CS1200: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 4: Reductions, Static Data Structures

Harvard SEAS - Fall 2025 *2025-09-11*

1 Announcements

- Salil's upcoming OH: today 3-3:45pm SEC 3.327; Mon 2-2:45pm Zoom.
- SRE 2 moved to **Thursday** 9/16. Some notes from your reflection surveys:
  - 93% of you found the exercise useful (slightly more for receivers than for senders).
  - Concrete examples are very helpful. If you are a sender, we recommend preparing concrete examples yourself in advance!
  - Most of you felt your partner did well, but still do reflect on what both parties can do to make the interaction most valuable as you prepare for future SREs.
  - Time allocated was appropriate, with some variance in feedback.
- Reminder to ask questions or provide feedback on the textbook in *Perusall*. Link on course website (or Canvas).

2 Recommended Reading

- Hesterberg–Vadhan, Chapter 3, Section 4.1–4.2
- CLRS Chapter 10
- Roughgarden II, Sec. 10.0–10.1, 11.1
- CS50 Week 5: https://cs50.harvard.edu/x/2022/weeks/5/

3 Interval Scheduling

Consider the following scenario: 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.

When confronted with an informally described computational problem as above, our first task as algorithmicists is to figure out how to model it mathematically, so that we make our task precise and can match our algorithmic toolkit to it. In this modeling, we want to be sure to capture all of the essential details, while abstracting away inessential ones.

Input: A collection of **Output:** YES if NO otherwise

Computational Problem INTERVALSCHEDULING–DECISION

Notice that this formulation as a computational problem has abstracted away lots of inessential details of our original problem, like the fact that it involves a radio station and segments of time. We can apply it equally well to the problem of allocating segments of sidewalk to food vendors along the route of the Boston Marathon. This points to another benefit of abstraction and mathematically modelling; it allows one solution (like an algorithm) to apply to many different problems.

There is a simple algorithm to solve INTERVALSCHEDULING–DECISION in $O(n^2)$ runtime.

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

Proposition .1. *There is an algorithm that solves* INTERVALSCHEDULING–DECISION *for n intervals in time $O(n \log n)$.*

Proof. We first describe the algorithm.

```
IntervalSchedulingViaSorting(C):
Input   : A collection C of intervals [a_0, b_0], ..., [a_{n-1}, b_{n-1}], where each a_i, b_i ∈ ℝ
          and a_i ≤ b_i
Output  : YES if intervals are disjoint, NO otherwise.
0 Set A =
1 Let ((a'_0, b'_0), ..., (a'_{n-1}, b'_{n-1})) =;
2 foreach i = 1, ..., n − 1 do
    |
3 return YES;
```

Algorithm .2: `IntervalSchedulingViaSorting()`

We now want to prove that `IntervalSchedulingViaSorting` has the claimed runtime of $O(n \log n)$ and is correct.

a: **Runtime analysis:**

b: **Proof of correctness:**

$\square$

Question: Define IntervalScheduling–Decision–OnFiniteUniverse to be a variant of IntervalScheduling–Decision where we are also given a universe size $U \in \mathbb{N}$ and the interval endpoints a_i, b_i are constrained to lie in $[U]$. Can you think of an algorithm for solving IntervalScheduling–Decision–OnFiniteUniverse in time $O(n + U)$?

Answer:

4 Reductions: Formalism

In the example above, note that it was not crucial for the correctness of the `IntervalSchedulingViaSorting` algorithm that we used `MergeSort`. *Any* algorithm that correctly solves Sorting would do; we do not need to know how a solution to Sorting is implemented. This is why we called this a *reduction* from the IntervalScheduling–Decision problem to the Sorting problem. Reductions are a powerful tool and will be a running theme throughout the rest of the course, so we now introduce terminology and notation to treat them more formally:

Definition .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 *query* $y \in \mathcal{J}$ returns an element of $g(y)$, or $\perp$ if no such element exists.

We can visualize a reduction as follows:

Next we describe our notation for reductions and some notions of their efficiency:

Definition .3.

- 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 $\Pi \leq_{T,q\otimes h} \Gamma$.

For example, our proof of Proposition .1 implicitly showed:

Proposition .4. *IntervalScheduling-Decision* $\leq_{\_,\_}$ Sorting.

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

Lemma .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 $T_A(n)$, and $\Pi \leq_{T_R,q\otimes h} \Gamma$, then*

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

Using Proposition .4 together with Item 3 with $T_R(n) = O(n)$, $q(n) = 1$, $h(n) = n$, and $T_A(n) = O(n \log n)$ yields Proposition .1 as a corollary.

Proof of Lemma .5.

