An Overview of Adaptive Sorting

Alistair Moffat[*] Ola Petersson[†]

Abstract: An algorithm is *adaptive* if 'easy' problem instances are solved faster than 'hard' instances. Here we give a tutorial overview of the field of adaptive sorting, considering in turn each of the three main paradigms for the design of sorting algorithms. We show that each of these paradigms leads to a corresponding family of adaptive algorithms.

Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—*sorting and searching*.

General Terms: Algorithms, Theory.

Keywords: adaptive sorting, measures of presortedness, nearly sorted.

1 Introduction

The problem of sorting is one of the most fundamental in computing, and has been intensively studied for many years. Moreover, it is also a very practical problem, and even trivial interactions with a computer system often involve sorting. For example, obtaining a directory listing or querying the current users of a system will probably involve sorted output.

Here we concentrate on *adaptive* sorting algorithms. An algorithm is said to be adaptive if 'easy' problem instances are solved faster than 'hard'. For example, the 'easiest' possible instance of the sorting problem is probably to 'sort' an already sorted list:

$$X_1 = (10, 11, 16, 18, 21, 27, 45, 64, 65, 67, 84, 85, 86).$$

As 'human computers' we have no difficulty in recognising the special nature of list X_1, and can quickly output—write down—the sorted equivalent. Not so 'easy' to process is list X_2:

$$X_2 = (67, 10, 65, 16, 64, 21, 85, 18, 84, 27, 45, 86, 11),$$

and somewhere in between, 'nearly sorted', is list X_3:

$$X_3 = (10, 11, 18, 21, 27, 45, 16, 64, 85, 65, 67, 84, 86).$$

An adaptive algorithm attempts to capture and act upon in some analytic manner this intuitive notion of 'nearly sorted'. To be of interest, an adaptive

[*]Dept. Comp. Sci., The University of Melbourne, Victoria 3052, Australia. alistair@cs.mu.OZ.AU.

[†]Dept. Comp. Sci., Lund University, Box 118, S-221 00 Lund, Sweden. ola@dna.lth.SE.

sorting algorithm should, as a minimum specification, consume[1] $O(n)$ time on a sorted list, and never more than $O(n \log n)$ time on any list, where it is supposed that the list contains n items and the only operation permitted of the algorithm is pairwise comparison of items. We would also hope that the transition between these two extremes of permitted behaviour should be 'smooth' in some sense. The $\Omega(n)$ requirement comes from the straightforward observation that no algorithm can claim to sort its input without every item being involved in at least one comparison; and the $\Omega(n \log n)$ worst case requirement is developed from an argument based upon the minimum depth of a binary decision tree with $n!$ leaves, where each leaf of the decision tree represents one possible input permutation [14, p.183].

The interest in adaptive sorting is motivated by two concerns: first, a purely academic curiosity, to see (asymptotically) just how fast certain categories of list can be sorted; and second, by an empirical observation that many lists to be sorted in practice are already nearly sorted, and applications requiring such sorting can be made to execute faster. For example, if a large sorted file of records is to be edited (old records altered or deleted, new records appended) and then re-sorted, it is likely that the number of records in the file that must be moved to regain sorted order is small, and the use of a $\Theta(n \log n)$ sort might be unnecessarily expensive. An understanding of adaptive sorting and the requirements of a particular application might lead to significant performance improvements.

2 Measures of Presortedness

To formalise the notion of 'nearly sorted' several different *measures of presortedness* [19] have been proposed. Perhaps the most intuitive of these is the number of ascending sequences or *Runs* in the input. For example, if $\mathit{Runs}(X)$ is the number of ascending runs in list X, $\mathit{Runs}(X_1) = 1$; $\mathit{Runs}(X_2) = 7$; and $\mathit{Runs}(X_3) = 3$. The maximum number of runs appears in a reverse sorted list:

$$X_4 = (n, n-1, n-2, \ldots, 2, 1).$$

An algorithm that is *adaptive with respect to Runs* has a running time that is a function both of the number of items n in the list and the number of runs in the list. As we shall see below, the best bound that can be achieved in terms of adaptivity with respect to *Runs* is[2] $\Theta(n \log \mathit{Runs}(X))$; an algorithm which attains this bound is said to be *optimally adaptive* (with respect to *Runs*).

Another obvious measure of presortedness is the number of *inversions*—pairs of items that are out of order. If x_i is the i'th item of the list X, then the number of inversions $\mathit{Inv}(X)$ in X is defined as

$$\mathit{Inv}(X) = |\{(i,j) : i < j \text{ and } x_i > x_j\}|.$$

[1]A function $f(n)$ is said to be $O(g(n))$ if $f(n) \leq c \cdot g(n)$ for some positive constant c and all sufficiently large values of n; is said to be $\Omega(g(n))$ if $g(n)$ is $O(f(n))$; is said to be $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$; and is said to be $o(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is *not* $\Theta(g(n))$.

[2]It is convenient to assume, within the big-O notation, that $\log x = \log_2 \max\{2, x\}$.

