An Adaptive Generic Sorting Algorithm that Uses Variable Partitioning

Vladimir Estivill-Castro and Derick Wood

Technical Report HKUST-CS96-19
June 1996

Abstract

A sorting algorithm is adaptive if its run time, for inputs of the same size $n$, varies smoothly from $O(n)$ to $O(n \log n)$ as the disorder of the input varies. Recently, researchers have focused their attention on sorting algorithms that are optimally adaptive with respect to several measures of disorder, since the type of disorder in the input is usually not known. They have demonstrated a need to develop tools for the construction of sorting algorithms that are adaptive with respect to large classes of measures.

We present Partition Sort, a new generic sorting algorithm for which we can establish adaptivity. It is based on divide-and-conquer in which the number of subproblems at each level of recursion depends on the disorder of the input. We apply this generic algorithm to obtain optimal adaptivity for several measures of disorder. In addition, we introduce Randomized Partition Sort, a randomized version of Partition Sort, that not only is adaptive in the expected case, but also simplifies implementations.

---

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada, from the Information Technology Research Centre of Ontario, and from the Research Grants Committee of Hong Kong.
An Adaptive Generic Sorting Algorithm thatUses Variable Partitioning

Vladimir Estivill-Castro \( ^2 \) \hspace{1cm} Derick Wood \( ^3 \)

Abstract

A sorting algorithm is adaptive if its run time, for inputs of the same size \( n \), varies smoothly from \( O(n) \) to \( O(n \log n) \) as the disorder of the input varies. It is well accepted that files that are already sorted are often sorted again and that nearly sorted files often occur. Recently, researchers have focused their attention on sorting algorithms that are optimally adaptive with respect to several measures of disorder, since the type of disorder in the input is usually not known. They have demonstrated a need to develop tools for the construction of sorting algorithms that are adaptive with respect to large classes of measures.

We present Partition Sort, a new generic sorting algorithm for which we can establish adaptivity. It is based on divide-and-conquer in which the number of subproblems at each level of recursion depends on the disorder of the input. We apply this generic algorithm to obtain optimal adaptivity for several measures of disorder. In addition, we introduce Randomized Partition Sort, a randomized version of Partition Sort, that not only is adaptive in the expected case, but also simplifies implementations (for example, it avoids median finding).

Keywords: algorithm design; analysis of algorithms; adaptive sorting.

1 Introduction

Intuitively, a sorting algorithm is adaptive if its run time, for inputs of the same size \( n \), varies smoothly from \( O(n) \) to \( O(n \log n) \) as the disorder or nearly-sortedness of the input varies and it adapts its performance without knowing the disorder in advance. It is well accepted that sorted files are often sorted again and that nearly sorted files often occur. Thus, adaptive sorting is not only theoretically interesting but also it promises to improve current practice [4].

One critical issue that we face when we design adaptive sorting algorithms is how to measure disorder or nearly-sortedness. There are a number of measures of disorder [5]; the most well-known measure is the number \( Inv(X) \) of inversions in a sequence \( X = \langle x_1, \ldots, x_n \rangle \), where an inversion is a pair \( (i, j) \) of positions in \( X \) such that \( i < j \) and \( x_i > x_j \). It is certainly useful to design a sorting algorithm that is adaptive for a specific measure; however, the diversity of measures of disorder suggests that an algorithm that is adaptive for several measures is much more useful, since in the general setting, the type of disorder in the input is usually not known. Initially, most developments arose from the study of specific measures, but increasingly, researchers have focused their attention on sorting algorithms that are adaptive for several measures [5]. Unfortunately, experience has shown that, whenever we obtain a sorting algorithm that is adaptive for more measures than previous algorithms, its implementation is more complex (for example, more complex data structures to represent the sorted sequence in Insertion Sort can make the algorithm adaptive.

\( ^1 \)This work was partially carried out under grants from the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre of Ontario.

\( ^2 \)School of Information Systems, Queensland University of Technology, Red Hill Q1059, Australia. E-mail: vlad@ics.qut.edu.au.

\( ^3 \)Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: dwood@cs.ust.hk.
for more measures [5]); thus, the new algorithm is impractical. There is a need to develop tools for the construction of practical sorting algorithms that are adaptive for large classes of measures of disorder.

