k-WAY MERGING AND k-ARY Sorts

William A. Greene
Computer Science Department
University of New Orleans
New Orleans, Louisiana 70148
WAGCS@UNO

Abstract: We present a divide-and-conquer algorithm for merging k sorted lists, namely, recursively merge the first \( \lfloor k/2 \rfloor \) lists, do likewise for the last \( \lfloor k/2 \rfloor \) lists, then merge the two results. We get a tight bound for the expense of the worst case behavior of this merge. We show the algorithm is cheapest among all similar divide-and-conquer approaches to k-way merging (here we have in mind different partitionings of the k source lists). We compute the expense of the k-ary sort; sometimes its expense is identical to that of the BinarySort.

This research originates from our interest in the MergeSort, which here will be called the BinarySort. The BinarySort achieves its low computational cost by a divide-and-conquer strategy: given a list to sort, cut it into its front and back halves, separately sort each half recursively, then merge those two sorted lists together to produce the sorted version of the input list. (The algorithm MERGE we have in mind for merging two sorted lists is the standard one.)

We ask, what is the computational cost of the k-ary sort? By the latter we mean a sort that (analogous to the BinarySort) divides its input list into some integral number \( k \geq 2 \) of disjoint sublists of (nearly) equal size, separately recursively sorts each of those \( k \) sublists, then merges those \( k \) sorted results back together to form the final result. Measuring computational cost as the number of comparisons made between list elements, it follows that all such cost is incurred by the merging operations.

So, we should investigate how we may most cheaply merge \( k \) sorted lists. Let \( k \) sorted lists be given (not necessarily of nearly equal length); let \( n \) be the sum of their lengths. Three choices for merging the \( k \) lists are:

1. LinearSearchMerge: at each step of merging, a linear search is used to find the smallest of \( k \) items. The (worst case) total cost of this approach to merging is approximately \( (k-1)n \) comparisons. Cheaper is:

2. HeapMerge: \( k \) items (one from each source list) are maintained in a heap. After copying the smallest (root), it is replaced by its neighbor in its source list; reheapifying has cost \( 2 \log k \). (here "log" means to base 2). The total cost of merging under this approach is approximately \( 2 \log k \). Cheaper still is:

3. Divide-and-Conquer-Merge: recursively merge the first \( \lfloor k/2 \rfloor \) lists, recursively merge the last \( \lfloor k/2 \rfloor \) lists, then MERGE the two results. For the cost we have:

Theorem: Assume the \( k \) sorted source lists are arranged in decreasing order of length. Let \( n \) be the sum of their lengths, and let

\[ C = n \left\lfloor \log k \right\rfloor - (n/k)^2 \log k + n - k + 1. \]

The Divide-and-Conquer-Merge performs at most \( C \) comparisons between list elements.

Note that amount \( C \) approximates \( n \left\lfloor \log k \right\rfloor \), so here we do half as many comparisons as with HeapMerge.

Corollary: If the \( k \) lists all have the same length, then Divide-and-Conquer-Merge performs exactly \( C \) comparisons (in the worst case). Thus \( C \) is a tight bound.

Like BinarySort, Divide-and-Conquer-Merge economizes by halving; it halves the number of lists to be merged. Other partitionings of the set of \( k \) source lists are conceivable. By a D&CM-Algorithm for merging \( k \) lists let us mean one whose behavior is described: partition the \( k \) lists into \( j \) subsets, where \( 1 < j < k \); recurse to merge each of the \( j \) subsets; recurse to merge the \( j \) results. (If \( k=2 \) then MERGE). Our next result shows that, when merging lists all of the same length, partitioning the list set into two halves is optimal.

Theorem: To merge \( k \) sorted lists which all have the same length and whose lengths sum to \( n \), any D&CM-Algorithm must do at least \( C \) comparisons between list elements in the worst case.

Regarding the k-ary sort, we obtain:

Theorem: (1) When list length \( n \) is a power of \( k \), the k-ary sort performs

\[ \left( 1 + \left\lfloor \log k \right\rfloor - (n/k)^2 \log k \right) \log k \log n - n + 1 \]

comparisons in the worst case.

(2) If also \( k \) is a power of 2, then the worst case behavior of the k-ary sort is identical to that of the BinarySort.

(3) The k-ary sort has runtime \( O(n \log n) \).