Using the same example lists, $\mathit{Inv}(X_1) = 0$ and $\mathit{Inv}(X_3) = 7$. It is straightforward to demonstrate that *Inv* and *Runs* are 'independent' (in some sense) by considering the two lists X_5 and X_6:

$$\begin{aligned} X_5 &= (n/2+1, n/2+2, n/2+3, \ldots, n, 1, 2, \ldots, n/2) \\ X_6 &= (2, 1, 4, 3, 6, 5, \ldots, n, n-1). \end{aligned}$$

List X_5 has a small number of runs, and so, according to *Runs*, is nearly sorted. However at $\Theta(n^2)$, X_5 has an almost maximal number of inversions, and so is completely *un*sorted according to *Inv*. Conversely, $\mathit{Inv}(X_6)$ is $O(n)$ and so X_6 is judged to be nearly sorted, while $\mathit{Runs}(X_6) = n/2+1$: that is, almost completely unsorted.

The third of the 'obvious' measures of presortedness is *Rem*: the minimum number of items that must be removed from the list such that the remaining items form a sorted subsequence. Using the same examples, $\mathit{Rem}(X_1) = 0$, and $\mathit{Rem}(X_3) = 2$ (remove 16 and 85). It is straightforward to verify that *Rem* and *Inv* are independent: list X_6 is unsorted according to *Rem* (but nearly sorted according to *Inv*); and list X_7

$$X_7 = (\sqrt{n}+1, \sqrt{n}+2, \ldots, n, 1, 2, \ldots, \sqrt{n})$$

has $\mathit{Rem}(X_7) = \sqrt{n}$ (which is low) and $\mathit{Inv}(X_7) = \Theta(n^{1.5})$ (that is, high).

More difficult to analyse is the relationship between *Rem* and *Runs*. List X_7 has only two runs, but is two runs 'better' than a *Rem* value of $\sqrt{n}$? More generally, a list of k runs must have $k-1$ pairs of items x_i, x_{i+1} such that $x_i > x_{i+1}$. In each of these pairs at least one of the items must contribute to *Rem*, since they cannot both remain in the sorted subsequence. Hence $\mathit{Runs}(X) \leq \mathit{Rem}(X) + 1$ for all X, and we might be tempted to conclude that *Runs* is, in some sense, a 'more accurate' measure than *Rem*.

However it is important to note that simple relationships such as $\mathit{Runs}(X) \leq \mathit{Rem}(X) + 1$, and indeed, $\mathit{Runs}(X) \leq \mathit{Inv}(X) + 1$ do *not* necessarily imply that *Runs* is a better measure than *Rem* or *Inv* (or vice versa). In the latter case it is perhaps obvious that we are comparing apples and oranges, since $1 \leq \mathit{Runs}(X) \leq n$ and $0 \leq \mathit{Inv}(X) \leq n(n-1)/2$; that is, the measures have different ranges. Similarly, even though *Runs* and *Rem* have almost identical ranges, it is erroneous to compare their values. It is not the numeric value that tells how close to sorted a measure judges a list, but the number of lists that are regarded by the measure to be *at least* as sorted as this particular list.

This notion was formalised by Mannila [19]. Let M be some measure of presortedness. Using a slightly different notation he defined

$$\mathit{below}_M(n, k) = \{\pi \in S_n : M(\pi) \leq k\},$$

where S_n is the set of all permutations of $\{1, 2, \ldots, n\}$. That is, the set $\mathit{below}_M(n, M(X))$ contains all n-permutations that are regarded by M as being at least as sorted as X.

Next, we introduce a notation for the number of comparisons needed to sort the permutations in a *below*-set [28]. Let M be a measure of presortedness, and

T_n the set of comparison trees for the set S_n. Then, for any $k \geq 0$ and $n \geq 1$,

$$C_M(n,k) = \min_{T \in T_n} \max_{\pi \in \mathit{below}_M(n,k)} \{\text{the number of comparisons spent by } T \text{ to sort } \pi\}.$$

That is, $C_M(n,k)$ tells us the best bound that we can hope for when presented with a list X with $M(X) = k$. Consequently, we say that a comparison-based sorting algorithm S that uses $T_S(X)$ steps on input X is *M-optimal*, or *optimal with respect to M,* if $T_S(X) = O(C_M(|X|, M(X)))$.

Proving that a sorting algorithm is optimal with respect to a measure M is done in two parts. One part is an upper bound on the time consumed by the algorithm, expressed in terms of the measure. The other part is a lower bound on $C_M(n,k)$, which is obtained by using the fact that [28]

$$C_M(n,k) = \Theta(n + \log |\mathit{below}_M(n,k)|).$$

Hence, it suffices to bound the cardinality of the *below*-set from below.

Let us give an example based upon the measures *Runs* and *Rem*. To give a lower bound on the size of $\mathit{below}_{\mathit{Runs}}(n,k)$ we must estimate the number of permutations in S_n that have k or fewer runs.

Consider any partitioning of $\{1, \ldots, n\}$ into k subsets $X_1, \ldots, X_k$ each of size n/k, where, for simplicity, we assume that k evenly divides n. Let π be the permutation corresponding to the concatenation of the *sorted* sets, taken in order. Then $\mathit{Runs}(\pi) \leq k$, and so $\pi \in \mathit{below}_{\mathit{Runs}}(n,k)$. It is easily seen that each partitioning gives rise to one and only one permutation π. Counting the number of different ways of performing the initial partitioning gives

$$|\mathit{below}_{\mathit{Runs}}(n,k)| \geq n! / \left(\frac{n}{k}!\right)^k, \tag{1}$$

which after taking the logarithm yields

$$C_{\mathit{Runs}}(n,k) = \Omega(n \log k).$$

Mannila [19] also gave this result, but with a more complex combinatorial construction.