Divide-and-conquer is a general and extremely successful strategy for the design and analysis of algorithms; in particular, it is the basis of many sorting algorithms for the comparison-based model of computation. Recently, Estivill-Castro and Wood [4] studied divide-and-conquer as the basis of a universal method for the development of adaptive sorting algorithms. The design of generic sorting algorithms has at least two advantages:

- Algorithm designers can focus their efforts on the combinatorial properties of the measures of disorder that are of interest, rather than in the combinatorial properties of the algorithm.
- Algorithm designers can regulate the trade-off between the number of measures for which the sorting algorithm is adaptive and the complexity of its implementation.

A generic adaptive sorting algorithm that we designed previously [4] requires that the division protocol partition the data into a fixed number of parts. Now, we present new generic adaptive sorting algorithm that uses variable partitioning. Not only does this new algorithm have the preceding advantages but also, because of the added flexibility, it enables us to obtain optimal adaptivity for even more measures.

In Section 2, we introduce some measures of disorder. In Section 3, we present Partition Sort, the new generic adaptive sorting algorithm and we present four applications of Partition Sort in Section 4. In Section 5, we present its randomized version and, in Section 6, we present an application in which we eliminate median finding. We conclude with some final remarks in Section 7.

In the following, we denote the base 2 logarithm with $\log$ and the set of all finite sequences of integers with $N^\mathbb{N}$. We denote the length of a sequence $X$ with $|X|$ and the cardinality of a set $S$ with $|S|$.

## 2 Evaluating disorder

To express the performance of a sorting algorithm in terms of the input length and disorder, we must assess the disorder in the input. Intuitively, a measure of disorder is a function that is minimized when a sequence has no disorder and depends on only the relative order of the elements in the sequence.

We describe the most common measures of disorder, many others can be found in the literature [5]. We may consider that, in terms of the disorder it represents, an inversion pair, whose elements are far apart, is more significant than an inversion pair, whose elements are closer. We define $Dis$ as the largest distance determined by an inversion. Often local disorder is not as important as global disorder; we define $Max$ as the largest distance an element must travel to reach its sorted position. Ascending runs constitute sorted segments of the data. We define $Runs(X)$ as the minimum number of contiguous up-sequences that are required to cover $X$. A natural generalization of $Runs$ is the minimum number $SUS(X)$ of ascending subsequences that are required to cover $X$. We can generalize this measure again and define $SMS(X)$ (for shuffled monotone subsequence) as the minimum number of monotone (ascending or descending) subsequences that are required to cover $X$. For example, for $W_0 = \{6,5,8,7,10,9,4,3,2,1\}$, $Runs(W_0) = 8$, $SUS(W_0) = \|\{6,8,10\},\{5,7,9\},\{4\}\| = 6$, and $SMS(X) = \|\{6,8,10\},\{5,7,9\},\{4,3,2,1\}\| = 3$.

These examples show that disorder can be evaluated in many ways and different measures take account of different types of disorder. We wish to design sorting algorithms that are optimally adaptive to as many measures as possible. A sorting algorithm is optimally adaptive with respect to a measure $M$ if it achieves, under the comparison-based model of computation, maximal adaptivity with respect to $M$ [11]. Thus, we concentrate on comparison-based sorting algorithms in which the numbers of data moves and other operations (for example, pointer assignments) are bounded by the number of comparisons. Also, when we say that a sorting algorithm is optimally adaptive
with respect to a measure we implicitly assume sequences of distinct elements [11]. We can handle sequences with repeated elements in at least two ways. One way is to extend the measures of disorder; for example, we can define $SUS$ to be the minimum number of nondecreasing sequences. Alternatively, if stability is important, then we can tag the elements with their initial position to obtain distinct composite keys. The performance bounds that we obtain hold for both points of view and they illustrate that the algorithms are also adaptive to the appearance of multiple keys in the input.

3 Partition Sort

The structure of the generic sorting algorithm, Partition Sort, should not be surprising: It uses divide and conquer. What is novel, is that we establish adaptivity with respect to a measure $M$ of disorder by ensuring that the partition protocol satisfies certain requirements.

Partition Sort divides the input into $s$ parts, in which the number $s$ of parts is not necessarily fixed and depends on the disorder in the input. If we cannot divide a sequence into smaller sequences or it is so close to sorted order that we can sort it by an alternative sorting algorithm, then we consider it to be simple. The sizes of the individual parts are not restricted, but the total size of the simple parts must be at least a fixed fraction of the total size of the parts. Sophisticated partition protocols that may require more than linear time may be used.

