CS1200: Intro. to Algorithms and their Limitations Prof. Anurag Anshu

Lecture 1: Sorting

Harvard SEAS - Spring 2026 *2026-01-26*

1 Announcements

- Watch course overview video (from last year's offering) and *read syllabus carefully* if you haven't already done so.
- Problem set 0 is available; due coming Saturday. It is required and substantial, so start early!

2 Recommended Reading

Throughout the course, the readings other than Hesterberg–Vadhan are optional, meant as supplements if you find them useful for a different presentation of the content or more details/examples/exercises.

- Hesterberg–Vadhan, Chapters 1.1-1.5 (Sorting), on Perusall.
- Cormen–Leiserson–Rivest–Stein Chapter 2
- Roughgarden I, Sections 1.4 and 1.5

3 Motivation

Unit 1: storing and searching data. Vast applicability, and clean setting to develop skills:

- How to mathematically *abstract* the *computational problems* that we want to solve with algorithms.
- How to *prove* that an algorithm correctly *solves* a given computational problem.
- How to *analyze and compare* the *efficiency* of different algorithms.
- How to *formalize* exactly what algorithms are and what they can and cannot do through precise *computational models.*

As a concrete motivation for data management algorithms, let us consider the problem of web search, which is one of the most remarkable achievements of algorithms at a massive scale. The World Wide Web consists of billions of web pages and yet search engines are able to answer queries in a fraction of a second. How is this possible?

Let's consider a simplified description of Google's original search algorithm from 1998 (which has evolved substantially since then):

1. Pagerank Calculation: For every URL on the entire World Wide Web ($\forall url \in \mathrm{WWW}$), calculate its *PageRank*, $\mathrm{PR}(url) \in [0,1]$.

2. KEYWORD SEARCH: Given a search keyword w, let S_w be the set of all webpages containing w. That is, compute $S_w = \{url \in \text{WWW} : w \text{ is contained on the webpage at } url\}$.

3. SORTING: Return the list of URLs in S_w, sorted in decreasing order of their pagerank.

The definition and calculation of PageRanks (Step 1) was the biggest innovation in Google's search, and is the most computationally intensive of these steps. However, it can be done offline, with periodic updates, rather than needing to be done in real-time response to an individual search query. Pageranks are outside the scope of CS 1200, but you can learn more about them in courses like CS 2241 (Algorithms at the End of the Wire) and CS 2252 (Spectral Graph Theory in Computer Science). KEYWORD SEARCH (Step 2) can be done by creating a *trie* data structure for each webpage, also offline. Tries are covered in introductory programming and data structure texts.

Our focus here is SORTING (Step 3), which needs to be extremely fast in response to real-time queries, and operates on a massive scale (e.g. millions of pages).

4 The Sorting Problem

Representing data items as key-value pairs (K, V), we can define the sorting problem as follows:

Input: An array A of key-value pairs $((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each key $K_i \in \mathbb{R}$.[a]

Output: An array A' of key-value pairs $((K'_0, V'_0), \ldots, (K'_{n-1}, V'_{n-1}))$ that is a *valid sorting* of A. That is, A' should be:

1. sorted by key values, i.e. $K'_0 \leq K'_1 \leq \cdots \leq K'_{n-1}$. and
2. a reordering of A, i.e. $\exists$ a permutation $\pi : [n] \to [n]$ such that $(K'_i, V'_i) = (K_{\pi(i)}, V_{\pi(i)})$
for $i = 0, \ldots, n-1$.

[a]When we introduce computational models later in the course, we'll stop working with arbitrary real numbers, and then keys could be restricted to integers or rational numbers.

Computational Problem SORTING

Above and throughout this text, $[n]$ denotes the set of numbers $\{0, \ldots, n-1\}$. In pure math (e.g. combinatorics), it is more standard for $[n]$ to be the set $\{1, \ldots, n\}$, but being computer scientists, we like to index starting at 0. Similarly, for us, the natural numbers are $\mathbb{N} = \{0, 1, 2, \ldots\}$.

