CS369N: Beyond Worst-Case Analysis
Lecture #1: Instance Optimality*

Tim Roughgarden†

February 19, 2010

1 Preamble

We begin with a provocative question.

Question 1.1 What is the point of theoretical research in the design and analysis of algorithms?

There is not, of course, a unique answer to this question. But certainly one answer — and to many, the most important answer — is *to give rigorous and accurate advice about how to solve important algorithmic problems.* An ideal theory would identify the "best" algorithm for a problem. At the very least, a successful theory should be able to determine which of two proposed algorithms is the better one.

To simplify matters, assume that we have agreed on a definition $\text{cost}(A, z)$ that measures the performance of an algorithm A on an input z. This measure could be the amount of resources consumed (time, space, I/O, etc.), or could be the quality of the solution computed by the algorithm (e.g., the objective function value of a proposed feasible solution). In practice defining an appropriate performance measure can be tricky, but this issue is outside the scope of this class.

Even with a well defined measure, comparing two algorithms A and B that solve the same problem can be difficult. Typically, one algorithm performs better on some inputs, and the other performs better on the other inputs. Of course, in the lucky case that one is *pointwise* better than the other, meaning that

$$\text{cost}(A, z) \leq \text{cost}(B, z) \tag{1}$$

for every input z, we can unequivocally declare A to be the better of the two algorithms. (Otherwise, we have to reason about trade-offs between the performance on different inputs.)

*©2009–2010, Tim Roughgarden.

†Department of Computer Science, Stanford University, 462 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

If an algorithm is better in this sense than *every* other algorithm, then without controversy we can call it *optimal.* This extremely strong type of optimality is called *instance optimality*, and it is the subject of this lecture.

Remark 1.2 (Worst-Case Analysis) In the current notation, *worst-case analysis* is the philosophy that algorithm A should be deemed "better" than algorithm B if and only if

$$\max_z \{\text{cost}(A, z)\} \leq \left\{\max_z \text{cost}(B, z)\right\},$$

where z ranges over all inputs (of a given size, say). Note that the worst-case input is typically different for different algorithms, so this approach does not correspond to any kind of "consistent belief" about which inputs are the most important or most likely. Rather, it is a belief in "Murphy's Law" — that whatever algorithm you choose to use, the world will conspire to provide you with the worst-possible input for that algorithm.

2 Identifying the Top k Objects from Sorted Lists

We first discuss some results by Fagin, Lotem, and Naor [2]. This paper was the first to emphasize instance optimality as an important concept (and it also named the concept). We discuss only some special cases of their results, to illustrate the main points and ideas.

2.1 The Setup

The problem is as follows. There is a very large set X of objects, such as Web pages. There is a small number m of attributes, such as the ranking (e.g., PageRank) of a Web page under m different search engines. To keep things simple, assume that attribute values lie in $[0, 1]$. Thus an object consists of a unique name and an element of $[0, 1]^m$.

We are also given a *scoring function* $\sigma : [0, 1]^m \rightarrow [0, 1]$ which aggregates m attribute values into a single score. We interpret higher attribute values and scores as being "better". We assume that the scoring function is *monotone*, meaning that its output is nondecreasing in each of its inputs. An obvious scoring function is the average, but clearly there are numerous other natural examples.

The algorithmic goal is, given a positive integer k, to identify k objects of X that have the highest scores (ties can be broken arbitrarily).

We assume that the data can only be accessed in a restricted way. It is presented as m sorted lists $L_1, L_2, \ldots, L_m$. Each list L_i is a copy of X, sorted in nonincreasing order of the ith attribute value. An algorithm can only access the data by requesting the next object in one of the lists. Thus an algorithm could ask for the first (highest) object of L_4, followed by the first object of L_7, followed by the second object of L_4, and so on. Such a request reveals the name of said object along with all m of its attribute values. We charge

an algorithm a cost of 1 for each such data access.[1] Thus, in our previous notation, we are defining $\text{cost}(A, z)$ to be the number of data accesses that the algorithm A needs to correctly identify the top k objects in the input z.

2.2 The Threshold Algorithm

We study the following *Threshold Algorithm (TA)*. The algorithm is natural but perhaps not the first algorithm one would write down for the problem. The reader is encouraged to think about "more obvious" algorithms, which probably won't enjoy the same instance optimality guarantee.

