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

Lecture 3: Reductions

Harvard SEAS - Fall 2022 *2022-09-08*

1 Announcements

- Raise your name placard to ask a question
- Log in to Ed during class — participate in live Q&A
- PS0 due last night
- PS1 out today
- Participation portfolios described on website: keep a portfolio of the best examples of where participation helped advance your classmates' or your own learning. 4 highlight submissions should include 3 examples of meaningful participation, at least one from SRE.
- Handout: Lecture notes 3 (PDF on google spreadsheet)
- Sender–Receiver exercise!

2 Sender-Receiver Exercise Wrap-Up

Note that at the end of the previous lecture, 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.

Fill out the Sender-Receiver Exercise Survey: link on Ed, on Lec 3 Megathread.

3 Loose Ends from Lec 2

4 Reductions

4.1 Motivating Problem: Duplicate Search

There are 270 students in CS 120, but when we sent email to the class list for the sender-receiver exercise earlier today, we sent 271 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?

Check every pair of $i, j \in [n]$ where $i \neq j$. If for any pair, $a_i = a_j$, return a_i. Otherwise, return $\perp$. Since there are $\binom{n}{2} = O(n^2)$ pairs, we have the desired runtime.

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

Proposition 4.1. *There is an algorithm that solves DuplicateSearch for n inputs in time $O(n \log n)$.*

Proof.
We first describe the algorithm.

- Sort the array of inputs, yielding a sorted array S.
- Given S, for each $i \in [n-1]$, check whether $S[i] = S[i+1]$. If so, return $S[i]$.
- If not, return $\perp$.

We now want to prove that our algorithm

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

- **Correctness:** If the input array has a duplicate element $a_i = a_j$, then after sorting, there are two adjacent equal elements in the sorted array. So, the algorithm will return some real number (when the loop reaches i or earlier).

  If the algorithm returns a real number a, it's just checked that two elements of the input array equal a, so a is a duplicate.

  If the input array has no duplicate element, we proved above that we don't return a real number a, so we return $\perp$, as appropriate for arrays with no duplicate elements.

- **Runtime:** We essentially give a linear time algorithm *given the ability to sort an array of length n*. Since we have an algorithm for sorting with runtime $O(n \log n)$, we are done. This is known as a **reduction** from Duplicate Search to Sorting. In more detail:
  - Sorting: $O(n \log n)$.
  - Checking $S[i] == S[i+1]$: $O(n)$.

Hence, we have shown: an $O(n)$-time algortihm for `DuplicateSearch` that makes one "oracle call" to `Sorting` on an array of size n, and we conclude that:

$$Time_{DuplicateSearch}(n) \leq O(n) + Time_{Sorting}(n) = O(n) + \Theta(n \log n) = \Theta(n \log n).$$

□

4.2 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 4.2. 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 4.1 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 4.3 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 4.3. *Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then:*

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

[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 $O(n)$".

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 there does not exist an algorithm solving* Γ *with runtime* $O(g(n))$*.*

Proof.

1. Let A be the algorithm solving Π using an oracle to Γ. By assumption, there exists an algorithm solving Γ, denoted B. We then use B in the place of A's oracle and obtain a standard (oracle-free) algorithm for Π.
2. This is the contrapositive of Item 1.
3. Let A be the algorithm solving Π with runtime $T(n)$ and a call to the oracle to Γ with size $f(n)$. By assumption, there exists an algorithm solving Γ with runtime $g(f(n))$ on inputs of size $f(n)$, denoted B. We then use B in the place of A's oracle and obtain a standard (oracle-free) algorithm for Π, which takes time $T(n)$ for the reduction and $g(f(n))$ for the replaced oracle call.
4. This is the contrapositive of Item 3.

□

Both of 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!*

4.3 Example Problem: Interval Scheduling

The following is a practice problem for reductions. If you're reviewing these notes, you may want to first look at the version with solutions blanked out and try to solve it first. The solution was also discussed at the beginning of lecture on 9/13.

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?

Check every possible pair of intervals. If any pair overlaps, return NO and otherwise return YES. Since there are $\binom{n}{2} = O(n^2)$ pairs, we have the desired runtime.

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

Proposition 4.4. *IntervalScheduling-Decision* $\leq_{O(n),n}$ *Sorting*

Proof.
We first describe the reduction algorithm.

- Form an array of (key, value) pairs where $(K_i, V_i) = (a_i, b_i)$ for all $i \in \{1, \ldots, n\}$.
- Sort this array by start time into a sorted array S, by an oracle call to Sorting.
- Given S, for each pair of adjacent elements $(K'_i, V'_i) = (a'_i, b'_i)$ and $(K'_{i+1}, V'_{i+1}) = (a'_{i+1}, b'_{i+1})$, check if $b'_i < a'_{i+1}$. Intuitively, this means that the left interval ended before the right interval, which indicates that there is no overlap. If this inequality holds for all pairs of adjacent elements, return YES and otherwise return NO.

We now want to prove our reduction algorithm

1. has the desired runtime and call size to the Sorting oracle, and
2. is correct.

- Forming an array in the first step takes time $O(n)$. Searching through it once in the last step takes time $O(n)$. So the total time taken by the reduction is $O(n)$. Also, the array passed to Sorting has size n, as claimed.
- The proof is somewhat tedious, but we can show that if a set of intervals contains no collisions, we will return YES, and if it does contain a collision, we will return NO.

□

Corollary 4.5. *There is an algorithm that solves IntervalScheduling-Decision for n intervals in time $O(n \log n)$.*

Proof. We have shown an $O(n)$-time algortihm for `IS-D` that makes one "oracle call" to `Sorting` on an array of size n, so we conclude that:

$$\text{Time}_{IS\text{-}D}(n) \leq O(n) + \text{Time}_{Sorting}(n) = O(n \log n)$$

by Lemma 3 □