Sorting and Selection with Structured Costs

Anupam Gupta*
Bell Laboratories
600 Mountain Ave.
Murray Hill NJ 07974.
anupamg@bell-labs.com

Amit Kumar†
Department of Computer Science
Cornell University
Ithaca NY 14853.
amitk@cs.cornell.edu

December 26, 2001

Abstract

The *(unit-cost) comparison tree* model has long been the basis of evaluating the performance of algorithms for fundamental problems like sorting and searching. In this model, the assumption is that elements of some total order are not given to us directly, but only through a black-box, which performs comparisons between the elements and outputs the result of the comparison.

While the comparison tree model has served us well, the recent interest in the concept of *priced information* [9, 10] encourages us to look more closely at this model. The study of the effect of priced information on basic algorithmic problems was initiated by the paper of Charikar et al. [5]. In this paper, we continue the study of sorting and selection in the priced comparison model, i.e., when comparison has an associated cost and answer some of the open problems suggested by [5]. If the comparison costs are allowed to be arbitrary, one can not get good approximation ratios. A different way to assign costs is based on the idea that one can distill out an intrinsic *value* for each item being compared such that the cost of comparing two elements is some "well-behaved" or "structured" function of their values. We feel that most practical applications will have some structured cost property.

In this paper, we study the problems of sorting and selection (which includes finding the maximum and the median) in the structured cost model. We get a variety of approximation results for these problems, depending on the restrictions we put on the structured costs. We show that it is possible to get much improved results with the structured cost model than the case when we do not have any assumptions on comparison costs.

*Part of this research was done when this author was visiting Cornell University.
†Supported in part by Lucent Bell Labs and the ONR Young Investigator Award of Jon Kleinberg.

1 Introduction

The *(unit-cost) comparison tree* model has long been the basis of evaluating the performance of algorithms for fundamental problems like sorting and searching. In this model, the assumption is that elements of some total order are not given to us directly, but only through a black-box, which performs comparisons between the elements and outputs the result of the comparison. This model has proved both very robust and interesting, and tight bounds are known on the performance of many basic problems like sorting, searching, and selection.

While the comparison tree model has served us well, the recent interest in the concept of *priced information* [9, 10] encourages us to look more closely at this model. Since many of the objects being compared are not directly available to us, the black-box comparison scheme seems very pertinent; however, the concept of having a uniform cost for all comparisons seems rather conservative.

The study of the effect of priced information on basic algorithmic problems was initiated by the paper of Charikar et al. [5]. They gave several results on the problem of evaluating AND-OR trees with the inputs having costs, and for searching a sorted list, with a price associated with each comparison. They also gave some preliminary results for finding the maximum of a total order under this priced comparison model.

In this paper, we continue the study of sorting and selection in the priced comparison model, and answer some of the open problems suggested by [5]. Since it turns out that even finding the maximum can cost as much as $\Omega(n)$ times the optimal proof when the costs are allowed to be arbitrary, the problem seems to be hopeless. However, there is more than one natural way to assign costs to the comparisons. The paper of Charikar et al [5] used the model where it was possible to give each comparison an arbitrary cost; however, as mentioned above (and noted independently by [5] and [7]), the situation seems very bleak. A different possible way to assign costs is based on the idea that one can distill out an intrinsic *value* for each item being compared such that the cost of comparing two elements is some "well-behaved" or "structured" function of their values. We feel that most practical applications will have some structured cost property.

In this paper, we concentrate on the latter structured cost model. One possible scenario where this may arise is when comparing databases, the comparison cost of two databases be just the time to scan the two databases, which is proportional to the *sum* of the sizes. Alternatively, it may involve comparing each element of one database to all the elements of the other, which would imply the comparison cost to be the product of the sizes. This are just two examples; different functions may suggest themselves based on the application in question.

Our results: In most of the following results, we compare the performance of our algorithms to the cost of the optimal solution. We denote the cost of this cheapest proof by *OPT*. E.g., when considering the problem of sorting, the cost of the the optimal solution is just the cost of the $n-1$ comparisons between the consecutive elements in the sorted order. Note that this is very similar the concept of the competitive ratio in online algorithms [4], where our (online) algorithm decides on which edges to query without knowing their outcomes, whereas the optimal solution knows the outcomes of all the comparisons and can base its proof on that knowledge. In the case of sorting, we also compare to our algorithm to all *online* algorithms.

We study the problems of sorting and selection (which includes finding the maximum and the median) in the structured cost model. For finding the maximum, we show that we can come within a factor of 2 of *OPT* for the addition function. We then extend this result to give an algorithm incurring a cost of 8 *OPT* for any *monotone* function, which is the class of functions whose values do not decrease on increasing any of the arguments.

For the problem of sorting, we give an algorithm that incurs a cost of $O(\log n)\,OPT$ for any monotone function. Note that this is the best possible (up to constants), since for sorting with unit costs, we know that any algorithm takes $\Omega(n \log n)$ comparisons, while *OPT* is $O(n)$. We then go on to show that, for the special case of addition, this algorithm comes within a constant of *any online* algorithm for sorting.

The case for median finding is more involved. For this, we give algorithms that come within a constant of *OPT* for some special functions, which include addition and multiplication.

We can also give some results for the case of arbitrary comparison costs. For finding the maximum, we give an algorithm that incurs a cost of $(n-1)\,OPT$. Furthermore, we show that any online algorithm must incur a cost of $(n-2)\,OPT$ For sorting, we give an algorithm within a factor of $\Omega(n)$ *OPT*. Finally, for the problem of merging two sorted lists, we give an algorithm that comes within $O(\log n)$ of *OPT*. This is also within constants of the best possible result, since it generalizes the problem of searching a sorted list, the priced version of which was studied by [5],

Related Work: The paper of Charikar et al. [5] mention the problems of sorting and selection in their paper; however, they look only at the arbitrary cost model. In a later version of their paper, they show an $O(n)$ approximation for finding the maximum independent of our work. Also independent of our work, this result has been improved to $(n-1)\,OPT$ and a near-matching lower bound has been given by by [7]. As far as we know, the structured cost model has not been considered prior to our work.

There have been other attempts to generalize the comparison-based model. One well-known recent example is the Nuts-and-Bolts problem studied by [2, 3, 8]. In this problem, the aim is to sort n nuts and n bolts, when comparisons can be made only between a nut and a bolt. This can be seem to be a special case of the sorting problem, where the comparison costs are 1 and ∞, and the cost-1 edges form a complete bipartite graph. Recently, Winkler and Zuckerman solved the 1-∞ case for complete multipartite graphs, and for graphs with suitable expansion properties [11]. Another line of research is in the papers of Adler et al. [1] and Feige et al. [6] studied the problem of finding the maximum element in the presence of probabilistic errors.

1.1 Notation

Let the elements be identified with V, the vertex set of a graph $G = (V, E)$. The *cost* of comparing vertices u and v is denoted by $c(u, v) = c(e)$, where e represents the edge $\{u, v\}$. Furthermore, let $K(u)$ be the *value* or *key* of a vertex u, and u *is greater than* v (or *defeats* v) if $K(u) > K(v)$. For ease of exposition, we will assume that all the values are distinct and we have a unique total order.

Clearly this can also be modeled as follows: each element u is assigned a *weight* w_u, and there is a (symmetric) function f such that $c(u, v) = f(w_u, w_v)$. In much of the following sections, we shall explore the *structured cost* model, where the function f is "well-behaved". For instance, we shall look at cases where f is the addition or the multiplication function, or when f is a monotonically increasing function.

2 Finding the Maximum

In this section, we look at algorithms to find the maximal element of the order. (I.e., an element x such that $K(x) > K(y)$ for all $y \in V$.) We compare the cost incurred by our algorithms to the cost of the optimal *proof* of optimality of such an x, which we denote by *OPT*. Note that this is just the cost of the min-cost branching rooted at x. We show that for any monotone function f, we can find the maximum with cost at most a constant factor of *OPT*. As the name suggests, a *monotone function* f has the property that its value never decreases on increasing either of its arguments.

