CS1200: Intro. to Algorithms and their Limitations Prof. Anurag Anshu

Lecture 3: Computational complexity of Sorting

Harvard SEAS - Spring 2026 *2025-09-09*

1 Announcements

- Anurag's OH: Wed 12:45-1:45pm SEC 3.323.
- Reminder about revision videos to improve your pset grades; don't despair or spend 15+ hours if you haven't solved everything.
- Student-led AI safety workshop (see Ed).

2 Recommended Reading

- Hesterberg–Vadhan, Sections 2.2–2.4.
- Roughgarden I, Ch. 2
- CLRS 3e, Ch. 8
- Lewis–Zax Ch. 21

3 Loose ends: Measuring efficiency

Informal Definition 3.1 (running time)**.** For an algorithm A, an input set $\mathcal{I}$, and input size function $\mathsf{size} : \mathcal{I} \to \mathbb{N}$, the *(worst-case) running time* of A is the function $T : \mathbb{R}^{\geq 0} \to \mathbb{N}$ given by:

$$T(n) =$$

This is referred to as *worst-case* running time because we take the *maximum* runtime over all inputs of size at most n. A couple of choices in the definition of $T(n)$ may seem unusual, but turn out to be technically convenient:

- We take the maximum over inputs of length *at most* n, not just equal to n.
- The domain of T is $\mathbb{R}^+$, not just $\mathbb{N}$.

For flexibility, we also introduce a variant definition that considers only inputs of size equal to n.

Informal Definition 3.2 (running time variant)**.** For an algorithm A, an input set $\mathcal{I}$, and input size function $\mathsf{size} : \mathcal{I} \to \mathbb{N}$, the *(worst-case) running time for fixed-size inputs* of A is the function $T^{=} : \mathbb{N} \to \mathbb{N}$ given by:

$$T^{=}(n) =$$

Note that for all $n \in \mathbb{N}$, $T^{=}(n) \leq T(n)$ and these are usually equal.

Remarks.

- *Basic operations:* Basic operations are arithmetic on individual numbers, manipulation of pointers, and writing/reading individual numbers to/from memory.

  Q: Should the Python function `A.sort()` count as a basic operation?

- *Worst-case runtime:*

- *Other notions of efficiency:*

To avoid having our evaluations of algorithms depend on minor differences in the choice of "basic operations" and instead reveal more fundamental differences between algorithms, we generally measure complexity with asymptotic growth rates, using big-O notation and variants.

4 Asymptotic Notation

To avoid having our evaluations of algorithms depend on minor differences in the choice of "basic operations" and instead reveal more fundamental differences between algorithms, we generally measure complexity with asymptotic growth rates, for which we review "asymptotic notation":

Definition 4.1. Let $h, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say:

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

- $h = \omega(g)$ if
  Equivalently:

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

5 Computational Complexity of Sorting

Let's analyze the runtime of the sorting algorithms covered so far. We assume that comparison between two integers (that is, whether $a < b$ is true or false) is a basic operation. This follows from the assumption that subtraction $b - a$ is a basic operation.

```
  ExhaustiveSearchSort($A$):
  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 $A$
0 foreach permutation $\pi : [n] \to [n]$ do
1     if $K_{\pi(0)} \le K_{\pi(1)} \le \cdots \le K_{\pi(n-1)}$ then
2         return $A' = ((K_{\pi(0)}, V_{\pi(0)}), (K_{\pi(1)}, V_{\pi(1)}), \ldots, (K_{\pi(n-1)}, V_{\pi(n-1)}))$
3 return $\perp$
```

Algorithm 5.1: `ExhaustiveSearchSort()`

Let $T_{exhaustsort}(n)$ be the worst-case running time of `ExhaustiveSearchSort`.