Input: a parameter k and m sorted lists.
Invariant: of the objects seen so far, S is those with the top k scores.

1. Fetch the next item from each of the m lists.
2. Compute the score $\sigma(x)$ of each object x returned, and update S as needed.
3. Let a_i denote the ith attribute value of the object just fetched from the list L_i, and set a threshold $t := \sigma(a_1, \dots, a_m)$.
4. If all objects of S have score at least t, halt; otherwise return to step 1.

Figure 1: The threshold algorithm (TA).

We first claim that the TA is correct — for every input, it successfully identifies the k objects with the highest scores (even if it halts well before encountering all of the objects of X).

Proof: By definition, the final set S returned by the TA is the best of the objects seen by the algorithm. If an object $x \in X$ has not been seen by the TA, then its ith attribute value x_i is at most the lowest attribute value a_i of an object fetched from the list L_i (since the lists are sorted). Since σ is a monotone scoring function, $\sigma(x)$ is at most $\sigma(a_1, \dots, a_m)$, which by definition is the final threshold t of the TA, which by the stopping rule is at most the score of every object in S. Thus every object in S has score at least as large as every object outside of S, as desired. ■

The main take-away point of the proof is: the threshold t acts as an upper bound on the

[1] This is not the most realistic cost model, but it serves to illustrate our main points in a simple way. In the terminology of [2], this corresponds to a sequential access cost of 1 and a random access cost of 0. More generally, Fagin et al. [2] charge some constant c_s for each data access of the type we describe and assume that accessing list L_i only reveals the value of the ith attribute; the other attribute values are then determined via $m - 1$ "random accesses" to the other lists, each of which is assumed to cost some other constant c_r. Analogous instance optimality results are possible in this more general model [2].

best-possible score of an unseen object. Once the best objects identified so far are at least this threshold, it is safe to halt without further exploration.

2.3 Instance Optimality of the Threshold Algorithm

Less clear is the following result, which in the vocabulary of [2] asserts that the threshold algorithm is *instance optimal with optimality ratio* m.

Theorem 2.1 (Instance Optimality of the TA) *For* every *algorithm* A *and* every *input* z,

$$cost(TA, z) \leq m \cdot cost(A, z). \tag{2}$$

In words, suppose you pre-commit to using the TA, and you have a competitor who is allowed to pick *both* an input z *and* a (correct) algorithm A that is specifically tailored to perform well on the input z. Theorem 2.1 says that even with this extreme advantage, your opponent's performance will only be a factor of m better than yours. Recall that we view m as a small constant, which makes sense in many natural motivating applications for the problem. While the guarantee (2) is not as strong as (1), because of the constant factor m, it is still amazing that any kind of per-input guarantee is possible. Also, we'll note below that no algorithm has an optimality ratio smaller than m.

Proof of Theorem 2.1: Consider a (correct) algorithm A and an input z. Suppose that A accesses the first $k_1, \ldots, k_m$ elements of the lists $L_1, \ldots, L_m$ en route to computing the (correct) output S on z. For each i, let b_i denote the ith attribute value of the last accessed object of L_i — the lowest such attribute value seen for an object fetched from L_i.

The key claim is that every object x in A's output S must have a score $\sigma(x)$ that is at least $\sigma(b_1, \ldots, b_m)$. The reason is: for all A knows, there is an unseen object y with attribute values $b_1, \ldots, b_m$, appearing as the (k_i+1)th object of list L_i for each i (recall that ties within an L_i can be broken arbitrarily). Thus, A cannot halt with $x \in S$ and $\sigma(x) < \sigma(b_1, \ldots, b_m)$ without violating correctness on some input z'. (Here z' agrees with z on the first k_i objects of each L_i, and has an object y as above next in each of the lists.)

Now, after $\max_i k_i$ rounds, the TA has probed at least as far as A into each of lists, and has discovered every object that A did (including all of S). Thus a_i, the ith attribute value of the final item fetched by the TA from the list L_i, is at most b_i. Since σ is monotone, $\sigma(a_1, \ldots, a_m) \leq \sigma(b_1, \ldots, b_m)$. Thus after at most $\max_i k_i$ rounds, the TA discovers at least k objects with a score at least its threshold, which triggers its stopping condition. Thus $\text{cost}(TA, z) \leq m \cdot \max_i k_i$; since $\text{cost}(A, z) = \sum_i k_i \geq \max_i k_i$, the proof is complete. ■

The reader should think about why the guarantee in Theorem 2.1 continues to hold even if A is a Las Vegas randomized algorithm or a nondeterministic algorithm.