Sorting can be considered a process in which the disorder in the input is reduced to zero. If the partition protocol repeatedly introduces disorder, then we are producing smaller and harder instances of sorting problems and we do not achieve adaptivity. Thus, the partition protocol that we use must ensure that the disorder has been substantially reduced in a fixed fraction of the parts. We formalize this restriction later.

For a sequence $X$, 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 value $p$ is called the size of the partition.

Partition Sort works as follows. 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 simple $X_i$ is sorted using a secondary sorting algorithm that sorts simple sequences. Each nonsimple part is sorted by a recursive call of Partition Sort. In the final step, all the sorted parts are merged; see Figure 1.

We can merge $\|P(X)\|$ sorted parts in $O(\|X\| (1 + \log(\|P(X)\| + 1)))$ time (pairing the parts $X_{2i-1}$ and $X_{2i}$, for $i = 1, \ldots, \|P(X)\|/2$, and merging them reduces the number of parts by a half in linear time). If Partition Sort takes no more time to compute the partition and resolve its recursive calls, then we obtain a sorting algorithm that takes $O(\|X\| (1 + \log(\|P(X)\| + 1)))$ total time. If $\|P(X)\|$ is related to a measure $M$ of disorder, then Partition Sort is adaptive with respect to $M$. We can ensure that Partition Sort resolves its recursive calls with the claimed worst-case performance only if a constant fraction of its 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.

**Theorem 3.1** Let $c, d, k, \in \mathbb{R}$ be constants with $0 < c \leq 1$, $d > 0$ and $k > 0$. Let $PP$ be a partition protocol such that, for all sequences $X \in N^\leq N$:

1. To detect that $X$ is nonsimple, to create a partition $P(X) = \{X_1, \ldots, X_{\|P(X)\|}\}$ of $X$ using $PP$, and to merge the sorted versions of $X_1, \ldots, X_{\|P(X)\|}$ takes no more than $d \|X\| (1 + \log(\|P(X)\| + 1))$ comparisons;

2. The sum of the lengths of the simple sequences in $P(X)$ is at least $c \|X\|$;

3. For all $X_i$ in $P(X)$, if $PP$ is applied to $X_i$, then no more than $\|P(X)\|$ sequences are obtained;

4. There is a sorting algorithm that sorts any simple $X_i$ making no more than $k \|X_i\| \log(\|P(X)\| + 1)$ comparisons.
Partition Sort($X$)

$X$ is sorted. Terminate.

$X$ is simple. Sort it using an algorithm for simple sequences.

$X$ 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 1: Partition Sort. The generic sorting algorithm.

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.

Sketch of the proof of Theorem 3.1: Since $\sum_{i \geq 1, |X_i| \text{ simple}} |X_i| \geq c |X|$, for all nonsimple sequences $X$, there is a constant $u$ with

$$u \geq \frac{d + k \sum_{i \geq 1, |X_i| \text{ simple}} |X_i|/|X|}{1 - \sum_{i \geq 1, |X_i| \text{ nonsimple}} |X_i|/|X|}.$$ (1)

Now it is straightforward to prove, by induction on the length of $X$, that, for all sequences $X \in N^N$, Partition Sort makes no more than $u |X|(1 + \log(\|P(X)\| + 1))$ comparisons.

We present only the induction step since the basis is immediate. If $X$ is simple, then $X$ is sorted by an alternative algorithm in the required bound. If $X$ is nonsimple, then Partition Sort makes at most $d |X|(1 + \log(\|P(X)\| + 1))$ comparisons to detect that $X$ is nonsimple, to construct the partition, and to merge the results of the recursive calls. We now account for the time required to resolve the recursive calls. For all simple $X_i$ in the partition, we use no more than $k \sum_{i \geq 1, |X_i| \text{ simple}} |X_i|(1 + \log(\|P(X)\| + 1))$ comparisons in total. For all nonsimple $X_i$ in the partition, by the induction hypothesis, we use no more than $\sum_{i \geq 1, |X_i| \text{ nonsimple}} u |X_i|(1 + \log(\|P(X)\| + 1))$ comparisons in total. Thus, Partition Sort requires no more than

$$d |X|(1 + \log(\|P(X)\| + 1)) + k \sum_{i \geq 1, |X_i| \text{ simple}} |X_i|(1 + \log(\|P(X)\| + 1)) + u \sum_{i \geq 1, |X_i| \text{ nonsimple}} |X_i|(1 + \log(\|P(X)\| + 1))$$

comparisons in total and, by Equation 1, it takes no more than $u |X|(1 + \log(\|P(X)\| + 1))$ comparisons.

