BIT 28 (1988), 764-774

ON THE DISTRIBUTION OF COMPARISONS IN SORTING ALGORITHMS

DANA RICHARDS and PRAVIN VAIDYA*

Dept. of Computer Science, University of Virginia, Charlottesville, VA 22903, USA

Dept. of Computer Science, University of Illinois, Urbana, IL 61801, USA

Abstract.

We considered the following natural conjecture: For every sorting algorithm every key will be involved in $\Omega(\log n)$ comparisons for some input. We show that this is true for most of the keys and prove matching upper and lower bounds. Every sorting algorithm for some input will involve $n-n^{\varepsilon}/2+1$ keys in at least $\varepsilon\log_2 n$ comparisons, $\varepsilon > 0$. Further, there exists a sorting algorithm that will for every input involve at most $n-n^{\varepsilon/c}$ keys in greater than $\varepsilon\log_2 n$ comparisons, where c is a constant and $\varepsilon > 0$. The conjecture is shown to hold for "natural" algorithms from the literature.

CR categories: F.2.2.

Keywords: Sorting, lower bounds, adversaries.

1. Introduction.

It is well-known that the information-theoretic lower bound for comparison-based sorting algorithms is $\lceil\log_2 n!\rceil = \Omega(n\log n)$ 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", i.e., no key being involved in just a few comparisons. We restrict our attention to algorithms that use pair-wise comparisons.

The motivation for studying the distribution of comparisons came from applications where the cost of comparing two keys is not uniform and depends on the keys being compared. Such situations arise when the keys have widely varying lengths and also when different keys reside in different parts of memory

* Current address: AT&T Bell Laboratories, Murray Hill, NJ 07974.

Received December 1987. Revised July 1988.

with differing access times. In such cases traditional algorithms such as quicksort, heapsort, etc. may no longer be suitable because of non-uniform costs of comparisons; under such circumstances a desirable sorting algorithm would try to involve a costlier key in fewer comparisons. Furthermore, the known $\Omega(n \log n)$ lower bound on the total number of comparisons would not necessarily translate into a meaningful lower bound on the worst case cost (running time) of a sorting algorithm. This suggests that a new program of research should be initiated, for comparison-based algorithms (for sorting, selection, etc.), that isolates the comparisons made by the individual keys. An initial detailed study for small n has appeared [9]. We return to this application in the conclusions.

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

Equitable Sorting 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 results of Ajtai, Komlos, and Szemeredi [1]. 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. (Is there a simple serial sorting algorithm such that every key is always in $O(\log n)$ comparisons?)

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. Lower bounds on costs of sorting algorithms can be obtained from these partial results for various distributions of costs of key comparisons. 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. Preliminary results.

Any comparison-based sorting algorithm can be presented as a decision tree. Each input (a set of n keys) forces the computation to follow some path in the decision tree from the root to a leaf. We will refer to inputs and such paths interchangeably, depending on the context. While it may be possible for some set of keys to be avoided to some extent, it is easy to see 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. Choose any key x and assume an arbitrary total ordering for the other $n-1$ keys. Modify the decision tree by removing all unnecessary comparisons, i.e., those that are answered by the assumptions. (See [7] for examples of such modifications.) 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 $\Omega(\log n)$. Since the nodes on such a path are all on a single path in the original tree the result follows. ■

Our principal result is that most, but not all, of the keys satisfy the lower bound of the conjecture. There has been no known previous work on this type of result; most lower bounds arguments are oblivious to the identities of the keys. However, the techniques of Atallah and Kosaraju [2] can be used. (Their work was only concerned with typical lower bounds.)

THEOREM 1: *Every sorting algorithm for some inputs will involve $n-n^{\varepsilon}+1$ keys in at least $\varepsilon \log_2 n$ comparisons, $\varepsilon > 0$.*

PROOF. We will just sketch the technique; the interested reader is referred to [2]. It can be shown, for any $1 \leq k < n$, that any $n-k$ keys can be made to be involved in at least $\log_2(k+1)$ comparisons each. Note that if $k = n/g(n)-1$ we have $n-n/g(n)+1$ keys each involved in $\log_2 n - \log_2 g(n)$ comparisons. We let $g(n) = n^{1-\varepsilon}$. So $n-n^{\varepsilon}+1$ keys are each in $\varepsilon \log_2 n$ comparisons. ■

As ε decreases more keys satisfy the conjecture, albeit with respect to a smaller coefficient. Essentially the same result will be derived in the next two sections. We feel each additional proof gives further insight into the problem.