2.4 A Matching Lower Bound on the Optimality Ratio

The factor of m in Theorem 2.1 cannot be improved, for the TA or any other algorithm. We content ourselves with the case of $k = 1$ and a scoring function σ with the property that

$\sigma(x) = 1$ if and only if $x_1 = x_2 = \cdots = x_m = 1$. More general lower bounds are possible [2], using extensions of the simple idea explained here.

The guarantee of instance optimality is so strong that proving lower bounds can be quite easy. Given an arbitrary correct algorithm A, one needs to exhibit an input z and a correct algorithm A' with smaller cost on z than A. Getting to choose A' and z in tandem is what enables simple lower bound proofs.

Precisely, suppose $k = 1$. We only need to use special inputs z of the following form:

- there is a unique object y with $\sigma(y) = 1$; and
- this object y appears first in exactly 1 of the lists $L_1, \ldots, L_m$. (Recall that arbitrary tie-breaking within a list is allowed.)

The lower bound follows from the following two observations. For every such input z, there is an algorithm A' with $\text{cost}(A', z) = 1$: it looks in the list containing y first, and on finding it can safely halt with y as the answer, since no other object can have a higher score. But for every fixed (deterministic) algorithm A, there is such an input z on which $\text{cost}(A, z) \geq m$: A must look at one of the lists last, and one can pick the z in which y is hidden in this last list.

We do not claim that this lower bound is interesting in its own right. Rather, the fact that lower bounds for instance optimality arise so trivially should give further appreciation for upper bounds, when they exist.

3 Maxima in the Plane

We next touch on results of Afshani, Barbay, and Chan [1], who give a number of interesting instance-optimality results for fundamental problems in computational geometry, including computing the convex hull of a point set in two or three dimensions. We discuss only their simplest result, for the following *2D Maxima* problem.

3.1 The Problem

The input is n points in the plane. Assume for simplicity that all coordinate values are distinct (this is not important). Say that x is *dominated* by y if y is bigger in both coordinates (i.e., lies to the northeast of x). A *maximal point* is one not dominated by any other. The goal is to compute the set of all maxima of the point set. See Figure 2.

We assume that an algorithm can only access the input via comparisons of coordinate values, and define $\text{cost}(A, z)$ as the number of comparisons that the algorithm A needs to compute the correct answer for the input z. We are now in a position to reason about instance-optimal algorithms for 2D MAXIMA.

One neat aspect of the main result of this section is that it proves the instance optimality of an existing algorithm for the problem — the classic and clever algorithm of Kirkpatrick and Seidel [3], which we review next.

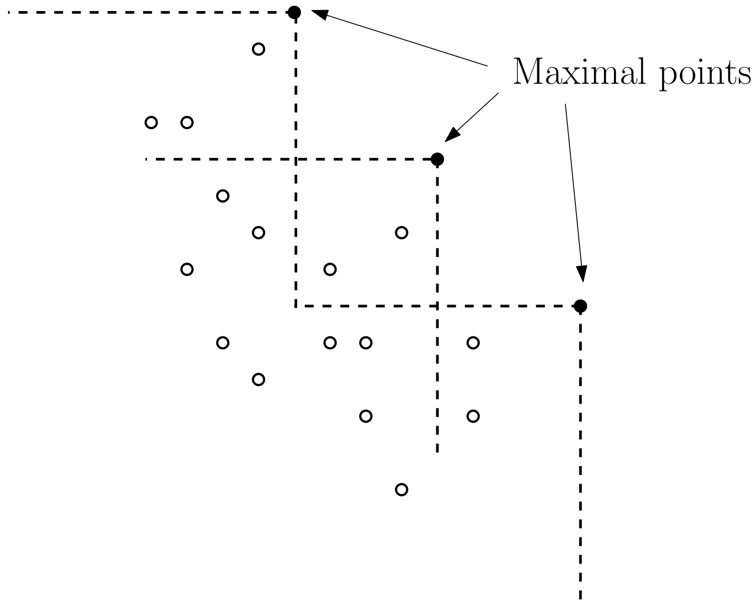

Figure 2: A point set and its maxima. Solid circles are the maximal points, hollow circles are dominated points. The dashed lines indicate the "region of domination" for each of the maximal points.

