Selection with Monotone Comparison Costs

Sampath Kannan* Sanjeev Khanna†

Abstract

We consider the problem of selecting the r^{th}-smallest element from a list of n elements under a model where the comparisons may have different costs depending on the elements being compared. This model was introduced by [3] and is realistic in the context of comparisons between complex objects. An important special case of this general cost model is one where the comparison costs are monotone in the sizes of the elements being compared. This monotone cost model covers most "natural" cost models that arise and the selection problem turns out to be the most challenging one among the usual problems for comparison-based algorithms. We present an $O(\log^2 n)$-competitive algorithm for selection under the monotone cost model. This is in contrast to an $\Omega(n)$ lower bound that is known for arbitrary comparison costs. We also consider selection under a special case of monotone costs — the `min` model where the cost of comparing two elements is the minimum of the sizes. We give a randomized $O(1)$-competitive algorithm for the `min` model.

1 Introduction

Charikar *et al* [3] introduced the problem of computing a function at optimal cost when each input to the function has an associated *price*. They use the framework of competitive analysis, comparing the cost incurred by their algorithms to compute the function value to the cost of an optimal *proof* that the function has that value. Under this measure, they provide optimal algorithms for computing arbitrary Boolean AND/OR trees and for the problem of searching in a sorted array. In their paper and subsequent papers including ours, it is assumed that the entire set of costs is available to the algorithm.

In this paper, we consider the selection problem in this framework. We are given a set of n elements where each element has a *value* and a *size* associated with it (value and sizes are independent), and an integer $r \in [1..n]$. The cost of comparing two elements is completely determined by their sizes. Our goal is to minimize the total cost of comparisons performed in determining the element rank r (based on the value). When the comparison costs are allowed to be an arbitrary function of the sizes, the problem becomes provably hard even for special cases of selection. For instance, Hartline *et al* [5], and independently, Gupta and Kumar [4], have shown that in the general model of comparison costs, any algorithm for computing the maximum of n elements has a competitive-ratio $\Omega(n)$. An $O(n)$-competitive algorithm for computing the maximum is known [3]. However, for other problems such as sorting and general selection, no non-trivial bounds on the competitive ratio are known. Since a comparison involves two elements, natural restrictions can be placed on the functions mapping pairs of sizes to the comparison cost. Some natural structured cost functions and situations where these functions might apply are listed below. In all cases assume that we are comparing two elements a_1 and a_2 with associated sizes s_1 and s_2.

sum Cost of comparison is $s_1 + s_2$.

product Cost of comparison is $s_1 s_2$. Both `sum` and `product` are motivated in [4] by the following application — a proxy can compare databases a_1 and a_2 of sizes s_1 and s_2 but in order to do so it must read both databases (at cost $s_1 + s_2$) or compare every entry in one database with every entry in the other (at cost $s_1 s_2$).

minimum Cost of comparison is $\min(s_1, s_2)$. If a_1 and a_2 are strings and the comparison is lexicographic, this function represents the worst-case cost of the comparison.

Many other functions are possible but one property common to the functions that we have described above and other natural functions is ***monotonicity***. We say that a comparison cost function is monotone if increasing the size of one of the elements being compared does not decrease the cost of the comparison. Coming up with competitive algorithms that work for arbitrary, monotone comparison costs is therefore an interesting problem. We note here that an example of non-monotone comparison costs is the nuts-and-bolts model where the comparison cost is 1 for a comparison between a nut and a bolt, and is ∞ otherwise [1, 2, 6]. To our knowledge, this is the only example in the literature of comparison costs that can not be modeled by a monotone cost function.

*Dept. of CIS, University of Pennsylvania, Philadelphia, PA 19104. E-mail: kannan@cis.upenn.edu. Supported in part by NSF Grants CCR0105337 and CCR9820885

†Dept. of CIS, University of Pennsylvania, Philadelphia, PA 19104. E-mail: sanjeev@cis.upenn.edu. Supported in part by an Alfred P. Sloan Research Fellowship and by an NSF Career Award CCR-0093117.

Selection is a challenging problem even in the monotone cost model. One reason is that the certificate cost is highly sensitive to the rank of the element being selected. This is in contrast to the uniform case where the optimal certificate cost is always $\Theta(n)$. Suppose we want to select the r^{th}-smallest element out of n elements. For each element x other than the element of rank r, the optimal certificate involves comparing x to an element y of smallest size that lies between x and the rank-r element. Many selection algorithms work by finding a pivot element whose rank approaches r over several iterations. However, note that even if we were lucky enough to find an element of rank $r+1$ and performed a single pivoting iteration with this element, we might have incurred a cost that is unboundedly higher than the cost of the optimal certificate. This can happen, for example, when the rank r element has significantly smaller size than the rank $r+1$ element. Note that this is a problem even for the `min` function. Even a pivot with respect to the correct rank r element may not be competitive with the optimal certificate when the rank-r element has large size. Also note that sorting all the elements is not an option. Such a procedure could be unboundedly worse than the optimal certificate for selection.