Before we present some applications of Partition Sort, we make two observations. First, Generic Sort [4] can be regarded as a variant of Partition Sort; the specific form of Generic Sort, however, results in a theorem that is easier to apply. Second, Chen and Carlsson [2] defined generic measures of disorder as “...minimal size of a specific type $\tau$ of partition;” however, they did not use this notion with the generality given by Theorem 3.1.

4 Applications of Partition Sort

We first apply Partition Sort to Natural Mergesort [9, page 161], which partitions the input into ascending runs. A sequence is simple if it is sorted and, in fact, in this application all parts of the
partition are simple. We obtain a partition into ascending runs, in linear time, by scanning the input once. Since the number of ascending runs is directly related to the measure \textit{Runs}, see Section 2, Natural Mergesort takes \(O(|X|(1 + \log \text{Runs}(X) + 1))\) time and is, therefore, \textit{Runs}-optimal; a new proof of Mannila's result [11].

Next, we use \textit{Partition Sort} to obtain a sorting algorithm that is optimally adaptive with respect to \textit{Dis}. For a given sequence \(X\), the partition protocol computes, in \(O(|X|)\) time, the value \(\text{Dis}(X)\) as described, by Altman and Igarashi [1]. Then, it divides \(X\) into \(\text{Dis}(X) + 1\) subsequences, each containing the elements that are \(\text{Dis}(X) + 1\) elements apart (for example, if \(\text{Dis}(X)\) is one, then the partition protocol gives two subsequences, one of elements in even positions and one of elements in odd positions). By definition, after partitioning, all parts are sorted and, therefore, simple. By Theorem 3.1, the algorithm sorts \(X\) in \(O(|X|(1 + \log \text{Dis}(X) + 1))\) time, which is optimally adaptive with respect to \textit{Dis} and \textit{Max} [3].

Our third application is to Skiena's \textit{Melsort}, which we now describe [13]. When \textit{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 \textit{encroaching lists} of \(X\). Since encroaching lists are sorted, the simple parts are sorted sequences and, after the partition, all parts are simple. In the final step, \textit{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 \(x_i\) \textit{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 that 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 \(\text{Enc}\) of disorder; thus, in the example, \(\text{Enc}(4, 6, 5, 2, 9, 1, 3, 8, 0, 7) = 3\). Since the encroaching set can be constructed in \(O(|X|(1 + \log \text{Enc}(X) + 1))\) time, \textit{Melsort} takes \(O(|X|(1 + \log \text{Enc}(X) + 1))\) time; therefore, \textit{Melsort} is \(\text{Enc}\)-optimal and we have obtained a new proof of Skiena's result [13].

Our fourth application is to \textit{Slab Sort} [10], a sorting algorithm that achieves optimality with respect to \textit{SMS}(\(X\)) (the minimum number of shuffled monotone subsequences of \(X\)) and, therefore, optimality with respect to \textit{Dis}, \textit{Max}, \textit{Runs} and \textit{SUS}. Although \textit{Slab Sort} is an important theoretical breakthrough it has limited practical value because it requires repeated median finding. Before describing \textit{Slab Sort}, we must define \textit{zig-zag shuffles} because, in this application, zig-zag shuffles are the simple sequences.

\textbf{Definition 4.1} Letting \(X \in N^{<N}\), we denote a subsequence of \(X\) by \(\langle x_{i(1)}, \ldots, x_{i(s)} \rangle\), where \(i : \{1, \ldots, s\} \to \{1, \ldots, |X|\}\) is injective and monotonically increasing. We say that a subsequence \(\langle x_{i(1)}, \ldots, x_{i(s)} \rangle\) is an up-sequence if \(x_{i(1)} < x_{i(2)} < \cdots < x_{i(s)}\). Similarly, we say that a subsequence \(\langle x_{i(1)}, \ldots, x_{i(s)} \rangle\) is a down-sequence if \(x_{i(1)} > x_{i(2)} > \cdots > x_{i(s)}\). A subsequence is \textit{monotone} if it is either a down-sequence or an up-sequence. We say that two subsequences \(\langle x_{i(1)}, \ldots, x_{i(s)} \rangle, \langle x_{j(1)}, \ldots, x_{j(t)} \rangle\) \textit{intersect} if \(\{i(1), i(1) + 1, \ldots, i(s)\} \cap \{j(1), j(1) + 1, \ldots, j(t)\} \neq \emptyset\). A sequence \(X\) is a \textit{zig-zag shuffle} if there is a partition of \(X\) into \(\text{SMS}(X)\) monotone subsequences such that no up-sequence intersects a down-sequence.

