Sorting with powerful primitive operations[1]

M. D. Atkinson
D. Nussbaum

School of Computer Science, Carleton University
Ottawa, Canada K1S 5B6

Abstract. The cost of a sorting algorithm is the number of primitive operations used in a worst case execution of the algorithm. In the standard model the primitive operation is a binary comparison, which sorts a pair of keys. Cost measures based on other primitive operations are considered. A general lower bound for the cost of a sorting algorithm is given which is valid for a wide class of possible primitives. For several special primitive operations sorting algorithms are given. The primitive operations studied are: sorting k keys, finding the largest among k keys, and merging lists of lengths i,j.

AMS Classification numbers: 06A10, 68C05

1. Introduction

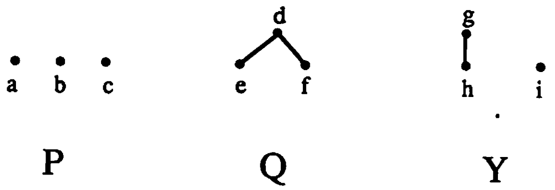
The standard comparison based model for the problem of sorting n keys into order assesses the cost of an algorithm as the number of (binary) comparisons of keys used in the algorithm (see [2] for the theory and practice of such algorithms). We may view this operation as one which discovers, in a subset of two keys, more order than was previously known. In this particular case the comparison operation begins with a subset about which (presumably) no order information is known and ends with complete information about the order relations of the subset. This paper investigates the power of more general primitive operations. Informally, we shall be considering operations which work in the following way: they are applied to subsets of some fixed size which already possess some order structure (perhaps, trivial structure) and their effect is to produce a fixed amount of additional order information (perhaps a total ordering of the subset). A more precise definition of such a primitive operation is given below. For the most general operation of this type we give a lower bound on the number of them required to sort n keys (in the worst case). We go on to consider some special cases of these operations and derive algorithms for sorting which attempt to utilise the particular operation as effectively as possible. For the classical case when the primitive operation is the binary comparison every sorting algorithm requires at least $\log_2 n!$ operations. It is traditional to use Stirling's approximation, neglect lower order terms, and express this as $n \log_2 n$; since both binary merge and insertion sort achieve this lower bound (again, apart from lower order terms) we can regard the classical case as essentially solved. For other types of primitive operation lower bounds of the form $\Omega(n \log n)$ and upper bounds $O(n \log n)$ can be obtained without difficulty; the main interest is in the implied constant factors. While we are able to narrow the gap between upper and lower bounds for the worst case complexity, only in a few cases can we find the actual constant which multiplies the $n \log n$ factor. This work is reminiscent of various models for sorting on parallel machines [1] but also contains elements of sequential computation.

We now give a formal definition of a primitive sorting operation. Suppose that P is an abstract poset with k elements and that Y is a set of k keys which has, because of

[1]This research was supported by the National Science and Research Council of Canada under Grant No. A4219

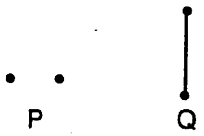
previously discovered order relations, a partially ordered set structure. If there is an order monomorphism $\alpha:P\rightarrow Y$ (that is, a one-to-one map α such that $x<y$ implies $\alpha(x)<\alpha(y)$) then we say that Y *has the structure of* P.
Suppose now that P,Q are two abstract posets each with k elements and that Y is a set of k keys which, by virtue of an order monomorphism $\alpha:P\rightarrow Y$, has the structure of P. A $(P\rightarrow Q)$ operation is an operation whose outcome is a function decomposition $\alpha=\beta\gamma$ where $\beta:Q\rightarrow Y$ and $\gamma:P\rightarrow Q$ are both order monomorphisms. The mapping γ gives Y the structure of Q (the $(P\rightarrow Q)$ operation has thereby discovered new order relations within Y) and the mapping β shows the precise way that, in Y, the structure of P corresponds, element by element, to the structure of Q.