Gupta and Kumar [4] have studied in detail the sorting and selection problems for specific cost functions. In particular, they give $O(1)$-competitive algorithms for selection in the `sum` and the `product` cost model. In addition, [4] provides a very simple algorithm for selection under monotone costs and claims that this algorithm achieves a competitive ratio of $O(\log n)$. However, this algorithm turns out to be flawed. It is easy to construct monotone cost functions on which the competitiveness of this algorithm is $\Omega(n)$ or worse.

Our results: Our main result is an algorithm for selection under the monotone cost model which achieves an $O(\log^2 n)$-competitive ratio. Thus the natural monotonicity assumption significantly alters the complexity of the selection problem. This algorithm carefully balances the amount of work it does with the currently known lower bound for the optimal certificate. To make the algorithm work we need to show that after performing comparisons costing no more than $O(\log^2 n)$ times the current lower bound on the cost of the certificate we have either found the rank-r element or raised the lower bound on the certificate cost. We essentially try to do this accounting on a per-element-basis, spending only $O(\log^2 n)$ times the cost of an element's certifying comparison on comparisons of this element with smaller elements. One big difficulty in making this work is that we may not immediately recognize when we have performed a certifying comparison for an element since recognizing this event requires comparisons involving other pairs of elements which we may not yet afford to perform!

We also present a randomized constant-factor-competitive algorithm for selection under the special case of the `min` function. Our algorithm strongly relies on both the structure of the `min` function and elementary properties of random sampling.

2 Preliminaries

We are given a set X of n elements, say $x_1, x_2, ..., x_n$, drawn from some totally ordered set. We are also given two non-negative functions: (i) a size function $s : X \to \mathrm{R}$, and (ii) a comparison cost function $f : \mathrm{R} \times \mathrm{R} \to \mathrm{R}$. We will focus on monotone comparison cost functions: we say f is monotone if $f(a,b) \geq f(a',b')$ whenever $a \geq a'$ and $b \geq b'$.

For ease of notation, we will use $|x_i|$ to denote $s(x_i)$. W.l.o.g. we assume that $|x_1| \leq |x_2| ... \leq |x_n|$. Finally, we will denote by X_i the set of first i elements, namely $\{x_1, x_2, ..., x_i\}$.

We give here a simple characterization of optimal certificate cost for selection. This characterization will be useful in analyzing the relative costs of the certificates produced by our online algorithms. Let σ be the permutation that sorts the elements of X in ascending order.

PROPOSITION 2.1. *Let r be any integer between 1 and n. For any element x_i, $i \neq \sigma(r)$, let ℓ_i denote the index of the element of smallest size that resides between x_i and $x_{\sigma(r)}$ (including $x_{\sigma(r)}$) in the sorted sequence. Then the optimal certificate for selecting the element of rank r, has cost $\sum_{x_i \in X \setminus x_{\sigma(r)}} f(|x_i|, |x_{\ell_i}|)$.*

We will refer to $f(|x_i|, |x_{\ell_i}|)$ as the optimal certificate cost for the element x_i.

3 Selection for Arbitrary Monotone Functions

We will now describe an algorithm that determines an element of rank r, for any $1 \leq r \leq n$, at a cost of $O(\log^2 n)$ times the optimal certificate cost.

We start by briefly describing the algorithm proposed in [4] to highlight some difficulties in dealing with general monotone functions. The algorithm in [4] partitions elements into lists $L_1, L_2, ..., L_k$ of geometrically increasing sizes. It then sorts L_1, and places all remaining elements among elements of L_1. This step identifies a subset S of elements in $L_2, ..., L_k$ that are candidates for being the element of rank r — all such elements lie between two consecutive elements in L_1. The algorithm now recursively searches for the rank r element in this subset. The analysis of this algorithm relies on the following two propositions. First, in an optimal certificate, each element in the subset S gets compared to an element in $L_2, ..., L_k$. Thus if an element x is compared to some element in L_i in an optimal certificate, the algorithm never performs any comparison between x and an element in $L_{i+1}, L_{i+2}, ..., L_k$. Moreover, the element x is compared only $O(\log n)$ times to elements in

$L_1, ..., L_i$. Second, since L_j's are geometrically increasing in sizes, comparisons performed between x and elements in $L_1, ..., L_i$ can be charged to the optimal certificate cost of x. Combining these propositions, an $O(\log n)$-approximate algorithm is concluded.

