CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 3: Data Structures

Harvard SEAS - Fall 2021 *Sept. 9, 2021*

1 Announcements

- First active learning excercise today!
- PS1 posted - goal is to be less time consuming than PS0
- Sections begin today
- Free SEC coffee until 10am!
- Video of the comparison based LB is on canvas
- PS0 Feedback Form is available at https://tiny.cc/ps0feedback

2 Recommended Reading

- CLRS 3e, Chapter 10
- Roughgarden II, Sec 10.1, 11.1
- CS50 Week 5: https://cs50.harvard.edu/x/2021/weeks/5/

3 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

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 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.

- Form an array of (item, key) pairs where $(K_i, I_i) = (a_i, b_i)$ for all $i \in \{1, \ldots, n\}$.
- Sort this array by start time into a sorted array S.
- Given S, for each pair of adjacent elements $(K'_i, I'_i) = (a'_i, b'_i)$ and $(K'_{i+1}, I'_{i+1}) = (a'_{i+1}, b'_{i+1})$, check if $I < K'$, i.e. $b'_i < a'_{i+1}$. 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 algorithm A: has the desired runtime, and B: is correct.

a: 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 interval scheduling to sorting. In more detail:

- Forming array: $O(n)$.
- Sorting: $O(n \log n)$.
- Checking $b_i < a_{i+1}$: $O(n)$.

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

$$Time_{IS-D}(n) \leq O(n) + Time_{Sorting}(n) = O(n \log n).$$

b: 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.

□

4 Static Data Structures

Suppose we've already solved the IntervalScheduling-Decision() problem, and determined that the intervals are all disjoint. Now another listener comes along and wants to purchase another interval $[a_n, b_n]$. Do we need to spend time $O(n \log n)$ again to decide whether we can make the sale?

Definition 4.1. A *(static) data structure problem* Π is given by:

- a (typically infinite) set $\mathcal{I}$ of *inputs* (or *instances*),
- a set $\mathcal{Q}$ of *queries*, and
- for every $x \in \mathcal{I}$ and $q \in \mathcal{Q}$, a set $f(x, q)$ of *valid answers*.

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 $x = ((I_0, K_0), \ldots, (I_{n-1}, K_{n-1}))$, with each $K_i \in \mathbb{R}$
Queries	: For any threshold $q \in \mathbb{R}$, return (I_i, K_i) such that $K_i = \max\{K_j : K_j \leq q\}$

Data-Structure Problem Static Predecessors

Using Static Predecessors for Interval Scheduling:

Many other applications: Predecessor Data Structures (and the equivalent Successor Data Structures) 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. For a (static) data structure problem $\Pi = (\mathcal{I}, \mathcal{Q}, f)$, a *solution* is a pair of algorithms (Preprocess, Eval) such that for all $x \in \mathcal{I}$ and $q \in \mathcal{Q}$, we have Eval(Preprocess$(x), q) = f(x, q)$.

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).

A solution to the Static Predecessor Problem:

- Preprocess(x): Sort the array x in $O(n \log n)$ time to obtain a sorted array x'.
- Eval(x', q): Search for the largest predecessor of q in the sorted array x' in $O(\log n)$ time.

Implementing Static Data Structures. As you saw on Problem Set 0, we can represent data structures in Python by using Python `class`es similarly to C `struct`s. However, 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. (This is the notion of an "abstract data type" that some of you may have seen in CS51.) For example:

```
class StaticPredecessor:
      def __init__(self,A: 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
```

```
    def extractkey(pair): return pair[1]
    self.sortedarray=sorted(A,key=extractkey)

  def eval(q: float):
    # put code for binary search here
    # ...
    return pair

MyDS = StaticPredecessor([("a",5),("b",3),("c",7),("d",2)])  # runs __init

MyDS.eval(4) # should return ("b",3)
```

Note: Despite the name, a Python `list` is implemented as an array rather than as a linked list.