Example

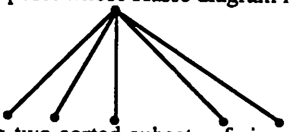


We take α to be $\alpha:(a,b,c)\rightarrow(g,h,i)$. The $(P\rightarrow Q)$ operation is essentially the operation of identifying the maximum among 3 keys. Two of the possible outcomes are
(i) $\beta:(d,e,f)\rightarrow(g,h,i)$ and $\gamma:(a,b,c)\rightarrow(d,e,f)$ (here the operation determined that g was the maximal element), and
(ii) $\beta:(d,e,f)\rightarrow(i,h,g)$ and $\gamma:(a,b,c)\rightarrow(f,e,d)$ (here the operation determined that i was the maximal element)

In this terminology an ordinary comparison is a $(P\rightarrow Q)$ operation with P and Q being the posets



Other examples of this concept (all of which will be studied in more detail in later sections) are
1. The operation of sorting a set of k keys (k fixed); here the poset P would be an anti-chain of size k while Q would be a chain of length k,
2. The operation of finding the maximum among k keys (k fixed); again, P would be an anti-chain and Q would be the poset whose Hasse diagram is



3. The operation of merging two sorted subsets of sizes i,j together (i,j both fixed); P consists of two disjoint chains of lengths i,j while Q is a chain of length i+j.
A $(P\rightarrow Q)$-algorithm is one which accomplishes its result (sorting or merging a set of n keys) by a succession of $(P\rightarrow Q)$ operations. After each $(P\rightarrow Q)$ operation the algorithm

examines the information it has discovered and chooses another subset on which to apply the next (P→Q) operation. It is assumed implicitly that no other order revealing operations (such as binary comparisons) can be used in a (P→Q)-algorithm. The execution cost of a (P→Q)-algorithm is the number of (P→Q) operations that it uses. Usually in what follows the particular (P→Q) operation is clear from the context and we shall refer to "algorithm" rather than "(P→Q)-algorithm". Before going on to study upper bounds on the worst case execution time algorithm costs for particular (P→Q) operations we give a general lower bound result.

Theorem 1 For any posets P and Q (defined on sets of the same size and such that order monomorphisms from P to Q exist) the number of (P→Q) operations required to sort n keys is, neglecting lower order terms, at least $\frac{n \log n}{\log L(P) - \log L(Q)}$,[1] where L(P) denotes the number of linear extensions of the poset P.

The proof is a generalisation of the standard information theoretic argument and requires the following simple lemma.

Lemma A Let $a_1 \geq a_2 \geq \ldots \geq a_n$ be n numbers with sum s, and suppose that $1 \leq m \leq n$. Then $a_1+a_2+\ldots+a_m \geq \frac{m}{n} . s$

Proof. Consider the sum $\Sigma_M \Sigma_{i \in M} a_i$ where the outer sum is over all subsets M of $\{1,2,\ldots,n\}$ of size m. In this sum each a_i occurs exactly ${}^{n-1}C_{m-1}$ times and so the value of the sum is ${}^{n-1}C_{m-1}.s$. However if the lemma is false then every subset of $\{a_1,a_2,\ldots,a_n\}$ would have value less than $\frac{m}{n}.s$ and so the value of $\Sigma_M \Sigma_{i \in M} a_i$ would be less than ${}^{n}C_m . \frac{m}{n} . s = {}^{n-1}C_{m-1}.s$.

Proof of Theorem 1. We shall show that each (P→Q) operation reduces the number of possibilities for the sorted order by a fraction at most equal to $\frac{L(Q)}{L(P)}$ in the worst case. The theorem will then follow since the number of (P→Q) operations required to reduce the n! initial possible orderings down to exactly one will be at least

$$\log_{L(P)/L(Q)} n! = \frac{\log n!}{\log(L(P)/L(Q))} = \frac{n \log n}{\log L(P) - \log L(Q)},$$ neglecting terms of lower order than n log n, by Stirling's approximation.