The mistakes in this analysis are in the second proposition above. First, the monotonicity of the comparison function does not imply that the comparision costs within any L_i are in a bounded range. In particular, if $f(|x|, |y|) = |x|^{|y|} + |y|^{|x|}$ then the cost of comparing two different pairs of elements within L_i may differ by an exponential factor. Thus we may incur a much larger cost than optimal in sorting any list L_i. Second, even if we do a more refined grouping where perhaps each element is in a separate list, elements of large size may get compared $\Omega(n)$ times by the algorithm whereas an optimal certificate performs only $O(1)$ comparisons that involve such elements. As a concrete example, consider the case where the element of rank r is x_{n-2} while x_{n-1} and x_n are elements of rank $r-1$ and $r+1$, respectively. Then the elements x_{n-1} and x_n survive through $n-2$ steps where lists $L_1, L_2, ..., L_{n-2}$ are considered, and are involved in a comparison at each step. It is easy to see that we can then create an $\Omega(n)$ gap between the optimal certificate cost and the algorithm's cost for even a simple monotone function like $f(|x|, |y|) = \max\{|x|, |y|\}$.

Overview of our algorithm: When compared with the optimal set of comparisons, an algorithm may perform many "unnecessary" comparisons in its search for a certificate. The idea of geometric grouping seems essential for absorbing the cost of unnecessary comparisons into more expensive "necessary" comparisons. However, as highlighted above, a direct grouping of elements runs into difficulties. An alternate approach is to do geometric grouping with respect to each element, i.e., for each element x, we partition the remaining elements into blocks of geometrically increasing costs w.r.t. x. This is the starting point for our approach.

DEFINITION 1. *For any element x_i, let $B_{i1}, B_{i2}, \ldots,$ be a partition of elements into a sequence of* blocks *based on geometrically increasing comparison costs with respect to x_i. We will refer to B_{i1} as the* initial block *for x_i. We refer to an element x_i as* doubling within k *if $|B_{i1}| \leq k$ and as* non-doubling *otherwise.*

Note that even for monotone cost functions, there is nothing monotone about the block structure. For example B_{i1} could be a subset or a superset of $B_{(i+1)1}$ and B_{ij} may not even overlap with $B_{(i+1)j}$. Therein lies some of the difficulty of this problem. If all elements have small initial blocks of size k for some small k, then one can hope to sort the first k elements (charging this cost to the $n-k$ other elements) and then place each of the other elements into the sorted list by binary searching. Let c_i be the optimal certificate cost for x_i. It is clear that in this process we charge only $O(c_i \log n)$ to each element; we either identify one of the first k elements as the rank r element or eliminate them altogether; we have doubled our initial estimate of the certificate cost of the surviving elements and hence can ignore the costs charged to the accounts of the surviving elements and recurse, producing an $O(\log n)$-competitive algorithm.

Unfortunately, all initial blocks may not be of the same size. If some element has an initial block of size smaller than k, it might not be able to pay its share for sorting the first k elements. On the other hand, if an element has an initial block of size larger than k, it might be charged repeatedly for sorting initial segments, so that the cumulative charges are a factor of $\Omega(n)$ greater than its certificate cost. However, such elements have such expensive certificate costs that we can still harness them to sort smaller cost elements. This is approximately what we do, taking great care not to charge repeatedly to these elements. We next describe the invariants maintained by the algorithm.

We will account for the costs incurred by our algorithm as follows. We maintain a global account, A, and a separate account $A(x_i)$ for each element x_i and maintain the following invariants.

- The account A is charged only when the problem size decreases by a constant factor. At each of these points A is charged no more than $O(\log n (\sum_i c_i))$.
- For element x_i let B_{ij} be the block containing the element to which x_i is compared in the optimal certificate. Then, for each $k \leq j$, $A(x_i)$ is charged at most $O(\log n)$ times the cost of a comparison between x_i and an element in B_{ik}. In fact, for any such k, we perform at most two binary searches for x_i within block B_{ik}. $A(x_i)$ can be charged for both of these binary searches. In addition, $A(x_i)$ may bear the cost of a constant number of comparisons involving smaller elements with elements in blocks less than or equal to j. These are the only charges made to $A(x_i)$.

The algorithm proceeds iteratively where each iteration consists of two stages. In the first stage, we identify an initial set of k elements, that we can afford to sort (by charging the cost initially to A) and maintain them in a sorted list L. In the second stage we keep track of the region of interest in L where the rank r element could still lie. We then process the surviving elements not in the sorted list very carefully — we pick an appropriate one of these elements, say x_i, in a manner that will be explained later. Suppose j is the smallest index for which there exists an element in $B_{ij} \cap L$ whose order with respect to x_i is not known. Then we binary search for x_i in $L \cap B_{ij}$. We repeat this process until we have identified a set of elements that are candidates for being the rank r element and that lie between two successive elements in L. We next describe these stages in detail.