Hence, an algorithm taking $O(n \log k)$ time to sort a list of n items and k runs is optimal with respect to *Runs*. As we shall see in Section 4, such algorithms do exist.

Let us now establish a similar bound for *Rem*. Let π be any permutation obtained by permuting the first $k+1$ items of the identity permutation in S_n. By construction, removing any k of the first $k+1$ items in π leaves a sorted list, and hence, $\mathit{Rem}(\pi) \leq k$. As there are $(k+1)!$ different π's that can be constructed this way, we have

$$C_{\mathit{Rem}}(n,k) = \Omega(n + \log((k+1)!)) = \Omega(n + k \log k).$$

Cook and Kim [5] described an adaptive sorting algorithm that runs in time $O(n + \mathit{Rem}(X) \log \mathit{Rem}(X))$, and so it is *Rem*-optimal.

We now return to the question of which is better, *Rem* or *Runs*. Consider the list X_8, a slight variant of X_7:

$$X_8 = (\sqrt{n}+1, \sqrt{n}+2, \ldots, n, \sqrt{n}, \sqrt{n}-1, \ldots, 2, 1).$$

We have $\mathit{Rem}(X_8) = \sqrt{n}$ and $\mathit{Runs}(X_8) = \sqrt{n}+1$. But according to the results above, a *Runs*-optimal algorithm may spend $\Theta(n \log \mathit{Runs}(X_8)) = \Theta(n \log n)$ time, while a *Rem*-optimal algorithm must complete the sorting of X_8 in $O(n + \mathit{Rem}(X_8) \log \mathit{Rem}(X_8)) = O(n)$ time. List X_5 suffices to complete the demonstration that *Runs* and *Rem* are independent.

The considerations outlined in the preceding paragraphs led us to make these further definitions [28]. Suppose that M_1 and M_2 are two measures of presortedness. Then

- M_1 is *superior* to M_2, denoted $M_1 \supseteq M_2$, if
$$C_{M_1}(|X|, M_1(X)) = O(C_{M_2}(|X|, M_2(X)));$$
- M_1 is *strictly superior* to M_2, denoted $M_1 \supset M_2$, if $M_1 \supseteq M_2$ and $M_2 \not\supseteq M_1$;
- M_1 and M_2 are *equivalent,* denoted $M_1 \equiv M_2$, if $M_1 \supseteq M_2$ and $M_2 \supseteq M_1$;
- M_1 and M_2 are *independent* if $M_1 \not\supseteq M_2$ and $M_2 \not\supseteq M_1$.

There are two important consequences of these definitions. Firstly, if algorithm S is M_1-optimal, and $M_1 \supseteq M_2$, then S is M_2-optimal. Secondly, if algorithm S is not M_2-optimal and $M_1 \supseteq M_2$ then S cannot be M_1-optimal.

We now briefly describe a number of other measures that have been discussed in the literature (see, for example, [14, 19, 26]).

The measure $\mathit{Exc}(X)$ counts the minimum number of exchanges of arbitrary items necessary to sort the input list; the measure $\mathit{Block}(X)$ counts the number of contiguous blocks of items in X that remain together in the sorted output (i.e., one more than the minimum number of 'cuts' that must be made if we were sorting the list using scissors and paper); and $\mathit{Max}(X)$ is the maximum distance any item in X is from its correct sorted position. For example, $\mathit{Exc}(X_3) = 7$; $\mathit{Block}(X_3) = 7$; and $\mathit{Max}(X_3) = 4$. The relationships between these measures and the measures already described will be discussed in Section 7.

Given these definitions, and the notions of *optimal adaptivity* and *superiority* there are several problems to be addressed:

- We should attempt to order (using $\supseteq$) existing measures, so that algorithm designers can know which measures or combinations of measures pose 'challenges';
- We should attempt to categorise current adaptive algorithms in terms of this ordering of measures;
- We should attempt to develop new measures that are strictly superior to current measures or combinations of current measures, to better capture the behaviour of particular algorithms. (We would also, however, like any new measures to still be 'natural'.)

In the sections that follow we give examples of each of these lines of investigation. We start by giving an overview of the main paradigms for the construction of sorting algorithms, and then show how each paradigm can also be applied to the development of adaptive sorting algorithms.

3 Sorting Paradigms

General purpose comparison based sorting algorithms fall into three broad categories.

Merge-based sorts subdivide the input list into a set of two or more sorted sublists, and by repeated merging of these lists reduce the set to one sorted list. A recursive formulation is often particularly convenient for describing the sequence of merges that take place. The standard example of this paradigm is *Straight Mergesort,* which sorts a list by recursively splitting the input list into two sublists of equal size, and then merges the two sorted halves using a straightforward linear time merge. Straight Mergesort requires at most $n \log_2 n - n + 1$ comparisons and $\Theta(n \log n)$ time.

Selection sorts repeatedly select an item from its input and place it into its correct location. Most selection sorts store the input list in a *priority queue,* and select the maximum item remaining. The simplest such algorithm is *Linear Selection Sort,* which uses the input array as priority queue, and takes $\Theta(n^2)$ time. Slightly more complex, but much more efficient, is *Heapsort*, which stores the items in a *heap.* As a heap can be built in linear time and supports extraction of the maximum item in logarithmic time, Heapsort runs in $\Theta(n \log n)$ time. An algorithm that does not select the maximum item, but *any* item, is *Quicksort,* which partitions the input about one of the items and then recursively sorts the two partitions. Quicksort consumes $\Theta(n^2)$ time in the worst case, but is $\Theta(n \log n)$ on *average,* and is widely held to be the best general purpose sorting method, because of its small implicit constant factor and the fact that it requires only $O(\log n)$ extra space.

