Chapter 51
Optimistic Sorting and Information Theoretic Complexity

Peter McIlroy*

Abstract

Entropy considerations provide a natural estimate of the number of comparisons to sort incompletely shuffled data, which subsumes most previous measures of configurational complexity. Simple modifications to insertion sort and merge sort improve their performance on such data. The modified merge sort proves efficient both in theory and in practice.

Introduction

The problem of sorting incompletely shuffled data in random access memory is intuitively quite simple. It is often easy to tell at a glance whether some configuration is easy to sort, that is, whether fewer than $\lg N!$ comparisons are necessary. For instance, in updating data bases, a common technique is to sort recent additions independently, then merge them with the previously sorted data. This requires $O(M \lg M + N)$ comparisons, where M is the number of recent additions. It is also common to use insertion sort for checking data in which a few elements are slightly misplaced. The question then arises: what are useful classes of configurations for which fewer than $O(N \lg N)$ comparisons are necessary? Various competing measures of disorder have been proposed, often together with special sorting algorithms tuned to those particular measures.[1,2,3] Until recently, measures have often described only worst case performance, or have given overly imprecise estimates of complexity. Recent work by Petersson and Moffat[4] has to a great extent unified these measures. This paper discusses an alternative approach using information theoretic and combinatorial techniques.

From an information theoretic viewpoint, the number of comparisons needed to sort a permutation is an upper bound on the Kolmogorov complexity of the permutation. A sorting algorithm generates a string of comparisons unique to each permutation. These strings form a prefix-free encoding for permutations. Since there are $N!$ permutations, most permutations will require at least $\lg N!$ comparisons. The challenge is to determine classes of permutations which require fewer, and to construct sorting algorithms whose comparison counts approach the information theoretic limit within these classes.

Desirable Properties

We list some properties that a sorting algorithm should have for its comparison count to resemble Kolmogorov complexity (K-complexity). Each desideratum reflects the length of some simple information theoretic encoding of permutations, and has a common analog in statistical mechanics. (The original motivation for these desiderata was the Kolmogorov-Sinai entropy[5] of statistical physics.) Most previous measures of disorder are subsumed by these properties.

Define $C(\pi)$ to be the number of comparisons needed to bring permutation π: $i \to \pi_i$ into order.

1. **Reversal.**

Let $-\pi$: $i \to \pi_{N+1-i}$. Define a sorting algorithm to be optimal over reversal when for any permutation π,

$$C(-\pi) = O(C(\pi)).$$

(Here $= O(C(\pi))$ represents $\leq aC(\pi) + bN$ for some constants a and b, and for all N.) The K-complexity of $-\pi$ is within a small additive constant of π, as $-\pi$ can be encoded as "π = [encoding]; print $-\pi$".

2. **Inversion.**

Define a sorting algorithm to be optimal over inversion when for any permutation π,

*Computer Science Research Group, University of California at Berkeley, Berkeley, CA 94720.

$$C(\pi^{-1}) = O(C(\pi)).$$

3. **Composition.**

Let $\rho\cdot\sigma$ be a decomposition of π. Define a sorting algorithm to be optimal over composition if

$$C(\pi) = O(C(\rho)) + O(C(\sigma))$$

for all decompositions of π. Later in the paper it is shown that full optimality over composition is not feasible.

These properties define relations between the comparison counts for different permutations without actually limiting the count for any one permutation. We now restrict the comparison counts for specific permutations.

4a. **Weak segmentation.**

Assume that π can be cut into S segments such that $\pi_j \le \pi_k$ for all j in segment i and all k in segment $i+1$. Let C_i be the number of comparisons to sort segment i alone. A sorting algorithm is optimal over weak segmentation if

$$C(\pi) = O(\sum_{i=1}^{S}(C_i + \lg N_i)) = O(N + \sum_{i=1}^{S} C_i)$$

Only $\sum \lg N_i = O(N)$ comparisons are necessary to distinguish the lengths of the S segments, as it is possible to express N_i in $2\lg N_i$ bits.