Stage 1: The first part of the algorithm is to identify how many elements we can sort. We will identify an integer k such that $x_1, x_2, ..., x_k$ are sorted and for each x_i, $i = 1, \ldots, k$, the cost of placing it correctly, can be charged to the global account A. However, if the current iteration does not eliminate an $\Omega(n)$-fraction of elements, we will reassign this charge to a suitable set of doubling elements since we are only allowed to charge to A when we do decrease the number of elements by a constant fraction.

Sorted List:
$L = x_1$; $k = 1$ /* L is the sorted list of length k*/
for $i = 2$ **to** n
  if $(|B_{i1}| \geq k)$
    binary insert x_{k+1} in L & (tentatively) charge to A
    $k = k + 1$
endfor
end Sorted List

CLAIM 1. *At the end of stage 1 if $|L| = k$ then there are at most k elements that are non-doubling in k. Moreover, A has been charged no more than $O(\log n)$ times the overall certificate cost.*

Proof. If there are at least $k + 1$ elements that are non-doubling in k, then when processing the $(k+1)^{th}$ of these elements, we would have inserted a $(k+1)^{th}$ element into L. It is clear that if x_k is inserted into L because $|B_{i1}| \geq k$, then the cost of inserting x_k is $O(c_i \log n)$. Since for each such element x_i we charge this amount exactly once to A, the overall charge to A is as claimed. □

Stage 2: We pick a threshold of $n/10$ and follow very different strategies for the two cases of $k \geq n/10$ and $k < n/10$.

Case 1: $\underline{k \geq n/10}$

In the case of large k, using binary search we place each element that doubles in k in the sorted subsequence of L formed by its initial block. We also do a complete binary search of L to place each of the elements that are non-doubling. At this point we have only paid at most a log-factor over the optimal certificate for each element.

For each element $x \in L$, we compute lower and upper bounds $L(x)$ and $U(x)$ on the rank of x. Clearly the set of x such that $r \in [L(x), U(x)]$ is an interval of L and this is the interval of interest. Let x_ℓ and x_h be the lowest and highest points in this interval.

To visualize this, we think of L as a list of $k+2$ elements (including two sentinel elements $-\infty$ and ∞). For each pair of elements $x_p < x_q \in L$, define the (possibly empty) set S_{pq} to consist of those $y \notin L$ for which the sharpest bounds known are $x_p < y < x_q$. Visually, we think of a box connecting x_p and x_q which is labeled with all the elements in S_{pq}. We say that the box S_{pq} *spans* the elements in L strictly between x_p and x_q. We say that a box is *maximal* if the subinterval of L it spans is maximal.

CLAIM 2. *Non-empty boxes form a nested treelike structure.*

Proof. The proof follows from the monotonicity of the function f. Suppose there are two non-empty intersecting boxes (x_{p_1}, x_{q_1}) and (x_{p_2}, x_{q_2}) containing elements y_1 and y_2 respectively such that neither is contained in the other. Assume w.l.o.g. that $x_{p_1} < x_{p_2} < x_{q_1} < x_{q_2}$. Now suppose $|x_{q_1}| \geq |x_{p_2}|$. Then x_{p_2} must belong to the initial block of y_1, contradicting the assumption that the box (x_{p_1}, x_{q_1}) represents the sharpest bounds on the rank of y_1. On the other hand, if $|x_{q_1}| < |x_{p_2}|$, x_{q_1} must belong to the initial block of y_2, contradicting the assumption that (x_{p_2}, x_{q_2}) represents the sharpest bounds on y_2. □

CLAIM 3. *There must be at least one element y such that both $y < x_\ell$ and $y > x_h$ are possible based on the set of comparisons that have been done so far.*

Proof. Suppose not. Then there are no boxes that span x_ℓ and x_h. By the nested structure of the boxes there must be an element x_k between x_ℓ and x_h (inclusive) whose exact rank is known, causing the elimination of at least one of x_ℓ and x_h. □

CLAIM 4. *If a box spans all the candidate elements in L, then for each element y in this box, at least one comparison involving y must be performed before the rank r element can be found.*

By the above claim the lower bound for the optimal certificate is increased. Thus for each x_i in the outermost box, we identify the set of elements in the next block for x_i which are contained in the sub-interval $[x_\ell, x_h]$ and binary insert x_i into this set at cost at most $\log n$ times the current lower bound on c_i. At the end of this stage either one of the elements in L has been identified as the desired element of rank r or all elements in L have been eliminated. Since $k > n/10$, we have eliminated a constant fraction of the elements charging at most a log factor over the optimal certificate for each of the elements. The entire cost of this iteration can therefore be charged to the global account A.

Case 2: $\underline{k < n/10}$

The difficulty in this case arises from the k or fewer non-doubling elements. If we are not careful, they will be charged for their pivoting and this charge could be already a factor of $\log n$ worse than the optimal certificate cost for these elements. Since we may not eliminate more than k elements at this stage, there could be an accrual of charges to these elements over stages that results in our algorithm being only $O(n)$ competitive. In addition, in the situation where we do not eliminate many elements we need to reassign the charge we made to the global account A for producing the sorted list L.

