CS120: Intro. to Algorithms and their Limitations Anshu & Hesterberg

Lecture 3: Reductions

Harvard SEAS - Fall 2023 *2023-09-12*

1 Announcements

2 Sender-Receiver Exercise Wrap-Up

We mentioned that there's a lower bound of $\Omega(n \log n)$ for the worst-case runtime of a sorting algorithm *that can only compare keys*. In today's exercise, you saw a sorting algorithm with a runtime of $o(n \log n)$; necessarily, this algorithm does not only compare keys (it treats them as numbers and uses them to index into arrays). The contrast between these highlights the need for a precise model of computation, which we'll see in ≤ 2 weeks.

3 Reductions

3.1 Motivating Problem: Duplicate Search

There are 227 students in CS 120, but when we sent email to the class list for the sender-receiver exercise earlier, we sent 228 emails. We suspected that this was because someone had signed up for the class twice, so we had their email twice. The same person can't be in the SRE twice, so we wanted to find the repeated person.

This gives rise to the following computational problem:

Input	: A list of real numbers $a_0, \ldots, a_{n-1}$
Output	: A duplicate element; that is, a real number a such that there exist $i \neq j$ such that $a_i = a_j = a$.

Computational Problem DuplicateSearch

Note that this is an example of a problem where, for some inputs $x \in \mathcal{I}$, the set of correct answers $f(x)$ is empty, if there are no duplicates. (Maybe Adam's email address was on the list.) If $f(x) = \emptyset$, an algorithm solving DuplicateSearch should return $\perp$.

Using its definition directly, we can solve this problem in time $O(n^2)$. How?

However, we can get a faster algorithm by a *reduction* to sorting.

Proposition 3.1. *There is an algorithm that solves DuplicateSearch for n inputs in time $O(n \log n)$.*

Proof.

We first describe the algorithm.

We now want to prove that our algorithm

1. has the claimed runtime of $O(n \log n)$, and
2. is correct.

- **Correctness:**

- **Runtime:**

□

3.2 Reductions: Formalism

The technique above, to use the solution to one problem to solve another, is so commonly useful that it has a name, *reduction*, and we'll treat reductions more formally:

Definition 3.2. Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ and $\Gamma = (\mathcal{J}, \mathcal{Q}, g)$ be two computational problems. A *reduction* from Π to Γ is an algorithm that solves Π using as a subroutine a(ny) *oracle* that solves Γ.

An *oracle* solving Γ is a function that, given any input $x \in \mathcal{J}$ returns an element of $g(x)$. This is the same as the requirement for an algorithm solving Γ, but without the requirement to have a well-defined "procedure"—the oracle might be an import from a Python library you don't control or a question to the Oracle at Delphi, who always answers questions correctly.

If there exists a reduction from Π to Γ, then we write $\Pi \leq \Gamma$ (read "Pi reduces to Gamma"). If there exists a reduction from Π to Γ which, on inputs (to Π) of size n, takes $O(T(n))$ time (counting each oracle call as *one* time step) and calls the oracle only once on an input (to Γ) of size at most $f(n)$, we write $\Pi \leq_{T,f} \Gamma$.[1]

For example, our proof of Proposition 3.1 gave a reduction from Π = DuplicateSearch to Γ = Sorting that, on a DuplicateSearch input of size n, runs in time $O(n)$ and makes 1 call to the Sorting oracle on an input of size n; that is, DuplicateSearch $\leq_{O(n),n}$ Sorting. This reduction takes time $O(n)$, not $O(n \log n)$, because in reductions, oracle calls are treated as one time step. They allow us to ignore how the oracle can be implemented as an algorithm (or whether it can be even implemented at all). As shown in Lemma 3.3 below, a reduction from Π to Γ can then be *combined* with an algorithm for Γ to obtain an oracle-free algorithm for Π.

Note that if Γ is a computational problem where there can be multiple valid solutions (i.e. $|g(x)| > 1$ for some $x \in \mathcal{J}$), then a valid reduction is required to work correctly for *every* oracle that solves Γ (i.e. no matter which valid solutions it returns).

The use of reductions is mostly described by the following lemma, which we'll return to many times in CS 120:

Lemma 3.3. *Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then:*

1. *If there exists an algorithm solving Γ, then*

[1] One might care about exactly one of the runtime for a reduction and whether it calls its oracle multiple times, but in most of CS 120's reductions we'll either care about both or neither. We might write out in words, e.g. "there is a reduction from Π to Γ that makes 4 oracle calls on inputs of length $O(n)$".

2. *If there does not exist an algorithm solving* Π*, then*

3. *If there exists an algorithm solving* Γ *with runtime* $g(n)$*, and* $\Pi \leq_{T,f} \Gamma$*, then*

4. *If there does not exists an algorithm solving* Π *with runtime* $O(T(n)+g(f(n))$*, and* $\Pi \leq_{T,f} \Gamma$*, then*

Proof.

□

These statements are not true if we flip the direction $\leq$ of the reduction. For instance, very easy problems can reduce to very hard problems (consider a sorting algorithm that first calls an oracle for a very hard problem on the array, then discards the result). But this doesn't imply that sorting is very hard.

For the next month or two of CS 120, we use reductions to show (efficient) solvability of problems, i.e. using Item 1 (or Item 3). Later, we'll use Item 2 to prove that problems are not efficiently solvable, or even entirely unsolvable! *Note that the direction of the reduction (*$\Pi \leq \Gamma$ *vs.* $\Gamma \leq \Pi$*) is crucial!*

3.3 Example Problem: Interval Scheduling

A small public radio station decided to raise money by allowing listeners to purchase segments of airtime during a particular week. However, they now need to check that all of the segments that they sold aren't in conflict with each other; that is, no two segments overlap.

This gives rise to the following computational problem:

Input	: A collection of intervals $[a_0, b_0], \ldots, [a_{n-1}, b_{n-1}]$, where each $a_i, b_i \in \mathbb{R}$ and $a_i \leq b_i$
Output	: YES if the intervals are disjoint from each other (for all $i \neq j$, $[a_i, b_i] \cap [a_j, b_j] = \emptyset$) NO otherwise

Computational Problem IntervalScheduling-Decision

Using its definition directly, we can solve this problem in time $O(n^2)$. How?

However, we can get a faster algorithm by a reduction to sorting.

Proposition 3.4. *IntervalScheduling-Decision* $\leq_{O(n),n}$ *Sorting*

Proof.

□

Corollary 3.5. *There is an algorithm that solves IntervalScheduling-Decision for n intervals in time $O(n \log n)$.*

Proof. □