3.2 The KS Algorithm and Its Correctness

We claim that the KS algorithm is correct. First, note that all deletions are of dominated points and are therefore justified. Second, observe that the point q is maximal in the point set Q: its x-coordinate is bigger than everything in Q_ℓ, and its y-coordinate is bigger than everything in Q_r. Third, recursively computed maxima of Q_r are also maxima in Q: such a point p is not dominated by anything in Q_r, nor by q (since it wasn't pruned), nor by anything in Q_l (all of which have smaller x-coordinates than p). Finally, recursively computed maxima of Q_ℓ are also maxima in Q: such a point p is not dominated by anything in Q_l, nor by q or anything in Q_r (since p is not dominated by q and q has the largest y-coordinate in Q_r). Thus every point is eventually either correctly deleted or correctly identified as a maximal point of the original point set.

3.3 Running Time Analysis of the KS Algorithm

An $O(n \log n)$ Bound. First, observe that the work done in a call to the KS algorithm (finding a median, a maximum, and pruning) is linear in the input size (not counting the work done by further recursive calls). Each recursive call is on at most half of the input. The standard MergeSort recurrence thus gives an upper bound of $O(n \log n)$ on the running time of the KS algorithm.

Input: a point set Q.

1. If $Q = \emptyset$ return; if $|Q| = 1$ return Q.
2. Split the input into left and right halves Q_ℓ and Q_r, by computing the median x-coordinate among points of Q.
3. Let q have the maximum y-coordinate in Q_r, and add q to the output set S.
4. Delete q and everything it dominates.
5. Recurse on (what's left of) Q_ℓ and Q_r.

Figure 3: The Kirkpatrick-Seidel (KS) algorithm.

An $O(n \log h)$ Bound. This $O(n \log n)$ upper bound is achieved for some inputs (as the reader should verify), but we can still pursue a refined upper bound that is sharper for some inputs. A common refinement in computational geometry is *output-sensitive* running time bounds. Consider an instance in which k of the n points are maximal (of course k could be n, but is often much less). We claim that the KS algorithm runs in $O(n \log h)$ time on every input.

The informal proof is as follows. Every recursive call successfully identifies a new maximal point (the point q). Thus the ith level of recursion identifies 2^i new maximal points. Thus there can only be $\log k$ recursion levels, and the total work at each level is $O(n)$.

Why is this not a proof? Because it is only recursive calls with *non-empty input* that identify new maximal points. The pruning step might render one (or both) recursive calls vacuous, which can cause the number of recursion levels to exceed $\log k$. But in this case the work of a function's single non-empty recursive call (on half as many points) can be charged directly against its own work with only a constant factor loss (by a geometric sum argument). We leave the details to the reader.

A Still More Refined Bound. To claim an instance-optimality result, we need an even more sensitive running time bound on the KS algorithm. To state it, suppose that we can partition the input set S into groups $S_1, \ldots, S_k$ where for each i:

1. either S_i is a single point; or
2. there is an axis-aligned box B_i such that its interior contains all of S_i and lies strictly below the "staircase" of S (recall Figure 2).

We call such a partition *legal*; see Figure 4 for an example. The intuition is that each group S_i of the second type represents a cluster of points that can conceivably be easily eliminated by an algorithm such as the KS algorithm in one fell swoop. Thus the bigger the S_i's, the easier one might expect the instance to be. Formally, we prove the following.

