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

Lecture 4: Reductions, Static Data Structures

Harvard SEAS - Fall 2024 *2024-09-12*

1 Announcements

- Please fill out PS0 feedback survey.
- Handout: Lecture notes 4 (PDF on course schedule)
- Avi Wigderson (Abel Prize '21, Turing Award '23) lecture Friday 3:45pm, in this room "The Value of Errors in Proofs". Highly recommended!
- Salil OH: after class today in SEC 3.327, Anurag OH: Fri 1:30-2:30 on Zoom.
- Anurag will be available to support DCE students for the SRE in the Study Lounge, Fri 2:30-3pm.
- Problem Set 1 to be posted shortly (due Wed 9/18).
- Next SRE: Thursday 9/19.

2 Recommended Reading

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

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

There is a simple algorithm to solve this problem in $O(n^2)$ runtime.

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. We first describe the algorithm.

```
1 IntervalSchedule(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.
2 Set A =


3 A' =;
4 foreach i = 1, ..., n − 1 do
  |
  |
5 return YES
```

Algorithm 1: FastIntervalScheduling

The algorithm forms an array A of key-value pairs from the collection C of intervals. The keys are the starts of the intervals a_i's and the values are the ends of the intervals b_i's. Then it invokes MergeSort on this array, which outputs a valid sort A' of A. Let $A' = ((a'_0, b'_0), \ldots, (a'_{n-1}, b'_{n-1}))$. Then, the algorithm checks if for each $i = 1, \ldots, n-1$, we have $a'_i > b'_{i-1}$. 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 that FastIntervalScheduling has the desired runtime, and is correct.

a: **Runtime analysis:**

b: **Proof of correctness**

□

Question: Define IntervalScheduling-Decision-OnFiniteUniverse to be a variant of IntervalScheduling-Decision where we are 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:

3.2 Reductions: Formalism

In the example above, note that, for the correctness of the IntervalScheduling-Decision algorithm, it was not crucial that we used MergeSort. *Any* algorithm that correctly solves Sorting would do; we do not need to know how it 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 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 *query* $y \in \mathcal{J}$ returns an element of $g(y)$, or $\perp$ if no such element exists.

Definition 3.3. Here we describe our notations and notions of 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*

Using Proposition 3.4 together with Item 3 with $T(n) = O(n)$, $q(n) = 1$, $h(n) = n$, and $R(n) = O(n \log n)$ yields Proposition 3.1 as a corollary. Reductions can also be used for two-parameter problems, for example SortingOnFiniteUniverse from SRE 1, where we have two size parameters, the length n of the array and the key-universe size U. Recall that we discussed an algorithm for IntervalScheduling-Decision-OnFiniteUniverse that used the Singleton Bucket Sort algorithm (which solves the SortingOnFiniteUniverse problem). By inspection of this algorithm, we actually showed the following reduction:

$$\text{IntervalScheduling-Decision-OnFiniteUniverse} \leq_{T(n,U),h(n,U)} \text{SortingOnFiniteUniverse}.$$

for $T(n, U) = O(n)$ and $h(n, U) = (n, U)$. The $O(n+U)$ algorithm for IntervalScheduling-Decision-OnFiniteUniverse that we discussed earlier can be viewed as coming from a natural two-parameter extension of Item 3.

Proof of Lemma 3.5.

□

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

4 Static Data Structures

Q: Suppose we have already solved IntervalScheduling using the algorithm of Proposition 3.1, 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 we can fit that interval in?

A:

The sorted array in the above solution is an example of a static data structure. Let's abstract what static data structures are supposed to do.

Definition 4.1. A *static data structure problem* 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 $q \in \mathcal{Q}$,

For such a data-structure problem, we want to design efficient algorithms that preprocess the input x into a data structure that allows for quickly answering 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 $K \in \mathbb{R}$:	
• `next-smaller`(K) for $K \in \mathbb{R}$:	

Data-Structure Problem Static Predecessors

To formalize these two types of queries using Definition 4.1, we can take

$$\mathcal{Q} =$$

Note that both types of queries may have no valid solution, or may have multiple solutions. (Why?) If we removed the `next-smaller` queries and only kept `search` queries, we would have the (static) *Dictionary* data structure problem, which we will study in a couple of weeks.

Using Static Predecessors for Interval Scheduling queries: Given an Interval Scheduling instance $[a_0, b_0], \ldots, [a_{n-1}, b_{n-1}]$, we check for conflicts in the input, and if there are none, we build a Static Predecessors data structure using input keys $K_i = a_i$ and values $V_i = b_i$.

Then, given an interval $[a^*, b^*]$ we can check whether it conflicts with any of the input intervals as follows:

-
-
-

Remark. The terminology *predecessor* comes from the fact that when K is a key in the input array x, then `next-smaller`(K) should return the *predecessor* of K.

Predecessor Data Structures (and the equivalent Successor Data Structures) have many applications. They enable one to perform a "range select" — selecting all of the elements of a dataset whose keys fall within a given range. This is a fundamental operation across many application and systems, including relational databases (e.g. a university database selecting all CS alumni who graduated in the 1990's), 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).

Definition 4.2. A *solution* to a (static) data structure problem $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, f)$ is a pair of algorithms (Preprocess, Eval) such that

Sometimes (e.g. in CS51 and in the study of Programming Languages) data structure problems are referred to as *abstract data types* and solutions are referred to as *implementations*.

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 solving data structure problems is figuring out what the form of Preprocess(x) should be. Our first example (from today) takes it to be a sorted array, but we will see other possibilities in next week's class (like binary search trees and hash tables).

Let's formalize our solution to the Static Predecessor Problem:

Theorem 4.3. *Static Predecessors+Successors has a solution in which:*

- Eval$(x', (\texttt{search}, K))$, Eval$(x', (\texttt{next-smaller}, K))$, Eval$(x', (\texttt{next-larger}, K))$ *all take time*
- Preprocess(x) *takes time*
- Preprocess(x) *has size*

Proof.
- Preprocess(x):
- Eval$(x', (\texttt{search}, K))$:
- Eval$(x', (\texttt{next-smaller}, K))$:

□

In the lecture notes, you can see some example Python code, showing how the mathematical formalism of data structures corresponds quite nicely to implementation with object-oriented programming.

5 Food for Thought

Next time, we will study *dynamic data structures*, which allow update operations (such as inserting and deleting key-value pairs) in addition to queries, and we ideally want these updates to be very fast.

Suppose we use a sorted array to implement a data structure storing a dynamically changing multiset S of key-value pairs with insertions and deletions. How efficiently can you perform each of the following operations (in the worst case), as a function of the current number n of elements of S? Possible answers are $O(1)$, $O(\log n)$, $O(n)$, and "other".

1. `insert`(K, V)
2. `delete`(K, V)
3. `min`(): return the smallest key K in S.
4. `rank`(K): return the *number* of pairs $(K', V') \in S$ such that $K' < K$.