4b. **Strong segmentation.**

Assume that the sequence of segments from 4a is scrambled without changing their internal order. An algorithm is optimal over strong segmentation if

$$C(\pi) = O(\sum_{i=1}^{S} C_i + S\lg N)$$

Overhand shuffling, or block optimality,[4,6] is a special case of strong segmentation, where

$$\sum_{i=1}^{S} C_i + S\lg N = \sum_{i=1}^{S}(N_i - 1) + S\lg N = N - S + S\lg N,$$

since perfectly ordered sequences require only $N-1$ comparisons. (Overhand shuffling is the familiar technique of dropping short runs of cards from one hand into the remainder of the deck held in the other.)

5a. **Repeated keys.**

Assume there are $S \le N$ distinct keys. A sorting algorithm is optimal over repeated keys if

$$C(\pi) = O(N + \lg\frac{N!}{\prod N_i!}) = O(N + \sum_{i=1}^{S} N_i \lg\frac{N}{N_i})$$

where N_i is the number of elements with key i, and $\lg(N!/\prod N_i!)$ is simply the entropy of a configuration with restricted keys. When $S=N$, this reduces to the usual $C(\pi) = O(\lg N!)$. Although a configuration with repeated keys is not strictly a permutation, it can be made so by requiring stability. This leads to the next case.

5b. **Riffle shuffles** (riffles).

Riffles are the other common form of card shuffling, where the deck is cut, and the two sections are interleaved randomly, without changing order within a section.

Definition of an S-way riffle. A permutation π is an S-way riffle if π can be constructed by dividing the identity into S segments with lengths N_i, and interleaving the segments so that each remains monotonic. This is equivalent to requiring that π^{-1} consist of S runs, within which π_{j-1}^{-1}, π_j^{-1}, and π_{j+1}^{-1} are monotonic.

A sorting algorithm is optimal over riffles if

$$C(\pi) = O(N + \sum_{i=1}^{S} N_i \lg\frac{N}{N_i}).$$

This is a simple generalization of optimality for repeated keys, which can be considered as a riffle whose segments consist of runs of equal keys.

5c. **Runs.**

Assume that π consists of S monotonic runs, with N_i the length of run i. Runs have the same complexity as riffle shuffles, and the same definition for optimality. Optimality over runs is a corollary of properties 5b and 2, as the inverse a permutation consisting of runs is a riffle. Note that this is not the usual measure of optimality over runs,[1] $N\lg S$, which best applies only to runs of equal length.

These properties subsume most previous measures of disorder, including *Hist*, *Loc*, and *SMS*.[4] The possible exception is *Reg*.*

$$Loc \equiv \sum_{i>1} \max(\lg|\pi_i - \pi_{i-1}|, 1)$$

$$Hist \equiv \sum_{i>1} \max(\lg|\pi_i^{-1} - \pi_{i-1}^{-1}|, 1)$$

$$Reg \equiv \sum_{i>1} \max(log \min_{t<i}\{(i-t)\times|\pi_i - \pi_t|\}, 1)$$

$SMS \equiv$ number of shuffled monotone subsequences.

Hist, *Loc*, and *SMS* together subsume all other measures of complexity.[4]

* In fact, *Hist*, *Loc*, and *Reg* are usually defined in terms of difference at the time of insertion by insertion sort, but this makes only $O(N)$ difference and complicates analysis. Also, the usual definition of *Reg* is $\sum_{min}\{\lg(t+|\pi_i - \pi_{i-t}|)\}$; replacing addition with multiplication supplies the necessary bits for expressing both t and $|\pi_i - \pi_{i-t}|$ and better reflects information theoretic complexity.

The five properties listed here form tighter bounds on complexity than other measures, which often are accurate only within a factor of two (*Hist*, *Loc*, and *Reg*), or only describe worst case behavior (*SMS*). For our purposes, it will also be necessary to distinguish strong segmentation from the other desiderata, a distinction not made by difference-based measures like *Hist*, *Loc*, and *Reg*.

Data Compression of Comparison Strings

Consider the comparison trace of any (stable) sorting algorithm. The ith bit represents the sense of the ith comparison, with 0 representing $\leq$ and 1 representing $>$. If the trace has fewer than $\lg N!$ bits, then the sorting algorithm is also a data compression algorithm. (It may be possible to compress the trace further after sorting is complete.) In the cases of insertion sort and merge sort, the two most common stable sorts, it is sometimes possible to compress data on the fly, reducing the number of comparisons. This is achieved by using different searching strategies for making insertions into an ordered list.

