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

Lecture 2: Measuring Efficiency

Harvard SEAS - Fall 2022 *2022-09-06*

1 Announcements

2 Loose Ends from Lec 1

3 Measuring Efficiency

Recommended Reading:

- CS50 Week 3: https://cs50.harvard.edu/college/2021/fall/notes/3/
- Roughgarden I, Ch. 2
- CLRS 3e Ch. 2, Sec 8.1
- Lewis–Zax Ch. 21

3.1 Definitions

Informal Definition 3.1 (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

- size(x) =
- basic operations:
- non-basic operations:
- to be made more formal next week!

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

- $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \le c \cdot g(n)$ for all sufficiently large n.
- $f = \Omega(g)$ if there is a constant $c > 0$ such that $f(n) \ge c \cdot g(n)$ 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 $f(n) \le c \cdot g(n)$ 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 $f(n) \ge c \cdot g(n)$ for all sufficiently large n. Equivalently, $\lim_{n\to\infty} f(n)/g(n) = \infty$. Equivalently, $g = o(f)$.

Definition 3.3. Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say:

- $f = O(g)$ if
- $f = \Omega(g)$ if
  Equivalently:
- $f = \Theta(g)$ if
- $f = o(g)$ if
  Equivalently:
- $f = \omega(g)$ if
  Equivalently:

Given a computational problem Π, our goal is to find algorithms whose running time $T(n)$ has the smallest possible growth rate among all of the algorithms that correctly solve Π. This minimal growth rate is informally called the *computational complexity* of the problem Π.

3.2 Computational Complexity of Sorting

Let's analyze the runtime of the sorting algorithms we've seen.

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

foreach *permutation* $\pi : [n] \to [n]$ **do**
  if $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$ **then**
    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

$T_{exhaustsort}(n) =$

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

$T_{insertsort}(n) =$

`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

$T_{insertsort}(n) =$

Exercise 3.4. Order $T_{exhaustsort}, T_{insertsort}, T_{mergesort}$ from fastest to slowest, i.e. T_0, T_1, T_2 such that $T_0 = o(T_1)$ and $T_1 = o(T_2)$.

Exercise 3.5. Which of the following correctly describe the asymptotic (worst-case) runtime of each of the three sorting algorithms? (Include all that apply.)

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

- $T_{exhaustsort}(n) =$
- $T_{insertsort}(n) =$
- $T_{mergesort}(n) =$

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)

Q: Why *worst-case* correctness and complexity?

3.3 Complexity of Comparison-based Sorting

All of the above algorithms are "comparison-based" sorting algorithms: the only way in which they use the keys is by comparing them to see whether one is larger than the other.

It may seem intuitive that sorting algorithms must work via comparisons, but in Thursday's Sender–Receiver exercise and Problem Set 1, you'll see examples of sorting algorithms that benefit from doing other operations on keys.

The concept of a comparison-based sorting algorithm can be modelled using a programming language in which keys are a special data type `key` that only allows the following operations of variables `var` and `var′` of type `key`:

- `var` = `var′`: assigns variable `var` the value of variable `var′`.
- `var` ≤ `var′`: returns a boolean (`true`/`false`) value according to whether the value of `var` is ≤ the value of `var′`

In particular, comparison-based programs are not allowed to convert between type `key` and other data types (like `int`) to perform other operations on them (like arithmetic operations). This can all be made formal and rigorous using a variant of the RAM model that we will be studying in a couple of weeks. (In the basic RAM model, all variables are of integer type.)

We will prove a *lower bound* on the efficiency of *every* comparison-based sorting algorithm:

Theorem 3.6. *If A is a comparison-based algorithm that correctly solves the sorting problem on arrays of length n in time $T(n)$, then $T(n) = \Omega(n \log n)$. Moreover, this lower bound holds even if the keys are restricted to be elements of $[n]$ and the values are all empty.*

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. No matter how clever computer scientists are in the future, they will not be able to come up with an asymptotically faster comparison-based sorting algorithm.

The key to the proof is the following lemma, which we state for input arrays consisting only of keys, since Theorem 3.6 holds even when the values are all empty.

Lemma 3.7. *If we feed a comparison-based algorithm A an input array $x = (K_0, K_1, \ldots, K_{n-1})$ consisting of elements of type `key` and the output $A(x)$ contains a variable K' of type `key`, then:*

1. *$K' = K_i$ for some $i = 0, \ldots, n-1$, and*
2. *The value of i depends only on the results of the boolean key comparisons that A makes on input x.*

Let's illustrate this with an example.

Example: insertion sort on the key array (K_0, K_1, K_2).

Proof Sketch of Lemma 3.7. Item 1 follows because a comparison-based algorithm does not have any way to create a variable of type `key` other than by copying (using the assignment operation $\texttt{var} = \texttt{var}'$).

For Item 2, the intuition is that A has access to no other information about the input other than the results of the comparisons it has made so far. We omit a formal proof. □

Proof of Theorem 3.6.

□