a number of processors operating under the control of a single instruction stream issued by a central control unit. The processors each have a small private memory for storing programs and data and operate synchronously. A number of metrics have been proposed for evaluating parallel algorithms. Akl describes the most important ones [Akl 1985]. The **parallel running time** of a parallel algorithm is defined as the total number of the two kinds of steps executed by the algorithm: routing steps and computational steps. For a problem of size n, the parallel worst-case running time of an algorithm is denoted by $w(n)$, and **the number of processors** it requires is denoted by $P(n)$. The **cost** $c(n)$ of a parallel algorithm is defined as the product $w(n)P(n)$. A parallel algorithm is said to be **cost optimal** if $c(n)$ is of the same order of time complexity as an optimal sequential algorithm.

When sorting on a shared-memory parallel computer, it is usually assumed that the data is stored in the shared memory. Since the introduction of Ajtai and his coworkers' $O(n \log n)$-cost sorting network, there are many parallel sorting algorithms that use $O(n)$ processors and take

$O(\log n)$ time; hence, they are cost optimal [Ajtai et al. 1983]. On a CRCW PRAM it takes constant time to test whether a sequence is sorted, however, in most other models with n processors, $\Omega(\log n)$ time is needed to test whether the input is sorted. Thus, the cost of verifying whether the input is sorted is the same as the cost of sorting.

Because of these observations, it seems that there is no raison d'être for parallel adaptive sorting. However, Levcopoulos and Petersson have taken an unusual approach [Levcopoulos and Petersson 1988; Levcopoulos and Petersson 1989b]. Rather than the usual notion of the number of processors required to solve a problem, they redefine this complexity measure as follows. At step t_i the parallel algorithm may decide to declare several processors idle and not be charged for them. The algorithm has full power to request more processors or to declare some processors to be idle. The **cost** of the algorithm is redefined to be the sum, over all time steps t_i, of the number of processors active at time t_i. In this model, they have described parallel sorting algorithms that are cost optimal with respect to the measures *Inv* and *Rem* [Levcopoulos and Petersson 1988; Levcopoulos and Petersson 1989b]. Although it is usually assumed that the number of processors does not change dynamically during execution of an algorithm, the idea that the algorithm can adapt its resource requirements (processors) according to the difficulty of the input is a step forward. With one processor, the algorithm's time complexity grows smoothly from $O(n)$ to $O(n \log n)$ according to the disorder in the input sequence. When we have $O(n)$ processors, the time complexity is always $\Omega(\log n)$ for all problem instances. Thus, a basic question is: Given P processors, can we design parallel sorting algorithms whose *running time* is a nondecreasing function of the disorder in the input sequence and, in this case, what does it mean to be optimal? We have almost no answers. For the measure *Runs* and the special case $P = n/\log n$, Carlsson and Chen provide a cost-optimal adaptive algorithm [Carlsson and Chen 1991]. For a more general case, McGlinn has generalized *CKsort* and studied its adaptive behavior empirically [McGlinn 1989].

ACKNOWLEDGEMENTS

This work was done while the first author was a Postdoctoral Fellow and the second author was a fulltime faculty member in the Department of Computer Science, University of Waterloo, Waterloo, Canada. It was carried out under Natural Sciences and Engineering Research Council of Canada Grant No.A-5692 and under an Information Technology Research Centre grant.

References

[1] M. Ajtai, J Komlós, and E. Szemerédi. An O($n \log n$) sorting network. In *Proceedings 11th Annual ACM Symposium on the Theory of Computing*, pages 1–9, April 1983.

[2] S. G. Akl. *Parallel Sorting Algorithms.* Notes and Reports in Computer Science and Applied Mathematics. Academic Press, Orlando, FL, 1985.

[3] C. Aragon and R. Seidel. Randomized search trees. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 540–545, 1989.

[4] J. L. Bentley, D. F. Stanat, and Steele J. M. Analysis of a randomized data structure for representing ordered sets. In *Proceedings of the 19th Annual Allerton Conference on Communication, Control and Computing*, pages 364–372, 1981.

[5] M.R. Brown and R.E. Tarjan. Design and analysis of data structures for representing sorted lists. *SIAM Journal on Computing*, 9:594–614, 1980.

[6] W.H. Burge. Sorting, trees and measures of order. *Information and Control*, 1:181–197, 1958.

[7] S. Carlsson and J. Chen. An optimal parallel adaptive sorting algorithm. *Information Processing Letters*, 39:195–200, 1991.

[8] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural merge sort. In Joho Shoi Gakkai, editor, *Proceedings of SIGAL International Symposium on Algorithms*, Springer-Verlag Lecture Notes in Computer Science 450, 1990.

[9] A. Cayley. Note on the theory of permutations. *London, Edinburgh and Dublin Philosophical Magazine and Journal of Science*, 34:527–529, 1849.

[10] J. Chen and S. Carlsson. A group-theoretic approach to measures of presortedness. Technical report, Department of Computer Science, Lund University, Box 118, S-2100 Lund, Sweden, 1989.