Consider the complexity of inserting an element x into an ordered list of $i-1$ elements, $[x_1, \ldots, x_{i-1}]$. Insertion sort uses l_i comparisons, where l_i is i minus the rank of x in $[x_1, \ldots, x_{i-1}]$. The total comparison count is at most $Inv(\pi)+N-1$, where $0 \leq Inv(\pi) \leq N^2/2$, is the number of nearest neighbor swaps needed to bring π into order. This can be improved in various ways. First, a binary search uses at most $\lg i$ comparisons. This is best conceived as insertion into a balanced tree. This uses

$$C(\pi) \leq \sum_{i=2}^{N} \lg i < N \lg N$$

comparisons. However, the excellent behavior of insertion sort for small Inv has been lost.

It can be regained by using an exponential search[7] beginning at x_{i-1}. An exponential search compares x first with x_{i-1}. Then x is compared with x_{i-2}, x_{i-4}, x_{i-8}, increasing by powers of two until $x > x_{i-j}$. From here, a binary search is made on the $j/2$ positions between x_{i-j} and $x_{i-j/2}$. This may be considered as a linear search up the rightmost branch of a balanced tree, followed by a binary search down the tree. Figure 1a shows an example of this.

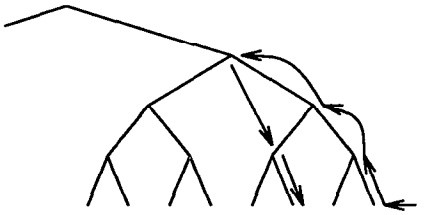

Figure 1a. Insertion sort with exponential search. Arrows demonstrate insertion between elements $i-4$ and $i-5$. $6 = 2\lg 5$ comparisons are necessary.

Insertion sort with exponential search uses

$$C(\pi) = \sum_{i=2}^{N} \max(\lg l_i, 1) < 2N \lg N$$

comparisons. Compared to insertion sort, the worst case occurs when every insertion is between elements x_{i-2} and x_{i-3}. In this case, at most $(4/3)Inv(\pi)$ comparisons are used. An analogous algorithm using a more complicated data structure can be found in Mehlhorn (8).

The algorithm can be extended further by replacing the linear search on the rightmost branch with an exponential search. An example is shown in Figure 1b.

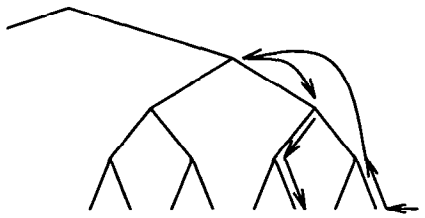

Figure 1b. Insertion sort with double exponential search. $\lg 5 + 2\lg\lg 5 = 6$ comparisons are again necessary.

Insertion sort with double exponential search uses

$$C(\pi) = \sum_{i=2}^{N} (\lg l_i + \max(2\lg\lg l_i, 1)) < N \lg N + 2N \lg\lg N$$

comparisons, and at most $(5/3)Inv(\pi)$ comparisons. Further levels of extended search are described in Bentley and Yao,[7] and lead to encoding l_i bits in $U(l_i)$ bits, where $U(l_i)$ is the universal code, requiring the sum of positive terms in

$$(\lg l_i + 1) + (\lg\lg l_i + 1) + (\lg\lg\lg l_i + 1) + \cdots$$

bits. All of these strategies reflect methods from elementary information theory for encoding a number l_i in $O(\lg l_i)$ bits.[9]

Theorem: Insertion sort with exponential search (ISES) is optimal over inversion and weak segmentation.

Proof: Note that $l_i \le |\pi_i - i| + 1$ for all i, and that insertion sort with exponential search uses at most

$$C(\pi) \le \sum_{i=2}^{N} \max(2\lg(|\pi_i - i| + 1), 1)$$

comparisons. Since $|\pi_i - i| = |\pi_{\pi_i}^{-1} - \pi_i|$, $C(\pi)$ is approximately $C(\pi^{-1})$. For weak segmentation, l_i remains the same for points on the interior of the segments. For the first endpoint of a segment, l_i is the length of the previous segment.