As before, we binary insert each doubling element into its initial block in L. At this point the non-doubling elements are exactly the ones in a box between the sentinel elements which we call the sentinel box. We then identify the candidate elements for rank r in L, as well as the boxes that span them.

Besides the sentinel box, the candidate elements may have several non-sentinel boxes spanning them. We call boxes whose span includes candidate elements, non-sentinel boxes of interest. If there is a non-sentinel box that spans all the candidate elements in L, then Claim 4 applies and we can afford to binary insert all the elements in this box into their next blocks. On the other hand, if there are one or more maximal, non-sentinel boxes which span portions of the candidate subinterval in L, we have a problem. To continue processing any of these boxes might be a mistake if the actual rank r element is one whose relationship to the elements in this box is already known.

Clearly, any algorithm needs further information about the placement of elements in the sentinel box before identifying the element of rank r. So, could we simply process the sentinel box first? Unlike Case 1, however, we need to be careful since we might not eliminate many elements at the end of this stage.

There are two possiblities for the next element to be placed: we can either place an element from a maximal non-sentinel box of interest or an element from the sentinel box. If we do the former, and the rank r element eventually turns out to be outside of the range spanned by the non-sentinel box, we perform potentially costly comparisons that cannot be paid for by our charging scheme. On the other hand, in the latter case, we may end up repeatedly charging against the certificate cost of non-doubling elements for $\Omega(n)$ iterations.

We therefore have to adopt a hedge strategy where we balance the cost of placing the elements from the sentinel box and the non-sentinel boxes. As before, by "placing" an element x_i we mean, binary inserting it into the subsequence $L \cap B_{ij}$ where B_{ij} is the first block containing elements whose order with respect to x_i is currently unknown. As the algorithm proceeds, the set of non-sentinel boxes of interest evolves. At any point in time, let H^- and H^+ denote the lowermost and the uppermost maximal non-sentinel box of interest respectively. At the beginning of the stage, let R^- consist of all *candidate* elements which are either in H^- or nested within H^- and define R^+ similarly. An element drops out of R^- or R^+ only if it ceases to be a candidate. Also, let H denote the sentinel box. Recall that the elements in H are the non-doubling elements, while the elements in H^+ and H^- are doubling elements.

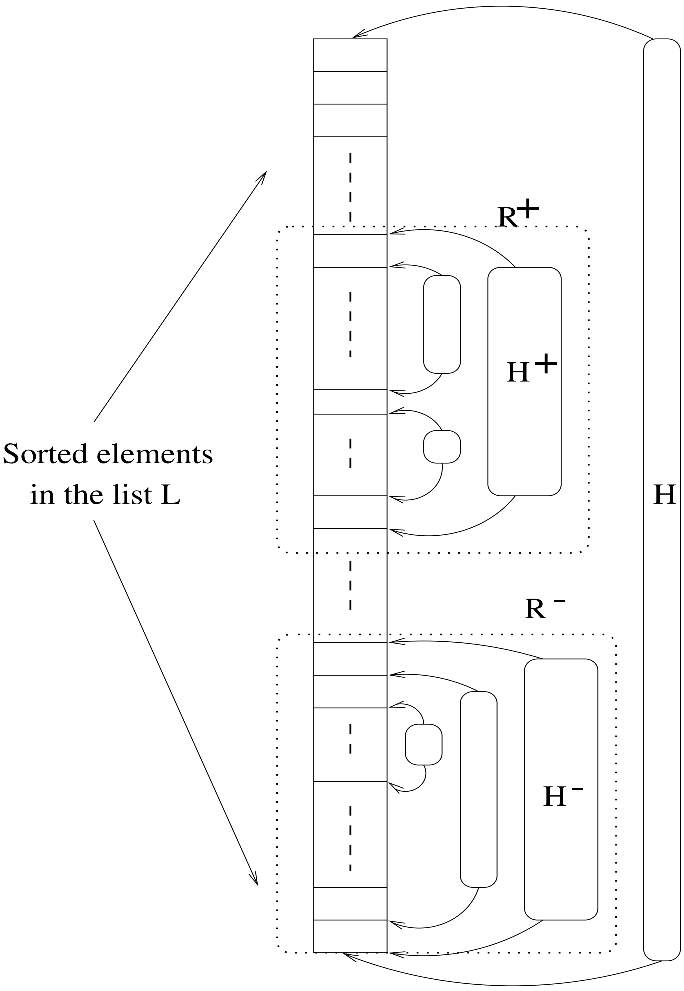

Figure 1: A schematic depiction of H^-, H^+, H, R^-, and R^+.