Throughout this section, we shall assume that the vertices are numbered $1, 2, \ldots, n$, and that $w_i \le w_j$ for $1 \le i < j \le n$. To begin, let us look at the case when f is the addition function; i.e., $c(x, y) = w_x + w_y$. In this case, we can get within a factor 2 of optimal by the following algorithm:

Algorithm Sum-Max:

> **let** $m \leftarrow 1$.
> **for** $j = 2, \ldots, n$ **do**
>   **if** $(K(j) > K(m))$ **then** $m \leftarrow j$
> **Output** m.

Theorem 2.1 *The algorithm Sum-Max returns the maximal element in V, and incurs a cost at most twice OPT.*

Proof: The correctness of the algorithm is obvious. To bound the cost, note that the i-th comparison is between m and $i + 1$, where m is no greater than i. Hence the cost of the i-th comparison is at most $w_i + w_{i+1}$, using the fact that the weights are in non-decreasing order. Now summing over all $n - 1$ comparisons gives a bound of $w_1 + 2\sum_{1<i<n} w_i + w_n$. However, *OPT* must examine each element at least once, and hence must incur a cost of at least $\sum w_i$. This is at least half the algorithm's cost, which proves the result. ■

But applying this strategy to more general functions can yield poor results. Indeed, suppose f is multiplication, and consider a list of 3 elements such that $K(1) < K(2), K(2) > K(3)$ and $K(1) > K(3)$. Furthermore, let $w_1 = 1$, while $w_2 = w_3 = C \gg 1$. If the algorithm orders them as $\langle 1, 2, 3 \rangle$, it will pay $C + C^2$, while the optimum need pay only $2C$.

However, it is possible to modify the above algorithm to handle monotone functions. The essential idea of the modification is to not compare a new element j with the current candidate for the maximum immediately (since this comparison could have a very high cost), but instead to maintain a budget and compare j to the maximum element x already examined for which $c(x, j)$ is no more than the budget. By doubling the budget at every step, the algorithm ensures that its cost does not overshoot the optimal cost by more than a constant factor. We shall formally state the algorithm in the following paragraphs.

Let us assume that the values of f are powers of 2. If they are not so, we can work instead with the function $\hat{f}(x, y) = 2^{\lceil \log f(x,y) \rceil}$, which is also monotone, and the cost incurred by this new algorithm is at most twice the original cost.

As noted above, any proof showing that an element m is maximal is a branching rooted at m such that if i is the parent of j then $K(i) > K(j)$. The cost of this proof is the sum of the costs of edges in this tree. We shall, in the course of the algorithm, also build up such a *witness tree*.

An element j is considered a *local winner at* i if it is the maximum element among the elements $\{1, 2, \ldots, i\}$. We maintain the invariant that at the moment the algorithm examines element j, it has already built up the list L of the local winners at all positions $i < j$ and furthermore, it knows their sorted order. (Hence it knows the maximum element among the first $j - 1$ elements.) Also, for each element $i < j$ that is not the local winner at j, it knows a $p(i) \in L$ which defeats i; this is the parent relation defining the witness tree for the algorithm.

Algorithm Monotone-Max:

let $L = \{1\}$. (L is the list of local winners).
for $j = 2, \ldots, n$ **do**
  Suppose $L = \langle i_1, \ldots, i_r \rangle$. (Note $i_1 < i_2 < \ldots < i_r$ and $K(i_1) < K(i_2) < \ldots < K(i_r)$.)
  Initialize $l = 1$, $flag$ = false.
  repeat
    let i be the node furthest to the right in L such that $c(j, i) = 2^l$.
    if such an element i does not exist **or** $K(i) < K(j)$
      $l \leftarrow l + 1$
      if $i = i_r$ **then**
        set $p(i_r) = j$, add j to the end of L, and set $flag$ to true.
    else if $K(i) > K(j)$
      set $p(j) = i$ and set $flag$ to true.
  until $flag$ = true.
Output the last element of L.

It is not difficult to see that the last element of L is the actual maximum. The following lemma (whose proof is in the appendix) shows that the cost of this algorithm is bounded.

Theorem 2.2 *The cost of comparisons performed by* **Monotone-Max** *is at most 4 times OPT when the costs are powers of 2, and hence at most* $8OPT$ *in general.*

It remains an interesting open problem to show broader class of natural functions for which good performance guarantees can be obtained.

3 Sorting

The optimal proof in the case of sorting is a path giving the total order, and hence if the order is $\langle \pi(1), \ldots, \pi(n) \rangle$, then *OPT* is $\sum_{i=1}^{n-1} c(\pi(i), \pi(i+1))$.

Note that, in contrast to the previous section, we cannot hope to be within a constant factor of *OPT* when sorting. Indeed, if $w_u = 1/2$ for all u, then we are in the realm of traditional sorting, and sorting requires $\Theta(n \log n)$ steps; however, *OPT* is just $(n - 1)$, and thus we cannot hope for a smaller gap than $\log n$.

Our first result of this section is a very simple algorithm which is within $O(\log n)$ of *OPT* for all monotone functions. Our second result is that for the special case of addition, this algorithm actually incurs a cost which is within a constant of that incurred by *any* online algorithm. This generalizes previous results showing that any sorting algorithm requires $\Omega(n \log n)$ comparisons.

3.1 An algorithm for sorting

For the algorithm, we use the procedure of Charikar et al. [5], which performs binary search with comparison costs. This procedure locates an element in a sorted list, incurring a cost at most $2\log n$ times the optimal proof. If an element x is not present in the list, this cost is merely the cost of comparing x to its neighboring elements in the list. (For a simpler version of this procedure which is less careful with constants, see Section 5.3.)

We start with an empty list L, and consider the elements in increasing order of their weights. As before, let us assume the elements are $1, 2, \ldots, n$ in this order. When processing element j, we use the binary search procedure to locate it in the current list L, and add it in that position. The cost of inserting j is at most $(c(j, j_<) + c(j, j_>))2\log j$, where $j_<$ and $j_>$ are the neighboring elements of j when it is inserted. Note that in the entire sorted list, if $j^*_<$ and $j^*_>$ are the neighbors of j, then it must be the case that $j_< \leq j^*_<$ and $j_> \leq j^*_>$, and thus by monotonicity, the cost of inserting j is at most $(c(j, j^*_<) + c(j, j^*_>))2\log j$. But *OPT*, the cost of the optimal proof, is just $\sum_j c(j, j^*_<)$, and hence the algorithm incurs a cost which is at most $O(\log n)$ times *OPT*.

For the special case of addition, it is easy to see that the above expression for the cost is at most $8\sum_j w_j \log j$. Let us assume that the weights are powers of 2, and let t_i (respectively, $t_{(i)}$) be the elements of weight exactly (respectively, at most) 2^i, then the cost is at most $8 \cdot \sum_i 2^i t_i \log t_{(i)}$. In the next section, we will show that this cost is within a constant of the cost incurred by *any* online algorithm, thus showing that it is close to the best possible.

3.2 Lower bounds for sorting

In this section, we shall show that for the sum function (i.e., when $c(i, j) = w_i + w_j$), the algorithm in the previous section incurs a cost which is within a constant of any online algorithm. For this, we need to exhibit a lower bound on the cost incurred by any online algorithm. It is easy to see that $\max_k 2^k t_k \log t_k$ is a lower bound on the cost of sorting, where t_k is the number of elements with weight 2^k. However, this can be very far off the mark; e.g., in the case where there is one element with weight $C \gg 1$ and all the others have weight 1, the best algorithm must incur about $C\log n$ in cost, but this bound merely gives us a C.

Let us assume that the weights are powers of some number $C \geq 2$, and L_i be the set of elements with weight C^i, with $t_i = |L_i|$. Let $L_{(i)}$ be $\cup_{j\leq i} L_i$, and $t_{(i)} = |L_{(i)}|$. Consider any (deterministic) algorithm, and let us look at the comparison tree T implicit in this algorithm. Each vertex v in this tree corresponds to some partial order $P(v)$, and also to some comparison (i, j) between two vertices in that partial order. For a permutation π on $L_{(i)}$, define the tree T_π to be the subtree obtained by dropping all the vertices $v \in T$ at which the partial order $P(v)|_{L_{(i)}}$ is not consistent with π.