By searching the leftmost branch whenever the previous insertion occurred to the left of the root, we can still get at most $(5/3)Inv(\pi)$ and $N\lg N + 2N\lg\lg N$ comparisons. This maintains the properties of ISES, with the addition of optimality over reversal.

For the final modification, insert element i by comparing it with the $i-1$st element of the original permutation, and search from there.[1,6] If an exponential search is used, at most

$$C(\pi) \le 2\sum_{i=2}^{N} \max(\lg|d_i|, 1) \le 2Loc(\pi)$$

comparisons are necessary, where d_i is the distance between the $i-1$st and ith insertions; now the quantity $|l_i - l_{i-1}| \approx |\pi_i - \pi_{i-1}|$ is compressed, rather than l_i alone.[4] We call this strategy insertion sort with differential search (ISDS).

Theorem: ISDS is optimal over reversal, weak and strong segmentation, and runs.
Proof: For the elements of run j, $\sum\max(2\lg|\pi_i - \pi_{i-1}|, 1)$ is at most $2N_j\lg(N/N_j)$, which occurs when elements in the run are uniformly spaced. Proofs for segmentation are as for ISES above.

Using $2\lg|\pi_i - \pi_{i-1}|$ comparisons to make insertion i is the same estimate of complexity achieved by encoding nearest neighbor differences in $2\lg|\pi_i - \pi_{i-1}|$ bits.

ISDS lacks optimality over inversion and composition.

Observation: No sorting algorithm can achieve optimality over composition while maintaining optimality over both runs and strong segmentation without analysis of the comparison trace. Let ρ be a permutation consisting of S equal runs, and σ be an overhand shuffle. Optimality over composition would require $C(\rho\cdot\sigma) = O(N\lg S + C(\sigma))$. The composition will consist of S pieces corresponding to the S runs of ρ. Each piece is a miniature of the overhand shuffle, with segments shorter by a factor of S. $O(C(\sigma) - \lg S) = O(C(\sigma))$ comparisons will be necessary to sort any one of these pieces. If each piece is sorted independently, a total of $O(SC(\sigma) + N\lg S)$ comparisons will be made. To achieve full optimality over composition, the configurations of some of the pieces must be sorted by extrapolation from the others. This requires analysis of the comparison trace.

However, insertion sort is optimal over composition for a limited class of permutations. Let $\prod\rho_k$ be a decomposition of ρ into permutations that require optimality only over runs, inversion, and weak segmentation for efficient sorting. Define an algorithm to be optimal over weak composition if

$$C(\rho) = O(\sum_k C(\rho_k))$$

Theorem: Insertion sort with differential search is optimal over weak composition.
Proof (sketch): composition of m runs with n runs leads to $O(mn)$ runs, and so requires $O(N(\lg m + \lg n))$ comparisons. Weak segmentation never changes relative positions by more than a fixed amount, and $\lg(|\pi_i - \pi_{i-1}| + C) \le \lg|\pi_i - \pi_{i-1}| + \lg C$. Since ISDS is not optimal over inversion, there are no further cases.

Further recursive iteration of exponential search[7] leads to an insertion sort with differential search requiring at most

$$C(\pi) \le \sum_{i=2} U(|\pi_i - \pi_{i-1}|)$$

comparisons, which corresponds to encoding nearest neighbor differences using their universal encodings. This too is optimal over runs, strong segmentation, and weak composition.

Merge Sort

To get optimality over inversion we must also require

$$C(\pi) = O(N + \sum_{i=1}^{N} \lg|\pi_i^{-1} - \pi_{i-1}^{-1}|) = O(N + Hist(\pi))$$

comparisons. If this is achieved while maintaining optimality over weak composition, all desiderata except full optimality over composition will be satisfied. With a simple modification, merge sort comes close.

