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

Lecture 2: Measuring Efficiency

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

1 Announcements

- Detailed Lecture 1 notes posted
- Raise your name placard to ask a question
- Log in to Ed during class — participate in live Q&A
- PS0 due tomorrow
- Adam OH after class
- Salil's OH today 4:15-5:15pm on Zoom
- Handout: Lecture notes 2 (PDF in Ed)
- Sender–Receiver exercise at start of class on Thursday; prepare and come on time!
- Participation portfolios: see guidelines on the course website

2 Loose Ends from Lec 1

- Recall exhaustive search, insertion sort, and merge sort.
- Abstract definition of computational problem and what it means for an algorithm to solve a computational problem.

2.1 Computational Problem

Definition 2.1. A *computational problem* is a triple $\Pi = (\mathcal{I}, \mathcal{O}, f)$ where

- $\mathcal{I}$ is the set of inputs/instances (typically infinity)
- $\mathcal{O}$ is the set of outputs.
- $f(x)$ is the set of solutions
- For each $x \in \mathcal{I}$, $f(x) \subseteq \mathcal{O}$

Definition 2.2. Let Π be a computational problem and let A be an algorithm. We say that A *solves* Π if

- For every $x \in \mathcal{I}$ such that $f(x) \neq \emptyset$, $A(x) \in f(x)$.
- $\exists$ a special symbol $\perp \notin \mathcal{O}$ such that for all $x \in \mathcal{I}$ such that $f(x) = \emptyset$, we have $A(x) = \perp$.

We can have multiple algorithms that return the correct output. We typically distinguish between algorithms by comparing their efficiency.

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

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 n of the array x of key-value pairs. In the upcoming Sender-Receiver Exercise on Counting Sort, we will measure the input size as a function of both the array length n as well as size U of the key universe.

Informal Definition 3.1 (running time)**.** [1] For an algorithm A and input size function size, the *(worst-case) running time* of A is the function $T : \mathbb{N} \to \mathbb{R}^+$ given by:

$$T(n) = \max_{x:\text{size}(x)\leq n} (\#\text{ "basic operations" perfomed by } A \text{ on input } x)\,.$$

The above definition is edited from the presentation in class in order to address questions raised in class, in particular how we can use $\Omega(\cdot)$ notation with worst-case runtimes.

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. $\text{size}(x)$ is a typically problem-dependent measure of size, such as the length of the input array in the case of sorting. Note that we are measuring *worst-case* running time; $T(n)$ must upper-bound the running time of A on *all* inputs of size at most n.

Note that we define $T(n)$ as a maximum over inputs of size *at most* n, not just equal to n, as a notational convenience to guarantee that our runtime functions $T(n)$ are defined for all real numbers n, not just integers, and are nondecreasing (i.e. $T(x) \geq T(y)$ whenever $x \geq y$); in other contexts it's more natural to $T(n)$ as a maximum over inputs of size *exactly* 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 3.2. Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say:

- $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all sufficiently large n.

[1] This definition is changed slightly from lecture to define not only upper bounds on the worst-case running time, but the worst-case running time itself, as a maximum over all inputs of a given size.

– **Example:** for $T(n) = 3n^2 + 2n + 5$, we can take $c = 3.1$ to show that $T(n) = O(n^2)$.

- $f = \Omega(g)$ 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, $\lim_{n\to\infty} f(n)/g(n) = \infty$. Equivalently, $g = o(f)$.

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, starting with 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

Let $T_{exhaustsort}(n)$ be the worst-case running time of Exhaustive-Search Sort.

Q: Why isn't the runtime just $n!$?

In the worst case (what case is that?), the loop in Line 1 will be executed $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$ times, because there are $n!$ permutations on n elements. Line 2 is not a "basic operation": it involves $n - 1 \leq$ comparisons and would be implemented with a loop from $i = 0$ to $n - 1$ that compares $K_{\pi(i)}$ to $K_{\pi(i+1)}$. Thus we have on the order of $n! \cdot (n - 1)$ basic operations, i.e.

$$T_{exhaustsort}(n) = \Theta(n! \cdot (n - 1))$$

We remark that in CS50, $O(\cdot)$ notation is used to upper-bound worst-case running time, and $\Omega(\cdot)$ to lower-bound best-case running time. However, our definitions of asymptotic notation can be applied to any positive function on $\mathbb{N}$, so it makes sense for us to write that $T_{exhaustsort}(n) = \Theta(n! \cdot (n-1))$, where $T_{exhaustsort}(n)$ is the worst-case running time as defined in Definition 3.1.

Now let's turn to Insertion Sort:

```
   Input    : An array A = ((K_0, V_0), ..., (K_{n-1}, V_{n-1})), where each K_i ∈ ℝ
   Output   : A valid sorting of A
1  /* "in-place" sorting algorithm that modifies A until it is sorted */
2  foreach i = 0, ..., n − 1 do
3  |  Insert A[i] into the correct place in (A[0], ..., A[i − 1])
4  |  /* Note that when i = 0 this is the empty list.  */
5  return A
```