Lemma 3.1 *There exists a permutation σ on $L_{(0)}$ such that the length of each root-leaf path in T_σ is at least $t_0 \log t_0$.*

Proof: If not, then each tree T_σ has a path of length less than $t_0 \log t_0$, and pasting these paths together will give us a decision tree with depth less than $t_0 \log t_0$ which sorts all the t_0 elements in $T_{(0)}$, which gives the contradiction. ■

Let σ be as in Lemma 3.1. We claim that we can prune the tree T_σ so that each branch corresponds to a non $(L_{(0)}, L_{(0)})$ comparison, and can (w.l.o.g.) assume that each branch costs at least

$C+1$. Indeed, if there is a branch in T_σ that compares two elements in $L_{(0)}$, then either only one of the two outcomes can be consistent with σ and the other branch will not belong to T_σ. An exception is when the result of the comparison is already decided, but then we can just delete one of the children of that branch without affecting the results.

Now each branching point in T_σ must have cost at least $C+1$, where as the other vertices cost at least 2. Now suppose we subtract 2 from each node of T_σ, then each root-path goes down by at least $2t_0 \log t_0$ (by Lemma 3.1). However, there are at least $t_{(1)}!/t_{(0)}!$ leaves in the tree, since we have to fix the positions of the elements in $L_{(1)}$, but only conditioned on the order of the elements in $L_{(0)}$. Thus there must be at least $\log(t_{(1)}!/t_{(0)}!)$ branches, each of which (remaining) cost at least $C-1$. Thus there must be a path in T_σ that costs at least $2t_0 \log t_{(0)} + (C-1)t_1 \log t_{(1)}$.

But we can extend this proof simply to more levels. Let us assume that the permutation on $L_{(1)}$ be σ_1, and focus on T_{σ_1}. Again, there must be $t_{(2)}!/t_{(1)}!$ leaves, and hence $\log(t_{(2)}!/t_{(1)}!)$ leaves, each of which have a remaining cost of $(C^2+1)-(C+1)$, since we have subtracted a total of $2+(C-1)$ from each vertex. Proceeding on the same lines, the lower bound for weights is:

$$2t_0 \log t_{(0)} + \sum_{i=1}^{k} (C^{i-1} - C^{i-2}) t_i \log t_{(i)}. \tag{3.1}$$

Now placing $C = 2$, we get a lower bound of the same order as the cost calculation of the algorithm in the previous Section 3.1, showing that it is optimal up to constants for the case of addition.

4 Median

We now consider finding the median of a list. We obtain constant factor approximations for some special cost functions, which include addition and multiplication.

The traditional median finding algorithm which recursively partitions the list by choosing elements from the list at random does not work with costs. Indeed, suppose the median is a low weight element, but the elements near it have very high weight. It is unlikely that a random element will weed out the high weight elements close to the median, and hence will incur a high cost. On the other hand, if we just pick a low weight element at random, it is not clear if we can partition the whole list evenly.

4.1 The Addition function

In this section, let f be the addition function. Again, we assume that each w_i is a power of 2, and this only introduces a factor of 2 loss in the objective function. Let there be t_k elements L_k of weight 2^k, $k = 0, \ldots, r$. A trivial lower bound on *OPT* is $\sum_{i \in V} w_i = \sum_{k=0}^{r} 2^k t_k$

Given a list L, and two indices i, j, $i < j$, let $L(i,j)$ denote the list of elements in L which lie between the i^{th} smallest and the j^{th} smallest elements of L (including these two elements). We will give an algorithm that, given L, i, j, finds the set of elements $L(i,j)$. For ease of exposition, we shall often allow i to be less than 0, and j to be greater than $|L|$ $i \leq 0$, and we actually mean $\max\{i, j\}$ and $\min\{j, |L|\}$ respectively. We now observe the following simple fact.

Lemma 4.1 *Let L', L'' be two disjoint lists, and let $L = L' \cup L''$. If $t'' = |L''|$, then $L(i,j) \subseteq L'(i-t'', j+t'') \cup L''$.*

During the execution of our algorithm, L' will be the low weight vertices and L'' will be the high weight vertices. Using the above lemma, we will reduce the length of L' close to the length of L'', which will allow us to charge our comparisons to the weight of L''. The algorithm is given more formally below.

Let $\mathcal{A}$ be an algorithm which, given a list of n elements, finds their median using at most αn comparisons, where α is a constant. Let L_i be the list of all elements of weight 2^i. Let $m = \lfloor n/2 \rfloor$, and $m' = \lceil n/2 \rceil$. Let $L_{(k)}$ denote $L_0 \cup \cdots \cup L_k$.

Algorithm Sum-Median:

> **initialize** $L_c = L_0, i = m - (t_1 + \cdots + t_r), j = m' + (t_1 + \cdots + t_r)$
> **for** $k = 1, \ldots, r$ **do**
>   Find the elements in $L_{(k-1)}(i, j)$ by running $\mathcal{A}$ on L_c.
> $L_c = L_{(k-1)}(i, j) \cup L_k$.
> $i = i + t_k, j = j - t_k$.
> Output the element at location m in V by running $\mathcal{A}$ on L_c.

Theorem 4.2 *The algorithm* **Sum-Median** *finds the median and incurs a cost of* $O(\sum_{i \in V} w_i)$.

Proof: To prove the correctness of the algorithm, we need to explain how we can find the elements $L_{(k-1)}(i, j)$ by only looking at L_c. Clearly, this is true when $k = 1$. Now, Claim 4.1 implies that we can find the elements $L_{(k)}(i, j)$ by just looking at L_k and the elements in $L_{(k-1)}(i - t_k, j + t_k)$.

Let us compute the cost incurred by this algorithm. When the k^{th} iteration finishes, the size of L_c is $t_k + 2(t_k + \cdots + t_r)$. Hence the next iteration does at most $2\alpha\,(t_k + 2(t_k + \cdots + t_r))$ comparisons, each costing at most $2w_k$. Thus, the cost incurred in this stage is at most $12\alpha(t_k + \cdots + t_r)2^k$. Thus, the cost of this algorithm is at most

$$12\alpha \sum_{k=0}^{r} 2^k (t_k + \cdots + t_r) \leq 12\alpha \sum_{k=0}^{r} (2^0 + \cdots + 2^k) t_k \leq 24\alpha \sum_{k=0}^{r} 2^k t_k.$$

But this is within a constant of *OPT*, which proves the result. ■

4.2 Medians with Multiplication

In this section, we give a brief outline of the constant factor approximation algorithm for median finding when the function f is multiplication — this method in fact generalizes to many other functions f which include constant degree polynomials. The complete algorithm is given in the appendix.

Let L be the list of elements in V. C is a large enough constant that we decide later. L is partitioned into lists $L_0, \ldots, L_k$, list L_i containing elements of weight C^i. Let $t_i = |L_i|$.

Suppose a weight 1 element is the median of L. Then, our algorithm should be able to find it by doing comparisons of elements with weight 1 elements only — other comparisons may turn out to be too costly because the function f is multiplication. But we don't know beforehand if the median is a weight 1 element. So, we first find the two weight 1 elements closest to the median on either side. We can then recurse on the sublist between these two weight 1 elements. Cost scaling ensures that the comparison cost involving elements which occur in several of these recursive steps scales geometrically. So, it is enough to find the weight 1 elements closest to the median.

To carry out our induction argument, we need a more general procedure. We say that a weight 1 element in a list L' is at position p if it is the highest element of weight 1 in L' which has at least p elements greater than or equal to it. Given a list L' and two positive integers n_1, n_2, $n_1 \geq n_2$, we define the partition $[n_1, n_2]$ as a partition of the list L' into three sets as follows — let x and y be the weight 1 elements at positions n_1 and n_2 in L' respectively. Then, L' should be partitioned into the elements less than x, elements between x and y (including x and y) and those greater than y. By a slight abuse of notation, we refer to the list $[n_1, n_2]$ as the list of elements between x and y.

For any list L', let L'_i denote the weight C^i elements and t'_i the size of L'_i. For a list L', $L'_{(i)}$ shall denote $L'_0 \cup \cdots \cup L'_i$. Let $t'_{(i)}$ denote the quantity $t'_0 + Ct'_1 + \cdots + C^i t'_i$.