To apply the abstract definition of SORTING to the web search problem, we can set:

- $K_i =$
- $V_i =$

Abstraction $\to$ many other applications! Database systems (both Relational and NoSQL), machine learning systems, ranking cat photos by cuteness, …

Note that some inputs to Sorting have multiple valid outputs. Specifically, if there are repetitions among the input keys (e.g. if the key is a person's age in years in a dataset of 1000 people), then there are multiple reorderings that all yield valid sortings.

In the subsequent sections, we will see three different sorting algorithms, and compare those algorithms to each other.

5 Exhaustive-Search Sort

We begin with a very simple sorting algorithm, which we obtain almost directly from the definition of the Sortingcomputational problem.

```
ExhaustiveSearchSort($A$):
Input   : An array $A = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each $K_i \in \mathbb{R}$
Output  : A valid sorting of $A$
0 foreach permutation $\pi : [n] \to [n]$ do
1     if $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$ then
2         return $A' = ((K_{\pi(0)}, V_{\pi(0)}), (K_{\pi(1)}, V_{\pi(1)}), \ldots, (K_{\pi(n-1)}, V_{\pi(n-1)}))$
3 return $\perp$
```

Algorithm 5.1: `ExhaustiveSearchSort()`

The "bottom" symbol $\perp$ is one that we will often use to denote failure to find a valid output.

Example 5.1. Let's run Exhaustive-Search Sort on $A = ((6, a), (1, b), (6, c), (9, d))$.

When we design an algorithm to solve a problem, we want to be *sure* that the algorithm does indeed correctly solve the problem, and that we haven't made some subtle mistake that causes it to err on some inputs. We will assure ourselves of this with mathematical *proofs*.

Theorem 5.2. `ExhaustiveSearchSort` *correctly solves the* Sorting *problem. That is, for every input array A,* `ExhaustiveSearchSort(`A`)` *returns a valid sorting of A (if one exists), and otherwise returns $\perp$.*

It turns out that every array A has a valid sorting, so `ExhaustiveSearchSort` will never return $\perp$, but we will not prove this yet.

Proof.

□

Question 5.3. If `ExhaustiveSearchSort` is so simple and provably correct, why isn't it taught in introductory programming classes?

Answer.

6 Insertion Sort

Now let's see a much more efficient, but still rather simple, sorting algorithm:

```
InsertionSort($A$):
Input       : An array $A = ((K_0, V_0), \dots, (K_{n-1}, V_{n-1}))$, where each $K_i \in \mathbb{R}$
Output      : A valid sorting of $A$
0 /* "in-place" sorting algorithm that modifies $A$ until it is sorted */
1 foreach $i = 0, \dots, n-1$ do
2     /* Insert $A[i]$ into the correct place among $(A[0], \dots, A[i-1])$.  */
3     Find the first index $j$ such that $A[i][0] \le A[j][0]$;
4     Insert $A[i]$ into position $j$ and shift $A[j \dots i-1]$ to positions $j+1, \dots, i$
5 return $A$
```

Algorithm 6.1: `InsertionSort()`

Above we are using the notation $A[k \dots \ell]$ as shorthand for the subarray $(A[k], \dots, A[\ell])$, for sake of readability.

Example 6.1. $A = ((6, a), (2, b), (1, c), (4, d))$.

As our algorithm runs, we produce the following sorted sub-arrays:

i	Array contents after iteration i
0	
1	
2	
3	

Above, we only informally described Lines 3 and 4. They are expanded into a more precise implementation in the textbook.

Theorem 6.2. `InsertionSort` *correctly solves the* SORTING *problem. That is, for every input array A,* `InsertionSort(`A`)` *returns a valid sorting of A.*

Note that, as a side-effect, here we will also prove that every array A has a valid sorting. This phenomenon, of proving that a mathematical object exists (e.g. a valid sorting of an arbitrary array A) by exhibiting and analyzing an algorithm to construct it, is a quite common and useful one. See Problem Set 0 for another example.

Proof. Notation:

Proof strategy: We'll prove by induction on i (from $i = 0, \ldots, n$) the following "loop invariant" $P(i)$:

$$P(i) =$$

Notice that the statement $P(n)$ says that $A^{(n)}$, which is the output of `InsertionSort(`A`)`, is a valid sorting of A, as desired. □