$T_{exhaustsort}(n) =$

Contrast the use of $\Omega(\cdot)$ to lower-bound the *worst-case* running time, as done above, with the use of $\Omega(\cdot)$ to lower-bound the *best-case* running time. The definition of $\Omega(\cdot)$ can be applied to either of those functions (or, indeed, to any positive function on $\mathbb{N}$). Some introductory programming classes such as Harvard's CS 50 use $\Omega(\cdot)$ (only) to bound best-case running times, giving upper and lower bounds on the time for *every* execution of the algorithm. Our purpose in giving an $\Omega(\cdot)$ lower bound on the *worst-case* running time of an algorithm such as exhaustive-search sort is to check whether our $O(\cdot)$ upper bound is tight.

```
  InsertionSort($A$):
  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 $A$
0 /* "in-place" sorting algorithm that modifies $A$ until it is sorted */
1 foreach $i = 0, \ldots, n-1$ do
2     /* Insert $A[i]$ into the correct place among $(A[0], \ldots, A[i-1])$.  */
3     Find the first index $j$ such that $A[i][0] \leq A[j][0]$;
4     Insert $A[i]$ into position $j$ and shift $A[j \ldots i-1]$ to positions $j+1, \ldots, i$
5 return $A$
```

Algorithm 5.2: `InsertionSort()`

$T_{insertsort}(n) =$

For the input keys $K_0 = n-1, K_1 = n-2, \ldots K_{n-1} = 0$, Line 3 will have to make about i shifts. Thus $T_{insertsort}(n) = \Omega(n^2)$, which means $T_{insertsort}(n) = \Theta(n^2)$.

```
  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 $A$
0 if $n \leq 1$ then return $A$;
1 else
2     $i = \lceil n/2 \rceil$
3     $B =$ MergeSort($((K_0, V_0), \ldots, (K_{i-1}, V_{i-1}))$)
4     $B' =$ MergeSort($((K_i, V_i), \ldots, (K_{n-1}, V_{n-1}))$)
5     return Merge $(B, B')$
```

Algorithm 5.3: `MergeSort()`

In order to analyze the runtime of this algorithm, we introduce a recurrence relation. From the description of the `MergeSort` algorithm, we find that

$$T_{mergesort}(n) \leq T_{mergesort}(\lceil n/2 \rceil) + T_{mergesort}(\lfloor n/2 \rfloor) + \Theta(n).$$

Solving such recurrences with the floors and ceilings can be generally complicated, but it is much simpler when n is a power of 2. In this case, When n is not a power of 2, we can let n' be the smallest power of 2 such that $n' \geq n \geq \frac{n'}{2}$. Then

$$T_{mergesort}(n) \leq T_{mergesort}(n') =$$

We can summarize what we've done above in the following "theorem," which will remain informal until we precisely specify our computational model and what constitutes a basic operation, which we will do in a couple of weeks.

Theorem 5.1 (runtimes of algorithms for Sorting, informal). *The worst-case runtimes of* `ExhaustiveSearchSort`*,* `InsertionSort`*, and* `MergeSort` *are $\Theta(n! \cdot n)$, $\Theta(n^2)$, and $\Theta(n \log n)$, respectively.*

To solidify your understanding, let's do the following exercise:

Problem .1 (Comparing runtimes of sorting algorithms). Let $T_{exhaustsort}, T_{insertsort}, T_{mergesort}$ be the worst-case runtimes of `ExhaustiveSearchSort`, `InsertionSort`, and `MergeSort`, respectively.

1. Order $T_{exhaustsort}, T_{insertsort}, T_{mergesort}$ from fastest- to slowest-growing,[1] i.e. number them T_0, T_1, T_2 such that $T_0 = o(T_1)$ and $T_1 = o(T_2)$.

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

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

Throughout this course, we will be interested in three very coarse categories of running time, of which our three sorting algorithms are exemplars:

(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)

All of the above algorithms are "comparison-based" sorting algorithms: the only way by which they use the keys is by comparing them to see whether one is larger than the other. It turns out that the worst-case running time of `MergeSort` cannot be improved by more than a constant factor if we stick with comparison-based sorting algorithms:

Theorem 5.2. *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.*

[1] Note that there exists sets of functions that cannot be so ordered, but the worst-case runtimes in this problem are sufficiently simple functions that they can be.

This is our first taste of what it means to establish *limits* of algorithms. A formalization of the concept of comparions-based algorithms and a proof of Theorem 5.2 is given in the lecture notes.

It may seem intuitive that sorting algorithms must work via comparisons, but in the Sender–Receiver Exercise you are about to do, you will see an example of a sorting algorithm that benefits from doing other operations on keys.