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

Lecture 2: Measuring Efficiency

Harvard SEAS - Fall 2021 *Sept. 7, 2021*

1 Announcements

- Name placards
- Introduce yourselves when speaking
- Log in to Ed during class
- PS0 due Wednesday
- My OH today 11:15-12 at 3.327
- Handout: Lecture notes 2 (PDF in Ed)
- Watch for active learning assignment for this class

2 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

2.1 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 of the array x of item-key pairs.

Informal Definition 2.1 (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 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 2.2. 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.
- $f = \Omega(g)$ 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, $g = o(f)$.

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.

2.2 Computational Complexity of Sorting

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

- Exhaustive-search sort:

  $T_{exhaust} \leq$ (# permutations on n elements) $\cdot$ (time to check if a permutation is valid).

- Insertion-sort:

  $T_{insert} \leq \sum_{i=1}^{n}$(time to insert element i).

- Merge sort:

  Rather than directly analyzing the runtime, we can write a recursive formula:
  $T_{merge}(n) = 2T_{merge}(n/2) +$ (Time to merge 2 lists of size $n/2$).

Exercise 2.3. Which of the following correctly describe the asymptotic runtime of each of the three sorting algorithms? (Include all that apply.)

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

- $T_{exhaust}(n) = \Theta(n! \cdot n) \quad (\neq O(n^2), o(2^n), \Theta(n \log n))$
- $T_{insert}(n) = \Theta(n^2) \quad (\neq \Omega(n!), \Theta(n \log n))$
- $T_{merge}(n) = \Theta(n \log n) \quad (\neq \Omega(n!))$

Exercise 2.4. 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).

2.3 Complexity of Comparison-based Sorting

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)
First comparison: $K_1 ? \leq K_0$.
If $K_1 \leq K_0$, check $K_2 \leq K_0$.
Else, check $K_1 \leq K_2$.

Generalizing the above example, we see that we can model a comparison-based algorithm as a binary tree (specifically a "decision tree"), whose depth 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 2.5. *If A is a comparison-based sorting algorithm that makes at most $T(n)$ comparisons on every 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. Observation 1:

$$\#\text{ distinct output permutations} \leq 2^{T(n)}$$

Observation 2:

$$\#\text{ distinct output permutations} \geq n!.$$

To justify Observation 2, we need to exhibit a family of input arrays that lead to 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 leads to output permutation $\pi = \sigma^{-1}$. Thus, there must be at least $n!$ possible different outputs.

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 $2^{T(n)}/n!$