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

In Lecture 3, We discussed three very coarse notions of running times -

(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)

For the Sorting problem, we found algorithms in each category - ExhaustiveSearch Sort, Insertion Sort and Merge Sort. Since Merge Sort solves the Sorting problem fast, we can - loosely speaking - say that Sorting is an 'easy' computational problem. There might be 'hard' computational problems out there, for which the best algorithms solving them are slow. But how do we know which problems are easy and which are hard? This lecture will introduce the technique of reduction, which is a powerful method to help with this question.

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 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

There is a simple algorithm to solve this problem in $O(n^2)$ runtime. It proceeds by checking 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 and checking overlap between pairs is an $O(1)$ time operation, we have the desired 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 = ((a_0, b_0), ..., (a_{n-1}, b_{n-1})) /* Form an array A of Key-Value pairs */
3 A' = MergeSort(A);
4 foreach i = 1, ..., n − 1 do
5 |   if A'[i][0] ≤ A'[i − 1][1] then return NO;
6 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:**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 the aforementioned reduction from interval scheduling to sorting. In more detail:

- Forming array (Line 2): $O(n)$.

- MergeSort (Line 3): $O(n \log n)$.
- Forloop and comparison on Lines 4-5: $O(n)$.

Thus, in total our runtime is $O(n \log n)$.

b: **Proof of correctness** If the input intervals are all disjoint, then we argue that in the sorted array $A' = ((a'_0, b'_0), \ldots, (a'_{n-1}, b'_{n-1}))$ we will have $a'_i > b'_{i-1}$ for all i, and thus the algorithm will output YES in Line 6. Suppose for contradiction that $b'_{i-1} \geq a'_i$ for some i. Then, since $a'_{i-1} \leq b'_{i-1}$ (A' is a valid sorting of A), the interval $[a'_{i-1}, b'_{i-1}]$ contains a'_i and hence intersects with the interval $[a'_i, b'_i]$, which contradicts the fact that the input intervals are all disjoint.

Conversely, suppose that the algorithm says YES. Then for all i, we have $a'_i > b'_{i-1}$. Then, using our input assumption that $a'_i \leq b'_i$ for all i, we have

$$a'_0 \leq b'_0 < a'_1 \leq b'_1 < \cdots a'_{n-1} \leq b'_{n-1},$$

and we can see by inspection that all of the intervals are disjoint.

□

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: The algorithm is very similar to FastIntervalScheduling, where we replace Merge Sort with Singleton Bucket Sort on the universe $[U]$, which has runtime of $O(n + U)$.

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.

Note that the requirement on an oracle solving Γ is the same as the requirement for an algorithm solving Γ, but without the requirement to have a well-defined "procedure." It's like a *hypothetical* algorithm, which you might instantiate with an import from a Python library you don't control or questions to the mythological Oracle at Delphi, who always answers questions correctly.

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 $\Pi \leq \Gamma$(read as "Pi reduces to Gamma").