We prove by induction on k the following : given any list L, and two numbers n_1, n_2, $n_1 \geq n_2$, we can find the partition $[n_1, n_2]$ and its subsequent interval by paying at most $\alpha(t_0 + \cdots + C^k t_k) + C^{k+1}(n_1 - n_2 + 1)$, where α is a constant. Suppose this fact is true for k and we are given a list L where the highest weight of an element is C^{k+1}.

We first reduce the size of $L_{(k)}$ to make it comparable to that of L_{k+1}. For this, we must also reduce the quantity $n_1 - n_2$. This is the goal of the cutting lemma (lemma B.3). So, suppose we are in this case. But unlike the case for addition, we still cannot compare the elements of L_{k+1} with each other. The second idea is to find a new sublist, which halves the size of L_{k+1}. This is the goal of the procedure **Partition** outlined in more detail in the appendix. We pick an element x from L_{k+1} which divides it almost evenly (using randomization). We compare x with all elements in $L_{(k)}$ to find the two consecutive elements in $L_{(k)}$ between which it lies — call these x and y. Using x and y — we can partition the whole list L into three parts — those less than x, between x and y and greater than y. The key thing to note is that this procedure doesn't involve any comparison between two elements of L_{k+1}. Suppose both the weight 1 elements that we want lie in the first of these three lists (other cases turn out to be similar), L'. We now apply the whole process to L', i.e., cutting lemma followed by **Partition**. This process continues till we find the desired elements.

5 Handling Arbitrary Costs

In this section, we shall consider the case when there is no special structure on the costs. In this case, we can show that any online algorithm for finding the maximum must incur a cost of $\Omega(n)$ *OPT* in the worst case; we also present a very natural algorithm which finds the maximum while incurring a cost of $(n-1)$ *OPT*. We also give an algorithm for sorting incurring a cost of $O(n)\,OPT$, and for merging two sorted lists with cost at most $O(\log n)$ *OPT*.

5.1 Finding the maximum

Let us give a very natural and simple algorithm for finding the maximum with cost $O(n)\,OPT$. Initially, all the vertices are *winners*. Whenever we compare two winners, the element with a smaller value becomes a *loser*. The algorithm looks at the edges in order of non-decreasing costs, but it does not perform any comparisons where both the end-points are losers. It stops when only one winner remains. Let *OPT* be the minimum cost of comparisons required to prove the maximality of an element, which is the cost of a minimum weight branching rooted at the maximal element m.

Lemma 5.1 *The above algorithm for finding the maximum incurs a cost of at most* $(2n-3)$ *OPT.*

Proof: Consider the optimal branching B, in which a non-root vertex v has parent $p(v)$. If the algorithm looks at edge $e = \{u, v\}$, where both u, v are non-root vertices, we claim that the cost of e must be at most

$$\max\{c(v, p(v)), c(u, p(u))\}. \tag{5.2}$$

Indeed, suppose $c(u, v)$ were more than this value. Now our algorithm must have looked at both the edges $(v, p(v))$ and $(u, p(u))$ before looking at e. But at this time, both u and v are losers, which contradicts the fact that we never do loser-loser comparisons. We can now charge the cost of e to the vertex that achieves the maximum in (5.2), and since each vertex u can have at most $n-2$ edges charged to it (excluding edges from vertices u and m) of cost at most $c(u, p(u))$, this proves that all edges not incident to the root contribute $(n-2)$ *OPT*. Further, there are at most $(n-1)$ edges adjacent to the root, and the edges queried all have weight at most the heaviest edge in B. This, in turn, is at most *OPT*, and combining the two factors completes the proof. ■

It can be easily seen that the above analysis is tight. However, if we also keep the transitive closure of the comparisons already performed, and never perform a comparison which is in this closure, we can give an improved performance guarantee, whose proof we defer to the appendix.

Theorem 5.2 *This algorithm incurs a cost which is at most* $(n-1)$ *OPT.*

This is almost the best result we can hope for, since there are inputs on which any online algorithm must incur a cost of $(n-2)$ *OPT*. (See Theorem C.1 for this lower bound.) After this research was done, we were informed that Charikar et al. [5], and Hartline et al. [7] have obtained the results in Theorems 5.2 and C.1 independently, with an arguably more involved algorithm.

5.2 Sorting

In this section, we give an $O(n)$-approximation algorithm for sorting with arbitrary comparison costs. As the usual preprocessing step, we round all the edge costs to the nearest power of 2; this affects our final cost by at most a factor of 2. Let $K(i)$ be the key given to element i; this defines the total order.

Our algorithm is the following: From the set of all edges not lying in the transitive closure of the previously queried edges, we pick an edge of minimum cost; if there is more that one such edge, we use a tie breaking rule to be described below. This process is continued until the entire total order has been inferred.

The *tie breaking* rule is equally simple. We define a partial order over the edges of minimum cost: if $e = (u, v)$ and $e' = (u', v')$ are two minimum cost edges, then $e \prec e'$ if we can infer (based on the edges the algorithm has previously queried) that $K(u) \leq K(u')$ and $K(v) \leq K(v')$. The rule now says that we must pick a maximal element in this partial order.

The following theorem is the main result of this section, the proof of which we defer to the appendix.

Theorem 5.3 *The above algorithm incurs cost at most $O(n)$ times the cost of the optimal proof.*

The problem of sorting even when the edge costs are either 1 or ∞ seems very hard. In fact, it is equivalent (to within constant factors) to a variant of the famous "Nuts-and-Bolts" problem [2, 3, 8]. In this variant, the set of n nuts and n bolts have to be sorted by performing only nut-bolt comparisons, but now each nut can be compared to only some subset of the bolts, and vice versa. To the best of our knowledge, no algorithms are known for this problem which perform a sub-quadratic number of comparisons in general.

5.3 Merging two lists

Finally, we consider the problem of merging two sorted lists. This generalizes the problem of binary searching considered by Charikar et al. [5], and is a special case of the sorting problem. For this problem, we can give the following result, the proof of which is given in Section C.

Theorem 5.4 *There is an algorithm for merging two sorted lists which incurs a cost of at most $O(\log n)$ times OPT.*

Note that the lower bound of $O(\log n)$ for binary search implies that this result cannot be improved in the worst case.

Acknowledgments

Many thanks to Ranjit Jhala, Jon Kleinberg, Éva Tardos and Peter Winkler for helpful comments.

References

[1] Micah Adler, Peter Gemmell, Mor Harchol-Balter, Richard M. Karp, and Claire Kenyon. Selection in the presence of noise: The design of playoff systems. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 564–572, 1994.

[2] Noga Alon, Manuel Blum, Amos Fiat, Sampath Kannan, Moni Naor, and Rafail Ostrovsky. Matching nuts and bolts. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 690–696, 1994.

[3] Noga Alon, Phillip G. Bradford, and Rudolf Fleischer. Matching nuts and bolts faster. *Inform. Process. Lett.*, 59(3):123–127, 1996.

[4] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, 1998.

[5] Moses Charikar, Ronald Fagin, Venkatesan Guruswami, Jon Kleinberg, Prabhakar Raghavan, and Amit Sahai. Query strategies for priced information. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 582–591, 2000.

[6] Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, 1994.

[7] J. Hartline, E. Hong, A. Mohr, E. Rocke, and K. Yasuhara. As reported in [3].

[8] János Komlós, Yuan Ma, and Endre Szemerédi. Matching nuts and bolts in $O(n \log n)$ time. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 232–241, 1996.

[9] Web page for Pricing Economic Access to Knowledge (PEAK). `http://www.lib.umich.edu/libhome/peak/papers.html`.

[10] S. Sairamesh, C. Nikolaou, D. F. Ferguson, and Y. Yemeni. Econimic framework for pricing and charging in digital libraries. *D-Lib Magazine*, February 1996.

[11] Peter Winkler and David Zuckerman. Personal Communication.