Let s be the number of possibilities for the sorted order at some general step in a sorting algorithm and let Y be a set with partial order structure P to which the next primitive (P→Q) operation is to be applied. These s possibilities fall into L(P) classes one for each linear extension of P. The effect of the (P→Q) operation is that only those classes which are consistent with Q can remain possibilities for the final ordering. There are L(Q) such classes. In the worst case these L(Q) classes are the largest and, by the lemma, their union has size at least $\frac{L(Q)}{L(P)}.s$.

[1] Here, and subsequently, all logarithms are to the base 2

We now consider upper bounds for the worst case complexity of sorting and merging with various types of $(P \rightarrow Q)$ operation. When merging several lists together we can avoid any difficulties caused by one of the input lists becoming exhausted by adding a dummy element $-\infty$ to each list. Our sorting algorithms are mostly based on merging algorithms and we obtain their execution costs by the following well known result.

Lemma B Let $M(r,n)$ be the worst case $(P \rightarrow Q)$ cost of merging r lists of total length n, and suppose that $M(r,n)$ satisfies $rM(r,n/r) \leq M(r,n)$ for all n. Then, except for small order terms, the worst case sorting cost $S(n)$ satisfies $S(n) \leq \frac{M(r,n)}{\log r}.\log n$.
Proof. The execution cost of the recursive sorting algorithm based on r-ary merge satisfies the recurrence $S(n) \leq rS(\frac{n}{r}) + M(r,n)$ and this has the solution given in the lemma.

2. The operation of computing the largest, next largest, ...

Theorem 2 Suppose that the primitive operation is that of finding the largest, second largest, ..., t th largest in a set of k elements. Then $M(k/t,n) \leq n/t$.
Corollary With this primitive operation then, except for smaller order terms, $\frac{n \log n}{\log k! - \log(k-t)!} \leq S(n) \leq \frac{n \log n}{t \log k/t}$. In particular, if t=1, $S(n) = \frac{n \log n}{\log k}$.
Proof. For simplicity we shall take k to be an exact multiple of t. Each step of the merging algorithm applies the given primitive operation to the union of the sets of t largest keys from each of the k/t lists being merged. The t largest elements returned by the operation are necessarily the t largest elements in the whole set of remaining keys. They are then removed from the set of keys and placed in an output buffer. Clearly, n/t such operations suffice to complete the merge.

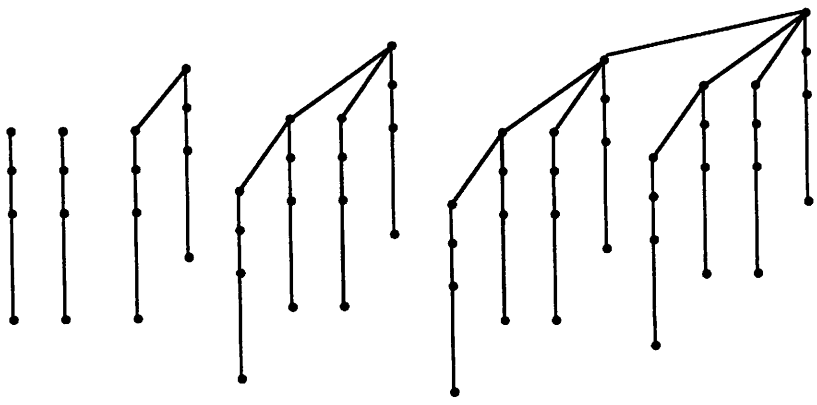
This proves the theorem. The corollary follows from Theorem 1 (lower bound) and from Lemma B (upper bound). The corollary is optimal when t=1. Moreover, Stirlings formula shows that, when k is large compared to t, the upper and lower bounds are fairly close.

3. The operation of sorting k elements

In this section we consider the operation σ_k which sorts a set of k elements into order.

