CS1200: Intro. to Algorithms and their Limitations	Anshu & Vadhan

Lecture 3: Reductions

Harvard SEAS - Fall 2024 *2024-09-10*

1 Announcements

- Lecture 2 detailed notes posted.
- Handout: Lecture notes 3 (PDF on Ed)
- Avi Wigderson (Abel Prize '21, Turing Award '23) lecture Friday 3:45pm, in this room "The Value of Errors in Proofs". Highly recommended!
- Sender–Receiver exercise today (partway through class)! Followed by a 5min in-class reflection survey (required for your participation grade).
- Don't burn yourself out on psets. Mistakes are part of learning, hence the possibility of revision videos. On psets 0 and 1, any grade can be revised even up to an R.

2 Loose Ends from Lecture 2

Definition 2.1. Let $h, g : \mathbb{N} \to \mathbb{R}^+$. We say:

- $h = O(g)$ if
- $h = \Omega(g)$ if
  Equivalently:
- $h = \Theta(g)$ if
- $h = o(g)$ if
  Equivalently:
- $h = \omega(g)$ if
  Equivalently:

Informal def: *computational complexity* of a problem Π = smallest possible growth rate of runtime among algorithms that solve Π.

2.1 Computational Complexity of Sorting

Let's analyze the runtime of the sorting algorithms covered so far (`ExhaustiveSearchSort`, `InsertionSort`, `MergeSort`).

$T_{exhaustsort}(n) =$

$T_{insertsort}(n) =$

And when n is a power of 2, $T_{mergesort}$ satisfies the following recurrence:

$T_{mergesort}(n) \leq$

From this, we can derive that when n is a power of 2,

$T_{mergesort}(n) =$

And the same holds true when n is not a power of 2 by rounding n up to the next-larger power of 2.

Exercise 2.2. Order $T_{exhaustsort}, T_{insertsort}, T_{mergesort}$ from fastest to slowest, i.e. T_0, T_1, T_2 such that $T_0 = o(T_1)$ and $T_1 = o(T_2)$.

In the detailed lecture notes for Lecture 2, you can find a proof (optional to read) that `MergeSort` is *asymptotically optimal* among *comparison-based* sorting algorithms:

Theorem 2.3. *If A is a comparison-based algorithm that correctly solves the sorting problem on arrays of length n in time $T(n)$, then $T(n) = \Omega(n \log n)$. Moreover, this lower bound holds even if the keys are restricted to be elements of $[n]$ and the values are all empty.*

We will be interested in three very coarse categories of running time:

(at most) exponential time $T(n) = 2^{n^{O(1)}}$ (slow)

(at most) polynomial time $T(n) = n^{O(1)}$ (reasonably efficient)

(at most) nearly linear time $T(n) = O(n \log n)$ or $T(n) = O(n)$ (fast)

3 Reductions

3.1 Motivating 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
Output	**:** YES if
	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 3.1. *There is an algorithm that solves IntervalScheduling-Decision for n intervals in time $O(n \log n)$.*

Proof.

$\square$

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 (reductions)**.** Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ and $\Gamma = (\mathcal{J}, \mathcal{P}, g)$ 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 $g(x)$, or $\perp$ if no such element exists.

Definition 3.3 (notation and efficiency for reductions)**.** If there exists a reduction from Π to Γ, then we write $\Pi \leq \Gamma$. 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 $h(n)$, we write $\Pi \leq_{T,h} \Gamma$. If there is a reduction from Π to Γ that makes at most $q(n)$ oracle calls of size at most $h(n)$, we write $\Pi \leq_{T,q\times h} \Gamma$.

For example, our proof of Proposition 3.1 implicitly showed:

Proposition 3.4. *IntervalScheduling-Decision* $\leq_{\_,\_}$ *Sorting.*

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

Lemma 3.5. *Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then:*

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

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

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

4. *If there does not exist an algorithm solving Π with runtime $T(n) + O(q(n) \cdot R(h(n)))$, and $\Pi \leq_{T,h} \Gamma$, then*

Proof.

$\square$

For the next month or two of the course, 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!*