Let lists $A = [A_1, A_2, \cdots A_m]$ and $B = [B_1, ..., B_n]$ be two adjacent lists to be merged (called hereafter "merge lists"), and assume $A_1 > B_1$. In this case element A_1 must be inserted into list B. The usual technique here is a linear

search of length l_i for insertion i. Again "compress" l_i bits into $\max(2\lg l_i, 1)$ bits, using the same method of an exponential search followed by a binary search. We name this algorithm merge sort with exponential search (MSES). The worst case again occurs for insertions after element two, where 4 comparisons are needed, rather than 3 as in traditional merge sort. This yields a worst case of $(4/3)N \lg N$ comparisons. For "random" data, where insertions occur after position j with frequency 2^{-j}, the expected number of comparisons is $\approx 1.08N(\lg N - 1)$. A variant of this algorithm was first described in (6), where natural merge sort was used to minimize recopying.

Theorem: Merge sort with exponential search is optimal over reversal, inversion, weak composition, runs, riffles, and weak segmentation. It is suboptimal for strong segmentation, where it requires an additional $O(S \lg N \lg(N/S)) = O(S \lg N \lg\lg N)$ comparisons. (A proof is sketched in the appendix.)

Optimality over strong segmentation would require an analysis of the comparison trace, as strong segmentation, or overhand shuffling, leads to correlations between different merge passes. However, within the class where MSES is optimal, further improvement is possible.* Using natural merge improves the behavior over runs by up to $N-1$ comparisons. However, this comes at the expense of about $.5N$ comparisons for random data. More consistent improvement comes from making a hybrid of exponential search and linear search (MSLS) merge sorts. This can be done with no memory of the comparison trace, simply switching to linear search when a short run occurs between insertions, and to exponential search when a long run occurs. An effective heuristic is to use linear search until a run of length 7 occurs, and to return to linear search when a run of length 1 occurs. The worst case versus MSES occurs for alternating segments of lengths near 30 and length 1, where the hybrid uses 3/2 times as many comparisons. (This can occur for carefully constructed overhand shuffles.) The worst case behavior versus MSLS remains $(4/3)N \lg N$. However, the average number of comparisons on random data is less than 1.003 times greater than MSLS. The behavior on runs and riffles is also close to the information theoretic limit, at

$$1.003 \lg \frac{N!}{\prod N_i!} + O(N).$$

* Similar improvement of insertion sort is also possible.

Analogous hybridization of natural and pairwise first passes makes a similar improvement in the $O(N)$ term for runs.

Figures 2a-c show the performance of MSES and the MSES/MSLS hybrid for some of the cases discussed previously. Figures 2a and 2c were generated by composition of 2-way and 3-way riffles and runs. The MSES/MSLS hybrid requires up to .6N extra comparisons for compositions of unbalanced riffles. Figure 2b was generated by extracting a run of length $N-LMS$, scrambling it, and riffling it back.

$$H = (N-LMS) + \lg((LMS)!) + 2((N-LMS)\lg(N/(N-LMS)) + LMS \lg(N/LMS))$$

This demonstrates the behavior for shuffled monotonic sequences with unbalanced lengths. In all cases, $N = 2^{18}$.

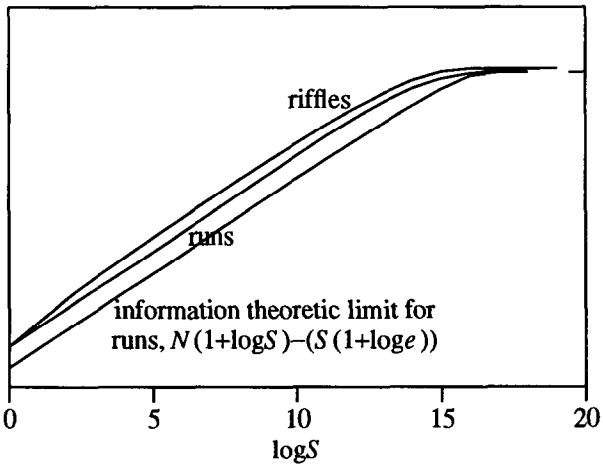

Figure 2a. Performance in comparisons of hybrid MSES/MSLS for runs and riffles. The tick-mark at right shows the information theoretic limit for random data, $\log(N!)$.