[11] J. Chen and S. Carlsson. On partitions and presortedness of sequences. In *Proceedings of the Second ACM–SIAM Symposium on Discrete Algorithms*, pages 63–71, 1991.

[12] C.R. Cook and D.J. Kim. Best sorting algorithms for nearly sorted lists. *Communications of the ACM*, 23:620–624, 1980.

[13] E.W. Dijkstra. Smoothsort, an alternative to sorting in situ. *Science of Computer Programming*, 1:223–233, 1982.

[14] R.J. Dinsmore. Longer strings for sorting. *Communications of the ACM*, 8(1):48, 1965.

[15] W. Dobosiewicz. Replacement selection in 3-level memories. *Computer Journal*, 27(4):334–339, 1984.

[16] P.G. Dromey. Exploiting partial order with Quicksort. *Software — Practice and Experience*, 14(6):509–518, 1984.

[17] V. Estivill-Castro. *Sorting and Measures of Disorder*. PhD thesis, University of Waterloo, 1991. Available as Department of Computer Science Research Report CS-91-07.

[18] V. Estivill-Castro, H. Mannila, and D. Wood. Right invariant metrics and measures of presortedness. *Discrete Applied Mathematics*, accepted 1989, to appear.

[19] V. Estivill-Castro and D. Wood. A new measure of presortedness. *Information and Computation*, 83:111–119, 1989.

[20] V. Estivill-Castro and D. Wood. External sorting, initial runs creation, and nearly sortedness. Research Report CS-91-36, Department of Computer Science, University of Waterloo, 1991.

[21] V. Estivill-Castro and D. Wood. Practical adaptive sorting. In F. Dehne, F. Fiala, and W.W. Koczkodaj, editors, *Advances in Computing and Information — Proceedings of the International Conference on Computing and Information*, Springer-Verlag Lecture Notes in Computer Science 497, pages 47–54, 1991.

[22] V. Estivill-Castro and D. Wood. Randomized sorting of shuffled monotone sequences. Research Report CS-91-24, Department of Computer Science, University of Waterloo, 1991.

[23] V. Estivill-Castro and D. Wood. Randomized adaptive sorting. *Random Structures and Algorithms*, 1992, to appear. Available as Department of Computer Science Research Report CS-91-21, University of Waterloo.

[24] V. Estivill-Castro and D. Wood. A generic adaptive sorting algorithm. *The Computer Journal*, to appear, 1992.

[25] V. Estivill-Castro and D. Wood. An adaptive generic sorting algorithm that uses variable partitioning. In preparation, 1992.

[26] V. Estivill-Castro and D. Wood. The use of exponential search to obtain generic sorting algorithms. In preparation, 1992.

[27] W.D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3):496–507, July 1970.

[28] W.D. Frazer and C.K. Wong. Sorting by natural selection. *Communications of the ACM*, 15(10):910–913, 1972.

[29] G.H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley Publishing Co., Reading, MA, 1984.

[30] S. L. Graham, P.B. Kessler, and M. K. McKusik. `gprof`: A call graph execution profiler. *The Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, SIGPLAN Notices*, 17(6):120–126, 1982.

[31] L.J. Guibas, E.M. McCreight, and M.F. Plass. A new representation of linear lists. In *The Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 49–60, 1977.

[32] J. D. Harris. Sorting unsorted and partially sorted lists using the natural merge sort. *Software — Practice and Experience*, 11:1339–1340, 1981.

[33] B. Huang and M. Langston. Practical in-place merging. *Communications of the ACM*, 31(3):348–352, March 1988.

[34] Y. Igarashi and T. Altman. Roughly sorting: Sequential and parallel approach. Technical Report 116-88, Department of Computer Science, University of Kentucky, 1988.

[35] T. Islam and K. B. Lakshman. On the error-sensitivity of sort algorithms. In S. G. Akl, F. Fiala, and W. W. Koezkodaj, editors, *Proceedings of the International Conference on Computing and Information*, pages 81–85, Canadian Scholar's Press, Toronto, 1990.

[36] W. Janko. A list insertion sort for keys with arbitrary key distributions. *ACM Transactions on Mathemathical Software*, 2(2):143–153, June 1976.

[37] R. M. Karp. Combinatorics, complexity and randomness. *Communications of the ACM*, 29(2):98–109, February 1986.

[38] J. Katajainen, C. Levcopoulos, and O. Petersson. Local insertion sort revisited. In *Proceedings of Optimal Algorithms*, Springer-Verlag Lecture Notes in Computer Science 401, pages 239–253, 1989.

[39] J. Katajainen and H. Mannila. On average case optimality of a presorting algorithm. Unpublished manuscript, 1989.

[40] M.G. Kendall. *Rank Correlation Methods.* Griffin, London, 4th edition, 1970.

[41] D.E. Knuth. *The Art of Computer Programming, Vol.3: Sorting and Searching.* Addison-Wesley Publishing Co., Reading, MA, 1973.

[42] H. W. Leong. Preorder Heapsort. Technical report, National University of Singapore, 1989.

[43] C. Levcopoulos and O. Petersson. An optimal parallel algorithm for sorting presorted files. In *In proceedings of the 8th Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer-Verlag Lecture Notes in Computer Science 338, pages 154–160, 1988.

