ON THE DISTRIBUTION OF COMPARISONS IN SORTING ALGORITHMS

Dana Richards
University of Virginia

Pravin Vaidya
University of Illinois

Computer Science Report No. TR-86-06
March 17, 1986

On the Distribution of Comparisons in Sorting Algorithms

Dana Richards
University of Virginia

Pravin Vaidya
University of Illinois

1. Introduction

It is well-known that the information-theoretic lower bound for comparison-based sorting algorithms is $\lceil \lg n! \rceil = \Omega(n \log n)$ comparisons, for n keys. We ask whether or not we can expect those comparisons to be evenly distributed over the keys. Clearly some algorithms on some inputs will use fewer total comparisons and some keys can be rarely used. However we want to know if every sorting algorithm has some inputs which cause the distribution of comparisons to be "equitable". We restrict our attention to pair-wise comparisons and assume the algorithms are presented as decision trees. If some keys are more costly, e.g. lengthy or inaccessible, then we would favor an algorithm that is biased against them. While it is possible a set of keys may be avoided it is clear that no particular key can be uniformly ignored.

Lemma 1: For every sorting algorithm any particular key will be involved in $\Omega(\log n)$ comparisons for some input.

Proof: Suppose some key x was ignored. Assume an arbitrary total ordering for the other $n-1$ keys. Modify the decision tree by removing all unnecessary comparisons. What remains must be the tree for an algorithm for inserting x into a sorted list of $n-1$ keys, with x involved in every comparison. It is well known that such a tree has a path of length $\Omega(\log n)$. □

Our original goal was to establish the following conjecture, which seems to be the best possible uniform conjecture.

Conjecture: Every sorting algorithm for some input will involve every key in $\Omega(\log n)$ comparisons.

The reason we say it is best possible is because of the result of Ajtai, Komlos, and Szemeredi [AJTA83]. They gave a sorting network that used only $O(\log n)$ time and $O(n)$ processors. It is an immediate consequence that a serialization of the network will involve every key in only $O(\log n)$ comparisons. Their result is of course much stronger so it is surprising it is the first serial algorithm with this property we have seen.

The conjecture is false. After investigating various complicated lower bound arguments we discovered a sorting algorithm which guarantees one key will be involved in at most 3 comparisons! This algorithm and its generalizations will be discussed in section 5. Our "partial results" in sections 2, 3, and 4 turned out to be close to the upper bounds. The main proof technique we have tried is the use of adversaries or oracles. Many different mechanisms for the adversaries have been tried and some will be described below. In section 6 we discuss the validity of the conjecture for "natural algorithms".

2. A Simple Adversary

We can show that most of the keys must satisfy the conjecture. We extend the arguments of Atallah and Kosaraju [ATAL81]. First the adversary partitions the keys and assumes some total ordering on the first k keys (i.e. $a_1<a_2< \cdots <a_k$) while the rest (i.e. $b_1, b_2, \cdots, b_{n-k}$) are free to be in any of the $k+1$ intervals defined by the a's. The adversary dynamically associates with each b_i a *range* of intervals that are still candidates for b_i. It tries to have the number of intervals in b_i's range shrink from $k+1$ to 1 slowly (by at most half per comparison).

There are three types of comparisons - $a_i:a_j$, $a_i:b_j$, and $b_i:b_j$. The first case is trivially answered. The $a_i:b_j$ case is easy if a_i is not in the range of b_j. If it is then answer so that the resulting range of b_j is as large as possible. The last case is best handled by cases. There are essentially four cases and it is an easy exercise to give the answer required.

The original range of each b_i is $k+1$ and the number of intervals in the range of b_i is at most halved after each comparison involving b_i and must eventually become 1. Therefore each b_i is in at least $\lg(k+1)$ comparisons giving an $\Omega((n-k)\log k)$ lower bound for sorting. Setting $k = n/2$ gives yet another proof of the $\Omega(n \log n)$ bound for sorting, though it is one of the few constructive adversary-based proofs. Those comparisons among the a's, which the adversary considers redundant, and among the b's that end in the same interval are not considered in the analysis.

Theorem 1: Every sorting algorithm for some inputs will involve $n - n^{\epsilon} + 1$ keys in at least $\epsilon \lg n$ comparisons, $\epsilon > 0$.

Proof: Note that if $k = n/g(n) - 1$ we have $n - n/g(n) + 1$ b's each involved in $\lg n - \lg g(n)$ comparisons. (In [ATAL81] $g(n) = \lg n$.) We let $g(n) = n^{1-\epsilon}$. So $n - n^{\epsilon} + 1$ keys are each in $\epsilon \lg n$ comparisons. $\square$

So as ϵ decreases more keys satisfy the conjecture, albeit more weakly. The same result will be independently derived in the next section.