CLAIM 5. *The total number of elements that are candidates* *for being the rank r element and are not contained in H or R^- or R^+ is at most k.*

Proof. Let $H^- = (x_{p_1}, x_{q_1})$ and $H^+ = (x_{p_2}, x_{q_2})$. It is easy to see that $L(x_{q_1}) \geq r - |H|$ since there are only $|H|$ elements whose order relative to $L(x_{q_1})$ is not known. Now suppose there are α candidate elements that are not contained in H, H^- and H^+. Then $L(x_{p_2}) \geq L(x_{q_1}) + \alpha$. The claim follows. □

The algorithm focuses on the blocks H, H^-, and H^+, and iteratively places elements from these blocks. We initialize three counters C_S, C^-, and C^+ where C_S tracks the total cost of placing elements in H, C^- the cost of placing elements in H^- and C^+ the cost of placing elements in H^+. Once an element is chosen for placement, the cost of placing it in its next block is added to the appropriate counter, and we recompute the blocks H^- and H^+. The next element to be placed is the smallest element in one of the three blocks. It is chosen with the goal of minimizing $\max\{C^-, C^+, C_S\}$ after the placement. This phase of the algorithm terminates when either H becomes empty or we exhaust all elements in $R^- \cup R^+$.

We analyze these scenarios as follows:

a) Suppose we place all elements in H. Then we are now in a similar scenario as in Case 1 where there is only one maximal non-sentinel block that spans all possible rank r candidate elements. Let t denote the total number of surviving elements. If $t < 2n/10$, then we charge the costs in C^-, C^+ and C_S to the global account A. Since the last step before exhausting H must have been the insertion of an element in H, it is clear that $C^- + C^+ + C_S \leq 3C_S$ and C_S is at most $O(\log n)$ times the certificate cost of the non-doubling elements. Since we have reduced the size of the subproblem by a constant factor, it is fine to charge this amount to A.

   On the other hand, if $t \geq 2n/10$, we are left with potentially $n - o(n)$ candidate elements. In this case, note that the elements that survive come from H and exactly one of R^- or R^+, say R^+, without loss of generality. Since H had at most $n/10$ elements to begin with, at least $n/10$ of the elements must be from R^+.

   We cannot charge any costs to A nor to any of the non-doubling elements. We first reassign the cost of creating L. Recall that when we inserted the j^{th} element into L in stage 1, it was because there is an element x_i which is non-doubling in j and hence could absorb the cost of this insertion. We revert to charging x_i for the cost of inserting x_j. If x_i is a doubling element, this charge is at most $\log n$ times its certificate cost and it can absorb this charge. If x_i is an element in H, we will need to further reassign the cost assigned to x_i since we do not want to charge it at all.

   Because of our hedging strategy, we know that after we place the smallest surviving element from H^+ in its next block, C^+ will exceed C^- and C_S. Thus the charges accumulated to C^- and C_S can be absorbed by elements in R^+ that have accumulated the charges in C^+. Since the sorting costs assigned to elements in H are dominated by C_S these can also be absorbed by the elements in H^+.

   Finally, we need to show that a doubling element does not pay too often for binary insertion into the same block. In any one iteration, a doubling element is inserted into any one of its blocks at most once. However, notice that for a doubling element x_i with B_{ij} being the last block with a non-empty intersection with L, it may be the case that L contains only an initial portion of B_{ij}. In this case we already charge $A(x_i)$ for inserting x_i into this initial portion of B_{ij}. In successive iterations we need to argue that we do not keep charging x_i for insertion into portions of B_{ij}.

   To see this, note that if in the next iteration L is a subset of B_{ij} then x_i will be a non-doubling element. Since we never charge non-doubling elements x_i will not be charged. In fact, the only iteration where $A(x_i)$ can get charged again for insertion into B_{ij} will be the iteration where L includes the last element in B_{ij}. Thus, each x_i is charged at most twice for binary insertion into each of its blocks up to and including the block used by the optimal certificate. For each of these blocks $A(x_i)$ is charged no more than $O(\log n)$ times the cost of a comparison between x_i and an element of this block and the asserted invariants hold.

b) Suppose both R^- and R^+ become empty. Then by the claim above, the total number of elements that are candidate for being rank r is at most $|H| + k \leq 2k$. In this case, we charge the costs in C^-, C^+, and C_S to the global account A. To see that this is okay, observe that the problem size has decreased by a constant factor. Also, note that C_S is at most $O(\log n)$ times the certificate cost for the non-doubling elements and that $C^- + C^+$ is at most C_S plus the cost of placing one more element from H. Thus $C^- + C^+$ is also bounded by $O(\log n)$ times the certificate cost for the elements in H and thus we charge A only an amount provided by the invariant.