Theorem 3 When the primitive operation is σ_k then $M(2^{k-1},n) \leq n+O(2^k)$.

Corollary With the primitive operation σ_k then, except for smaller order terms, $\frac{n \log n}{\log k!} \leq S(n) \leq \frac{n \log n}{k-1}$.
Proof The merging algorithm begins by defining the structure of a set of complete binomial heaps on the set of maximal elements of the 2^{k-1} lists to be merged, one heap of each size $2,4,8,\ldots,2^{k-2}$, and two heaps of size 1. As shown in the following figure, the partial order structure so obtained has a diagram which consists of several components. We shall say that a component has *type* 2^j if it involves 2^j of the lists being merged.



To create the binomial heap structure the standard algorithm for creating binomial heaps is used. This algorithm uses $O(2^k)$ binary comparisons and so $O(2^k)$ of the σ_k operations. Once the binomial heap structure is established each step of the algorithm applies the σ_k operation to the k maximum elements of each component. Suppose that it is discovered that the largest element is found in the component of type 2^j. This element is output and the component then splits into connected components which involve, respectively, $1,1,2,4,8,....,2^{j-1}$ lists; moreover the maximum elements remaining in these lists have the structure of complete binomial heaps of size $1,1,2,4,8,....,2^{j-1}$. However the σ_k operation has had another consequence: the original components of types $1,1,2,4,8,....,2^{j-1}$ have been given the structure of a component of type 2^j. This is because the two components of type 1 become (through their maximal elements being ranked) a component of type 2; this component taken together with the original component of type 2 yields a component of type 4, and so on. In other words, at the cost of one σ_k operation, one more output element has been identified and the remaining elements continue to be structured into components of types $1,1,2,4,8,...,2^{k-2}$. After n-k of these operations only k elements remain and then one final σ_k operation completes the merge.

4. The operation of merging

In this section we consider the primitive operation $\mu(i,j)$ to be that of merging two sorted lists of lengths i,j.

Theorem 4 Suppose that the primitive operation is $\mu(k,k)$. Then $M(2^j,n) \leq (2^j-1)n/k$.
Proof. We perform a balanced binary merge taking initially pairs of lists of total length $n/2^{j-1}$, continuing with pairs of lists of length $n/2^{j-2}$, and so on. When two lists are being merged we repeatedly apply the operation to the k largest elements in the first list and the k largest in the second list. This yields the next k elements to be output, in order, they are now deleted from the input and the operation is repeated. Since k elements are output at every stage the number of stages is $n/k\sum_{r=0}^{j-1}2^r$.

Corollary With this primitive operation then, except for smaller order terms,

$$\frac{n \log n}{\log\binom{2k}{k}} \leq S(n) \leq \frac{n \log n}{k}.$$

Proof. The lower bound follows from Theorem 1 and the upper bound follows from Lemma B.

The result of the Corollary has an alternative proof which allows a generalisation to the $\mu(i,j)$ operation.

Theorem 5 Suppose that the primitive operation is $\mu(i,j)$. Then, except for smaller order terms,

$$\frac{n \log n}{\log\binom{i+j}{j}} \le S(n) \le \frac{2n \log n}{(i+j)\log(i+j) - i \log i - j \log j}$$

Proof. First we observe that two lists of lengths m,n may be merged in $\frac{m}{i} + \frac{n}{j}$ operations of type $\mu(i,j)$. The algorithm is the natural one: it continually applies $\mu(i,j)$ to the i largest elements $a_1, a_2, \ldots, a_i$ of the first list and j largest elements $b_1, b_2, \ldots, b_j$ of the second list. If $a_i < b_j$ then i elements of the first list at least may be output to the result list, and otherwise j elements of the second list at least may be output to the result list.
We now use this merging result in a binary merge sorting algorithm. The merge is, however, not a balanced merge: whenever a sorted list of length k is required it is created by splitting the set of k keys into subsets of sizes in the ratio i:j, sorting recursively, and merging as above. The execution time T(n) of this algorithm therefore satisfies

$$T(n) = T(\tfrac{ni}{i+j}) + T(\tfrac{nj}{i+j}) + \frac{ni}{(i+j)i} + \frac{nj}{(i+j)j}$$

and the obvious inductive proof then establishes the claimed upper bound. The lower bound follows from Theorem 1.