3. Poset-Based Adversaries

In this section we consider adversaries that make use of the poset formed by their prior answers to comparisons. Note that "adversaries" are cooperating in that they answer in a way that will hopefully send us down the best branch of the decision tree. It can inspect the two possible posets resulting from a comparison and choose, by some criterion, the poset that promises to be the most "equitable".

We have investigated several functions the adversary could use based on known techniques, e.g. [SAKS85] and [FUSS79]. These attempts, detailed in [RICH84], were not successful as we found no mechanisms for isolating the direct and indirect effects on individual keys over time.

Griggs [GRIG82] studied certain functions on the elements of posets. In particular for the poset P, he considered $h(x)$ where $x \in P$ and $0 \leq h(x) \leq 1$. Let $h(S) = \sum_{x \in S} h(x)$, where

S is some subset of P. A condition in his paper is that $h(C) \leqslant l$ for every chain C in P, where l is a constant. We do not use his theorems. However a desire to find such an $h(.)$ for dynamically changing posets led naturally to the adversary described in the next paragraph.

Initially P is an antichain and $h(x) = 1$ for each $x \in P$. We want to continue to have $h(C) \leqslant 1$ for all (maximal) chains as we sort. When we answer $a{:}b$ we halve $h(a)$ and $h(b)$; therefore if $h(a) = 2^{-k}$ we know a has been in k comparisons. Further we look at $P_<$ and $P_>$, the two possible resulting posets, and choose the one with the lightest (i.e. least $h(C)$) chain through a and b. If, inductively, $h(C) \leqslant 1$ for all C in P, then it will be true in the poset chosen.

Lemma 2: Each comparison $a{:}b$ can be answered so that $h(C) \leqslant 1$ for each chain C in the resulting poset.

Proof: The basis of inductive argument is clear. Consider the maximal chains above and below a and b before the comparison. Let U_a be the maximum cost of a chain with a as it least element with the cost of a, $h(a)$, subtracted out. Let D_a be the maximum cost of a chain with a as it greatest element, again with $h(a)$ subtracted out. Suppose that there must be a chain C with $h(C) > 1$, then it follows

$$(U_a + \frac{h(a)}{2} + \frac{h(b)}{2} + D_b) + (D_a + \frac{h(a)}{2} + \frac{h(b)}{2} + U_b) > 2.$$

Therefore there was a chain through a or b before that violates the inductive assertion. □

Clearly when we are done we have just one chain and $h(P) \leqslant 1$. Alternatively, $\sum_{x \in P} 2^{-c(x)} \leqslant 1$ where x was involved in $c(x)$ comparisons. This is a interesting inequality. As an aside we note it provides yet another constructive adversary-based proof of the lower bound for sorting, due to the following observation.

Lemma 3: $\sum_{x \in P} 2^{-c(x)} \leqslant 1$ implies $\sum_{x \in P} c(x) = \Omega(n \log n)$.

Proof: A familiar result relating the geometric and arithmetic means (e.g. [WILF85]) states

$$(y_1 y_2 \cdots y_n)^{1/n} \leqslant \frac{(y_1 + \cdots + y_n)}{n}.$$

Assume $n = 2^k$. We know

$$\frac{(2^{k-c(x_1)} + \cdots + 2^{k-c(x_n)})}{n} \leqslant 1$$

so

$$(2^{k-c(x_1)} 2^{k-c(x_2)} \cdots 2^{k-c(x_n)})^{1/n} \leqslant 1$$

and

$$n \lg n \leqslant \sum c(x).$$

The case of $n \neq 2^k$ is similar. □

This inequality alone states that not many elements can have small $c(x)$'s but does not disallow a few. It gives us the following result which improves the result in section 2 only by a constant.

Theorem 2: Every sorting algorithm for some input will involve $n - n^{\epsilon}/2 + 1$ keys in at least $\epsilon \lg n$ comparisons, $\epsilon > 0$.

Proof: Note that only one $c(x)$ could be 1 since it contributes ½ to the sum $\sum 2^{-c(x)}$. More "harm" could be done by 3 $c(x)$'s being 2 with a net contribution of ¾. Clearly the maximum number of keys with $c(x) < \epsilon \lg n$ is achieved if each such key has $c(x) = \epsilon \lg n - 1$. The number of such keys is bounded by $2^{\epsilon \lg n - 1} - 1$ which gives the result. □

If we could show that $h(P) \leqslant n^{-\epsilon}$ then the conjecture would follow. However the following result shows that this is not nearly true.