The third of the three standard paradigms is sorting by *insertion.* These algorithms maintain the first $i-1$ items in some kind of sorted list, and repeatedly insert the i'th item, with i growing from 1 to n. The sorted list is initially empty, corresponding to $i = 1$; when i reaches n all of the items have been inserted into the sorted list and the algorithm is done. A well known example of this paradigm is *Linear Insertion Sort,* where the sorted component of the list is simply maintained in the first $i-1$ positions in the array being sorted. Linear Insertion Sort is normally regarded as being 'an $O(n^2)$ sorting algorithm'. More accurate is to say that Linear Insertion Sort requires $\Theta(n + \mathit{Inv}(X))$ time, and so on a sorted list will take linear time. However, in the worst case—a reverse sorted list—Linear Insertion Sort spends $\Theta(n^2)$ time.

Descriptions of these algorithms and further examples of the three paradigms can be found in Knuth [14].

4 Adaptive Merge Sorting

The essence of the merging paradigm is loosely captured in the following pseudo-code:

```
procedure Adaptive Merge Sort ($X$: list)
    Divide $X$ into ascending lists $X_1, X_2, \ldots, X_k$
    Let $Q$ be a queue containing the $k$ lists
    while $Q$ contains more than 1 list do
        Remove the first two lists from $Q$, and merge them
        Append the resultant list at the tail of $Q$
    endwhile
    return the single list in $Q$
end
```

The adaptivity of a merge sort depends on how the division and the merging is performed. One of the oldest adaptive sorting algorithms, Knuth's *Natural Mergesort* [14], differs from Straight Mergesort only in how the division is carried out. Rather than merging a set of lists each initially of length 1, as is done by Straight Mergesort, it first scans the input list locating the ascending runs. This pre-scan requires linear time, and will identify the starting point of each of the $k = \mathit{Runs}(X)$ ascending sublists in the input. Then the first run is merged with the second, the third with the fourth, and so on, resulting in $\lceil k/2 \rceil$ longer runs. These longer runs are then repeatedly merged pairwise until there is just one run left, which is the sorted list. Since the number of runs is approximately halved in each pass, each item takes part in $\lceil \log_2 k \rceil$ merges. Moreover, each item causes no more than constant cost per merge, and hence, Natural Mergesort runs in $O(n \log k)$ time, which is optimal with respect to *Runs*. Harris [11] describes a number of experiments in which Natural Mergesort is compared empirically with other sorting algorithms.

Carlsson, Levcopoulos, and Petersson [3] demonstrated that the adaptivity of Natural Mergesort can be extended significantly by applying an adaptive merging algorithm. This merging algorithm runs in sublinear time if the lists taking part in any particular merge are disjoint, or 'easily' merged in some other way. Their algorithm achieves optimality with respect to *Runs*, *Block*, and *Max*, but is not *Inv*-optimal.

Moffat [22] described an adaptive mergesort—*Margesort*—that attains *Runs*-optimality *without* pre-scanning the input identifying runs, instead making use of an adaptive merging algorithm similar to that of Carlsson, Levcopoulos, and Petersson. Moffat, Petersson, and Wormald [24] further analysed the behaviour of Margesort, and showed it to be optimal with respect to *Inv* and *Rem*, and also gave a counter-example that showed it to be *not* optimal with respect to *Block*.

The merging paradigm also leads to adaptive algorithms for other measures. Levcopoulos and Petersson [16] and Skiena [31] have developed algorithms that identify ascending subsequences where the items in each subsequence are not necessarily adjacent in the input. The behaviour of these algorithms is captured by a number of new measures. One such measure is $SUS(X)$, defined as the minimum number of shuffled upsequences into which X can be decomposed. For example the lists X_5 and X_6 of Section 2 can both be formed by the shuffle of two upsequences. In the case of X_5 there is no shuffling required; and in the case of X_6 the shuffle is 'perfect', with items in X_6 being drawn alternately from each of the two shuffles. Since a list with k ascending runs is, de facto, a member of $below_{SUS}(n, k)$, we have $C_{SUS}(n, k) = \Omega(n \log k)$ by eq. (1). Here we briefly describe an SUS-optimal algorithm that attains this bound.

The algorithm operates in two phases as follows. In the first phase each item in the input list is considered and appended to the j'th of a set of sorted lists X_k, chosen to be the smallest j such that x_i is greater than or equal to the item most recently appended to X_j. If x_i is less than all of the tail items, a new list is started, and x_i becomes the first item in that list. Item x_1 will always be the first item into X_1.

Then, in the second phase, the lists are merged, exactly as for Natural Mergesort.

Suppose that at the end of the first phase there are k lists. Each of the n items was appended to one of the lists; this takes $O(1)$ time per item. To identify the list takes longer. However the set of tail items in the lists always form a decreasing sequence, and if pointers to the tails are kept in an array, the correct list for x_i can be identified with a binary search in $O(\log k)$ time. Over all items, this totals $O(n \log k)$ time.