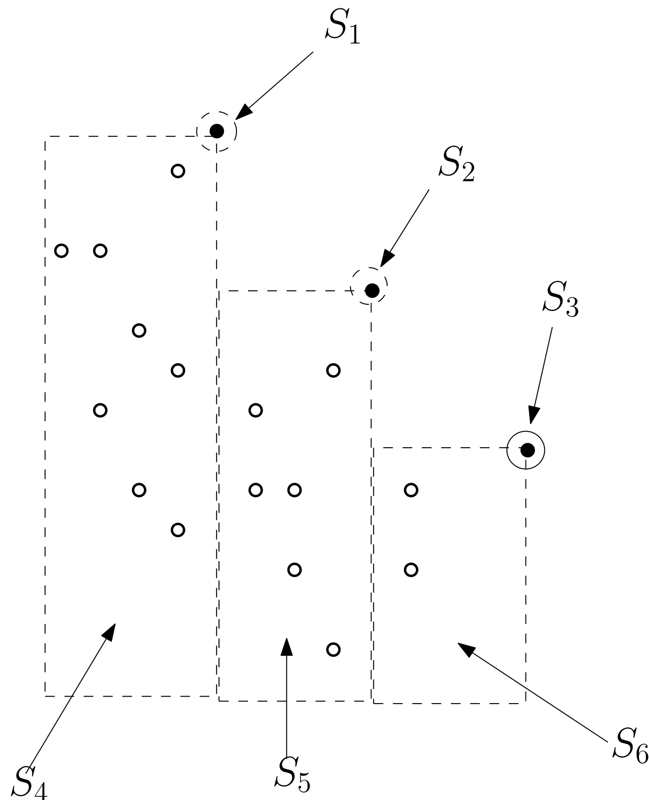

Figure 4: A legal partition of a point set.

Theorem 3.1 ([1]) *For every such partition of an input set, the running time of the KS algorithm is*

$$O\left(\sum_{i=1}^{k} |S_i| \log \frac{n}{|S_i|}\right).$$

The reader should verify that the bound in Theorem 3.1 subsumes the previous $O(n \log h)$ bound (take the S_i's to be "vertical slabs" as in Figure 4 and use the convexity of the function $x \log x$). It is tempting to interpret the bound as some type of "entropy measure" for point sets (that is specific to the problem of computing maxima).

Proof of Theorem 3.1: Consider such a partition, as above. We bound each set S_i's contribution to the running time separately, and then sum over all of the S_i's.

So consider a set S_i. The set's contribution to the work done at recursion level j is linear in the number of points of S_i that have not yet been pruned by the algorithm. Obviously there are at most $|S_i|$ such points. The key step in the proof is the following alternative upper bound.

Key Claim: Every set S_i has $O(n/2^j)$ unpruned points remaining at the jth recursion level of the KS algorithm.

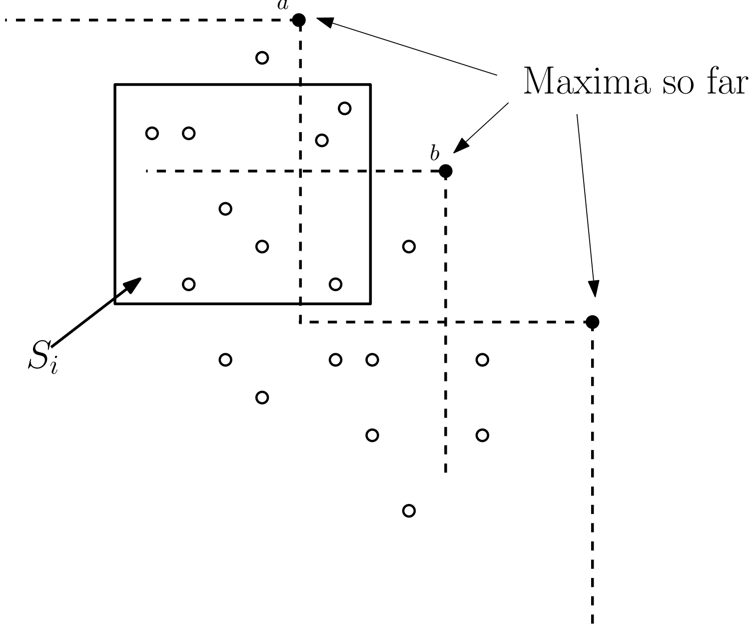

Figure 5: Proof of key claim for Theorem 3.1.

Let's prove the theorem assuming the key claim. Observe that the bound of $|S_i|$ is better than the bound of $n/2^j$ for the first $\approx \log_2(n/|S_i|)$ recursion levels. Summing over all recursion levels j, the total contribution of the points of S_i to the running time of the KS algorithm is

$$\leq \underbrace{|S_i| + |S_i| + \cdots + |S_i|}_{\log(n/|S_i|) \text{ times}} + \frac{|S_i|}{2} + \frac{|S_i|}{4} + \frac{|S_i|}{8} + \cdots$$

which is $O(|S_i| \log \frac{n}{|S_i|})$. Summing over all the S_i's proves the theorem.