Lemma 4: For some sorting algorithms we can have

$$h(P) \geqslant \frac{1}{3} + O(2^{-n/2})$$

for the total order P.

Proof: Consider the following algorithm which begins by sorting a set S of 3 keys. After some processing we will increase the size of S to 5 sorted keys, then 7, and so on. For each size of S we compare all the keys not in S to the second smallest key of S, and, in all cases, the second smallest key wins the comparison. This is consistent with lemma 2. After that we increase the size of S by two by sorting the smallest key of S with two keys not in S. Iterate until S contains all n keys.

Because of the regularity of the algorithm we can easily predict the final values of the $h(x)$'s and sum them. The important point is that $h(x)$ remains comparatively large for the largest, the third largest, and so on. For n odd we get

$$h(P) = \frac{(1-2^{1-n})}{3} + 2^{2-n} + 2^{3-n} + 2^{(3-n)/2}.$$

We get a similar expression for n even. □

It is possible to generalize this approach to take into account how the comparisons were resolved [PLES85]. In particular, let $w(x)$ be the number of comparisons x "won" and $l(x)$ be the number of losses; $c(x) = w(x) + l(x)$. Of course $\sum w(x) = \sum l(x)$. Define $h_r(x) = r^{w(x)}(1-r)^{l(x)}$, $0<r<1$, so that $h_{.5}(x) = h(x)$. A result analogous to lemma 2 can devised, i.e. we can maintain $h_r(C) \leqslant 1$ for all C. However it is not clear how to take advantage of this formulation.

4. Binary Tree Based Adversaries

We can derive the result in theorem 2 by yet another approach. We present an adversary that responds in an automatic fashion based on a binary tree data structure it maintains. Since a sorting algorithm could take advantage of the adversary's determinism it is surprising that we can get stronger results than with the poset-based adversary. This adversary was independently used to prove (unrelated) results about searching with preprocessing [BORO81, LYNC78].

The data structure is an infinite binary tree with tokens distributed over the nodes. It is convenient to regard the infinite subtrees without any tokens as being pruned away. The tokens, labeled $1, 2, \cdots, n$, are identified with the keys. Initially all n tokens are at the root. Let $n(i)$ be the node containing token i. To answer a comparison about the i th and j th keys the adversary locates the tokens labeled i and j in the tree. It maintains the following invariant:

> If $n(i)$ is an ancestor of $n(j)$ then the corresponding keys are incomparable. Otherwise, if $n(i)$ is to the left of $n(j)$, relative to their least common ancestor, then the i th key is less than j th key.

The adversary does not move any tokens if the comparison is already answered by the invariant. If, say, $n(i)$ is a proper ancestor of $n(j)$ then the token j is not moved and the token i is moved to the right (left) son of $n(i)$ if $n(j)$ is in its left (right) subtree. If $n(i) = n(j)$ then, arbitrarily, token i is moved to its left son and token j is moved to its right son.

As the sorting process progresses the tokens move down away from the root. It is clear that the invariant is maintained. Therefore, when sorting is done no token can be the ancestor of another. The following claim is easily verified: No node has tokens in one of its subtree while having no tokens in its other subtree. We can now give an alternate proof of theorem 2.

Proof (Theorem 2): From the above claim we see that the infinite tree when pruned after sorting gives a full binary tree with $n-1$ internal nodes and n leaves; each leaf with a single token on it. Let $depth(v)$ be the depth of a node v in the tree. Since a token only moves during a comparison we have $c(i) \geqslant depth(n(i))$, where $c(i)$ is the number of comparisons the i th key is involved in, much as before.

It is known (e.g., [KNUT68], problem 2.3.4.5-3) that if $l_1, l_2, \cdots, l_n$ are the leaves of a full binary tree then $\sum 2^{-depth(l_i)} = 1$. Therefore it follows that $\sum 2^{-c(i)} \leqslant 1$. The remainder of the proof follows as before. □

It is bothersome, due to the definition of the adversary, that $c(i)$ can be much greater than $depth(n(i))$. One attempt to correct for this is to have a token move after every comparison involving it. Unfortunately this causes more difficulties than it resolves. However we can go further than the above proof.

Let v be a node in the tree and $L(v)$ and $R(v)$ be the set of tokens in its left and right subtrees, respectively. For a set of tokens T, let $C(T)$ be the sum of the comparison counts for the keys associated with those tokens, i.e. $C(T) = \sum_{i \in T} c(i)$. The following theorem states that while the resulting binary tree can be skewed, with perhaps only a few tokens in the left subtree, the distribution of comparisons is not as skewed. One, of many possible, corollaries is given to show how to apply the result. Let $S(n)$ be the number of comparisons needed to sort n keys.