The merging of the second phase will also require $O(n \log k)$ time. To demonstrate that this algorithm is SUS-optimal all that remains is to show that $k = SUS(X)$. Consider the last item in list X_k. At the time it was appended to X_k the last item in list X_{k-1} was larger, and appeared earlier in the input list X. Similarly, there must be an item in X_{k-2} both larger than this item (in X_{k-1}) and earlier in X. Continuing in this fashion, there must be an item in X_1 larger than all of these items, and appearing before any of them. These k items, one in each list X_k, form a decreasing subsequence in the original list X, and so must of necessity appear in different shuffles. Hence $SUS(X) \geq k$. Since the lists X_i are a decomposition of X into upsequences, we must have $SUS(X) = k$.

This algorithm is thus optimal with respect to SUS. Furthermore, since $SUS(X) \leq Runs(X)$ for any list X, we have

$$\begin{aligned} C_{SUS}(|X|, SUS(X)) &= \Theta(|X| \log SUS(X)) \\ &= O(|X| \log Runs(X)) \\ &= O(C_{Runs}(|X|, Runs(X))). \end{aligned}$$

That is, we have shown that $SUS \supseteq Runs$. It is not difficult to show that $SUS \supset Runs$. Consider the list X_6 described earlier. This list must be sorted in linear time by an SUS-optimal algorithm, but since $Runs(X_6) = n/2 + 1$, a $Runs$-optimal algorithm is permitted to spend $\Theta(n \log n)$ time.

Skiena's *Melsort* [31] is also optimal with respect to these two measures, as well as the superior measure Enc, where Enc is the number of 'encroaching lists', defined operationally to be the number of sorted lists in the decomposition produced by the first pass of Melsort. Levcopoulos and Petersson's *Slabsort* [16], a hybrid of Melsort and Quicksort, is optimal with respect to all three, as well as the superior measure *SMS*, the number of shuffled monotone (i.e., either ascending *or* descending) sequences. Margesort has also recently been shown to be optimally adaptive with respect to *SMS* [25]. The merging paradigm can thus be seen to lead to a whole family of efficient adaptive algorithms.

Other merge-based adaptive sorting algorithms have been described by Chen and Carlsson [4], Estivill-Castro and Wood [8, 9], and Levcopoulos and Petersson [17, 18].

5 Adaptive Selection Sorting

For selection sorts that repeatedly select the maximum item there are essentially two different ways to take advantage of existing order within the input. The first possibility is devising an adaptive data structure for implementing the priority queue. That is, a data structure that does not always attain its worst-case bound, but for which the time complexities of the operations depend on some ordering in the original list. The first algorithm in which this idea was adopted is Dijkstra's *Smoothsort* [6]. Smoothsort implements a priority queue by an ordered forest of complete heaps, which is computed in linear time, where the roots are in ascending order and the sizes decrease exponentially. Dijkstra claimed that Smoothsort is adaptive without mentioning with respect to what. The adaptivity of Smoothsort was later investigated by Hertel [12], who proved that it was not optimal with respect to the number of inversions. In fact, to date it is not known whether Smoothsort is optimal with respect to any known measure of presortedness. Recently, however, Chen and Carlsson [4] demonstrated that *Inv*-optimality can be obtained by spending linear time rearranging the input before building the forest of heaps.

The second way of achieving adaptivity is motivated by the observation that instead of storing *all* items in the priority queue, only the ones that can possibly be the maximum of the remaining items, the *max-candidates,* need to be stored. To support this scheme we employ some data structure that provides max-candidates and interacts with the priority queue. The following describes a generic adaptive selection sort algorithm based on this approach:

```
procedure Adaptive Selection Sort ($X$: list)
    Construct a data structure $\mathcal{S}(X)$ to support the priority queue
    Insert max-candidates from $\mathcal{S}(X)$ into priority queue $P$
    for $i := 1$ to $n$ do
        Extract the maximum item from $P$
        if new max-candidates are needed then
            Retrieve max-candidates from $\mathcal{S}(X)$
            Insert max-candidates into $P$
        endif
    endfor
    return the list of items extracted from $P$
end
```

Suppose the priority queue is implemented by a data structure that supports the operations in logarithmic time, e.g., a heap. If the input is close to being sorted, the supporting data structure will economise on the number of supplied items. Hence, the priority queue will contain few items during most operations, and the algorithm will complete in $o(n \log n)$ time. On the other hand, in the worst case the priority queue will consist of a linear number of items during most operations, in which case the algorithm runs in $\Theta(n \log n)$ time.

In the following we outline two adaptive selection sorts that are instances of the above generic algorithm.

Multiway Mergesort [14] is a sorting algorithm that adapts to the measure *Runs*. Presented with a list X, it starts by finding the ascending runs in X in linear time. These runs constitute the data structure $\mathcal{S}(X)$. It then builds a heap consisting of the maximum item from each run. After the extraction the max-candidate is the next item from the run to which the extracted item belonged. It is easy to see that the heap will never contain more than one item from each run, and thus, Multiway Mergesort runs in $O(n \log \mathit{Runs}(X))$ time, which, as we have seen above, is *Runs*-optimal. Petersson [27] showed that, if one is careful when implementing the heap operations, Multiway Mergesort becomes optimal with respect to *Block* as well.

Inspired by the *sweep-line technique* in Computational Geometry [21], Levcopoulos and Petersson [15] devised a more sophisticated adaptive selection sort, called *Adaptive Heapsort.* This algorithm uses a *Cartesian tree* [32] to support the heap. The (max-)Cartesian tree for a list X of length n is the binary tree with root $x_i = \max\{x_1, \ldots, x_n\}$. Its left subtree is the Cartesian tree for $(x_1, \ldots, x_{i-1})$ and its right subtree is the Cartesian tree for $(x_{i+1}, \ldots, x_n)$.