□

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 ($\Pi \le \Gamma$ vs. $\Gamma \le \Pi$) is crucial!*

5 Static Data Structures

Q: Suppose we have already verified that an instance of IntervalScheduling–Decision has no conflicts using Algorithm .2, and another interval $[a^*, b^*]$ is given to us (e.g. another listener tries to buy some airtime). Do we need to spend time $O(n \log n)$ again to decide whether or not we can fit that interval into the schedule?

A:

The sorted array in the above solution is an example of a (static) *data structure*: a way of encoding our input data that allows us to answer certain queries about the data efficiently. As with computational problems, we'll want to distinguish the problem that data structures are supposed to solve from the way in which we solve them. Let's begin by formalizing the former:

Definition .6. A *static abstract data type* is a quadruple $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, f)$ where:

- $\mathcal{I}$ is a (typically-infinite) set of possible inputs x, and $\mathcal{O}$ is a (sometimes-infinite) set of possible outputs y.
- $\mathcal{Q}$ is
- for every $x \in \mathcal{I}$ and $q \in \mathcal{Q}$,

Given such an abstract data type, we want to design efficient algorithms that preprocess the input x into an encoding that allows us to quickly answer queries q that come later. For example, to be able to determine whether a new interval conflicts with one of the original ones, it suffices to solve the following data-structure problem.

Input: An array of key-value pairs $x = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, with each $K_i \in \mathbb{R}$

Queries:

- `search`(K) for $K \in \mathbb{R}$:
- `predecessor`(K) for $K \in \mathbb{R}$:
- `successor`(K) for $K \in \mathbb{R}$:

Abstract Data Type StaticPredecessors+Successors

To formalize these three types of queries using Definition .6, we can take

$$\mathcal{Q} =$$

Note that both types of queries may have no valid answers, or may have multiple answers. (Why?) If we removed the `predecessor` queries and kept only `search` queries, we would have the (static) DICTIONARY abstract data type, which we will study in a couple of weeks.

Now that we have seen how to formalize the problems we want to solve with data structures, we can turn to formalizing what it means to solve them:

Definition .7. Let $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, f)$ be a static abstract data type. A *(static) data structure* for Π is a pair of algorithms (Preprocess, Eval) such that

Here's an illustration:

Sometimes (e.g. in the study of programming languages) data structures are referred to as *implementations* of an abstract data type.

Our goal is for Eval to run as fast as possible. (As we'll see in examples below, sometimes there are multiple types of queries, in which case we often separately measure the running time of each type.) Secondarily, we would like Preprocess to also be reasonably efficient and to minimize the memory usage of the data structure Preprocess(x).

Note that there is no pre-specified problem that the algorithms Preprocess and Eval are required to solve individually; we only care that *together* they correctly answer queries. Thus, a big part of the creativity in designing data structures is figuring out what the form of Preprocess(x) should be. Our first example (from the discussion above and encapsulated in the theorem below) takes it to be a sorted array, but we will see other possibilities in subsequent sections (like binary search trees and hash tables).

Theorem .8. STATICPREDECESSORS+SUCCESSORS *has a data structure in which:*

- $\mathrm{Eval}(x', (\texttt{search}, K))$, $\mathrm{Eval}(x', (\texttt{predecessor}, K))$, $\mathrm{Eval}(x', (\texttt{successor}, K))$ *all take time*
- Preprocess(x) *takes time*
- Preprocess(x) *has size*

Proof.

- Preprocess(x):
- $\mathrm{Eval}(x', (\texttt{search}, K))$:
- $\mathrm{Eval}(x', (\texttt{predecessor}, K))$:

- Eval(x', (`successor`, K)):

$\square$

PREDECESSORS+SUCCESSORS have many applications. They enable one to perform a RANGESELECT—selecting all of the elements of a dataset whose keys fall within a given range. This is a fundamental operation across many applications and systems, including relational databases (e.g. a university database selecting all CS alumni who graduated in the 1990s), NoSQL data stores (e.g. selecting all users of a social network within a given age range), and ML systems (e.g. filtering intermediate results during neural network training sessions).

Other Queries. `search`, `predecessor`, and `successor` are only a few of the kinds of queries that are useful to ask on a dataset of key-value pairs. Some additional examples are:

1. `min`(K): return some (K_i, V_i) such that $K_i = \min\{K_j : j \in [n]\}$.
2. `rank`(K): return the *number* of pairs $(K_i, V_i) \in S$ such that $K_i < K$.

Q: How long does it take to answer these queries (in the worst case) if we encode our data using a sorted array?

In the next class, we will consider *dynamic* data structures, where we also need to be able perform updates such as the following:

1. `insert`(K, V): add a key–value pair (K, V) to those being stored.
2. `delete`(K): remove a key–value pair (K_i, V_i) with $K = K_i$ from those being stored.

Q: How long does it take to perform these updates if we encode our data using a sorted array?