Theorem 3: For any node v in the binary tree after the sorting process

$$C(L(v)) \geqslant |R(v)| + 2S(|L(v)|) + depth(v)|L(v)|$$

as well as with L and R interchanged.

Proof : Every comparison for a key now in the left subtree of v involved a second key that is now in the right subtree, the left subtree, or elsewhere in the tree. Every key now in the right subtree got there by a comparison of the first type. This gives the $|R(v)|$ term. Only keys in the left subtree are relevant in sorting those keys, hence the second term. Note that each such comparison is double-counted in the summation. Finally, the total number of comparisons answered before the tokens arrived at node v is bounded by the third term. □

Corollary 1: If one key was involved in only 2 comparisons then another key was involved in at least $n-2$ comparisons.

Proof : Let the i th key be involved in just two comparisons, i.e. $c(i)=2$. By the theorem $n(i)$ can not be a son of the root, so $n(i)$ is at depth 2. The sibling node of $n(i)$ must contain a token i'. Otherwise the theorem would be violated at their mutual parent v. Hence $c(i') + c(i) \geqslant (n-2) + 2 + 0$, since $S(2)=1$. □

5. Upper Bounds

We begin by showing that there exists an algorithm that can effectively avoid one of its keys. The algorithm sorts by repeatedly inserting a new key into a previously sorted sublist S. Initially S is created by sorting three keys; obviously not using more than two comparisons per key. The algorithm finds the least used key x in S and compares the new key z with the keys w and y which are just less and greater than x in S, respectively. If $w<z<y$ then compare z with x and stop, otherwise avoid x. Note that w or y may not exist creating simpler special cases.

Theorem 4: There exists a sorting algorithm that involves at least one key in at most three comparisons for every input.

Proof: A simple inductive proof can be based on the above algorithm. By hypothesis x has been in at most 3 comparisons. If z is compared against x then z assumes x's role in the hypothesis. □

We have been able to show that even more keys can be "shy". The following lemma for $f(n)$ keys, $f(n)$ an arbitrary function, is the principle result.

Lemma 5: There exists a sorting algorithm which will involve at least $f(n) > 1$ keys in $O(\log f(n))$ comparisons.

Proof: We will show that some $f(n)$ keys will be in at most $c_1 \lg f(n)$ comparisons, c_1 a constant, for the following insertion sorting algorithm. As above, the algorithm, after some preprocessing, has a sorted sublist S. On each iteration it picks a previously unused key and inserts it into S. We identify S with its total order and speak of a key being (immediatedly) above or below another. It is convenient to add the two keys ∞ and $-\infty$ to S.

First sort $2f(n)$ keys. (The case $n < 2f(n)$ follows directly from the following discussion.) Recall there exists a sorting algorithm for m keys that does not involve any key in more than $c_2 \lg m$ comparisons, c_2 a constant [AJTA83]. Using that algorithm during preprocessing we will not use more than $c_2 \lg f(n) + c_2$ comparisons per key. This provides the basis, with $c_1 > 2c_2$, for our inductive assertion: After each insertion there are $f(n)$ keys in S, each involved in at most $c_1 \lg f(n)$ comparisons and, further, they are separated in S by other keys.

The algorithm identifies these keys, $x_1 < x_2 < \cdots < x_{f(n)}$, and for each x_i it knows w_i and y_i, the keys immediatedly below and above x_i, respectively. Note that y_i may be w_{i+1}. A binary search with an unused key is conducted over the w's and y's. This requires at most $\lg f(n) + 2$ comparisons. If the final interval contains an x_i then do a final comparison with it, and the new key replaces x_i in the inductive assertion. Otherwise proceed with the insertion leaving the x's unaffected. We see the inductive assertion continues to hold. □

The next theorem follows directly from the previous result by setting $f(n) = n^{\epsilon/c_1}$, where c_1 is the constant in the above proof.

Theorem 5: There exists a sorting algorithm that will for every input involve at most $n - n^{\epsilon/c}$ keys in at least $\epsilon \lg n$ comparisons, where c is a constant and $\epsilon > 0$.

6. Known Algorithms

In this section we discuss "natural algorithms", i.e. algorithms in the literature, as opposed to those in the previous section which were designed to defeat the conjecture. All the natural algorithms we have analyzed have satisfied the conjecture. We will restrict our attention to the simplest presentation of an algorithm (e.g., in [KNUT73]) since further elaborations were not intended to defeat the conjecture.