After building the Cartesian tree in linear time, Adaptive Heapsort inserts its root into a heap. Then, each extraction of the maximum item is followed by the insertion into the heap of the children of the extracted item in the Cartesian tree; that is, these are the max-candidates. Levcopoulos and Petersson [15] proved that Adaptive Heapsort runs in $O(n \log(\mathit{Osc}(X)/n))$ time, where

$$\mathit{Osc}(X) = \sum_{i=1}^{n} |\{j : 1 \leq j < n \text{ and } \min\{x_j, x_{j+1}\} < x_i < \max\{x_j, x_{j+1}\}\}|.$$

Moreover, they showed that $C_{\mathit{Osc}}(n, k) = \Omega(n \log(n/k))$, and so the algorithm is *Osc*-optimal, and thus optimal with respect to several other measures as well (see Section 7).

Another adaptive selection sort that is optimal with respect to the measure *Max* is described by Igarashi and Wood [13].

Finally, let us briefly mention the status of adaptive selection sorts that do not select the maximum remaining item. Adaptive variants on Quicksort have been proposed [7, 30, 33], but the analysis of these has tended to be empirical rather than analytic, and no optimality results have been shown. Recently, Estivill-Castro and Wood [9] investigated the adaptivity of a Quicksort that uses the median as pivot, and that invokes a sorting check prior to each recursive call. They proved that it runs in $O(n \log \mathit{Exc}(X))$ time; however, this is not *Exc*-optimal since $C_{\mathit{Exc}}(n, k) = \Theta(n + k \log k)$ [3].

6 Adaptive Insertion Sorting

The insertion paradigm is described as follows:

```
procedure Adaptive Insertion Sort ($X$: list)
    Construct an empty dictionary $D$
    for $i := 1$ to $n$ do
        Insert $x_i$ into $D$
    endfor
    return the (sorted) set of items in $D$
end
```

It has also provided many adaptive sorting algorithms. Linear Insertion Sort is adaptive with respect to *Inv* (but not optimally adaptive), and this behaviour has meant that it is often used in tandem with Quicksort for practical in-memory sorting [1, 29].

Guibas *et al.* [10] and Mehlhorn [20] described *Inv*-optimal insertion sorts. Both algorithms are based upon *finger search trees*—dynamic data structures that allow searching and insertion in a set of n items in $O(\log n)$ time; but also allow any item to be 'fingered' for fast access. Searching from a finger in the tree requires $O(\log d)$ time, where d is the number of keys between the finger and the accessed item. Finger search trees also provide insertion adjacent to a fingered node in $O(1)$ amortised time [20]. One possible implementation of finger search trees using level-linked 2-3 trees was described by Brown and Tarjan [2]

Given the operations possible on a finger search tree, a number of adaptive sorting algorithms follow. Mehlhorn's *A-Sort* fingers the largest item in the tree, and then, using this finger, searches for the insertion point of the next item to be inserted. The cost of the i'th search will be $O(\log h_i)$, where h_i is the number of items preceding x_i that are larger than x_i. Since $\mathit{Inv}(X) = \sum_{i=1}^{n} h_i$, the total cost is proportional [20] to

$$\sum_{i=2}^{n} \log h_i = \log \prod_{i=2}^{n} h_i = n \log \left(\prod_{i=2}^{n} h_i \right)^{1/n} \leq n \log \frac{\sum_{i=2}^{h} h_i}{n} = O(n \log \frac{\mathit{Inv}(X)}{n})$$

with the inequality following because the geometric mean is never greater than the arithmetic mean. To show that A-Sort is *Inv*-optimal we must give a lower bound on the size of $\mathit{below}_{\mathit{Inv}}(n,k)$. Let $h = k/n$, and partition the numbers $1, 2, \ldots, n$ into subsequences of length h. Let X_i be any permutation of the i'th of these subsequences. For example, X_1 is a permutation of $1, 2, \ldots, h$; and X_2 is a permutation of $h+1, h+2, \ldots, 2h$. Finally, take $X = X_1 X_2 \cdots X_{n/h}$. By construction, each item in X has fewer than h larger items preceding it, and so $\mathit{Inv}(X) \leq nh = k$. Moreover, there were $h!$ possible permutations in each of n/h subsequences, and so

$$\log |\mathit{below}_{\mathit{Inv}}(n,k)| \geq \log(h!)^{n/h} = \Omega(n \log \frac{k}{n}).$$

Mannila [19] improved this algorithm by making the simple observation that nothing is lost if the finger is moved to the most recently inserted item rather than left on the largest item. This gives rise to *Local Insertion Sort*, which has running time proportional to $\sum_{i=2}^{n} \log d_i$, where d_i is the number of (previous) items between successive insertions:

$$d_i = |\{x_j : j < i \text{ and } \min\{x_{i-1}, x_i\} < x_j < \max\{x_{i-1}, x_i\}\}| + 1.$$

Mannila not only showed that $\sum d_i \leq 2\mathit{Inv}(X)$, and thus that Local Insertion Sort is *Inv*-optimal; but also that it is both *Runs* and *Rem* optimal. Petersson [26] has shown Local Insertion Sort to be *Block*-optimal.