A Proofs from Section 2

Proof of Theorem 2.2: Let T and T^* denote the witness trees produced by our algorithm and by the optimum strategy respectively. Consider a node i which is not the maximum, and let j and j^* be the parents of i in T and T^* respectively. We will show that the cost of processing i will be at most $2c(i, j^*)$. Let us look at two cases:

- $j^* < i$: In this case, j^* has already been processed when i is considered. Note that either j^* is a local winner (and hence in L) or it is a child of a local winner — let x denote the local winner in either case. Hence, $K(x) \geq K(j^*) > K(i)$. Let $c(x, i) = 2^r$, and x' be the node with the largest value in L at this point with $c(x', i) = 2^r$; clearly $k(x') \geq k(x)$. Since the algorithm works by doubling the value of l at each step, the cost of processing i is at most $2 + \cdots + 2^r \leq 2^{r+1} = 2c(x, i)$.

  If $x = j^*$ then we are done. If not, then j^* is not a local winner. But then we claim that x was considered before j^*; if not, the only other way that j^* can be a child of x is when j^* is a local winner. Thus $w_x \leq w_{j^*}$, which implies that $c(i, x) \leq c(i, j^*)$, and the cost is thus at most $2c(i, j^*)$.

- $j^* > i$: Consider the tree T after i gets processed. Suppose i is not the root, and is defeated by j. This implies that $j < j^*$, and hence $w_j \leq w_{j^*}$ and $c(i, j) \leq c(i, j^*)$. Arguing as above, we can show that the cost of inserting i is at most $2c(i, j)$. On the other hand, suppose that i is the root of T at this time. If i' is the (unique) child of i at this point; we can now show that the cost of processing i is at most $2f(i, i') \leq 2f(i, j^*)$.

Finally, we have to account for the cost of processing the maximum element m. Clearly, m will be the root of the witness tree when it is processed, and the cost can be bounded by $2c(m, m')$, where m' is the unique child of m at this point. Let us consider the path from m to m' in T^*, and let m'' be the closest node to m in this path. If $m'' = m'$, we are done. If not, then $K(m'') \geq K(m')$. Suppose $w_{m''} < w_{m'}$, then m'' was considered before m', and hence m'' was already in T when m was being inserted. But then, by our tie-breaking rule, m'' should have been the root of T, and hence m' could not be the child of m. Thus, $w_{m''} \geq w_{m'}$ and hence, $c(m, m') \leq c(m, m'')$.

Now the cost of the algorithm is just the cost of processing all its elements. But we have shown that this cost is at most 4 times the cost of T^*, and hence the theorem follows. ■

B Proofs of Section 4

B.1 Details of finding medians with multiplication

In this section, we give a constant factor approximation when the function f is multiplication. This method in fact generalizes to many other functions f which include constant degree polynomials.

Let L be the list of elements in V, whose weights are powers of C, which is a large enough constant we specify later. L is partitioned into lists $L_0, \ldots, L_k$, with list L_i containing elements of weight C^i. Let $t_i = |L_i|$.

The basic idea is the following: Suppose the median of L is a weight 1 element. Then our algorithm should be able to find it by comparing elements only with weight 1 elements, since any other comparisons may turn out to be too costly (because f is multiplication). Of course, we do not know if the median is a weight 1 element, and hence our procedure finds the weight 1 elements closest to the median on either side. We then apply the same procedure recursively to the set of higher weight elements between these two elements. Cost scaling ensures we do not pay too much in this recursive process.

We say that a weight 1 element in a list L' is at *position p* if it is the maximum weight 1 element in L' which has at least p elements greater than or equal to it. For simplicity of notation, if all weight 1 elements have less than p elements above it, define p to be the position of the smallest weight 1 element in L'. Further, if $p < 1$, then we assume $p = 1$. Given a list L' and two positive integers n_1, n_2 with $n_1 \geq n_2$, we define the partition $[n_1, n_2]$ as a partition of the list L' into three sets as follows — let x and y be the weight 1 elements at positions n_1 and n_2 in L' respectively. Then, L' should be partitioned into the elements less than x, elements between x and y (including x and y) and those greater than y. By a slight abuse of notation, we refer to the list $[n_1, n_2]$ as the list of elements between x and y. Note that the list $[n_1, n_2]$ can contain more than $n_1 - n_2 + 1$ elements, but it contains at most $n_1 - n_2 + 1$ weight 1 elements. Define the *subsequent interval* of $[n_1, n_2]$, $sub[n_1, n_2]$ as follows : if the list $[n_1, n_2]$ is singleton, then $sub[n_1, n_2]$ is the empty set. Otherwise, let x and y be the elements at position n_1 and n_2 respectively. Let x' be the weight 1 element succeeding x in the (ascending) sorted order. Define $sub[n_1, n_2]$ as the set of elements between x' and y. The reason why we need the list $sub[n_1, n_2]$ is that the size of this list is always at most $n_1 - n_2$.

We prove by induction on k the following : given any list L, and two numbers n_1, n_2, $n_1 \geq n_2$, we can find the partition $[n_1, n_2]$ and its subsequent interval by paying at most $\alpha(t_0 + \cdots + C^k t_k) + C^{k+1}(n_1 - n_2 + 1)$, where α is a constant. Suppose this fact is true for k and we are given a list L where the highest weight of an element is C^{k+1}.

Let us give some intuition for our proof. Using techniques similar to Lemma 4.1, we first reduce the size of list $L_{(k)}$ to make it comparable to t_{k+1} — this implies that we must also reduce the quantity $n_1 - n_2$. This is the goal of lemma B.3. So, suppose we are in this case. Unlike addition, we are not done. We still cannot compare elements in L_{k+1} with each other. The second idea is to choose an element from from L_{k+1} which partitions it almost evenly (using randomization) — this allows us to partition the whole list into two almost even pieces and then we can recurse on the appropriate part.

In our algorithm, we shall reduce the list L by removing certain elements at various stages. But we shall always maintain the following invariant — if a weight 1 element x has not been deleted from L, then we know which deleted elements are greater than (and so, which are less than) x.

This shall help us in the following way — suppose we find the required partition in a sublist L' of L, i.e., the partition $[n_1, n_2]$ of L restricted to L', and suppose the two weight 1 elements at position n_1 and n_2 in L are also in L'. Then we can actually recover the desired partition from this.

For any list L', let L'_i denote the weight C^i elements and t'_i the size of L'_i. For a list L', $L'_{(i)}$ shall denote $L'_0 \cup \cdots \cup L'_i$. Let $t'_{(i)}$ denote the quantity $t'_0 + Ct'_1 + \cdots + C^i t'_i$. We now worry about finding the partition $[n_1, n_2]$ only and shall show how to find $sub[n_1, n_2]$ later. We begin with two simple lemmas, whose proofs are in Appendix B.

Lemma B.1 Outer Cutting Lemma *Suppose we are given a list L' and two positive integers n_1, n_2, $n_1 \geq n_2$. We can reduce the problem of finding the partition $[n_1, n_2]$ in L' to finding a partition $[n'_1, n'_2]$ in a sublist L'' of L' such that $n'_1 - n'_2 \leq n_1 - n_2$ and $t''_0 + \cdots + t''_k \leq n_1 - n_2 + t'_{k+1} + 1$. This reduction requires at most $\alpha t'_{(k)} + (n_1 - n_2 + 1)C^{k+1} + t'_{k+1}C^{k+1}$ comparison cost.*

Proof of Theorem B.1: By induction, find the partition $[n_1, n_2 - t'_{k+1}]$ and its subsequent interval in $L'_{(k)}$. Let m be the number of elements in the third list of this partition. Let I be the list $[n_1, n_2 - t'_{k+1}]$. It is easy to see that all elements in $L'_{(k)}$ which are also in the list $[n_1, n_2]$ in L' lie in I. So, we can throw away all other elements of $L'_{(k)}$. If the subsequent interval is empty, we are done.

Let x be the element at position n_1 in $L'_{(k)}$ and x' be the weight 1 element succeeding it in this list. By induction, we know both x, x' and the set of elements in $L'_{(k)}$ lying strictly between these two elements, call it S. Let $m' = |S|$. Define $L'' = L'_{k+1} \cup (I - S)$. Note that $I - S$ has at most $n_1 - n_2 + t'_{k+1} + 1$ elements.