Algorithm 2: Insertion Sort

In contrast, here the outer loop (Line 2) executes a fixed number of times (namely n times). Expanding Line 3 into a loop of its own, we see that it takes on the order of i basic operations. Thus, the overall runtime is:

$$T_{insertsort}(n) = \Theta\left(\sum_{i=0}^{n-1} i\right) = \Theta(n^2).$$

Finally, let's look at Merge Sort:

```
1  MergeSort(A)
   Input    : An array A = ((K_0, V_0), ..., (K_{n-1}, V_{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 ((K_1, V_1), (K_0, V_0));
5  else
6  |  i = ⌈n/2⌉
7  |  A_1 = MergeSort(((K_0, V_0), ..., (K_{i-1}, V_{i-1})))
8  |  A_2 = MergeSort(((K_i, V_i), ..., (K_{n-1}, V_{n-1})))
9  |  return Merge (A_1, A_2)
10 |
```

Algorithm 3: Merge Sort

Rather than directly analyzing the runtime, we can write a recurrence. For $n \geq 3$, we have

$$\begin{aligned} T_{mergesort}(n) &= T_{mergesort}(\lceil n/2 \rceil) + T_{mergesort}(\lfloor n/2 \rfloor) + T_{merge}(n) + \Theta(1) \\ &= T_{mergesort}(\lceil n/2 \rceil) + T_{mergesort}(\lfloor n/2 \rfloor) + \Theta(n). \end{aligned}$$

It's a bit messy to solve such recurrences with the floors and ceilings, but when n is a power of 2, we can solve it by unrolling (*omitted in lecture due to time*):

$$\begin{aligned} T_{mergesort}(n) &= 2 \cdot T_{mergesort}(n/2) + c \cdot n \\ &= 2 \cdot (2 \cdot T_{mergesort}(n/4) + c \cdot (n/2)) + c \cdot n \\ &= 4 \cdot T_{mergesort}(n/4) + 2c \cdot n \\ &= 4 \cdot (2 \cdot T_{mergesort}(n/8) + c \cdot (n/8)) + 2c \cdot n \\ &= 8 \cdot T_{mergesort}(n/8) + 3c \cdot n \\ &= \cdots \\ &= 2^\ell \cdot T_{mergesort}(n/2^\ell) + \ell c \cdot n \end{aligned}$$

for $\ell = 0, 1, \ldots, (\log n) - 1$, where here and throughout CS 120 all logs are base 2 unless otherwise specified. Taking $\ell = (\log n) - 1$, we get

$$T_{mergesort}(n) = \frac{n}{2} \cdot T_{mergesort}(2) + \Theta(n \log n) = \Theta(n \log n).$$

When n is not a power of 2, we can let n' be the smallest power of 2 such that $n' \geq n$. Then

$$T'_{mergesort}(n) \leq T_{mergesort}(n') = \Theta(n' \log n') = \Theta(n \log n).$$

The first inequality follows from the fact that we are taking the maximum running time over inputs of length at most n, so increasing n can only increase the maximum. The last equality follows from the fact that $n \leq n' \leq 2n$.

Exercise 3.3. 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 3.4. Which of the following correctly describe the asymptotic (worst-case) runtime of each of the three sorting algorithms? (Include all that apply.)

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

- $T_{exhaustsort}(n) = \Theta(n! \cdot n) \quad (\neq O(n^2), o(2^n), \Theta(n \log n))$
- $T_{insertsort}(n) = \Theta(n^2) \quad (\neq \Omega(n!), \Theta(n \log n))$
- $T_{mergesort}(n) = \Theta(n \log n) \quad (\neq \Omega(n!))$

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 $T(n) = n^{O(1)}$ (reasonably efficient)

(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).

3.3 Complexity of Comparison-based Sorting

We did have time to discuss this in class, and instead just gave an informal overview of the idea behind it in the 9/8 lecture. It is recommended that you read this section through the statement of Theorem 3.5 and the paragraph immediately after it, but study the proof only if you are interested!

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` $\leq$ `var'`: returns a boolean (`true`/`false`) value according to whether the value of `var` is $\leq$ 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 3.5. *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 3.5 holds even when the values are all empty.

Lemma 3.6. *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).

1. First comparison: $K_1 \leq K_0$?
2. If $K_1 \leq K_0$, check $K_2 \leq K_0$?
3. Else, check $K_1 \leq K_2$?

Proof Sketch of Lemma 3.6. 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 `var` = `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 3.5. For a permutation $\sigma : [n] \to [n]$, define the input array

$$x_\sigma = (K_0, \ldots, K_{n-1}) = (\sigma(0), \sigma(1), \ldots, \sigma(n-1)).$$

That is, the keys are the numbers $0, \ldots, n-1$ permuted by σ. Let's consider the behavior of $A(x_\sigma)$ for each of the $n!$ permutations σ. Since A is a correct sorting algorithm,

$$A(x_\sigma) = (K_{\pi(0)}, K_{\pi(1)}, \ldots, K_{\pi(n-1)}),$$

for a permutation π such that $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$. The only permutation that satisfies this property is $\pi = \sigma^{-1}$ (i.e. the one that "undoes" the initial permutation σ).

By Lemma 3.6, the permutation π depends only on the the boolean results of the comparisons made by A on input x_σ. Since A can make at most $T(n)$ comparisons if it runs in time T and each comparison has only two possible results (`true` or `false`), there are at most $2^{T(n)}$ possibilities for the output permutation π.

But each of the $n!$ choices for σ must lead to a different output permutation π (namely $\pi = \sigma^{-1}$). Thus:

$$2^{T(n)} \geq n! \geq n \cdot (n-1) \cdot (n-2) \cdots (n/2) \geq \left(\frac{n}{2}\right)^{n/2}.$$

Taking logarithms, we have:

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

□