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; thus, \(W_0\) is not a zig-zag shuffle. We can obtain a geometric interpretation of zig-zag shuffles as follows: Given a sequence \(X\), for each \(x_i\), we plot a point \((i, \text{rank}(x_i))\) in the plane, where \(\text{rank}(x_i)\) is the position of \(x_i\) when \(X\) is sorted. 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 shuffle if no down-sequence curve intersects an up-sequence curve; for example, see Figure 2.

Zig-zag shuffles are important because if $X$ is a zig-zag shuffle and $\text{SMS}(X) \leq z$, then $\text{Enc}(X) \leq z$ and $\text{Melsort}$ sorts $X$ in $O(|X| (1 + \log[z + 1]))$ time; hence, $\text{Melsort}$ can be used to sort zig-zag shuffles. Moreover, for any $X$, if $\text{Melsort}$ attempts to construct an encroaching set with more than $z$ sorted lists, then we have discovered in $O(|X| (1 + \log[z + 1]))$ time that $X$ is nonsimple.

The partition protocol of $\text{Slab Sort}$ first finds a value $z$ such that $\text{SMS}(X) \leq z \leq (\text{SMS}(X))^2$, using $O(|X| (1 + \log[z + 1]))$ comparisons, we give the details of this procedure later. Next, the partition protocol constructs a partition of the input sequence $X$ by stably partitioning it into $p = \lceil z^2/2 \rceil$ parts of almost equal size using the $\lceil 1 + |X|/p \rceil$-th, $\lceil 1 + 2|X|/p \rceil$-th, ..., $\lceil 1 + (p-1)|X|/p \rceil$-th elements as pivots. These elements are found by repeated median finding that make a total of $O(|X| \log[p+1])$ comparisons. The partitioning is similar to the partitioning in $\text{Quicksort}$, but with $p-1$ pivots, and it can be carried out in $O(|X| (1 + \log[p+1]))$ time, as is shown by Frazer and McKellar for the partitioning in $\text{Samplesort}$ [6].

To apply Theorem 3.1, we must ensure that a constant fraction of the elements in $X$ belong to a simple subsequence. Let $X$ be a sequence with $\text{SMS}(X) \leq z$. Assume that a minimum partition by monotone sequences has $z_1$ down-sequence and $z_2$ up-sequence 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 $\text{Slab Sort}$ stably partitions $X$ into $p = \lceil z^2/2 \rceil$ parts $X_1, \ldots, X_p$, at least half of these parts are zig-zag shuffles with $\text{SMS}(X_i) \leq z$. Hence, at least half of the parts are simple and, since all parts are almost the same size, at least half of the elements in the original sequence belong to simple sequences.

We now explain how the partition protocol obtains a value $z$ such that $\text{SMS}(X) \leq z \leq (\text{SMS}(X))^2$. The partition protocol starts with a guess of $z = 2$, partitions $X$ into $p = \lceil z^2/2 \rceil$ parts, and tests whether half of these parts are simple using $\text{Melsort}$. If this condition is met, the partition protocol continues to construct the partition; otherwise, it squares its guess of $z$ and repeats the test.

## 5 Randomized Partition Sort

In contrast to the pessimistic view taken by a worst-case analysis, expected-case analysis provides a more practical view, because worst-case situations are normally unlikely. There are two approaches to expected-case complexity [7], the **distributional approach** and the **randomized approach**. The distributional approach may be inaccurate, since the probabilistic assumptions needed to carry out the analysis may be false and do not lead to useful notions of adaptivity [5]. The behavior of randomized sorting algorithms is, however, independent of the distribution of the instances to be sorted and they can be adaptive to the disorder in the input. Moreover, with randomization, we can often simplify sorting algorithms and their data structures; for example, we give a version of $\text{Slab Sort}$ that does not require median finding.

We define **Randomized Partition Sort**, a randomized version of **Partition Sort**. It is based on
Randomized Partition Sort ($X$)

$X$ is sorted. Terminate.

$X$ is simple. Sort it using an algorithm for simple sequences.