In Theorem 5 the Stirling approximation indicates that the lower and upper bounds differ by a multiplicative factor close to 2 (although some caution should be used in applying this approximation since Stirling's formula is an asymptotic formula and i and j are constants). Next we consider an algorithm which brings the lower and upper bounds closer together when j is very much larger than i. For convenience we shall take j to be an exact multiple of i.

Theorem 6 Suppose that the primitive operation is $\mu(i,j)$ where $j=qi$. Then, except for terms of lower order,

$$\frac{n \log n}{\log\binom{i+j}{j}} \le S(n) \le \frac{n \log n}{i \log(i+j) - i \log i}$$

Proof. The lower bound is given in Theorem 5. For the upper bound an algorithm similar to insertion sort is used. This algorithm inserts i elements at a time into a sorted list of length m ($m=0,i,2i,\ldots$). The cost of such an insertion step is $\log_{q+1} m$ + constant (as shown below) and hence the total cost is

$$\sum_{m=0 \bmod i}^{m=n} \log_{q+1} m = \sum_{k \le n/i} \log_{q+1} ki = \frac{\log(n/i)!}{\log q+1} = \frac{n \log n}{i \log(j/i + 1)}$$

(where the equalities in the equations above neglect terms of lower order than n log n) as required.
All that remains is to describe a typical insertion step. Let $L = \{y_1, y_2, \ldots, y_m\}$ be a sorted list of length m and let $x_1, x_2, \ldots, x_i$ be i keys to be inserted into L. We maintain i sets $I_1, I_2, \ldots, I_i$, initially all defined as $[1..m+1]$, such that each I_k is the range of candidate positions where x_k should be inserted: in other words, for some $j \in I_k$, $y_j < x_k < y_{j+1}$. To begin with we sort $x_1, x_2, \ldots, x_i$ Of course, this must be done with the $\mu(i,j)$ operation which is not very well suited for this task; however, with a $\mu(i,j)$ operation we can perform

an ordinary comparison by introducing dummy keys with value ∞ and the sorting can be done with cost bounded by a constant.

Next, $\log_{q+1} m$ of the $\mu(i,j)$ operations are performed; each one reduces the sizes of I_1, $I_2, \ldots, I_i$ by a factor of q+1. A typical $\mu(i,j)$ operation is applied as follows. We select q equally spaced points from each distinct set I_k (thereby dividing I_k into q+1 subsets of equal size); let $y_{k1}, y_{k2}, \ldots, y_{kq}$ be the keys indexed by these points. The set of (distinct) y_{kr} is a sorted set of size at most $qi=j$. The $\mu(i,j)$ operation is now applied to the list of all y_{kr} and the list $x_1, x_2, \ldots, x_i$. From the result of the operation we can determine which of the q+1 equally sized subsets of each I_k contains the insertion point for x_k; this subset becomes the new I_k, and the desired effect has been achieved.

An elementary calculation shows that

$$\frac{n \log n}{i \log(i+j) - i \log i} \le \frac{2n \log n}{(i+j) \log(i+j) - i \log i - j \log j - i}$$

Thus, when j is large compared to i, the upper bound in Theorem 6 is almost twice as good as that in Theorem 5.

References

1. S.G. Akl, Parallel Sorting Algorithms, Academic Press (Orlando, London), 1985.
2. D.E. Knuth, Sorting and Searching, Volume 3 of The Art of Computer Programming, Addison-Wesley (Reading, Massachusetts),1973.