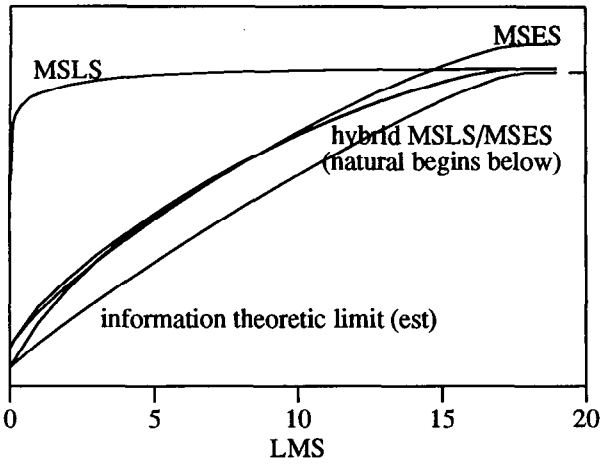

Figure 2b. Performance on Longest Monotonic Subsequence (LMS). Comparison counts for MSES, MSLS, the hybrid, and the hybrid with natural merging are shown. The steeper of the two hybrid curves is modified natural merge. The information theoretic limit is shown for comparison.

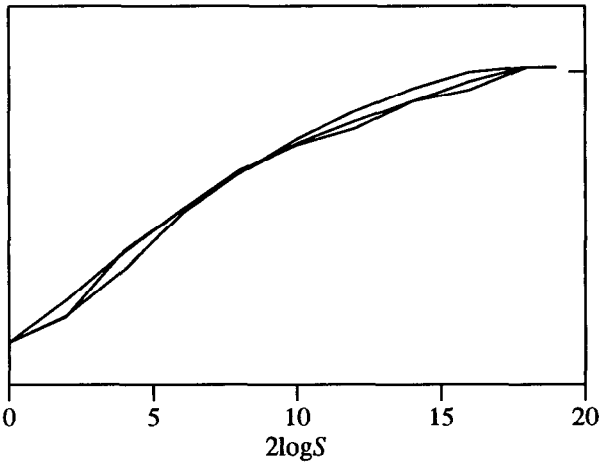

Figure 2c. Performance of hybrid on shuffled monotone sequences. The three curves result from different orders of composition of log$S$ runs with log$S$ riffles.

Performance

We have shown that merge sort with exponential search, and its hybrid with MSLS, have comparison counts near the information theoretic limit for a large class of permutations. We have shown that further improvement, to find other classes of low-complexity permutations, would be impractical, as analysis of the comparison trace would be required. (This is a time-intensive proposition!) Whether MSES is in fact practical, that is whether it actually reduces the time spent sorting, can only be found by experiment.

In tests on the Sun SPARC station, on random data, hybrid MSES/MSLS is at most 1% slower (in CPU time, not just comparisons) than MSLS alone. We'll not get in to what might comprise "real" data here. However, for sorting lines of C code, the text of this paper, and PostScript, the MSES/MSLS hybrid used between $\lg(N!)-2.5N$ and $\lg(N!)-.5N$ comparisons, and between 10% and 30% less time than MSLS alone. On this paper sorted as words (as for a spelling checker), the hybrid used about $\lg(N!)-.5N$, or about $.95\lg(N!)$ comparisons. Hybrid MSES/MSLS is competitive with Quicksort, even on random data. Versus BSD UNIX* system qsort, the hybrid is faster for all data. Versus the fastest readily available qsort,[10] the hybrid is 25% slower on random integers; for random strings, time differences are negligible.

Discussion

We have defined several intuitively reasonable properties corresponding to various classes of presortedness. For these classes, we have made accurate estimates of information theoretic complexity. For a large and important subset of this class, merge sort with hybrid exponential and linear search is near-optimal. Further qualitative improvement of merge sort, to include optimality over a larger class, would come at a penalty in both time and space, and likely would prove impractical.

Recent studies by Moffat et al[11] has shown that splaysort, and other tree-based strategies, are optimal over strong segmentation, runs/riffles and inversion. In particular, their experiments indicate that splaysort is *Reg*-optimal. However, splaysort requires $\approx 1.45N\lg N$ comparisons for random data, significantly more than the MSES/MSLS hybrid. Merge sort with competitive search strategies is likely to maintain two notable advantages. Its multiplicative constant near unity ($\tilde{}1.003N\lg N$) gives expected behavior only slightly worse than MSLS alone on random data, and a much larger set of permutations where it actually uses fewer than $\lg N!$ comparisons. In comparison with MSLS, hybrid MSES/MSLS has minimal computational overhead, requiring no additional memory, about 2 additional operations per comparison, and about 1% more CPU time on random data. Merge sort with hybrid exponential and linear search is thus the strategy of choice for even slightly presorted data.

