CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 1: Sorting

Harvard SEAS - Fall 2021 *Sept. 2, 2021*

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: 1-2pm in the SEC 3.327; zoom option available from my webpage
- TF sections and OH 0
- Revised problem set 0 posted
- Join/follow Ed even during class. We'll use it for quizzes and continuous chat/Q&A.
- Collaboration policies
-

2 Recommended Reading

- CS50 Week 3: https://cs50.harvard.edu/x/2021/weeks/3/
- Cormen-Leiserson-Rivest-Stein Chapter 2 and Section 8.1
- Roughgarden I, Sections 1.4 and 1.5, Chapter 2
- Lewis-Zax Chapter 21
- 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Simplified and outdated description Google's original search algorithm: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, $S_k = \{url \in \mathrm{WWW} : w \text{ is contained on the webpage at } url\}$.

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 item-key pairs $((I_0, K_0), \ldots, (I_{n-1}, K_{n-1}))$, where each key $K_i \in \mathbb{R}$

Output : An array A' of item-key pairs $((I'_0, K'_0), \ldots, (I'_{n-1}, K'_{n-1}))$ 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 $(I'_i, K'_i) = (I_{\pi(i)}, K_{\pi(i)})$ 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. Similarly, for us, the natural numbers are $\mathbb{N} = \{0, 1, 2, \ldots\}$

- Application to web search:
  - Items = urls
  - Keys = $1 - \text{PR}$ (Note that we flip the pageranks so higher PR appears towards the start of the list)
- 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 $A = ((I_0, K_0), \ldots, (I_{n-1}, K_{n-1}))$, where each $K_i \in \mathbb{R}$
Output : A valid sorting of A

```
1 foreach permutation π : [n] → [n] do
2     if K_π(0) ≤ K_π(2) ≤ ··· ≤ K_π(n−1) then
3         return (I_π(0), K_π(0)), (I_π(1), K_π(1)), ..., (I_π(n−1), K_π(n−1))
4
```

Algorithm 1: Exhaustive-Search Sort

Example: $A = ((a, 6), (b, 1), (c, 6), (d, 9))$.

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

Permutation is valid if in increasing order (i.e., keys in order). So the first valid permutation is 1 0 2 3, hence the output is $(b, 1), (a, 6), (c, 6), (d, 9)$. Note that the valid sorting is not unique. E.g., in this case, $a(, 6)$ and $(c, 6)$ can be permuted.

For correctness, we check two things:

1. If exhaustive sort produces an output, it is a valid output (i.e. sort).
2. Every array has a valid sort.

Note that we need to show the algorithm *always outputs* a valid sorting, but this follows from the definition of the algorithm, since if there is a valid permutation, we will find it.

6 Insertion Sort

Input : An array $A = ((I_0, K_0), \ldots, (I_{n-1}, K_{n-1}))$, 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 = 1, ..., n − 1 do
3     /* loop invariant: (A[0], A[1], ..., A[i − 1]) is a valid sorting of the first
          i elements of the original input array                            */
4     Insert A[i] into the correct place in (A[0], ..., A[i − 1])
5 return A
```

Algorithm 2: Insertion Sort

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

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

Iteration	Sorted Sub-Array
i=0	((a,6))
i=1	((b,2),(a,6))
i=2	((c,1),(b,2),(a,6))
i=3	((c,1),(b,2),(d,4),(a,6))

Correctness: proof by induction on i that prior to the i-th loop iteration, the following holds:

$$A[0], A[1], \ldots, A[i-1].$$

I.e., it is a valid sorting of the first i elements of the original array.

7 Merge Sort