Local Insertion Sort is fast when the list being sorted exhibits *spatial locality*—when most of the insertions are not too far (in space) from the most recently inserted item. In [23] we introduced the orthogonal concept of *temporal locality*, in which a list is judged to be nearly sorted if most of the insertions are adjacent to an item that was itself inserted 'not too long ago' in *time*. For example, the list X_9:

$$X_9 = (1, n/2+1, 2, n/2+2, 3, \ldots, n/2-1, n-1, n/2, n)$$

has no spatial locality, but does exhibit temporal locality, since each item after the second is inserted adjacent to an item that was itself inserted just two operations previously.

This definition leads naturally to the notion of *historical searching*, where we require fast access to items that have recently been accessed. For example, a move-to-front list provides reaccess to recently accessed items in $O(t)$ time, where t is the number of distinct items that have been accessed since the most recent access to this item. In [23] we introduced a data structure we call a *Historical Search Tree* that provides $O(\log t)$ reaccess to recently accessed items, and allows insertion of new items in $O(\log t)$ comparisons, where t in this case is the number of items that have been accessed since the most recent access to either of the neighbours of the new item. By repeated insertion into a Historical Search Tree we arrived at a *Historical Insertion Sort* that is sensitive to temporal locality. We also introduced two measures *Loc* and *Hist* to capture the notions of spatial and temporal presortedness, and to precisely model the running times of Local Insertion Sort and Historical Insertion Sort respectively.

It is also possible to combine these two assessments, and to accept as nearly sorted any list in which most of the insertions are not too far away (in space) from an item that was itself inserted not too long ago (in time). This is a very natural definition. For example, if we lose our car keys, we search not only in the immediate vicinity of where we first notice that we no longer have them, but also in the vicinity of where we have been recently, both tracing our steps further and further backward in time and also searching in a wider and wider radius from each intervening point.

Our *Regional Insertion Sort* [23] does exactly that. Perhaps in part because this is a particularly 'intuitive' searching strategy the associated measure of presortedness—*Reg*—can be shown to be superior to all previous measures [28], and so a *Reg*-optimal algorithm will be optimal with respect to all of the measures of presortedness described here.

Both Historical Insertion Sort and Regional Insertion Sort require efficient historical searching. The Historical Search Tree [23] supports fast reaccess to items recently inserted, but to date we have been unable to implement all of the desired operations in the necessary time bounds, although we can meet the requirements in terms of comparisons. The development of such a data structure remains an important open problem, and to date *Hist*-optimal and *Reg*-optimal (in terms of running time) algorithms have not been completely described.

7 A Partial Order on Measures of Presortedness

The relation $\supseteq$ allows the construction of a *partial order* on the set of measures of presortedness. The relationships in the partial order are best illustrated with the *Hasse diagram* of Figure 1, where an upward edge from M_2 to M_1 indicates that $M_1 \supset M_2$. For example, there is an upward edge to show that $SUS \supset Runs$. Details of the relationships between measures, and, where there is no relationship and the measures are independent, of the lists that are counter-examples, may be found in [26, 28].

Since $\supset$ is transitive, each edge in the diagram is a containment relation on optimality, with all optimal algorithms for the higher measure automatically inheriting optimality for any connected lower measures. For example, the edge from *Rem* to *Block* reflects the fact that every *Block*-optimal algorithm must be *Rem*-optimal. Conversely, the presence of upward paths from *Block* to *Loc*, *Hist*, and *Reg* means that an algorithm that is not *Block*-optimal cannot be optimal with respect to any of the higher measures.

Each adaptive sorting algorithm corresponds to a *descriptor* line across the diagram. For example, the descriptor for Local Insertion Sort crosses (immediately) below *Hist*, *Reg*, and *SUS*—it is not optimal with respect to any of these three, but *is* optimal with respect to everything below [19, 23, 28].

To establish lower bounds on the location of the descriptor corresponding to some algorithm we need proofs of optimality, such as the example proof given here for Natural Mergesort. To establish upper bounds on the location of the descriptor for an algorithm we need to describe lists that are nearly sorted according to the measure, but for which the algorithm is not optimal.

The partial order on measures of presortedness can thus be used as a yardstick to measure the importance (or otherwise) of new adaptive algorithms; and as well serves as a framework within which new measures of presortedness can be evaluated. The goal of the algorithm designer must be to show that any new algorithm is optimal with respect to the highest possible combination of measures; and an algorithm will only be of interest if its descriptor lies above all other algorithm descriptors at at least one point in the partial order.

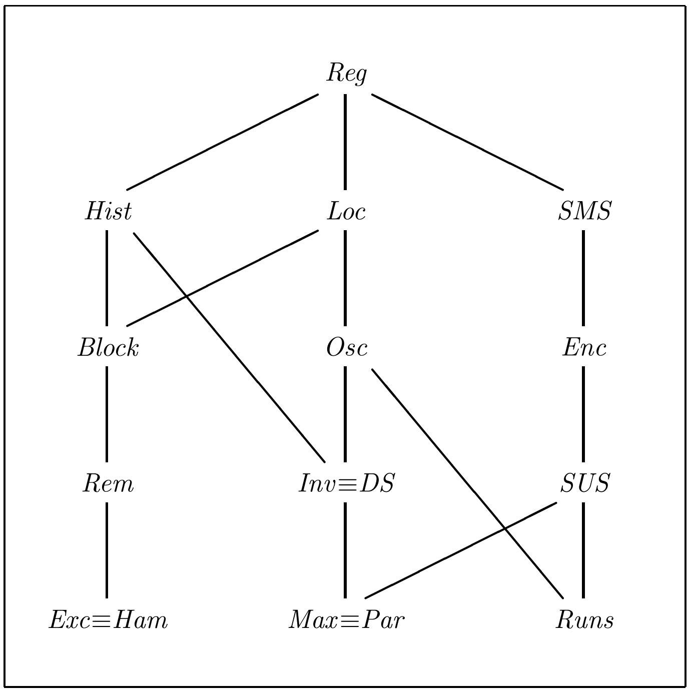

Figure 1: Partial order on measures of presortedness.

