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

Lecture 1: Sorting and General Computational Problems

Harvard SEAS - Fall 2023 *2023-09-05*

1 Announcements

- Watch course overview video if you haven't already done so
- Staff introductions
- HCS-organized pset 0 party/CS 120 social at office hours right after this class.
- Instructor OHs.
- TF sections and OH 0
- Revised syllabus available; e.g. more about participation grading
- Problem set 0 is available; due next Wednesday.
- Join/follow Ed even during class. We'll use it for continuous chat/Q&A.
- Collaboration policies
- Today's goals: learn what "algorithm", "computational problem", and "proof of correctness" are.

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 Motivating Problem: Web Search

Simplified and outdated description of Google's original search algorithm:

1. (Calculate Pageranks) For every URL *url* on the entire world-wide web 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, $S_w = \{url \in \text{WWW} : w \text{ is contained on the webpage at } url\}$.

3. (Sort Results) 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 222 (Algorithms at the End of the Wire) and CS 229r (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. 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

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 item-key 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. $K'_0 \leq K'_1 \leq \cdots K'_{n-1}$. 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 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.

- Application to web search:
  - Keys = 1 − 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
- Many other applications! Database systems (both Relational and NoSQL), machine learning systems, ranking cat photos by cuteness, ...
- Is the output uniquely defined?

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

5 Exhaustive-Search Sort

```
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 $(K_{\pi(0)}, V_{\pi(0)}), (K_{\pi(1)}, V_{\pi(1)}), \ldots, (K_{\pi(n-1)}, V_{\pi(n-1)})$
```

Algorithm 1: Exhaustive-Search Sort

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 and see if they produce a sorted array.

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

A permutation is valid if the keys are in increasing order. So the first valid permutation is 1 0 2 3, since the output of our algorithm is $(1, b), (6, a), (6, c), (9, d)$, and the keys are ordered $1 \leq 6 \leq 6 \leq 9$. Note that the valid sorting is not unique, e.g., in this case, $(6, a)$ and $(6, c)$ can be permuted.

To prove correctness of an algorithm such as this one, we must prove that, for every input (here, an array A of key-value pairs), the algorithm gives a correct output (here, a valid sorting of the input array). We'll prove that by breaking it up into three statements:

1. For every input array, there is a permutation π that defines a valid sorting.
2. If there is a valid sorting π of the input array, the algorithm returns something.
3. If the algorithm returns something, it returns a valid sorting of the input array.

Together these imply that for every input array, Exhaustive-Search Sort will always produce an output that is a valid sorting of the input array.

We look into the details of each of these things:

1. There exists at least one sorted order of the whole list (one can prove this from the definition of $\leq$, but that's a level of detail we won't go into).
2. If there's a valid sorting π, (that is, a permutation for which $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$) and the for loop reaches π, then we return it. Since we check all possible permutations in the algorithm, the only way the for loop could not reach π is if we return something else first. Either way, our algorithm must return something.
3. We return something only in line 3. What we return is a permutation of the input array, and we only reach line 3 if π was a valid permutation in line 2.

6 Insertion Sort

```
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 in $(A[0], \dots, A[i-1])$
    /* Note that when $i = 0$ this is the empty list. */
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	Sorted Sub-Array
i=0	((6,a))
i=1	((2,b),(6,a))
i=2	((1,c),(2,b),(6,a))
i=3	((1,c),(2,b),(4,d),(6,a))

Proof of correctness:

We'll prove by induction on i the statement $P(i)$ that at the end of iteration i of the loop,

$$A[0], A[1], \dots, A[i]$$

is a valid sorting of the first $i+1$ elements of the original array.

Base Case: The base case is the statement $P(0)$ that at the end of iteration 0 of the loop, the list $A[0]$ is a valid sorting of the first 1 elements of the original array. The first element of the original array is $A[0]$, and a one-element list is automatically validly sorted.

Inductive Step: Suppose, by induction, that statement $P(i)$ is true; that is, at the end of iteration i of the loop, the list $A[0], A[1], \dots, A[i]$ is a valid sorting of the first $i+1$ elements of the original array, so the keys are in ascending order $A[0][0] \le A[1][0] \le \dots \le A[i][0]$. In the $(i+1)$st iteration, we insert $A[i+1] = (k, v)$ into its correct place in the $A[0], A[1], \dots, A[i]$. We define (and implement somehow in the algorithm) the "correct place" to be position j if it is both greater than the element before it (if there is one), and less than the element after it (if there is one):

1. $A[j-1][0] \le k$ if $j > 0$, and

2. $k \le A[j][0]$ if $j < i$.

(We omit the proof that a correct position exists.) So, after inserting $A[i+1]$ into a correct position j, we have

1. $A[0][0] \le A[1][0] \le \dots \le A[j-1][0]$ by induction,

2. $A[j-1][0] \le A[j][0] \le A[j+1][0]$ by the definition of "correct position", and

3. $A[j+1][0] \le A[j+2][0] \le \dots \le A[i+1][0]$ by induction, since we've shifted the old indices j through i up by one to $j+1$ through $i+1$

Hence the statement $P(i+1)$ also holds, completing induction. In particular, we've proved the statement $P(n-1)$ that at the end of iteration $n-1$ of the loop, $A[0], A[1], \ldots, A[n-1]$ is a valid sorting of all n elements of the original array, completing a proof of correctness. This style of proof of correctness is called a proof of a *loop invariant*: here, the loop invariant is that $(A[0], A[1], \ldots, A[i-1])$ is a valid sorting of the first i elements of the original input array. 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.

7 Merge Sort

We did not cover this in lecture, since most of you have already seen Merge Sort in CS50. But you should review it on your own!

```
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 if $n = 2$ and $K_0 \leq K_1$ then return $A$;
else if $n = 2$ and $K_0 > K_1$ then return $((K_1, V_1), (K_0, V_0))$;
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: $A = (7, 4, 6, 9, 7, 1, 2, 4)$.

We sort $(7, 4, 6, 9)$ and $(7, 1, 2, 4)$ independently and obtain $(4, 6, 7, 9)$ and $(1, 2, 4, 7)$. We then merge the two sorted halves and obtain $(1, 2, 4, 4, 6, 7, 9)$.

For the proof of correctness, we use strong induction. The base cases are that we sort arrays of length 1 and 2 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`.)

8 Computational Problems

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

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

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 are multiple valid solutions, 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: this is 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 solution on input x, the algorithm A must find one. Condition 2 says that if there is no solution, 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 $\mathcal{I}$.

Important point: 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.