Compare x with all elements of L'_{k+1}. After this, we know the exact position of x in L'_{k+1}, and so, we know if it as at position n_1 in L' or not. Suppose x is at position n_1 in L'. Define $n'_1 = n_1 - m' - m$. Otherwise, delete x from L'' and define $n'_1 = n_1 - m$. Define $n'_2 = n_2 - m$. It is not difficult to show that we have reduced the problem to that of finding the interval $[n'_1, n'_2]$ in L''.

Also, we maintained the invariant whenever we deleted any element from the list L'. ■

Lemma B.2 Inner Cutting Lemma *Suppose we are given a list L' and two positive integers n_1, n_2, $n_1 \geq n_2$. We can reduce the problem of finding the partition $[n_1, n_2]$ in L' to finding a partition $[n'_1, n'_2]$ in a sublist L'' of L' such that $n'_1 - n'_2 \leq t'_{k+1} + 1$. The comparison cost in this reduction is at most $(\alpha + 1)t'_{(k)} + (n_1 - n_2 + 1)C^{k+1} + C^{k+1}t_{k+1}$.*

Proof of Theorem B.2: Assume $n_1 - n_2 > t'_{k+1} + 1, n_2 > 0$, otherwise we are done. Let x_i be the weight 1 element at position n_i in L'. Find the partition $[n_1 - t'_{k+1}, n_2]$ in $L'_{(k)}$, let y_i be the two weight 1 elements which define this partition, $y_1 \leq y_2$. Let I denote the middle list in this partition. Let I' be the interval I with y_1 and y_2 deleted from it. Define $m = |I'|$. It is easy to observe that $x_i \notin I'$. Following cases can happen (note that given the partition, we can easily find which case has happened) :

- Both y_1 and y_2 do not have at least n_2 elements (in $L'_{(k)}$) above it : in this case, $y_1 = y_2$ is the smallest weight 1 element. Further, $x_1 = y_1$, though x_2 may not equal y_2. Define n'_1 to be the number of elements in L' which are at least x_1. Then, it follows that $n'_1 \leq n_2 + t'_{k+1}$. So, the problem reduces to finding the partition $[n_2 + t'_{k+1}, n_2]$ in L'.

- y_1 does not have more than $n_1 - t'_{k+1}$ elements above it in $L'_{(k)}$, but there are at least n_2 elements in $L'_{(k)}$ above y_2 : again, $x_1 = y_1$ is the smallest weight 1 element. Find the weight 1 element succeeding y_2, call it y_3. Compare all elements in $L'_{(k)}$ with y_3 to find out the elements in $L'_{(k)}$ which (strictly) lie between y_2 and y_3 — call this S, and let $m' = |S|$ (in case y_2 is the largest weight 1 element, S is just the set of all elements in $L'_{(k)}$ greater than y_2). Let $L'' = L' - (S \cup I')$.

  Compare y_3 with all elements in L'_{k+1} (we already know how y_3 compares with other elements of L') — this will tell us if x_2 is greater than or equal to y_3, i.e., if $x_2 = y_2$ or not. If $x_2 = y_2$, we are done. Note that there are at most $n_2 + t'_{k+1} + 1$ elements of L'' which are at least y_1. So, the problem reduces to finding the partition $[n_2 + t'_{k+1} + 1, n_2]$ in L''.

- There are at least $n_1 - t'_{k+1}$ and n_2 elements in $L'_{(k)}$ above y_1 and y_2 respectively : define S, m' as in the case above. We can argue as above that $|I' \cup S| = m + m' \geq n_1 - t'_{k+1} - n_2 - 1$. Again, compare y_3 with all elements of L'_{k+1} to decide if $x_2 = y_2$. Suppose $x_2 = y_2$. Let S' be the elements in $L'_{(k)}$ greater than y_2. Let $m'' = |S'|$. Clearly, $m + m'' \geq n_1 - t'_{k+1} - 2$. Define $L'' = L' - S'$, $n'_1 = n_1 - (m + m'') \leq t'_{k+1} + 2, n'_2 = 1$. Otherwise, define $L'' = L' - (S \cup I)$, $n'_1 = n_1 - (m + m') \leq n_2 + t'_{k+1} + 1, n'_2 = n_2$.

Again, it is easy to check that we maintain the invariant. ■

Combining the two lemmas above, we get the main cutting lemma:

Lemma B.3 Cutting Lemma *Suppose we are given a list L' and two positive integers n_1, n_2, $n_1 \geq n_2$. We can reduce the problem of finding the interval $[n_1, n_2]$ in L' to finding an interval $[n'_1, n'_2]$ in a sublist L'' of L' such that $n'_1 - n'_2 \leq t'_{k+1} + 1$ and $t''_0 + \cdots + t''_k \leq (n_1 - n_2 + 1) + t'_{k+1}$. Further, this can be achieved by at most $\alpha t'_{(k)} + 2(n_1 - n_2 + 1)C^{k+1} + 3C^{k+1}t'_{k+1}$ comparison cost, provided $C \geq \alpha + 1$.*

Start with the input list L. Apply the cutting lemma to L, n_1, n_2, to get a new list $L^{(1)}$ and interval $[n_1^{(1)}, n_2^{(1)}]$. We now outline a procedure **Partition(L')** which takes a list and outputs two disjoint sublists of L'.

Algorithm Partition(L')

repeat
    Pick element $x \in L'_{k+1}$ uniformly at random.
    Find two consecutive elements of L'_0 between which x lies.
        Let these two elements be y and z.
    Partition L'_{k+1} into three lists M_1, M_2, M_3.
        M_1 is the set of elements $\leq y$, M_2 those elements between y and z, and M_3 those $\geq y$.
    Let N_1 be elements of $L'_{(k)}$ which are $\leq y$, and N_2 the elements of $L'_{(k)}$ which are $\geq z$.
    Define $L'' = N_1 \cup M_1, L''' = N_2 \cup M_3$.
until $(|M_1| \leq 3|L'_{k+1}|/4$ **and** $|M_3| \leq 3|L'_{k+1}|/4)$
Output L'' and L'''.

Lemma B.4 *The expected cost of comparisons made by the above procedure is at most $(t'_0 + 2t'_{k+1})C^{k+1} + 2t'_{(k)}$.*

Proof: The expected number of iterations, i.e., the expected number of steps until x lies in the middle half of L'_{k+1}, is two. In each iteration, the operations on L'_0 and $L'_{(k)}$ cost $t'_0 C^{k+1} + 2t'_{(k)}$, while the partitioning of L'_{k+1} costs at most $2t'_{k+1}C^{k+1}$. ■

We apply **Partition(**$L^{(1)}$**)** to get lists $L^{(1)'}$ and $L^{(1)''}$. Suppose the desired interval lies entirely in one of these two lists, call it $\tilde{L}^{(2)}$. Let the interval in this list be $[\tilde{n}_1^{(2)}, \tilde{n}_2^{(2)}]$. We now apply the cutting lemma to $\tilde{L}^{(2)}$ and $[\tilde{n}_1^{(2)}, \tilde{n}_2^{(2)}]$ to get $L^{(2)}$, $[n_1^{(2)}, n_2^{(2)}]$. The procedure **Partition** is again applied to this, and the process repeated till we get a list $L^{(r)}, [n_1^{(r)}, n_2^{(r)}]$ for which **Partition(**$L^{(r)}$**)** yields two lists $L^{(r)'}$ and $L^{(r)''}$, such that each of these two lists contain one of the two weight 1 elements that we want. After this, we shall maintain two lists instead of just one.

