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

Lecture 1: Sorting

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

The readings here (and throughout the course) are optional, meant as supplements if you find them useful for a different presentation of the content or more details/examples/exercises.

- 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

Our first unit is *storing and searching data*. In it, we will cover algorithms to solve problems have vast applicability both in practical data management and in the design of algorithms for other important problems. Perhaps even more importantly, this topic will provide us with a clean setting to introduce key skills for the study of algorithms, namely:

- 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: $\forall url \in \text{WWW}$), calculate its *PageRank*, $\text{PR}(url) \in [0, 1]$.
2. (Keyword Search) Given a search keyword w, let S_w be the set of all webpages containing w. That is, $S_w = \{url \in \text{WWW} : w \text{ is contained on the webpage at } url\}$.
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

Many problems in data management involve collections of *key-value pairs* (K, V). Each key-value pair represents a single data item, where we have separated the information into the two parts according to what computation we wish to perform. For example, if we want to sort a spreadsheet of people by an age column, we would set the key of each row to be the number in the age column and the value to be the entries in the other columns (such as name, address, etc.) or perhaps the entire row. Sorting is relevant for datasets where keys can be ordered, so we will assume for simplicity that each key is a real number, though other ordered domains can also be used (e.g. alphabetic ordering on words).

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

Output : An array A' of key-value pairs $((K'_0, V'_0), \dots, (K'_{n-1}, V'_{n-1}))$ 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 $i = 0, \dots, n-1$.

Computational Problem Sorting

That is, we want to *reorder* the input array (using the permutation π) so that the keys are in ascending order.

Above and throughout the course, $[n]$ denotes the set of numbers $\{0, \dots, n-1\}$. In pure math (e.g. combinatorics), it is more standard for $[n]$ to be the set $\{1, \dots, n\}$, 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 $= 1 - \text{PR}$. (Note that we flip the pageranks, so sorting in descending order makes higher PRs appear towards the start of the list).
- Values $=$ *url*s.

The advantage of using an abstract definition of Sorting, rather than focusing only the problem of sorting web-search results, is that it eliminates distracting details (such as how the specific PageRanks are computed) and gives us a much more general tool that can be applied in many other settings. Indeed, Sorting has a vast array of applications, being used throughout database systems (both Relational and NoSQL), in Machine Learning Systems, and in the design of other algorithms.

Q: Is the output uniquely defined?

A: No. When there are repetitions among the keys (e.g. consider the case when the key is a person's age in years), then there are multiple permutations of the input array that all yield valid sortings.

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 $A = ((K_0, V_0), \dots, (K_{n-1}, V_{n-1}))$, where each $K_i \in \mathbb{R}$
Output : A valid sorting of $A$
1 foreach permutation $\pi : [n] \to [n]$ do
2     if $K_{\pi(0)} \le K_{\pi(1)} \le \cdots \le K_{\pi(n-1)}$ then
3         return $A' = ((K_{\pi(0)}, V_{\pi(0)}), (K_{\pi(1)}, V_{\pi(1)}), \dots, (K_{\pi(n-1)}, V_{\pi(n-1)}))$
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 $A = ((6, a), (1, b), (6, c), (9, d))$.

As our algorithm runs, we try the following permutations in order until we find one that produces a valid sorting.

$\pi(0)$	$\pi(1)$	$\pi(2)$	$\pi(3)$
0	1	2	3
0	1	3	2
0	2	1	3
0	2	3	1
0	3	1	2
0	3	2	1
1	0	2	3
1	0	3	2

The first permutation that leads to keys being in sorted order is 1 0 2 3, which yields output $(1, b), (6, a), (6, c), (9, d)$, with the keys ordered $1 \le 6 \le 6 \le 9$. Note that the valid sorting is not unique, e.g., in this case, $(6, a)$ and $(6, c)$ could be swapped.

Correctness Proof: Whenever we give an algorithm, we will want to be sure that it solves the computational problem that it is intended to solve. We will do so by giving a *correctness proof*, which shows that for every input x (e.g. an array A of key-value pairs), the algorithm returns a valid output if there is one (e.g. a valid sorting A' of A).

In the case of sorting, a valid output A' is defined by the existence of a permutation π^* that reorders the input array A into the array A' so that the permuted keys satisfy $K_{\pi^*(0)} \le K_{\pi^*(1)} \le \ldots \le K_{\pi^*(n-1)}$. If there is such a permutation π^* (which there always is, though we don't need that fact here), then the algorithm is guaranteed to halt at some permutation π and produce an output array A', rather than outputting $\perp$. (π may be different than π^* if there are multiple valid sortings. But we know it definitely won't continue past π^* in the loop, since that is a valid sorting.) Lines 2 and 3 ensure that when the algorithm does output an array A', it is always a valid sorting of A.

