CS120: Intro. to Algorithms and their Limitations Hesterberg Vadhan

Lecture 1: Sorting and General Computational Problems

Harvard SEAS - Fall 2022 *2022-09-01*

1 Announcements

- Put up a name tent with your name and an emoji
- Watch course overview video if you haven't already done so
- Staff introductions
- My OH today: 11:15am-12:30pm in SEC 3.327; zoom option available on Canvas
- 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 quizzes and continuous chat/Q&A.
- Collaboration policies
- CS 121.5 optional advanced topics seminar: email gabrielwu@college.harvard.edu if interested.
- Today's goals: learn what "algorithm", "computational problem", and "proof of correctness" are.

2 Recommended Reading

- CS50 Week 3: https://cs50.harvard.edu/x/2021/weeks/3/
- Cormen-Leiserson-Rivest-Stein Section 1.3
- Roughgarden I, Sections 1.4 and 1.5
- We've ordered all of these at the Coop and for reserve through Harvard libraries (check on status through HOLLIS); apparently CLRS is already available to read as an e-book.

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Simplified and outdated description of Google's original search algorithm: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, $S_k = \{url \in \text{WWW} : w \text{ is contained on the webpage at } url\}$.

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 higher PR appears towards the start of the list)
  - Values = urls
- Many other applications! Database systems (both Relational and NoSQL), Machine learning systems, Ranking professional surfers by points accumulated,...
- 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 $A = ((6, a), (1, b), (6, c), (9, d))$.

For correctness, we check three things:

1. For every input array, there is at least one permutation π that defines a valid sorting.

2. If there is a valid sorting π of the input array, we return something.

3. If we return something, we return a valid sorting of the input array.

Together these imply that Exhaustive-Search Sort will always produce an output, and it will always be a valid sorting of the input array.

Each of these things can be proven by examination of the code:

6 Insertion 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 $A$
/* "in-place" sorting algorithm that modifies $A$ until it is sorted */
foreach $i = 0, \ldots, n-1$ do
    Insert $A[i]$ into the correct place in $(A[0], \ldots, A[i-1])$
    /* 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:

7 Merge Sort

```
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 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($((K_0, V_0), \dots, (K_i, V_i))$)
    $A_2 =$ MergeSort($((K_{i+1}, V_{i+1}), \dots, (K_{n-1}, V_{n-1}))$)
    return Merge $(L_1, L_2)$

```

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 $\lceil \frac{n+1}{2} \rceil \leq n$), 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 $(\mathcal{I}, f)$ where:

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

Example: sorting

- $\mathcal{I} = \{$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 $\Pi = (\mathcal{I}, f)$ if for every input $x \in \mathcal{I}$, we have:

1. If $f(x) \neq \emptyset$, then $A(x) \in f(x)$.
2. If $f(x) = \emptyset$, then $A(x) = \perp$.
3. If $f(x) \neq \emptyset$, then $A(x) \neq \perp$.

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). Finally, Condition 3 is a technical condition reserving $\perp$ for use as a failure code, not a regular output. (We could instead have required in the definition of a computational problem that $\perp \notin f(x)$; this distinction is not important.)

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.