[44] C. Levcopoulos and O. Petersson. Heapsort — adapted for presorted files. In F. Dehne, J.R. Sack, and N. Santoro, editors, *Proceedings of the Workshop on Algorithms and Data Structures*, Springer-Verlag Lecture Notes in Computer Science 382, pages 499–509, 1989.

[45] C. Levcopoulos and O. Petersson. A note on adaptive parallel sorting. *Information Processing Letters*, 33:187–191, 1989.

[46] C. Levcopoulos and O. Petersson. Sorting shuffled monotone sequences. In J.R. Gilbert and R. Karlsson, editors, *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*, Springer-Verlag Lecture Notes in Computer Science 447, pages 181–191, 1990.

[47] C. Levcopoulos and O. Petersson. An optimal in-place sorting algorithm. In *Proceedings of the 11th Conference on Foundations of Software Technology and Theoretical Computer Science*, 1991.

[48] C. Levcopoulos and O. Petersson. Splitsort—an adaptive sorting algorithm. *Information Processing Letters*, 39:205–211, 1991.

[49] M. Li and P.M.B. Vitanyi. A theory of learning simple concepts under simple distributions and average case complexity for the universal distribution. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 34–39, 1989.

[50] H. Mannila. *Instance Complexity for Sorting and NP-Complete Problems.* PhD thesis, University of Helsinki, Department of Computer Science, 1985.

[51] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34:318–325, 1985.

[52] R. McGlinn. A parallel version of Cook and Kim's algorithm for presorted lists. *Software — Practice and Experience*, 19(10):917–930, October 1989.

[53] K. Mehlhorn. Sorting presorted files. *Proceedings of the 4th GI Conference on Theory of Computer Science*, Springer-Verlag Lecture Notes in Computer Science 67:199–212, 1979.

[54] K. Mehlhorn. *Data Structures and Algorithms, Vol 1: Sorting and Searching.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin/Heidelberg, 1984.

[55] A. Moffat. How good is splay sort. Technical report, Department of Computer Science, The University of Melbourne, Parkville 3052, Australia, 1990.

[56] A. Moffat. Adaptive merging and a naturally natural merge sort. In *Proceedings of the 14th Australian Computer Science Conference*, pages 08.1–08.8, 1991.

[57] A. Moffat and O. Petersson. Historical searching and sorting. In *Second Annual International Symposium on Algorithms*, Springer-Verlag Lecture Notes in Computer Science, 1991.

[58] A. Moffat and O. Petersson. A framework for adaptive sorting. In *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory*, Springer-Verlag Lecture Notes in Computer Science, 1992.

[59] A. Moffat, O. Petersson, and N. Wormald. Further analysis of an adaptive sorting algorithm. In *Proceedings of the 15th Australian Computer Science Conference*, pages 603–613, 1992.

[60] J.I. Munro and P.M. Spira. Sorting and searching in multisets. *SIAM Journal on Computing*, 5(1):1–8, March 1976.

[61] T. Papadakis, J. I. Munro, and P.V. Poblete. Analysis of the Expected Search Cost in Skip Lists. In J.R. Gilbert and R. Karlsson, editors, *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*, Springer-Verlag Lecture Notes in Computer Science 447. pages 160–172, 1990.

[62] O. Petersson. *Adaptive Sorting.* PhD thesis, Lund University, Department of Computer Science, 1990.

[63] O. Petersson. Adaptive selection sorts. Technical Report LU-CS-TR:91-82, Department of Computer Science, Lund University, Lund, Sweden, 1991.

[64] W. Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[65] M.J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* Supercomputing and Artificial Intelligence, McGraw-Hill, New York, NY, 1987.

[66] B. Salzberg. *File Structures: An Analytic Approach.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1988.

[67] B. Salzberg. Merging sorted runs using large main memory. *Acta Informatica*, 27:195–215, 1989.

[68] R. Sedgewick. *Quicksort.* Garland Publishing Inc., New York and London, 1980.

[69] S.S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28:755–784, 1988.

[70] N.J.A. Sloane. Encrypting by random rotations. In T. Beth, editor, *Proceedings of Cryptography, Burg Feuerstein 82*, Springer-Verlag Lecture Notes in Computer Science 149, pages 71–128, 1983.

[71] T.C. Ting and Y. W. Wang. Multiway replacement selection sort with a dynamic reservoir. *Computer Journal*, 20(4):298–301, 1977.

[72] A. Van Gelder. Simple adaptive merge sort. Technical report, University of California, Santa Cruz, 1991.

[73] R.L. Wainwright. A class of sorting algorithms based on Quicksort. *Communications of the ACM*, 28:396–402, 1985.

[74] L.M. Wegner. Quicksort for equal keys. *IEEE Transactions on Computers*, C-34:362–367, 1985.

[75] A.C. Yao. Probabilistic computations — toward a unified measure of complexity. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 222–227, 1977.

[76] L.-Q. Zheng and P.-Å. Larson. Speeding up external mergesort. University of Waterloo, Department of Computer Science Research Report CS-92-??, 1992.