If $X$ is neither sorted nor simple,

Using a randomized partition 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 3: A relaxed, randomized partition protocol results in Randomized Partition Sort.

divide-and-conquer with a randomized partition protocol. We relax the requirement, however, that deeper recursive levels have no more parts than the partition at the first level. For a given 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 3. The number of parts $\|P(X)\|$ in the partition is a random variable; thus, the expected time taken for the merge is no more than

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

where $\gamma > 0$ is a constant and $E[\|P(X)\|]$ is the expected value of $\|P(X)\|$.

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

Theorem 5.1 Let $c, d$ and $k$ be constants with $0 < c \leq 1$, $d > 0$ and $k > 0$. Let RDP be a randomized partition protocol such that, for all sequences $X \in \mathbb{N}^N$:

1. If $X$ is nonsimple, then the RDP realizes this and 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;

2. For all parts of all partitions, $E[\|P(X)\|] \leq E[\|P(X)\|]$;

3. If $J(P(X))$ is the set of indices of the simple parts, then

$$c[X] \leq \sum_{P(X)} Pr[P(X)] \sum_{i \in J(P(X))} |X_i|;$$

4. There is a sorting algorithm $S$ that sorts any simple sequence $X_i$ in $P(X)$, making no more than $k[X](1 + \log[E[\|P(X)\|] + 1])$ 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 initial partition.

Proof: Let $\gamma > 0$ be a constant such that the merging of the results of recursive calls takes no more than $\gamma[X](1 + \log[E[\|P(X)\|] + 1])$ time. Let $u > (d + k + \gamma)/c$ and let $E[T_{RP S}(X)]$ denote the expected number of comparisons performed by Randomized Partition Sort on input
X. We prove by induction on |X| that \( E[T_{RPS}(X)] \leq |X|\left[1 + \log\left(E\left[\|P(X)\|\right] + 1\right)\right] \). We present only the induction step since the basis is immediate. We assume that if Z is a sequence with |Z| < |X|, then \( E[T_{RPS}(Z)] \leq |Z|\left[1 + \log\left(E\left[\|P(Z)\|\right] + 1\right)\right] \). Now, \( E[T_{RPS}(X)] \) is the sum of four quantities. The first is the expected number of the comparisons used to partition X and it is bounded by \( d|X|\left(1 + \log\left(E\left[\|P(X)\|\right] + 1\right)\right) \). The second is the expected number of comparisons used to identify and sort the simple parts and it is bounded by \( k|X|\left(1 + \log\left(E\left[\|P(X)\|\right] + 1\right)\right) \). The third is the expected number of comparisons used to merge the sorted parts and it is bounded by \( \gamma|X|\left(1 + \log\left(E\left[\|P(X)\|\right] + 1\right)\right) \). The fourth is the expected number of comparisons used by the recursive calls on the nonsimple parts of the partition. Let \( X_i \), for \( i = 1, \ldots, \|P(X)\| \), be the parts of X with respect to the partition \( P(X) \), and let \( J(P(X)) = \{ i \mid 1 \leq i \leq \|P(X)\| \text{ and } X_i \text{ is simple} \} \). Thus, by the induction hypothesis,

\[
E[T_{RPS}] \leq (d + k + \gamma)|X|\left[1 + \log\left(E\left[\|P(X)\|\right] + 1\right)\right] + \sum_{P(X) : j \in J(P(X))} E[T_{RPS}(X_i)]
\]

\[
\leq (d + k + \gamma)|X|\left[1 + \log\left(E\left[\|P(X)\|\right] + 1\right)\right] + u(1 + \log\left(E\left[\|P(X)\|\right] + 1\right)) \sum_{i \in J(P(X))} E[\|X_i\|]
\]

\[
\leq (d + k + \gamma + (1 - c)u)|X|\left[1 + \log\left(E\left[\|P(X)\|\right] + 1\right)\right]
\]

\[
\leq u|X|\left[1 + \log\left(E\left[\|P(X)\|\right] + 1\right)\right],
\]

which completes the induction step. \( \square \)

6 An Application of Randomized Partition Sort

We apply Randomized Partition Sort to obtain Randomized Slab sort, a randomized version of Slab Sort. Rather than finding elements that partition the input sequence X into parts of almost equal length, a sample of \( p = \lceil k^2/2 \rceil \) elements is chosen at random. These elements are used as pivots to stably partition X. This partition protocol is analogous to Samplesort [6]. Samplesort is a generalization of Randomized Quick sort in which, rather than choosing a single pivot at random, a sample of several pivots is chosen at random. To select any subset of p elements with equal probability, the sample is selected as described by Knuth [8, Algorithm S, page 137]. Thus, the stable partitioning can be carried out with \( O(|X|\left(1 + \log\left(p + 1\right)\right)) \) comparisons and with a much smaller multiplicative constant than with median finding; however, the length of the i-th part is now a random variable with expectation \((|X| - p)/(p + 1)\) [6].