3. Poset-based adversaries.

In this section we consider adversaries that make use of the poset formed by their prior answers to comparisons. 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", in some sense. We have investigated several functions the adversary could use based on known techniques, e.g. [10] and [4]. These attempts, detailed in [8], were not successful as we found no mechanisms for isolating the direct and indirect effects on individual keys over time.

The adversary in this section uses a function on the elements of a poset. In particular for the poset P, we 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 the elements of P. The adversary maintains the condition that $h(C) \leq 1$ for every chain C in P. Griggs [5] studied such functions but did not consider dynamically changing posets. Maintaining such a function 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 $h(C) \leq 1$ 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. (For our analysis we can assume a and b are incomparable.) 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. The following results shows that if $h(C) \leq 1$ for all C in the poset before a comparison then it will be true in the poset resulting from the adversary's resolution of the comparison.

LEMMA 2: *For every sorting algorithm any comparison $a:b$ can be answered so that in the resulting poset $h(C) \leq 1$, for each chain C, provided that $h(C) \leq 1$ held for each C before the comparison.*

PROOF. The basis of an 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 that

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

However, by hypothesis

$$(U_a + h(a) + D_a) + (U_b + h(b) + D_b) \leq 2. \qquad \blacksquare$$

Clearly when sorting is completed the poset P is a total order, a single chain, and therefore $h(P) \leq 1$. Alternatively, $\sum_{x \in P} 2^{-c(x)} \leq 1$ where x was involved in $c(x)$ comparisons. This is an interesting inequality. As an aside we note it provides still another constructive adversary-based proof of the lower bound for sorting, due to the following observation.

LEMMA 3: $\sum_{x \in P} 2^{-c(x)} \leq 1$ *implies* $\sum_{x \in P} c(x) = \Omega(n \log n)$ *for the total order P.*

PROOF. A familiar result relating the geometric and arithmetic means (e.g. [11]) states

$$(y_1 y_2 \ldots y_n)^{1/n} \leq (y_1 + \ldots + y_n)/n.$$

Assume $n = 2^k$. We know

$$(2^{k-c(x_1)} + \ldots + 2^{k-c(x_n)})/n \leq 1$$

so

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

and

$$n \log_2 n \leq \sum c(x).$$

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

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

THEOREM 2: *Every sorting algorithm for some input will involve $n - n^\varepsilon/2 + 1$ keys in at least $\varepsilon \log_2 n$ comparisons, $\varepsilon > 0$.*

PROOF. Note that only one $c(x)$ could be 1 since it contributes $\frac{1}{2}$ to the sum $\sum 2^{-c(x)}$. Similarly only three $c(x)$'s could be 2 with a net contribution of $\frac{3}{4}$. Clearly the maximum number of keys with $c(x) < \varepsilon \log_2 n$ is achieved if each such key has $c(x) = \varepsilon \log_2 n - 1$. The number of such keys is bounded by $2^{\varepsilon \log_2 n - 1} - 1$ which gives the result. ■

It is easy to show that if $h(P) \le n^{-\varepsilon}$ was always true for the total order then the uniform sorting conjecture would follow. However the following result shows that this is not nearly true.

LEMMA 4: *For some sorting algorithms we can have*

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

for the total order P.

PROOF. Consider the following algorithm which begins by sorting a subset 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 $h(x)$ 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

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

We get a similar expression for n even. ■

It is possible to generalize this approach to take into account how the comparisons were resolved.[1] 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 be devised, i.e. we can maintain $h_r(C) \le 1$ for all C. However, it is not clear how to take advantage of this formulation.

[1] Personal communication from M. Pleszkoch, 1985.

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 invented independently and used to prove (unrelated) results about searching with preprocessing [3][2].

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 $1, 2, \ldots, n$, 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 ith and jth 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 ith key is less than jth key.

The adversary does not move any tokens in response to a comparison that 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 subtrees while having no tokens in its other subtree. We can now give an alternative 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 $c(i) \geq depth(n(i))$, where $c(i)$ is the number of comparisons the ith key is involed in, as before.