* UNIX is a trademark of AT&T.

Acknowledgements

Thanks to Keith Bostic and CSRG for their support. Also thanks to J. Bentley and R. Karp for encouragement and criticism, to M. D. McIlroy for editorial comment, and to J. P. Linderman for help in testing.

Appendix

• Merge sort with exponential search (MSES) has nearly optimal efficiency when merging two lists.

Let l_i be the length of the ith insertion-free run. MSES requires

$$\sum_i \max(2\lg l_i, 1)$$

comparisons. Assume the positions of the $i-2$ and $i+2$ insertions into the second list are known. If both positions are the same, no comparisons will be needed for "insertion" i. If they are different, $\lg(l_{i-2}+l_i) > \lg l_i$ comparisons are needed. Thus the complexity of merging two sequences is at least

$$H \geq \sum_i \max(\lg l_i, 1).$$

Further modifications that choose between competitive search strategies (such as hybrid MSES/MSLS) will bring MSES nearer to the information theoretic limit.

When merging a list of length N_1 with one of length $N_2 \geq N_1$, this translates to a worst case behavior of

$$C \leq 2N_1 \lg \frac{N_2+1}{N_1} \leq 2(N_1 \lg \frac{N_1+N_2}{N_1} + N_2 \lg \frac{N_1+N_2}{N_2}).$$

MSES ignores correlations between merge passes, which would be required for optimality over strong segmentation, and long-range correlations which would be required for full optimality over composition. For the other properties, MSES is nearly optimal, that is, they correspond to variations in the lengths of insertion-free runs (density of insertions).

• MSES is optimal over riffle shuffles.

Define N_i the length of the ith shuffled subsequence; its elements will be consecutive after sorting. After $\lg(N/N_i)$ passes, there will be at most N_i runs to be joined end-to-end. In the worst case, each run will have one element of the original subsequence. The first $\lg N_i$ passes require at most $(4/3)N_i \lg(N/N_i)$ comparisons for sequence i. Each additional pass will double the length of consecutive runs into which no additional insertions will be made. This requires $O(N_i \sum_{j=1}^{\lg N} j 2^{-j}) = O(N_i)$ comparisons. For future reference, we call a monotonic sequence η in contiguity when the number of merge lists containing elements of η begins to decrease exponentially.

• MSES is optimal over runs.

When adjacent lists of length 2^j are merged, either they are both subsequences of a longer run in the original configuration, and $O(j)$ comparisons are made, or they include adjacent runs from the initial configuration, and $O(2^j)$ comparisons are necessary. The first case adds to $O(N_i)$ comparisons for run i over the first $\lg N_i$ passes. the second contributes $O(N_i)$ comparisons for each additional pass, for a total of $O(N_i(\lg N - \lg N_i))$. The argument can be made more rigorous by showing that the algorithm behaves gracefully when merging two lists that include subsequences of adjacent runs. It does, because adjacent runs need be merged only once, even when they are merged incrementally. The details are omitted; there are about 5 special cases.

• $O(\sum_{i=1}^{S} N_i \lg(N/N_i))$ comparisons are needed to merge S monotonic sequences with lengths N_i, independently of the order of merging. First reconsider disjoint sequences, or runs.

Assume that on the final merge pass sequences $\{1, ..., k\}$ are merged with sequences $\{k+1, ..., S\}$. Assume further that the first passes required at most

$$2\sum_{i=1}^{k} N_i \lg(\sum_{j \leq k} N_j / N_i) + 2\sum_{i=k+1}^{S} N_i \lg(\sum_{j>k} N_j / N_i)$$

comparisons. The final pass merges $\sum_{i=1}^{k} N_i$ elements with $\sum_{i=k+1}^{S} N_i$ elements, requiring at most

$$2(\sum_{i=1}^{k} N_i)\lg(N / \sum_{i=1}^{k} N_i) + 2(\sum_{i=k+1}^{S} N_i)\lg(N / \sum_{i=k+1}^{S} N_i)$$

comparisons for a total of at most $2\sum_{i=1}^{S} N_i \lg(N/N_i)$.