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

A: It is far too slow! In the worst case, it can enumerate all permutations on $[n]$, of which there are $n!$, which is (more than) exponential in n. The reason we are covering it is to illustrate a phenomena that will recur throughout the course. Most of the computational problems we will study have a simple, exhaustive-search algorithm that follows immediately from the definition, but this algorithm will often be very slow, at least exponential time. In some cases, but not all, we will be able to find much more efficient algorithms by exploiting the structure of the problem and a toolkit of algorithm design techniques.

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 $A[i][0]$ $leqA[j][0]$;
 5     Insert $A[i]$ into position $j$ and shift $A[j \ldots i-1]$ to positions $j+1, \ldots, i$
 6 return $A$
```

Algorithm 2: Insertion Sort

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

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

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

Iteration i	Array contents after iteration i
0	((6,a),(2,b),(1,c),(4,d))
1	((2,b),(6,a),(1,c),(4,d))
2	((1,c),(2,b),(6,a),(4,d))
3	((1,c),(2,b),(4,d),(6,a))

Above, we only informally described Lines 4 and 5. They are expanded into a more precise implementation below.

```
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 $A$
/* "in-place" sorting algorithm that modifies $A$ until it is sorted */
foreach $i = 0, \dots, n-1$ do
    /* Insert $A[i]$ into the correct place among $(A[0], \dots, A[i-1])$.  */
    $j = 0$
    while $A[i][0] > A[j][0]$ do                          /* find place to insert */
        $j = j + 1$
    temp $= A[j]$;
    $A[j] = A[i]$ ;                                        /* insert $A[i]$ into $A[j]$ */
    while $j < i$ do                                      /* shift over rest of array */
        $j = j + 1$
        temp$'$ $= A[j]$
        $A[j] =$ temp
        temp $=$ temp$'$
return $A$
```

Algorithm 3: Insertion Sort, in more detail

Proof of correctness:

We will prove by induction on the number of for-loop iterations executed, that every-increasing prefixes of the array A become sorted while suffixes remain as they were in the input array. To formalize this, we carefully introduce notation to allow us to distinguish the array contents at different stages of the algorithm while maintaining readability. For $i = 0, \dots, n$, let $A^{(i)}$ be the contents of the array A before iteration i and/or after iteration $i-1$ of the for-loop. In particular, $A^{(0)}$ is the input array and $A^{(n)}$ is the output array. For readability, we will write $K^{(i)}[j]$ for $A^{(i)}[j]$.

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

> $P(i)$ = "$A^{(i)}[0 \dots i-1]$ is a valid sorting of $A^{(0)}[0 \dots i-1]$ and $A^{(i)}[i \dots n-1] = A^{(0)}[i \dots n-1]$."

Base Case ($P(0)$): Both prefix arrays in $P(0)$ are of length $i = 0$, i.e. they are both the empty array. The empty array is a valid sorting of itself. Furthermore, since $i = 0$, it immediately holds that $A^{(i)}[i \dots n-1] = A^{(0)}[i \dots n-1]$.
Inductive Step ($P(i) \Rightarrow P(i+1)$): Suppose, by induction, that statement $P(i)$ is true. That is,

1. $A^{(i)}[0 \dots i-1]$ is a permutation of $A^{(0)}[0 \dots i-1]$.
2. $K^{(i)}[0] \leq K^{(i)}[1] \leq \cdots \leq K^{(i)}[i-1]$.
3. $A^{(i)}[i \dots n-1] = A^{(0)}[i \dots n-1]$

We now want to prove that the analogous three conditions will hold for the array $A^{(i+1)}$ at the end of the i'th loop iteration. In the first while loop, the algorithm finds the smallest j such that $K^{(i)}[i] \leq K^{(i)}[j]$. Note that $j \leq i$, since $K^{(i)}[i] \leq K^{(i)}[i]$. In the remainder of the for-loop the key-value pair $A^{(i)}[i]$ is inserted into position j, and the elements previously in positions $j, \dots, i-1$ are shifted over by 1 to make room for it. That is,

$$\begin{aligned}&\Big(A^{(i+1)}[0 \dots j-1], A^{(i+1)}[j], A^{(i+1)}[j+1 \dots i], A^{(i+1)}[i+1 \dots n-1]\Big)\\ &= \Big(A^{(i)}[0 \dots j-1], A^{(i)}[i], A^{(i)}[j \dots i-1], A^{(i)}[i+1 \dots n],\Big)\,. \qquad (1)\end{aligned}$$

From this equation and Item 1 of our inductive hypothesis, we see that $A^{(i+1)}[0 \dots i]$ is a permutation of $(A^{(0)}[0 \dots i-1], A^{(i)}[i])$, which equals $A^{(0)}[0 \dots i]$ by Item 3 of our inductive hypothesis. Thus we've established the analogue of Item 1 for $P(i+1)$. The analogue of Item 3 follows by observing that the i'th for-loop iteration does not write to array positions above i. So we only need to check the analogue of Item 2; that is, the first $i+1$ keys in $A^{(i+1)}$ are in sorted order. By Equation 1, it suffices to check that:

$$K^{(i)}[0] \le K^{(i)}[1] \le \dots \le K^{(i)}[j-1] \le K^{(i)}[i] \le K^{(i)}[j] \le \dots \le K^{(i)}[i-1].$$

This follows from Item 2 of our inductive hypothesis, and the fact that j was chosen to be the first index such that $K^{(i)}[i] \le K^{(i)}[j]$.

Remarks.

- Note that Insertion Sort always outputs an array of key-value pairs, never the $\perp$ symbol. Thus, by proving its correctness, we also prove that every array of key-value pairs has a valid sorting. This is an example where reasoning about an algorithm yields an interesting pure-mathematical statement; there are many much more nontrivial of this "proof by algorithm" phenomenon. On Problem Set 0, you will see an example in the other direction, where you start by proving a pure mathematical statement (about binary trees), but then turn the proof into an efficient algorithm.

- Induction is a common tool for reasoning about algorithms with a loop or with recursion (see also MergeSort below). In the case of loops, as above we often use induction to prove statements like $P(i)$ that hold after an unbounded number of loop iterations. Such statements $P(i)$ are often called *loop invariants*; it's common practice to state a loop invariant as a comment in code. Choosing the right loop invariant is often the hardest part of a proof of correctness. Still, it's important to emphasize that many proofs of algorithm correctness are not primarily based on induction, even for algorithms with loops; see the proof for Exhaustive-Search Sort above for an example.

- The above proof may seem like a lot of work to verify the correctness of a rather intuitive algorithm like Insertion Sort. In such cases, the role of the proof is to help ensure that we haven't missed some subtle detail that leads to erroneous behavior, which happens often with implementations of algorithms. Still, this proof falls short of a truly *formal* proof, in that we leave it to the reader to verify some easier but tedious claims, such as the effect of the insert-and-shift steps inside the for loop. Proofs that address all such low-level details quickly become too long and cumbersome for humans to write and read. Thus, there is an entire subfield of computer science, known as *formal verification*, where automated tools are used to assist in finding and verifying formal proofs of correctness and other properties of computer programs (as well as proofs of pure mathematical statements). We'll get a brief taste of some such tools later in the course (SAT Solvers and SMT Solvers), and you can learn more about such tools in some courses on Programming Languages (CS x5xx).

As the algorithms we reason about and their proofs of correctness become more sophisticated, the low-level steps that we skip over will also become more substantial. Still, you should state any intermediate claims precisely, even if you are skipping their proof, and be sure that (a) you are completely convinced of their correctness (e.g. could write down a detailed proof if forced), and (b) you are not omitting the main point or idea of the overall argument.

7 Merge Sort

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