Overall we have shown that for any element x_i, the account $A(x_i)$ is charged at most $O(c_i \log n)$ throughout the course of the algorithm and that the global account A is charged at most $O(\log^2 n)$ times the cost of the optimal certificate proving our claim.

4 Selection in the `min` Model

We now consider the comparison cost function $f(a,b) = \min(|a|,|b|)$. We will present an $O(1)$-competitive randomized algorithm for the `min` cost function. As before, the algorithm works by iteratively refining the space of elements that are candidates for being a rank r element. However, we can now exploit the structure of the `min` function to do a direct geometric grouping of elements based on their sizes. At each iteration, the algorithm either identifies an element of the smallest size group that can be used as a pivot for reducing the candidate set by a constant factor, or eliminates the smallest group altogether, raising the lower bound on the optimal certificate cost for the remaining elements. The difficulty here is in efficiently identifying a good pivot. We note that it is straightforward to get an $O(\log n)$-competitive algorithm by simply sorting the lowest class, binary inserting every other element into this sorted list and recursing on the surviving candidates.

We now describe our $O(1)$-competitive algorithm in detail. We partition the elements $x_1, x_2, \ldots, x_n$ into blocks of geometrically increasing size. More precisely, the partition has blocks $S_0, S_1, \ldots, S_b$ where $S_i = \{x_j | 2^i \leq \frac{|x_j|}{|x_1|} < 2^{i+1}\}$. For notational convenience, let s_i represent the maximum size of any element in S_i. (If S_i is empty, $s_i = 0$.) The algorithm maintains a set of elements, C, which are candidates for being the elements of rank r.

Let S_i be the lowest indexed block which has a nonempty intersection with C. Then the algorithm always maintains the invariant that $|S_i \cap C| \leq |C - S_i|$ by performing the following reduce step `reduce`(S_i, C) when necessary: Let x_b be the element that has rank $r - |C - S_i|$ in S_i and x_u be the element that has rank r in S_i. It is clear that the only elements in S_i which are candidates for the overall rank r element are those between x_b and x_u and there are clearly $|C - S_i|$ of them. The reduce step works by selecting x_b and x_u and pivoting on these elements. The cost of the reduce step is $O(|S_i \cap C| s_i)$. Henceforth we will assume that C is always thus reduced.

Initially the algorithm finds the median x_m of the block S_0. It compares a random sample of the elements in higher blocks with x_m to determine if x_m is a good pivot. If more than $3/4$-fraction of the sample elements are greater than x_m or more than $3/4$-fraction of the sample elements are smaller than x_m then the algorithm recurses with the appropriate half of S_0 replacing S_0. Once a good pivot, say x_m, has been found (i.e., each side of x_m contains less than $3/4$-fraction of the sample elements), all elements in $C - S_0$ are compared against it. This immediately tells us which side of x_m is the new set of candidate elements.

With high probability, this pivoting step will eliminate a constant fraction of elements from blocks higher than S_0. Thus in $O(\log |C|)$ good pivot steps we will have either eliminated all elements from higher blocks (in which case we can just solve the selection problem on the remaining candiate elements using a standard selection algorithm) or we will have eliminated all elements from S_0 (in which case we can move on to the next non-empty block).

The key invariants maintained by the algorithm are the following.

1. Every comparison made by the algorithm involves an element from the lowest block in which candidates still exist.
2. Let n_i be the size of the candidate set when S_i becomes the smallest-indexed block with a candidate remaining. Then the total cost of comparisons in which the smaller sized element comes from S_i is at most $O(n_i s_i)$.

The brief description of the algorithm above makes clear that the first invariant is maintained. In order to maintain the second invariant, if we have a candidate set C we will use a sample of size approximately $|C|/\log|C|$. (If the smallest surviving block is S_i, we will pick a sample by picking each candidate element from a block higher than i with probability $1/\log|C|$.) However, if $|C|$ is less than some constant threshold T, we pick all elements in $C - S_i$ in the sample.

The optimal certificate has cost $\Omega(\sum_{i=0}^{k} n_i s_i)$ and it is clear that our algorithm is constant factor competitive.

A more formal treatment of the algorithm and its analysis is given below. The probability p in the code below is usually $1/\log|C|$ but is 1 if $|C|$ is below threshold.

$C = \{x_1, x_2, \ldots x_n\}$
for $i = 0$ **to** k
    `reduce`(S_i, C)
    $temp = (C \cap S_i)$
    while $(temp \neq \emptyset)$ **and** $(C - S_i \neq \emptyset)$
        $x_m = \texttt{select} - \texttt{median}(temp)$
        $S = \emptyset$.
        foreach $x \in C - S_i$
            with probability p, $S = S \cup \{x\}$
        $L = 0$
        foreach $x \in S$
            if $(x > x_m)$, $L = L + 1$
        if $(|S|/4 < L < 3|S|/4)$ **or** (all of $C - S_i$ has been compared)
            compare all $x \in C$ with x_m
                update C and r
            `reduce`(S_i, C)
        elseif $(L \leq |S|/4)$
            $temp =$ elements in S_i less than x_m
        else $temp =$ elements in S_i greater than x_m
    endwhile
endfor

The analysis of the algorithm requires a lemma using Chernoff bounds which follows along standard lines.