We now apply **Partition** to both $L^{(r)'}$ and $L^{(r)''}$ to get lists $L^{(r)'}_{(1)}, L^{(r)'}_{(2)}$ and $L^{(r)''}_{(1)}, L^{(r)''}_{(2)}$ respectively. One of the weight 1 elements will be in one of the two lists $L^{(r)'}_{(1)}, L^{(r)'}_{(2)}$, call it $\tilde{L}^{(r+1)'}$. Similarly, the other weight 1 element lies in one of the two lists $L^{(r)''}_{(1)}, L^{(r)''}_{(2)}$, call it $\tilde{L}^{(r+1)''}$. Define Define $\tilde{L}^{(r+1)} = \tilde{L}^{(r+1)'} \cup \tilde{L}^{(r+1)''}$. So, our problem basically reduces to finding an interval $[\tilde{n}_1^{(r+1)}, \tilde{n}_2^{(r+1)}]$ in $\tilde{L}^{(r+1)}$. We apply the cutting lemma to $\tilde{L}^{(r+1)}$, $[\tilde{n}_1^{(r+1)}, \tilde{n}_2^{(r+1)}]$ to get a list $L^{(r+1)}$, $[n_1^{(r+1)}, n_2^{(r+1)}]$. Define $L^{(r+1)'}$ and $L^{(r+1)''}$ as $L^{(r+1)} \cap \tilde{L}^{(r+1)'}$ and $L^{(r+1)} \cap \tilde{L}^{(r+1)''}$ respectively.

Now, we apply the same process to the lists $L^{(r+1)'}$ and $L^{(r+1)''}$, and so on. If we compare the situations when $i \leq r$ and $i > r$, the only difference (in terms of number of comparisons) is that in the latter stage, we need to call the procedure **Partition** twice. But since **Partition** is being applied to two disjoint pieces of the list $L^{(r+1)}$, the expected cost is the same as that of applying **Partition** to the whole list $L^{(r+1)}$ once. So, we can assume without loss of generality that $r = 1$.

Let us now compute the total cost of comparisons. Observe that $t_0^{(1)} \leq (n_1 - n_2 + 1) + t_{k+1}$, $t_{(k)}^{(1)} \leq C^k(n_1 - n_2 + 1) + C^k t_{k+1}$. So, **Partition**$(L^{(1)})$ costs at most $5C^{k+1}t_{k+1} + (2C^k + C^{k+1})(n_1 - n_2 + 1)$.

Claim B.5 *Let* $i \geq 1$. $t_{k+1}^{(i)} \leq \left(\frac{3}{4}\right)^{i-1} t_{k+1}, n_1^{(i)} - n_2^{(i)} \leq \left(\frac{3}{4}\right)^{i-1} t_{k+1} + 1, t_0^{(i+1)} + \cdots + t_k^{(i+1)} \leq 2\left(\frac{3}{4}\right)^{i} t_{k+1} + 1.$

Proof: At the i^{th} step, we have lists $L^{(i)}, L^{(i)'}$ and $L^{(i)''}$. After applying **Partition** to the latter two lists, we get new lists which combine to give a list $\tilde{L}^{(i+1)}$. The definition of **Partition** implies that $\tilde{t}_{k+1}^{(i+1)} \leq 3/4 t_{k+1}^{(i)}$. Clearly, $\tilde{n}_1^{(i+1)} - \tilde{n}_2^{(i+1)} \leq n_1^{(i)} - n_2^{(i)}$ and $\tilde{t}_j^{(i+1)} \leq t_j^{(i)}$ for any $j \leq k$.

Now, we apply the cutting lemma to $\tilde{L}^{(i+1)}$ to get the list $L^{(i+1)}$. So $t_{k+1}^{(i+1)} \leq \tilde{t}_{k+1}^{(i+1)} \leq 3/4 t_{k+1}^{(i)}$. Furthermore, $t_0^{(i+1)} + \cdots + t_k^{(i+1)} \leq (\tilde{n}_1^{(i+1)} - \tilde{n}_2^{(i+1)} + 1) + \tilde{t}_{k+1}^{(i+1)} \leq t_{k+1}^{(i+1)} + n_1^{(i)} - n_2^{(i)} + 1$, and $n_1^{(i+1)} - n_2^{(i+1)} \leq \tilde{t}_{k+1}^{(i+1)} + 1$. This implies the claim. ■

Let us now compute the total cost at step i. **Partition** and the cutting lemma together cost at most $(\alpha + 2)t_{(k)}^{(i)} + 2(n_1^{(i)} - n_2^{(i)})C^{k+1} + 5C^{k+1}t_{k+1}^{(i)} + t_0^{(i)}C^{k+1}$. Adding this over all values of i, (note that the number of steps is at most t_{k+1}) and using the above lemma, we get the total cost over all the steps is at most $(15C^{k+1} + 9(\alpha+2)C^k)t_{k+1} + (C^{k+1} + (\alpha+2)C^k)(n_1 - n_2 + 1)$. Thus, the total comparison cost to find the partition $[n_1, n_2]$ is at most $\alpha t_{(k)} + (4C^{k+1} + (\alpha+4)C^k)(n_1 - n_2 + 1) + (23C^{k+1} + 9(\alpha+2)C^k)t_{k+1}$.

We have yet to show how to find the subsequent interval. Let x and y be the weight 1 elements at position n_1 and n_2 in the list L respectively. Note that we first apply the outer cutting lemma to the input list L. Let L'' denote the list output by the outer cutting lemma. We have shown that x and y must lie in L''. Moreover, L'' contains all elements between x and y. We find the weight 1 element z succeeding x in L'' — this costs at most t''_0. Now, we compare all elements in L'' with z — this costs at most $Ct''_1 + \ldots + C^{k+1}t''_{k+1}$. Thus, the total cost is at most $(t''_0 + \cdots + t''_k)C^k + C^{k+1}t_{k+1} \leq (n_1 - n_2 + 1)C^k + (C^k + C^{k+1})t_{k+1}$.

Thus, the total cost is at most $\alpha t_{(k)} + (4C^{k+1} + (\alpha+5)C^k)(n_1 - n_2 + 1) + (24C^{k+1} + 9(\alpha+3)C^k)t_{k+1}$. Choose $\alpha = 25$, $C = 9(\alpha+3)$. Then, we see that this cost is at most $\alpha t_{(k+1)} + C^{k+2}(n_1 - n_2 + 1)$. In the special case, $n_1 = n_2$, we see that there is a constant α' ($\alpha' < \alpha + C$) such that we can find the partition $[n_1, n_1]$ by at most $\alpha' t_{(k+1)}$ comparison cost.

Let x^* be the median in the list L and C^k be the highest weight of any element in L. Let $L^{(0)}$ denote the original list. Let $x^{(i)}$ and $y^{(i)}$ be the weight C^i elements which are closest to x^* on the left and on the right respectively. Define $L^{(i)}$ as the list of elements lying (strictly) between $x^{(i-1)}$ and $y^{(i-1)}$ — note that $L^{(i)}$ does not contain any element of weight C^{i-1} or less. It is easy to see that optimum cost of any proof that shows that x^* is the median is at least (let $L^{(k+1)}$ be the empty list)

$$\sum_{i=0}^{k} \sum_{x \in L^{(i)} - L^{(i+1)}} C^i c(x), \text{ where } c(x) \text{ denotes the weight of } x$$

$$= \sum_{i=0}^{k} C^i \left(\sum_{j=i}^{k} C^j (t_j^{(i)} - t_j^{(i+1)}) \right) \geq 1/2 \sum_{i=0}^{k} \sum_{j=i}^{k} C^{i+j} t_j^{(i)} = 1/2 \sum_{i=0}^{k} C^i t_k^{(i)}$$

We start with the list $L^{(0)}$. We maintain the following invariant when we process a list $L^{(i)}$: we always know at exactly what position in the sorted order the median x^* lies in $L^{(i)}$. When we process the list $L^{(i)}$ we can use exactly the same algorithm described above with the additional modification that now weight 1 elements get replaced by weight C^i elements. So, if x^* lies at position r_i (in the sorted order) in $L^{(i)}$, then we can find the partition $[r_i, r_i]$ in $L^{(i)}$ by paying at most $\alpha' C^i t_k^{(i)}$ comparison cost. This will give us the element $x^{(i)}$ (as defined above). Similarly, we can get the element $y^{(i)}$. Note that by definition of partition, we also get the list $L^{(i+1)}$ and we know the position of x^* in this list. So, we can recurse on the list $L^{(i+1)}$.

Thus, we can find the median x^* by incurring at most $2\alpha' \sum_{i=0}^{k} C^i t_k^{(i)}$ comparison cost, which is within a constant factor of the optimum cost.