```
MergeSort($A$)
Input      : An array $A = ((K_0, V_0), \dots, (K_{n-1}, V_{n-1}))$, 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($((K_0, V_0), \dots, (K_{i-1}, V_{i-1}))$)
    $A_2 =$ MergeSort($((K_i, V_i), \dots, (K_{n-1}, V_{n-1}))$)
    return Merge $(A_1, A_2)$

```

Algorithm 4: Merge Sort

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

Example: $A = ((7,a),(4.b),(6,c),(9,d),(7,e),(1,f),(2,g),(4,h))$. We sort $((7,a),(4.b),(6,c),(9,d))$ and $(7,e),(1,f),(2,g),(4,h)$ recursively and obtain $((4,b),(6,c),(7,a),(9,d))$ and $((1,f),(2,g),(4,h),(7,e))$. We then merge the two sorted halves and obtain $((1,f),(2,g),(4,b),(4,h),(6,c),(7,a),(9,d))$.

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." The base cases are that we sort arrays of length 0 and 1 correctly. If we assume by induction that we sort arrays of size up to n correctly, then the recursive calls to `MergeSort` are to smaller lists (since $\lceil n/2 \rceil$ and $n - \lceil n/2 \rceil$ are both strictly smaller than n when $n > 1$), so they return sorted lists, and to complete the inductive step we only need to show that calling `Merge` on two sorted lists produces a sorted list. (We omit this proof, since we omitted the definition of `Merge`.)