Using Theorem 5.1, we now show that Randomized Slab sort achieves the same asymptotic complexity as Slab Sort, but in the expected case. Since we do not know the lengths of the parts that are successfully sorted by Melsort, the analysis is different from the analysis of Samplesort [6] and Slab Sort [10]. We verify the conditions of Theorem 5.1.

Let \( L_Z(X) \) be the sum of the lengths of the parts that are zig-zag shuffles and let \( L_N(X) \) be the sum of the lengths of the parts that are not zig-zag shuffles. \( L_Z(X) \) and \( L_N(X) \) are random variables over all partitions of X into p parts with \( L_Z(X) + L_N(X) = |X| \). Moreover, \( L_Z(X) \) is always larger than the sum of the lengths of the \( \lfloor p/2 \rfloor \) smallest parts.

Let \( L(X) \) denote the length of the \( \lfloor p/2 \rfloor \) smallest parts. The expected length of the i-th smallest part, when partitioning X into p pieces, is

\[
\frac{|X|}{p} \sum_{j=1}^{i} \frac{1}{p - (j - 1)} \quad \text{[12, page 66]}
\]

thus,

\[
E[L(X)] = \frac{\lfloor p/2 \rfloor}{p} \sum_{i=1}^{\lfloor p/2 \rfloor} \frac{|X|}{p} \sum_{j=1}^{i} \frac{1}{p - (j - 1)}.
\]
Now, let $H_n = \sum_{i=1}^{n} 1/i$ denote the $n$-th harmonic number. Algebraic manipulation shows that $E[L(X)] = |X| H_{|p/2|} - H_p + 1/2$; therefore, $E[L(X)] \geq (1 - \ln(2)) |X| / 2$ and we have proved the following result.

**Lemma.** Let $X \in N^{\leq N}$ and $S$ be a uniform sample of $p$ elements. Then, there is a constant $c_0$ such that $0 < c_0 < 1$ and the expected length of the smallest $|p/2|$ parts is at least $c_0 |X|$.

Therefore, there is a constant $0 < c_0 < 1$ such that $E[L_2(X)] \geq c_0 |X|$ and $E[L_N(X)] \leq (1 - c_0) |X|$, which allows us to apply Theorem 5.1 to obtain the following result.

**Theorem 6.1** Let $T_S(X)$ denote the number of comparisons taken by Randomized Slabsort($X$). Then, there is a constant $d > 0$ such that, for all sequences $X \in N^{\leq N}$, $E[T_S(X)] \leq d |X| (1 + \log [S M S(X) + 1])$.

Thus, we have obtained a sorting algorithm that is optimally adaptive for $S M S$, $S U S$, $R u n s$, $D i s$ and $M a x$ and does not use median finding.

7 Final Remarks

There are many measures of disorder; for example, the number of operations that are required to rearrange a sequence into sorted order may be our first concern. We define $E x c$ as the minimum number of exchanges that are required to sort a sequence. We may also consider that disorder is produced by the insertion of some records into the wrong positions. We define $R e m$ as the minimum number of elements that must be removed to obtain a sorted subsequence. A sorting algorithm that is optimally adaptive for $S M S$ is also optimally adaptive for $R u n s$, $M a x$, $D i s$, and $S U S$ [10], but, by combining partition protocols, we can also achieve optimality for $I n v$, $E x c$ and $R e m$ [5]. This result illustrates the power of our approach: concentrating on design of partition protocols and, then, alternating them to obtain adaptivity for more measures.

All known adaptive sorting algorithms that are based on divide-and-conquer achieve adaptivity by choosing an appropriate partition protocol in the division step. An interesting open question is: can we achieve adaptivity by choosing an appropriate method of combining subsequences? For example, we might assume that subproblems are only nearly sorted. In some sense, such approach is used in *bitonic sort* [9], where bitonic sequences are merged to give sorted sequences.

References


