CS120: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 1: Sorting and General Computational Problems

Harvard SEAS - Fall 2024 *2024-09-03*

1 Announcements

- Watch course overview video if you haven't already done so.
- Staff introductions.
- Updated syllabus posted, with midterm date (10/17) and more about participation grading and generative AI policy.
- Problem set 0 is available; due next Wednesday. It is required and substantial, so start early!
- Instructor OHs.
- TF OH and Sections start today; see Ed.
- Ask live questions in class! If you've missed an opportunity to, the TFs will also monitor Ed for questions during class.

2 Recommended Reading

- CS50 Week 3: `https://cs50.harvard.edu/x/2022/weeks/3/`
- Cormen-Leiserson-Rivest-Stein Chapter 2
- Roughgarden I, Sections 1.4 and 1.5
- We've ordered all of the course texts for purchase at the Coop, for reserve through Harvard libraries (should be available to read as e-books through HOLLIS), and physical copies that can be read in the SEC 2nd floor reading room.

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* what exactly algorithms are and what they can and cannot do through the 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. (Calculate PageRanks) For every URL on the entire World Wide Web (in mathematical notation: $\forall url \in \mathrm{WWW}$), 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, $S_w = \{url \in \mathrm{WWW} : w \text{ is contained on the webpage at } url\}$.
3. (Sort) 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 120, 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).

The keyword search (Step 2) can be done by creating a *trie* data structure for each webpage, also offline. Tries are covered in CS50.

Our focus here is Sorting (Step 3), which needs to be extremely fast (unlike `my.harvard`!) 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 $K_i \in \mathbb{R}$

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

Computational Problem Sorting

Above and throughout the course, $[n]$ 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.

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

- Keys =
- Values =

Abstraction → many other applications! Database systems (both Relational and NoSQL), machine learning systems, ranking cat photos by cuteness, . . .

Q: Is the output uniquely defined?

A:

In the subsequent sections, we will see pseudocode for 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 Sorting computational problem.

```
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 $A$
1 foreach permutation $\pi : [n] \to [n]$ do
2     if $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$ then
3         return $A' = ((K_{\pi(0)}, V_{\pi(0)}), (K_{\pi(1)}, V_{\pi(1)}), \ldots, (K_{\pi(n-1)}, V_{\pi(n-1)}))$
4 return $\perp$
```

Algorithm 1: Exhaustive-Search Sort

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

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

Correctness Proof:

Q: If Exhaustive-Search Sort is so simple and provably correct, why isn't it taught in introductory programming classes like CS50?

A:

6 Insertion Sort

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

```
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 $A$
1 /* "in-place" sorting algorithm that modifies $A$ until it is sorted */
2 foreach $i = 0, \ldots, n-1$ do
3     /* Insert $A[i]$ into the correct place among $(A[0], \ldots, A[i-1])$.  */
4     Find the first index $j$ such that $A[i][0] > A[j][0]$;
5     Insert $A[i]$ into $A[j]$ and shift $A[j \ldots i-1]$ to $A[j \ldots (i+1)]$
6 return $A$
```

Algorithm 2: Insertion Sort

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

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

Iteration i	Sorted Sub-Array
0	
1	
2	
3	

Proof of correctness:

Notation:

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

$P(i) =$

Base Case ($P(0)$):

Inductive Step ($P(i) \Rightarrow P(i+1)$):

Remarks.

- Correctness of InsertionSort $\Rightarrow$ every array of key-value pairs has a valid sorting. ("proof by algorithm")
- Not all correctness proofs for algorithms are induction! (cf. Exhaustive-Search Sort.)
- This was not a fully formal proof. It is often necessary to skip steps to make such proofs manageable for humans, but you should be careful when do so. Be sure that (a) you are completely convinced of the correctness of your claims, and (b) you are not omitting the main point or idea of the argument.

7 Merge Sort

Finally, you have probably already seen the following, even more efficient sorting algorithm.

```
MergeSort($A$)
  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 $A$
if $n \leq 1$ then return $A$;
else
    $i = \lceil n/2 \rceil$
    $A_1 =$ MergeSort($((K_0, V_0), \ldots, (K_{i-1}, V_{i-1}))$)
    $A_2 =$ MergeSort($((K_i, V_i), \ldots, (K_{n-1}, V_{n-1}))$)
    return Merge ($A_1, A_2$)
```

Algorithm 3: Merge Sort

We omit the implementation of Merge, which you can find in the readings.

Example: $A = (7, 4, 6, 9, 7, 1, 2, 4)$.

For the proof of correctness (which we only sketch here), we use strong induction on the statement $P(n) =$ "`MergeSort` correctly sorts arrays of size n."

8 Computational Problems

In the theory of algorithms, we want not only study and compare a variety of different algorithms for a single computational problem like Sorting, but also study and compare a variety of different computational problems. We want to classify problems according to which ones have efficient algorithms, which ones only have inefficient algorithms, and which ones have no algorithms at all. We also want to be able to relate different computational problems to each other, via the concept of *reductions* that we will see next week. All of this requires having an abstract definition of what is a computational problem, and what it means for algorithm to solve a computational problem.

Definition 8.1. A *computational problem* Π is a triple $(\mathcal{I}, \mathcal{O}, f)$ where:

- $\mathcal{I}$ is a (typically infinite) set of possible inputs (a.k.a. *instances*) x, and $\mathcal{O}$ is a (sometimes infinite) set of possible outputs y.
- For every input $x \in \mathcal{I}$, a *set* $f(x) \subseteq \mathcal{O}$ of *valid outputs* (a.k.a. *valid answers*).

Example: Sorting:

- $\mathcal{I} = \mathcal{O} = \{$All arrays of key-value pairs with keys in $\mathbb{R}\}$
- $f(x) = \{$All valid sorts of $x\}$

Note that there can be multiple valid outputs, which is why $f(x)$ is a set.

Informal Definition 8.2. An *algorithm* is a well-defined "procedure" A for "transforming" any input x into an output $A(x)$.

Note that this is only an informal definition. We will be more formal in a couple of weeks.

Definition 8.3. Algorithm A *solves* computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ if the following holds:

1. For every input $x \in \mathcal{I}$ with $f(x) \neq \emptyset$, we have $A(x) \in f(x)$.
2. There is a special output $\perp \notin \mathcal{O}$ such that for every input $x \in \mathcal{I}$ with $f(x) = \emptyset$, we have $A(x) = \perp$.

Condition 1 is the most important one: that when there is a valid output on input x, the algorithm A must find one. Condition 2 says that if there is no valid output, the Algorithm A must report so with the special output $\perp$ (a failure code).

Note that we want a *single* algorithm A (with a fixed, finite description) that is going to correctly solve the problem Π for *all of the* (infinitely many) inputs in the set $\mathcal{I}$.

A fundamental point in the theory of algorithms is that we distinguish between computational problems and algorithms that solve them. A single computational problem may have many different algorithms that solves it (or even no algorithm that solves it!), and our focus will be on trying to identify the most efficient among these.

Note that our proofs of correctness of sorting algorithms above are exactly proofs that the algorithms fit Definition 8.3 for an algorithm that solves the computational problem of sorting.