Proof of the Key Claim: Let a, b denote the x-coordinates of the maximal points q_1, q_2 identified so far that flank the right-hand side of S_i's box B_i (see Figure 5). Take $a = -\infty$ or $b = +\infty$ if need be (if q_1 or q_2 doesn't exist). By definition, all points of S_i have x-coordinate less than b. Since the interior of B_i lies below the staircase of S (recall Figure 4) and q_1 is a maximal point, q_1 lies at or above the top of B_i and thus all surviving points of S_i have x-coordinate at least a. We can therefore finish the proof by showing that, in recursion level j, there are at most $O(n/2^j)$ surviving points with x-coordinate strictly between those of two neighboring already-identified maximal points.

By induction on j, the $\leq 2^j$ level-j recursive calls bucket by x-coordinate the surviving points into $\leq 2^j$ buckets with population $\leq n/2^j$ each. These recursive calls identify a different maximal point in each of the (nonempty) buckets. After these are identified, there are at most $2 \cdot n/2^j$ surviving points strictly between two successive maximal points (which occurs with two consecutive buckets that have maximal points at the "far left" and "far right", respectively). Thus by recursion level j, only $O(n/2^j)$ points of S_i remain to contribute to

the running time of the KS algorithm at this level, completing the proof of the claim and the theorem. ■

3.4 Instance Optimality of the KS Algorithm

We would like to assert that the KS algorithm is instance-optimal in the comparison model. This requires proving that for every input z, every (correct) algorithm A requires

$$\Omega\left(\min_{\text{legal } \{S_i\}} \left\{ \sum_i |S_i| \log \frac{n}{|S_i|} \right\}\right) \tag{3}$$

comparisons to compute the maximal points of z.

But there is a problem: *this is clearly false.* The irritating reason is that an algorithm can "memorize" (i.e., be pre-programmed with) the correct answer for a particular input z, and regurgitate it without any non-trivial work. Precisely, for a fixed input z, consider the following algorithm (which is defined for every input w):

1. Check if the real input w is z.

2. If so, output the (previously memorized) correct answer for z.

3. If not, run (say) the KS algorithm on w.

This algorithm is clearly correct. Since each of the first two steps takes linear time, this algorithm runs in linear time on the input z. We see again that the ambition of instance optimality is so strong that lower bounds arise for trivial reasons.[2]

A general pep talk: *annoying counterexamples are not a good reason to abandon the quest for an interesting theorem.* If you had a particular theorem in mind, perhaps the model or the theorem statement can be slightly perturbed to obtain a true and meaningful result? To the credit of Afshani et al. [1], they persevered in the face of the stupid example above and were able to prove an instance-optimality-like guarantee for the KS algorithm.

The motivation for the guarantee formulated and proved in [1] is the following observation: our running time bound for the KS algorithm depends only on the *set* S of input points, and is independent of the *order* in which these points happened to listed in the input to the algorithm. On the other hand, the stupid algorithm above does *not* have this property: if only the unordered set S were memorized in advance, then the first step (verifying that the input is in fact S, listed in some order) would require $\Omega(n \log n)$ comparisons and the algorithm would pose no barrier to an instance optimality result. Intuitively, one expects the running time bound of a "natural" algorithm to be "order-oblivious" in the same sense as the KS algorithm.

Now, an obvious (but still ambitious) goal would be to formulate a definition of a "natural" or "order-oblivious" algorithm, and then show the following:

[2]Why did this problem not occur in Section 2? Because in that setting we were only interested in sublinear time bounds anyway — by the time an algorithm verified the input (as in Step 1 above), the threshold algorithm likely would have stopped early and left it in the dust.

Theorem 3.2 ((Informally) [1]) *For every input z of* 2D MAXIMA*, every "order-oblivious" algorithm requires*

$$\Omega\left(\min_{\text{legal } \{S_i\}}\left\{\sum_i |S_i| \log \frac{n}{|S_i|}\right\}\right)$$

comparisons.