8 Summary

We have shown that each of the three main sorting paradigms leads naturally to corresponding adaptive algorithms. We have also surveyed a wide range of measures of presortedness, and shown how measures of presortedness can, in some cases, be compared. This relationship leads to a partial ordering on measures of presortedness, and provides a framework not only for the evaluation of new measures of presortedness, but also for the evaluation of all adaptive sorting algorithms.

Acknowledgements

We thank the referees for their helpful comments and suggestions. This work was supported by the Australian Research Council.

References

[1] J.L. Bentley. Programming pearls: How to sort. *Communications of the ACM*, 27:287–291, 1984.

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

[3] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and Natural Mergesort. *Algorithmica.* To appear. Prel. version in *Proc. Internat. Symp. on Algorithms,* pages 251–260, LNCS 450, Springer-Verlag, 1990.

[4] J. Chen and S. Carlsson. On partitions and presortedness of sequences. *Acta Informatica.* To appear. Prel. version in *Proc. 2'nd ACM-SIAM Symposium on Discrete Algorithms*, pages 62–71, 1991.

[5] C.R. Cook and D.J. Kim. Best sorting algorithms for nearly sorted lists. *Communications of the ACM*, 23(11):620–624, 1980.

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

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

[8] V. Estivill-Castro and D. Wood. A new measure of presortedness. *Information and Computation*, 83(1):111–119, 1989.

[9] V. Estivill-Castro and D. Wood. Practical adaptive sorting. In *Proc. International Conference on Computing and Information*, pages 47–54. LNCS 497, Springer-Verlag, 1991.

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

[11] J.D. Harris. Sorting unsorted and partially sorted lists using the Natural Merge Sort. *Software—Practice and Experience*, 11(12):1339–1340, 1981.

[12] S. Hertel. Smoothsort's behaviour on presorted sequences. *Information Processing Letters*, 16(4):165–170, 1983.

[13] Y. Igarashi and D. Wood. Roughly sorting: a generalization of sorting. *Journal of Information Processing*, 14(1):36–42, 1991.

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

[15] C. Levcopoulos and O. Petersson. Adaptive Heapsort. *Journal of Algorithms.* To appear. Prel. version in *Proc. 1989 Workshop on Algorithms and Data Structures,* pages 499–509, LNCS 382, Springer-Verlag, 1989.

[16] C. Levcopoulos and O. Petersson. Sorting shuffled monotone sequences. *Information and Computation.* To appear. Prel. version in *Proc. 2nd Scandinavian Workshop on Algorithm Theory*, pages 181–191, LNCS 447, Springer-Verlag, 1990.

[17] C. Levcopoulos and O. Petersson. An optimal adaptive in-place sorting algorithm. In *Proc. 8th International Conference on Fundamentals of Computation Theory*, pages 329–338. LNCS 529, Springer-Verlag, 1991.

[18] C. Levcopoulos and O. Petersson. Splitsort—an adaptive sorting algorithm. *Information Processing Letters*, 39(4):205–211, 1991.

[19] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34(4):318–325, 1985.

[20] K. Mehlhorn. *Data Structures and Algorithms, Vol. 1: Sorting and Searching.* Springer-Verlag, Berlin, Germany, 1984.

[21] K. Mehlhorn. *Data Structures and Algorithms, Vol. 3: Multidimensional Searching and Computational Geometry.* Springer-Verlag, Berlin/Heidelberg, Germany, 1984.

[22] A.M. Moffat. Adaptive merging and a naturally Natural Mergesort. In *Proc. 14'th Australian Computer Science Conference*, pages 08.1–08.8. University of New South Wales, February 1991.

[23] A.M. Moffat and O. Petersson. Historical searching and sorting. In *Proc. ISA'91 International Symposium on Algorithms*, pages 263–272. LNCS Volume 557, Springer-Verlag, December 1991.

[24] A.M. Moffat, O. Petersson, and N.C. Wormald. Further analysis of an adaptive sorting algorithm. In *Proc. 15'th Australian Computer Science Conference*, pages 603–613. University of Tasmania, January 1992.

[25] A.M. Moffat, O. Petersson, and N.C. Wormald. Sorting and/by merging finger trees. Manuscript, 1992.

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

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

[28] O. Petersson and A.M. Moffat. A framework for adaptive sorting. In *Proc. 3'rd Scandinavian Workshop on Algorithm Theory*, July 1992. To appear. Also available as Technical Report LU-CS-TR:91–81, Department of Computer Science, Lund University, Sweden, October 1991.

[29] R. Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21:847–857, 1978.

[30] R. Sedgewick. *Quicksort.* Garland Publishing, Inc., New York & London, 1980.

[31] S.S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28(4):775–784, 1988.

[32] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–237, 1980.

[33] R.L. Wainwright. A class of sorting algorithms based on Quicksort. *Communications of the ACM*, 28(4):396–402, 1985.