The class of adjacent-interchange algorithms clearly satisfy the conjecture. In the same spirit, Quicksort supports the conjecture for those inputs which give rise to the $\Theta(n^2)$ worst-case performance. Mergesort is trivially seen to satisfy the conjecture, since each key cannot be avoided on each phase and there are $\Omega(\log n)$ phases.

The proofs for Heapsort and Binary Insertion Sort are less obvious. With Binary-insertion Sort it seems some keys might be ignored. A generalization of this theorem applies to the Ford-Johnson algorithm.

Theorem 6: Binary-insertion Sort satisfies the conjecture.

Proof : A basis for an induction proof is trivial. Assume after 2^k-1 keys have been inserted, those keys have been in at least $k-1$ comparisons each. Now as we insert the next 2^k elements, each time "target" one of the original 2^k-1 keys (and one twice) to be compared. Hence all the original elements will have been in k comparisons, and each of the new keys have been k comparisons too. □

Heapsort is an algorithm in which it can be hard to keep track of each key. It is important to initially arrange the keys on the tree of the heap so that each key travels a distance equal to the height of the tree cumulatively over the "heapify" and sorting stages.

Theorem 7: Heapsort satisfies the conjecture.

Proof : We sketch the proof for $n = 2^k-1$. Start with the keys in sorted order (so that, for example, the smallest is at the root and the largest are at the leaves) and then heapify. The effect is that the $2^{k/2}-1$ smallest keys have gone from the internal nodes to the leaves while the $2^{k/2}$ largest keys have moved up, one level at a time, to the internal nodes. (Actually two of the large keys did not move but their subsequent behaviors are the same as the other large keys.)

The theorem follows from the following observations: During the first $2^{k/2}$ steps, i.e. *deletemax*'s, in every case the key brought to the root percolates back to a leaf. An induction proof can be built on the fact that after these $2^{k/2}$ steps the resulting heap is identical to the original heap constructed in the case of $n = 2^{k/2}-1$, and therefore the observation above can be applied again. It follows that every key is at some point at a leaf node and does not decrease its depth without being in a comparison. The proof for other values of n is similar. □

Finally we consider the class of non-adaptive sorting algorithms and sorting networks; these satisfy the conjecture. Recall that lemma 1 states that we can force any particular key to be involved in $\Omega(\log n)$ comparisons. Since these algorithms are unresponsive they must have that property for every key. Another more complex proof using the results of Yao and Yao [YAO76] can be devised.

7. Conclusions

Theorems 2 and 5 show that there is only a small gap between the upper and lower bounds. Further, we conjecture that the constant in theorem 5 can be much smaller than the constructive proof might indicate. In fact it may be 1.

The most interesting avenue is to determine the "profiles" of sorting algorithms. The profile is the sequence of n integers: how often the most compared key is used, $\cdots$, how often the least compared key is used. What we have presented are two-step characterizations of these profiles. More complete characterizations should be investigated, possibly using theorem 3.

8. References

[AJTA83] M. Ajtai, J. Komlos and E. Szemeredi, An O(n log n) Sorting Network, *Proc 15th ACM Symp Theory of Computation*, 1983, pp. 1-9.

[ATAL81] M. Atallah and S. Kosaraju, An Adversary-Based Lower Bound for Sorting, *Info Proc Let*, **13**, 1981, pp. 55-57.

[BORO81] A. Borodin, L. J. Guibas, N. A. Lynch and A. C. Yao, Efficient Searching Using Partial Ordering, *Info Proc Let*, **12**, 1981, pp. 71-75.

[FUSS79] F. Fussenegger and H. Gabow, A Counting Approach to Lower Bounds for Selection Problems, *J ACM*, **26**, 1979, pp. 227-238.

[GRIG82] J. R. Griggs, Poset Measure and Saturated Partitions, *Studies in Applied Math*, **66**, 1982, pp. 91-93.

[KNUT68] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, 1968.

[KNUT73] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley, 1973.

[LYNC78] N. A. Lynch, Insert-Search Tradeoff, unpublished manuscript, 1978.

[PLES85] M. Pleszkoch, , personal communication, 1985.

[RICH84] D. S. Richards, Problems in Sorting and Graph Algorithms, UIUCDCS-R-84-1186, Ph.D. Thesis, University of Illinois, 1984.

[SAKS85] M. Saks, The Information Theoretic Bound for Problems on Ordered Sets and Graphs, in *Graphs and Orders*, I. Rival (ed.), Reidel, 1985, 137-168.

[WILF85] H. Wilf, Some Examples of Combinatorial Averaging, *Am Math Monthly*, **92**, 1985, pp. 250-261.

[YAO76] A. C. Yao and F. F. Yao, Lower Bounds on Merging Networks, *J ACM*, **23**, 1976, pp. 566-571.