It is known (e.g., [6], problem 2.3.4.5-3) that if $l_1, l_2, \ldots, l_n$ are the leaves of a full binary tree then $\sum 2^{-depth(l_i)} = 1$. Therefore it follows that $\sum 2^{-c(i)} \leq 1$. 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 has a token move after

[2] This was treated in more depth in the unpublished manuscript "Insert-Search Tradeoff," by N. Lynch, 1978.

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*

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

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 involving the tokens before they 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 ith key be involved in just two comparisons, i.e. $c(i) = 2$. By the theorem $n(i)$ cannot 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, applying the theorem at the root, $c(i') + c(i) \geq$ $\geq (n-2)+2+0$, since $S(2) = 1$. ■

5. Upper bounds.

We begin by showing that there exists an algorithm that can effectively avoid using at least one of its keys in many comparisons. 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 just 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 principal 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 $c_1 \log_2 f(n)$ 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 (immediately) 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 $c_2 \log_2 m$ comparisons, c_2 a constant [1]. Using that algorithm during preprocessing we will not use more than $c_2 \log_2 f(n) + c_2$ 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 $c_1 \log_2 f(n)$ comparisons and, further, they are separated in S by other keys.

The algorithm identifies these keys, $x_1 < x_2 < \ldots < x_{f(n)}$, and for each x_i it knows w_i and y_i, the keys immediately 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 $\log_2 f(n) + 2$ 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 $f(n) = n^{\varepsilon/c_1}$, 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 $n - n^{\varepsilon/c}$ keys in more than $\varepsilon \log_2 n$ comparisons, where c is a constant and $\varepsilon > 0$.*

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 since further elaborations were not intended to defeat the conjecture. For any definitions not given here see [7].

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. Similarly for Shellsort, when 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 in 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 $2^{k-1} - 1$ smallest keys have gone from the internal nodes to the leaves while the 2^{k-1} 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-1} 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-1} steps the resulting heap is identical to the original heap constructed in the case of $n = 2^{k-1} - 1$, 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, by one level, 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.

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.

Recall that this work was motivated by considering keys with varying costs and therefore the comparisons they are involved in are not unit-cost operations. Results about the distribution of comparisons in sorting algorithms can be used to obtain lower bounds. For example, the result that every sorting algorithm for some input must involve $n - n^{\varepsilon}/2 + 1$ keys in at least $\varepsilon \log_2 n$ comparisons leads to lower bounds for various natural distributions of costs of key comparisons. In particular, if the keys are numbered 1 to n and the cost of comparing i and j is $i + j$ then we obtain an $\Omega(n^2 \log n)$ lower bound on the worst case cost of any sorting algorithm.

The results of section 5 do not immediately imply that desirable sorting algorithms exist; i.e., algorithms which reduce total cost by avoiding expensive keys. For example, if there is just one expensive key Theorem 4 does not imply that it can be in just a few comparisons; Lemma 1 actually states the opposite. It is an open question whether desirable algorithms exist. A conjectured result is that if there is a large enough subset of relatively expensive keys then some of those could be "shy" (as in Theorem 5). The best algorithm, to our knowledge, seems to be the sorting network serialization discussed above [1] since it minimizes the possible deficit from the expensive keys.

Another interesting avenue in this line of research would be to determine the "profiles" of sorting algorithms. The profile is the sequence of n integers: how often the most compared key is used, ..., 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.

REFERENCES

1. 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.
2. M. Atallah and S. Kosaraju, *An adversary-based lower bound for sorting*, Info. Proc. Let., 13, 1981, pp. 55–57.

3. 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.
4. F. Fussenegger and H. Gabow, *A counting approach to lower bounds for selection problems*, J. ACM, 26, 1979, pp. 227–238.
5. J. R. Griggs, *Poset measure and saturated partitions*, Studies in Applied Math., 66, 1982, pp. 91–93.
6. D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, 1968.
7. D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley, 1973.
8. D. S. Richards, *Problems in Sorting and Graph Algorithms*, UIUCDCS-R-84-1186, Ph.D. Thesis, University of Illinois, 1984.
9. D. S. Richards, *Sorting with expensive comparands*, International Journal of Computer Mathemathics, 16, 1984, pp. 23–45.
10. M. Saks, *The information theoretic bound for problems on ordered sets and graphs*, in *Graphs and Orders*, I. Rival (ed.), Reidel, 1985, 137–168.
11. H. Wilf, *Some examples of combinatorial averaging*, Am. Math. Monthly, 92, 1985, pp. 250–261.