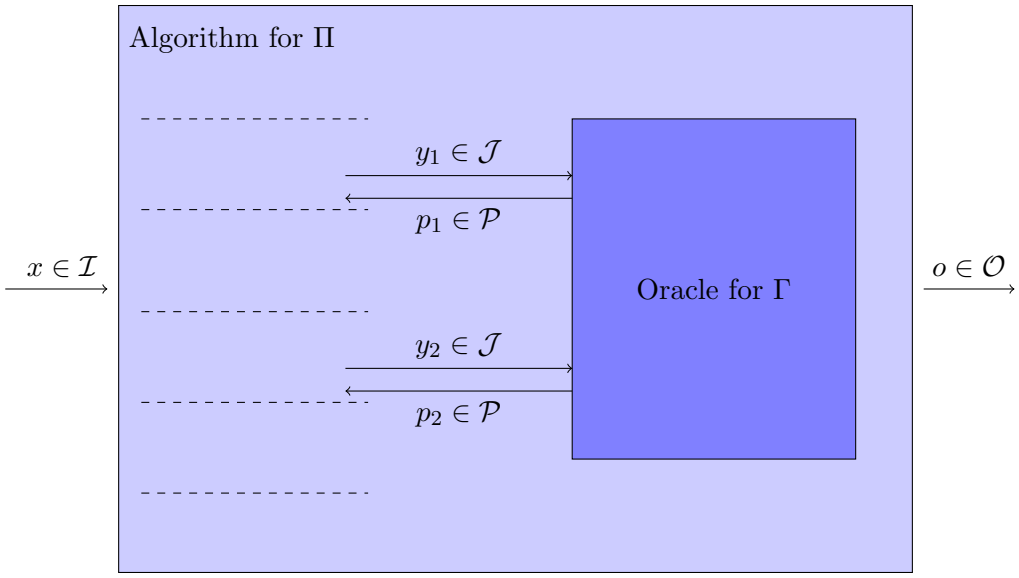

Figure 1: Reduction algorithm solves Π by including lines of code (dashed lines) that call an oracle that solves Γ.

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

Sometimes it is useful to distinguish the sizes of the different oracle calls, in which case we will just write it out in words, e.g. "the reduction makes n oracle calls of size $O(n)$ and 1 oracle call of size $O(n^2)$."

For example, our proof of Proposition 3.1 gave a reduction from Π = IntervalScheduling-Decision to Γ = Sorting that runs in time $O(n)$ and makes 1 call to the Sorting oracle on an input of size n. That is:

Proposition 3.4. *IntervalScheduling-Decision* $\leq_{O(n),n}$ *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 3.5 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 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 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* $R(n)$*, and* $\Pi \leq_{T,q\times h} \Gamma$*, then there exists an algorithm solving* Π *with runtime* $T(n) + O(q(n) \cdot R(h(n)))$*.*

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

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 `SingletonBucketSort` 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.

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 $T(n)$, making at most $q(n)$ calls to the Γ oracle of size at most $h(n)$. By assumption, there exists an algorithm solving Γ with runtime $R(h(n))$ on inputs of size $h(n)$, 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 $O(R(h(n)))$ for each of the $q(n)$ replaced oracle calls, allowing a $O(\cdot)$ overhead for passing inputs and outputs between the subroutine for B and the algorithm solving A.

4. This is the contrapositive of Item 3.

□

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 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: We can use binary search to find where $[a', b']$ would be in the sorted array and check for conflicts with the adjacent time slots. Using binary search on a sorted array takes $O(\log n)$ time.

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 a set of *queries*, and
- for every $x \in \mathcal{I}$ and $q \in \mathcal{Q}$, $f(x, q) \subseteq \mathcal{O}$ is a set of *valid answers* (or *valid outputs*).

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 $K \in \mathbb{R}$: output some (K_i, V_i) such that $K_i = K$.
- `next-smaller`(K) for $K \in \mathbb{R}$: output some (K_i, V_i) such that $K_i = \max\{K_j : K_j < K\}$.

Data-Structure Problem Static Predecessors

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

$$\mathcal{Q} = \{(\texttt{search}, K) : K \in \mathbb{R}\} \cup \{(\texttt{next-smaller}, K) : K \in \mathbb{R}\}$$

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: We can use a solution to Static Predecessors to solve our problem about Interval Scheduling queries, but we also need to support `next-larger` queries, defined analogously to `next-smaller` queries:

- `next-larger`(K) for $K \in \mathbb{R}$: output (K_i, V_i) such that $K_i = \min\{K_j : K_j > K\}$.

A solution to this *Static Successors* problem can be obtained from a solution to Static Predecessors (how?), though typical solutions for one will easily give a solution to the other at the same time (so we get a single data structure that simultaneously supports `next-smaller` and `next-larger` queries) without doubling the storage.