This is precisely the spirit of the results of Afshani et al. [1], and thus *the KS algorithm is instance optimal among the class of order-oblivious algorithms.*

In fact, the precise statement in [1] is even cooler. There remains the lingering question of how to formally define an "order-oblivious" algorithm — and such struggles to usefully and meaningfully define "natural" algorithms will be a recurring theme in these lectures. This issue is sidestepped in [1] by proving a stronger result. Rather than restricting the class of possible algorithms, we allow an *arbitrary* algorithm A and evaluate its running time under a *worst-case ordering* of a given point set S. In our abstract notation, we now think of A as a vector indexed not by (ordered) inputs but by (unordered) point sets, and we define

$$\text{cost}(A, S) = \max_{\pi} \left\{\text{cost}(A, \pi(S))\right\}, \tag{4}$$

where S denotes a point set, $\pi(S)$ denotes the input in which the points of S are specified by the ordering π, and $\text{cost}(A, \pi(S))$ denotes the number of comparisons used by A to correctly solve the input $\pi(S)$. Thus we (conceptually) run the algorithm $|S|!$ different times — once for each ordering of S — and take the largest number of comparisons used. Obviously, for every "order-oblivious" algorithm — one for which (our upper bound on) its running time is order-independent, like the KS algorithm — this extra max operator has no effect. But an algorithm that memorizes an input no longer performs well by this measure. Indeed, the KS algorithm is "order-oblivious instance-optimal" in the following sense.

Theorem 3.3 ([1]) *For every point set S and every (correct) algorithm A,*

$$\text{cost}(A, S) = \Omega\left(\min_{\text{legal } \{S_i\}}\left\{\sum_i |S_i| \log \frac{n}{|S_i|}\right\}\right)$$

where cost(A, S) is defined as in (4).

The proof of Theorem 3.3 is non-trivial and we do not present it here. Very vaguely, what needs to proved is similar to what we accomplished in the proof of Theorem 2.1: for every point set S and the best legal partition $\{S_i\}$ for it, and for every algorithm A, as long as the number of comparisons used by A so far is too small (less than (3)), then we can order the points of S so that A cannot yet be sure of what the correct answer is (i.e., there are two different inputs consistent with the comparisons so far but with different sets of maximal points). See [1] for the details.[3]

[3] See also [1] for a stronger instance optimality result for the KS algorithm, which replaces the right-hand side of (4) by the average of $\text{cost}(A, \pi(S))$ over all orderings π.

3.5 Key Contributions

The 2D Maxima analysis in the Afshani et al. paper had three main points.

1. A novel measure of the "input complexity" of a point set, and a point set-by-point set matching upper bound for the running time of the KS algorithm.[4]
2. A model that sidesteps the annoying issue of "algorithms that memorize the correct answer". An interpretation of the solution in [1] is that only "order-oblivious" algorithms are allowed as competitors, but the actual statement is stronger and more elegant.
3. A highly non-trivial adversary argument showing that, point set-by-point set, the above measure of input complexity lower bounds the number of comparisons of every algorithm under a worst-case (or even average-case) ordering of the point set.

4 Discussion and Take-Home Points

1. Instance optimality is a pointwise guarantee, and as such is essentially the strongest notion of optimality one could hope for. (Of course, the value of the constant factor is important.) Such input-by-input domination is the only uncontroversial way to argue that one algorithm is better than another.
2. Because of its strength, few instance optimality guarantees are known. Finding more, across all sorts of problem domains and cost measures, is a great research direction.
3. The definition of instance optimality can be weakened in various ways to make such guarantees more feasible. Section 3 considered point set-by-point set guarantees, rather that input-by-input guarantees. (More generally, one can have groups of inputs, and ask about group-by-group guarantees.) A different approach is to restrict the competition to a class of "natural" algorithms. We will return to this point in future lectures (see especially Lecture #9).
4. A prerequisite for an instance optimality result is an input-by-input lower bound on the performance of every algorithm. Thus, one should only seek instance optimality results in models where there are lower bound techniques. For running time analyses, this would seem to limit the possibilities primarily to problems solvable in close to linear time (as in Section 3). But in models that admit information-theoretic lower bounds — like in Section 2, or in online algorithms, streaming algorithms, and mechanism design — there would seem to be many opportunities.

[4] In Section 2, our implicit measure of input complexity was the depth to which the threshold algorithm probed into the sorted lists.

References

[1] P. Afshani, J. Barbay, and T. M. Chan. Instance-optimal geometric algorithms. In *Proceedings of the 50th Annual Symposium on Foundations of Computer Science (FOCS)*, pages ??–??, 2009.

[2] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003. Preliminary version in *PODS '01*.

[3] D. G. Kirkpatrick and R. Seidel. Output-size sensitive algorithms for finding maximal vectors. In *Proceedings of the First Annual ACM Symposium on Computational Geometry*, pages 89–96, 1985.