C Proofs of Section 5

Theorem C.1 *There are inputs on which any deterministic strategy (or even randomized strategies against oblivious adversaries) incurs cost at least $\Omega(n)$ OPT.*

Proof: Let the element 1 (resp. 2) be connected to $S = \{3, 4, \ldots, n\}$ by edges of cost 0 (resp. 1). Let there also be an edge of cost n between 1 and 2. Lastly, let 1 defeat all elements in S. Now if any algorithm queries $(1, 2)$ before querying all edges between S and 2, then the adversary can make one of the remaining edges between S and 2 defeat 2, thus making *OPT* = 1. Else, we have to query all the edges between S and 2, which will cost $n - 2$ as well.

Showing this for randomized strategies against oblivious adversaries is equivalent to finding an element in an unordered list of length $(n-2)$, which has a lower bound of $(n-2)/2$. ■

Proof of Theorem 5.2: Let B be the optimal branching that proves that m is the maximum. We start off by the simple observation that the algorithm queries all the edges in B. To see this, note that each edge $e = (u, v) \in B$ is the cheapest edge that defeats v, else we can replace e by such a cheaper edge and reduce the cost of the branching. Hence v will be a winner at the time e is examined, which also means there cannot be a set of implications showing that u defeats v.

Now let us modify the argument above, and separate E', the edges queried into two parts; those which go between elements related in B (denoted by E'_1) and those which do not (denoted by E'_2). Note that the edges in E'_2 go between two eventual losers, and hence they can be charged to edges in B in exactly the same way as above.

If an edge $e = (u, v) \in E'_1$ is examined, it must be the case that there is some highest edge $(u', v') \in B$ on the directed path between u and v in B that has not been yet been examined, and hence $c_{u',v'} > c_{u,v}$. We will now charge e to $(u', v') \in B$. It now remains to show that for any edge in B, at most $n-1$ edges are charged to it.

To see this, let us look at an edge $e = (p(u), u) \in B$. Let k be the number of descendents of u in B (including u itself). It is clear any edge in E'_2 which is charged to e must be incident to u, and must go to an element not related to u. There can be at most $n-1-k$ elements. We now claim that at most k edges in E'_1 can be charged to e, which will complete the proof of the theorem.

To prove this, let us prove the fact that if edges $(x, y), (x', y) \in E'_1$ are queried by the algorithm and charged to e, then $x = x'$; this will show that there is at most one edge per descendent y of u in E'_1 that is charged to e, which will prove the claim.

Indeed, let x be an ancestor of x' in B. Then walking from x to y in B, we shall encounter $x', p(u), u$ in that order. Note that since we charged e for the (x, y) comparison, it must be that we must have already examined the path from x to x' (and in fact, to $p(u)$) at the time we examine (x, y). Suppose $c_{x,y} > c_{x',y}$. Since (x', y) must have been examined when we consider (x, y), we can infer (x, y) by transitivity and will not examine it. In the other case, when $c_{x,y} < c_{x',y}$, both x' and y must be losers (due to the path from x to x', and the edge (x, y) respectively) when $(x'y)$ is considered, and will not be examined. This completes the proof. ■

Proof of Theorem 5.3: Let us assume, for the sake of the proof, that the vertices are $\{1, 2, \dots, n\}$, and that $K(1) < K(2) < \cdots < K(n)$. Let edge e_i be $(i, i+1)$, and P be the path $\{e_i\}_{i=1}^{n-1}$. The cost of the optimal proof is simply $\sum_{i=1}^{n-1} c(e_i)$.

As before, we shall charge each edge the algorithm examines to some edge $e_i \in P$. Suppose it looks at edge $e = (i, j)$, where $i < j$. Let P_{ij} denote the portion of P between i and j. We claim that at least one of the edges in P_{ij} has cost at least $c(e)$. To see this, let us assume to the contrary. Since all these edges have cost less than $c(e)$, they must have been considered before e; and further, they must have been queried by the algorithm, since the outcome of e_k cannot be inferred by the outcomes of querying edges other than e_k. Thus all edges in P_{ij} must have been queried, and hence e would be implied by transitivity; this gives the desired contradiction. Thus one of the edges in P_{ij} must have cost at least $c(e)$; we charge the cost of edge e to such an edge which is closest to i.

It now suffices to show that, for any edge $e_k \in P$, the amount charged to it is at most $2nc(e_k)$. It is clear that an edge (i, j) charged to the edge e_k must have $i \le k < k+1 \le j$. Fixing a j, we now show that the total cost of edges (i, j) charged to e_k is at most $2\,c(e_k)$. Let these edges be

$\{(i_1, j), (i_2, j), \ldots, (i_r, j)\}$, where $i_r < \cdots < i_2 < i_1$, and f_l denote the edge (i_l, j). We claim that $c(f_1) > c(f_2)$. Indeed, consider the portion P_{i_2,i_1} between i_2 and i_1; any edge in P_{i_2,i_1} must have been queried before f_2, else f_2 would have been charged to such an edge. But now the algorithm must have queried f_1 after f_2, else f_2 could be inferred by transitivity, and hence $c(f_1) \geq c(f_2)$. To show that equality cannot hold, note that if $c(f_1)$ were equal to $c(f_2)$, our tie-breaking rule would have picked f_1 before f_2. Proceeding in this manner, we can prove that $c(f_r) < \cdots < c(f_2) < c(f_1)$. However, since the costs are powers of 2, these costs add up to at most $2\,c(f_1) \leq 2\,c(e_k)$. ■

Proof of Theorem 5.4: We give an $O(\log n)$ approximation algorithm for merging of two sorted lists. Let the list (sorted in ascending order) be $X = (x_1, \ldots, x_n), Y = (y_1, \ldots, y_m)$. We introduce four artificial elements, two of them x_0, y_0 of value smaller than any other value in X and Y, with $K(x_0) < K(y_0)$, and x_{n+1}, y_{m+1} with values greater any other element with $K(y_{m+1}) > K(x_{n+1})$. The comparison cost of these new elements with any element is 0. As always, we assume that all costs are powers of 2.

We can write the merged list as $(x_0, M_1, N_1, \ldots, M_s, N_s, y_{m+1})$, where L_i is a sequence of elements from Y, and N_i a sequence of elements from X. Note that L_1 begins with y_0 and M_s end with x_{n+1}. Let $L_i = (y_{l_i}, \ldots, y_{l_{i+1}} - 1)$ and $M_i = (x_{m_i}, \ldots, x_{m_{i+1}} - 1)$. Hence *OPT* is $\sum_{i=1}^{s} \big(c(x_{n_i-1}), y_{m_i}) + c(y_{m_{i+1}-1}, x_{n_i})\big)$.

Let us suppose we know x_{n_i} and x_{n_i-1}, i.e., the first element of N_i, and the last element of N_{i-1}. Furthermore, suppose we are given an element $y \in M_i$. We now show how to find $y_{m_{i+1}-1}$, the last element of M_i, while paying at most $O(c(y_{m_{i+1}-1}, x_{n_i}) + c(y_{m_{i+1}}, x_{n_{i+1}-1})$. This algorithm is as follows :

Procedure:

> **let** $y_c \leftarrow y$.
> **for** $K = 1, 2, 4, \ldots$ **do**
>   let Y' be elements of Y greater than y_c whose comparison cost with x_{n_i} is K.
>   locate x_{n_i} in Y' by binary search.
>   Let the element in Y' immediately smaller than x_{n_i} be y'.
>   (in case x_{n_i} is smaller than all elements in Y', $y' = y_c$.)
>   Update y_c to y'. Let y'' be the element in Y following y'.
>   let X' be elements in X greater than x_{n_i} whose comparison cost with y'' is at most K.
>   let x' be the smallest element in X'.
>   Compare x' with y''. If $K(x') < K(y'')$, then y_c is the desired element.
> **endfor**

It is not difficult to show that this algorithm has the desired properties. Now we can apply the same procedure with the roles of x and y reversed.

Thus the merging procedure is as follows: we know that y_0 lies between x_0 and x_1. Now using the above procedure, we can find y_{n_2-1}. Now we know M_1, since we know the last element of M_1 and the first element of M_2. Furthermore, we know the first element of N_1, which is x_1, and so we can apply the same procedure again. It is not difficult to show that the total cost is at most a constant times *OPT*. ■