```
1  MergeSort(A)
   Input  : An array A = ((I_0, K_0), ..., (I_{n-1}, K_{n-1})), where each K_i ∈ ℝ
   Output : A valid sorting of A
2  if n ≤ 1 then return A;
3  else if n = 2 and K_0 ≤ K_1 then return A;
4  else if n = 2 and K_0 > K_1 then return ((I_1, K_1), (I_0, K_0));
5  else
6  |  i = ⌈n/2⌉
7  |  A_1 = MergeSort(((I_0, K_0), ..., (I_i, K_i)))
8  |  A_2 = MergeSort(((I_{i+1}, K_{i+1}), ..., (I_{n-1}, K_{n-1})))
9  |  return Merge (L_1, L_2)
10 |
```

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, we use induction. We show the base case (that we sort length $1, 2$ arrays correctly), and that if we sort arrays of size k correctly, we also sort arrays of size $k + 1$ correctly.

8 Evaluating Algorithms

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 item-key pairs with keys in $\mathbb{R}\}$

- $f(x) = \{\text{All valid sorts of } x\}$

(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 $A(x) \in f(x)$.

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 $\mathcal{I}$. Now to measure the efficiency of an algorithm, we consider how its computation time *scales* with the size of its input. That is, for every input $x \in \mathcal{I}$, we associate one or more *size* parameters $\text{size}(x) \geq 0$. For example, in sorting, we typically let $\text{size}(x)$ be the length of the array x of item-key pairs.

Informal Definition 8.4 (running time)**.** For a function $T : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$, We say that algorithm A has running time T if for every input x, the number of "basic operations" performed by $A(x)$ is at most $T(\text{size}(x))$.

We will make this definition more formal in a couple of weeks, but for now think of a "basic operation" as arithmetic on individual numbers, manipulating pointers, and stepping through a line of code. However, using a sophisticated built-in Python function like `A.sort()` does not count as a single "basic operation". Indeed, this function is implemented using a combination of Merge Sort and Insertion Sort. Note that we are measuring *worst-case* running time; $T(n)$ must upper-bound the running time of A on all inputs of size n.

To avoid our evaluations of algorithms depending too much on minor distinctions in the choice of "basic operations" and bring out more fundamental differences between algorithms, we generally measure complexity with asymptotic growth rates. Recall:

Definition 8.5. Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say:

- $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all sufficiently large n.
- $f = \Omega(g)$ if there is a constant $c > 0$ such that $f(n) \geq c \cdot g(n)$ for all sufficiently large n. Equivalently, $g = O(f)$.
- $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.
- $f = o(g)$ if for every constant $c > 0$, we have $f(n) \leq c \cdot g(n)$ for all sufficiently large n. Equivalently, $\lim_{n\to\infty} f(n)/g(n) = 0$.
- $f = \omega(g)$ if for every constant $c > 0$, we have $f(n) \geq c \cdot g(n)$ for all sufficiently large n. Equivalently, $g = o(f)$.

9 Complexity of Sorting

What approach do we use to estimate the asymptotic running time of the sorting algorithms we've seen?

- Exhaustive-search sort:
- Insertion-sort:
- Merge sort:

Exercise 9.1. Which of the following correctly describe the asymptotic runtime of each of the three sorting algorithms? (Include all the apply.)

$$O(n^2), \omega(n), o(2^n), \Omega(n!), \Theta(n \log n)$$

- $T_{exhaust}(n) =$
- $T_{insert}(n) =$
- $T_{merge}(n) =$

Exercise 9.2. Order $T_{exhaust}, T_{insert}, T_{merge}$ from fastest to slowest, i.e. T_0, T_1, T_2 such that $T_0 = o(T_1)$ and $T_1 = o(T_2)$.

We will be interested in three very coarse categories of running time:

(at most) exponential time $T(n) = 2^{n^{O(1)}}$ (slow)

(at most) polynomial time $T(n) = n^{O(1)}$ (reasonable)

(at most) nearly linear time $T(n) = O(n \log n)$ or $T(n) = O(n)$ (fast)

Why do we measure correctness and complexity in the worst-case? While this can sometimes give an overly pessimistic picture, it has the advantage of providing us with more general-purpose and application-independent guarantees. If an algorithm has good performance on some inputs and not on others, we may need think hard about which kinds of inputs arise in our application before using it. When good enough worst-case performance is not possible, then one may need to turn to alternatives to worst-case analysis (mostly beyond the scope of this course).

10 Comparison-based Sorting Algorithms

All above algorithms are "comparison based": the permutation of the input list depends only on comparisons between input values. So this provides one way we can measure complexity.
Example: insertion sort on the array (K_0, K_1, K_2)

Generalizing the above example, we see that we can model a comparison-based sorting example as a binary tree (specifically a "decision tree"), whose height is the worst-case number of comparisons made by the algorithm. With this model of computation, we can prove a *lower bound* on the efficiency of any sorting algorithm.

Theorem 10.1. *If A is a comparison-based sorting algorithm that makes at most $T(n)$ comparisons on any array of size n, then $T(n) = \Omega(n \log n)$.*

This is our first taste of what it means to establish *limits* of algorithms. From this, we see that `MergeSort()` has asymptotically *optimal* complexity among comparison-based sorting algorithms.

Proof. For any permutation $\sigma : [n] \to [n]$, consider an input array x where $(K_1, \ldots, K_n) = (\sigma(1), \ldots, \sigma(n))$. Then $A(x)$ produces a correct output on x if and only if it produces permutation $\pi = \sigma^{-1}$. Thus, there must be at least $n!$ possible different outputs.

However, since A's outputs are determined by a binary tree of comparisons of depth $T(n)$, A can output at most $2^{T(n)}$ distinct permutations.

Thus

$$2^{T(n)} \geq n! \geq \left(\frac{n}{2}\right)^{n/2}.$$

Therefore $T(n) \geq \frac{n}{2} \log_2 \left(\frac{n}{2}\right) = \Omega\left(n \log n\right)$. □

Extensions:

1. The lower bound of $\Omega(n \log n)$ for comparison-based sorting holds even when the keys are restricted to be from the set $[n]$.

2. If we feed A an input array x where the keys are a uniformly random permutation σ of $[n]$, then the probability that A correctly sorts the array is at most $2^{T(n)}/n!$