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+Successors 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:

- Call `search`(a^*) and check that it returns $\perp$.
- Call `next-smaller`(a^*), which returns some (a_i, b_i). Check that $a^* > b_i$.
- Call `next-larger`(a^*), which returns some (a_j, b_j). Check that $b^* < a_j$.

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. The CLRS textbook only defines the predecessor problem where the query is a key of already in the set. However, the more general formulation we have given (where K can be any real number) is more standard and more useful.

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

- for all $x \in \mathcal{I}$ and $q \in \mathcal{Q}$ such that $f(x, q) \neq \emptyset$, we have Eval(Preprocess$(x), q) \in f(x, q)$, and
- there is a special output $\perp \notin \mathcal{O}$ such that for all $x \in \mathcal{I}$ and $q \in \mathcal{Q}$ such that $f(x, q) = \emptyset$, we have Eval(Preprocess$(x), q) = \perp$.

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* $O(\log n)$
- Preprocess(x) *takes time* $O(n \log n)$
- Preprocess(x) *has size* $O(n)$

Proof.

- Preprocess(x): Sort the array x of key-value pairs in $O(n \log n)$ time to obtain a sorted array $x' = ((K'_0, V'_0), \ldots, (K'_{n-1}, V'_{n-1}))$.
- Eval($x', (\texttt{search}, K)$): Use binary search to find $i \in [n]$ such that $K'_i = K$ in time $O(\log n)$ and return (K'_i, V'_i). If no such i exists, return $\perp$.
- Eval($x', (\texttt{next-smaller}, K)$): Use binary search to find $i = \max\{j : K'_j < K\}$ in time $O(\log n)$ and return (K'_i, V'_i). If no such i exists (namely, if $K'_0 \geq K$), return $\perp$.

□

5 Implementing Data Structures

This will not be covered in class, but you may find it useful to see how the mathematical descriptions above translate into concrete code.

As you saw on Problem Set 0, we can represent data structures in Python by using Python `class`es similarly to C `struct`s. We can attach *methods* that implement the Preprocess and Eval functions of a data-structure solution. The implementation details of these methods (as well as the private attributes) can be hidden from a user of the class, creating an "abstraction barrier" that allows the user to focus only on the data-structure problem that it solves, without concern for the particular solution.[1] For example:

```
class StaticPredecessor:

    def __init__(self, array: list): # preprocessing method
        # Python's built-in sorting doesn't take keys as explicit inputs,
        # but asks the user to specify a function that defines the keys
        self.sorted_array = sorted(array) # by default, sorts by the first element of the tuple

    def search(self, K: float):
        """
        Binary search to find the smallest element in the sorted array that
        is equal to K. Returns None if no such element exists.
```

[1] This is the notion of an "abstract data type" that some of you may have seen in CS51. It is also similar to how our notion of *reduction* $\Pi \leq \Gamma$ above abstracts away how the problem Γ is solved when using it to solve Π.

```
        Assumes data is sorted in ascending order.

        """
        left = 0
        right = len(self.sorted_array) - 1
        mid = left

        while left <= right:
            mid = (left + right) // 2
            mid_element = self.sorted_array[mid][0]
            # If K is greater, ignore left half
            if mid_element < K:
                left = mid + 1

            # If K is smaller, ignore right half
            elif mid_element > K:
                right = mid - 1

            else: # K is equal to element at ‘mid‘
                return self.sorted_array[mid]

        # if no solution is found
        return None
    def next_smaller(self, K: float):
        # not implemented
        pass

    def next_larger(self, K: float):
        # not implemented
        pass


data = [(5,"a"),(3,"b"),(7,"c"),(2,"d")]

MyDS = StaticPredecessor(data)  # runs __init
MyDS.search(3) # should return (3,"b")
MyDS.search(4) # should return None
```

Note: Despite the name, a Python `list` is implemented as an array rather than as a linked list, this allows it to have O(1) look up time.

6 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 set 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 $K' < K$.