The same proof holds for shuffled sequences, once they have been brought into contiguity. Shuffled monotonic sequences must first be brought into contiguity, requiring the same number of comparisons as riffle shuffles. Then they must be merged. Note that two sequences can be merged with each other only once, even if different fractions are merged on different merge passes. Thus the argument for runs merged in arbitrary order still holds. As expected, shuffled monotone sequences are twice as complex as the same number of riffles or runs.

This proof can be extended to composition an arbitrary numbers of runs with a different number of riffles, and to allow optimality over weak segmentation. In all cases, sequences can be considered to have two densities.

Define $\pi(i) \equiv \pi_i$ to avoid double subscripts.

Let X represent a monotonic sequence with elements $X(1) \cdots X(|X|)$.

Define density in position for sequence X:

$$\rho_p = 1/ < | \pi^{-1}(X(i)) - \pi^{-1}X(i-1) | >$$

Define density in value for sequence X:

$$\rho_v = 1/ < | \pi(X(i)) - \pi(X(i-1)) | >$$

For S sequences $X_1 \cdots X_S$, the complexity of merging is

$$O(N - \sum_{\alpha=1}^{S} |X_\alpha| (\lg\rho_p(X_\alpha) + \lg\rho_v(X_\alpha)))$$

Riffle shuffles are reflected in ρ_p; runs in ρ_v. Weak segmentation is reflected in variations in density, and in the positions of endpoints of subsequences.

Merge sort with exponential search behaves sub-optimally over strong segmentation because there is a strong correlation (low relative entropy) between the positions of insertions in different passes. For a segment of length l, merge sort requires $O(\lg l)$ comparisons to relocate the segment boundaries on each pass, for a total of $O(\lg l \lg N)$ comparisons. However the complexity per segment is only $\lg N$, or $O(1)$ per pass. In the worst case, the segment length is $O(\lg N)$, and the comparison count is multiplied by $\lg\lg N$.

References

1. J. Mannila, "Measures of Presortedness and Optimal Sorting Algorithms," *IEEE Trans. Computing*, vol. 34, pp. 318-325, 1985.
2. C. Levcopoulos and O. Petersson, "Sorting Shuffled Monotone Sequences," in *Proc. 2nd Scandinavian Workshop on Algorithms & Theory*, Lecture Notes in CS, vol. 447, Springer-Verlag 1990.
3. S. S. Skiena, "Encroaching Lists as a Measure of Presortedness," *Bit*, vol. 28, pp. 775-784, 1988.
4. O. Petersson and A. Moffat, "A Framework for Adaptive Sorting," in *Proc. Scandinavian Workshop on Algorithms & Theory*, LNCS, Springer-Verlag, 1992.
5. A. J. Lichtenberg and M. A. Lieberman, *Regular and Stochastic Motion*, pp. 260-268, Springer-Verlag, 1983.
6. C. Levcopoulos, O. Petersson, and S. Carlsson, "Sublinear Merging and Natural Mergesort," in *Proc SIGAL International Symposium on Algorithms*, Lecture Notes in CS, vol. 450, Springer-Verlag, 1990.
7. J. Bentley and A. Yao, "An Almost Optimal Algorithm for Unbounded Searching," *Information Processing Letters*, vol. 5, pp. 82-87, 1976.
8. K. Mehlhorn, "Sorting Presorted Files," in *Proc. 4th GI Conference on Theoretical CS*, Lecture Notes in CS, vol. 67, Springer-Verlag, 1979.
9. T. M. Cover and J. A. Thomas, *Elements of Information Theory*, pp. 144-150, Wiley & Sons, 1991.
10. J. Bentley and D. McIlroy, "Engineering a Sort Function," *(submitted)*.
11. A. Moffat, G. Eddy, and O. Petersson, "How Good is Splaysort?," in *Australian Computer Conference (Proceedings—to be published)*, Feb. 1993.