LEMMA 4.1. *Let C be a set and α be the fraction of elements from C which are greater than a given element x_m. If a random sample of size R is chosen from C then the probability that the fraction of the sample that is greater than x_m exceeds $\alpha(1+\epsilon)$ is upper bounded by $e^{-\epsilon^2 \alpha R/3}$. The probability that the fraction of the sample that is greater than x_m is smaller than $\alpha(1-\epsilon)$ is upper bounded by $e^{-\epsilon^2 \alpha R/2}$.*

Proof. Let $X_1, X_2, \ldots, X_R$ be i.i.d. random variables where X_1 is 1 if a random, uniformly distributed element from C is greater than x_m and 0 otherwise. Each of these random variables has mean α and the lemma follows from applying Chernoff bounds (see, [7], for instance). □

The detailed description of the algorithm clearly shows that invariant 1 holds. Notice that all comparisons are made with x_m which is always a member of the lowest indexed block surviving in the candidate set. If this lowest indexed block is i, note that for each of the surviving candidates the optimal certificate involves a cost of at least s_i.

We will view the algorithm above as being performed in stages where the i^{th} stage corresponds to the i^{th} iteration of the outer for-loop. Let C be the size of the candidate set at the beginning of the i^{th} stage. We will call a comparison between the sample S and an element x of S_i a *sample pivot*. When we compare all of $C - S_i$ with an element x of S_i we call this a pivot step. A pivot step is *good* if the split it produces has at least a fraction 1/8 of the elements of $C - S_i$ on either side of x.

We divide stage i into epochs where each epoch ends with a pivot step. From the description of the algorithm it is clear that the number of sample pivots in an epoch is $O(\log |C|)$. Each sample pivot at stage i has expected cost $|C|s_i/\log |C|$ and thus the total cost of sample pivots in an epoch is $|C|s_i$. The expected number of pivot steps before a good pivot step is a constant since each pivot step has a high probability of being good. Each good pivot step leads to a constant factor reduction in $|C - S_i|$.

Thus the number of epochs is $O(\log |C - S_i|)$ and the cost of the epochs decreases geometrically. Thus the overall cost of stage i is dominated by the cost of the first epoch which is $O(|C|s_i)$ as desired.

5 Concluding Remarks

We have an $O(\log^2 n)$-competitive algorithm for the selection problem with an arbitrary monotone comparison cost function. This result is in strong contrast to an $\Omega(n)$ lower bound that is known for finding even the maximum element under unrestricted comparison cost functions. The currently best known lower bound for monotone cost functions is a constant (e.g., can be shown for `sum` function). A natural question is if there exists an $O(1)$-competitive algorithm for arbitrary cost functions. We conjecture that the competitive ratio of this problem is $\Omega(\log n)$.

We also gave a randomized $O(1)$-competitive algorithm for the `min` model which is a natural special case of monotone functions. The performance guarantee of our algorithm strongly relies on ability to choose good pivots using samples of sublinear size. An interesting question is if a deterministic $O(1)$-competitive algorithm can be shown for this problem.

References

[1] N. ALON, M. BLUM, A. FIAT, S. KANNAN, M. NAOR, AND R. OSTROVSKY. Matching nuts and bolts. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms,* page 690–696, 1994.

[2] N. ALON, P. G. BRADFORD, AND R. FLEISCHER. Matching nuts and bolts faster. *Information Processing Letters,* 59(3), pp. 123–127, 1996.

[3] M. CHARIKAR, R. FAGIN, V. GURUSWAMI, J. KLEINBERG, P. RAGHAVAN, AND A. SAHAI. Query strategies for priced information. In *Proceedings of 32nd ACM Symposium on Theory of Computing*, pp. 582–591, 2000.

[4] A. GUPTA AND A. KUMAR. Sorting and Selection with Structured Costs. In *Proceedings of 42nd IEEE Symposium on Foundations of Computer Science*, pp. 582–591, 2001.

[5] J. HARTLINE, E. HONG, A. MOHR, E. ROCKE, AND K. YASUHARA. As reported in [3]. Nov. 2000.

[6] J. KOMLOS, Y. MA, AND E. SZEMEREDI. Matching nuts and bolts in $O(n \log n)$ time. In *Proceedings of 7th ACM-SIAM Symposium on Discrete Algorithms*, pp. 232–241, 1996.

[7] R. MOTWANI AND P. RAGHAVAN. *Randomized Algorithms*. Cambridge University Press (1995).