CS1200: Intro. to Algorithms and their Limitations Vadhan

Lecture 3: Complexity of Sorting, Interval Scheduling

Harvard SEAS - Fall 2025 *2025-09-09*

1 Announcements

- Salil's upcoming OH: Thu 3-3:45pm SEC 3.327.
- Reminder about revision videos to improve your pset grades; don't despair or spend 15+ hours if you haven't solved everything.

2 Recommended Reading

- Hesterberg–Vadhan, Sections 1.4–2.5.
- Roughgarden I, Ch. 2
- CLRS 3e Ch. 8
- Lewis–Zax Ch. 21

3 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 .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.

4 Computational Complexity of Sorting

Let's analyze the runtime of the sorting algorithms covered so far (more rigorously than last time).

```
   ExhaustiveSearchSort(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
0  foreach permutation π : [n] → [n] do
1  |  if K_{π(0)} ≤ K_{π(1)} ≤ ··· ≤ K_{π(n-1)} then
2  |  |  return A' = ((K_{π(0)}, V_{π(0)}), (K_{π(1)}, V_{π(1)}), ..., (K_{π(n-1)}, V_{π(n-1)}))
3  return ⊥
```

Algorithm .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), ..., (K_{n-1}, V_{n-1})), where each K_i ∈ ℝ
   Output       : A valid sorting of A
0  /* "in-place" sorting algorithm that modifies A until it is sorted */
1  foreach i = 0, ..., n − 1 do
2  |  /* Insert A[i] into the correct place among (A[0], ..., A[i − 1]).  */
3  |  Find the first index j such that A[i][0] ≤ A[j][0];
4  |  Insert A[i] into position j and shift A[j ... i − 1] to positions j + 1, ..., i
5  return A
```

Algorithm .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 comparisons. 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), ..., (K_{n-1}, V_{n-1})), where each K_i ∈ ℝ
   Output     : A valid sorting of A
0  if n ≤ 1 then return A;
1  else
2  |  i = ⌈n/2⌉
3  |  B = MergeSort(((K_0, V_0), ..., (K_{i-1}, V_{i-1})))
4  |  B' = MergeSort(((K_i, V_i), ..., (K_{n-1}, V_{n-1})))
5  |  return Merge (B,B')
```

Algorithm .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 .2 (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 .3. *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. A formalization of the concept of comparions-based algorithms and a proof of Theorem .3 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.

5 Interval Scheduling

Consider the following scenario: A small public radio station decided to raise money by allowing listeners to purchase segments of airtime during a particular week. However, they now need to check that all of the segments that they sold aren't in conflict with each other; that is, no two segments overlap.

When confronted with an informally described computational problem as above, our first task as algorithmicists is to figure out how to model it mathematically, so that we make our task precise and can match our algorithmic toolkit to it. In this modeling, we want to be sure to capture all of the essential details, while abstracting away inessential ones.

[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.

Input: A collection of

Output: YES if
NO otherwise

Computational Problem IntervalScheduling–Decision

Notice that this formulation as a computational problem has abstracted away lots of inessential details of our original problem, like the fact that it involves a radio station and segments of time. We can apply it equally well to the problem of allocating segments of sidewalk to food vendors along the route of the Boston Marathon. This points to another benefit of abstraction and mathematically modelling; it allows one solution (like an algorithm) to apply to many different problems.

There is a simple algorithm to solve IntervalScheduling–Decision in $O(n^2)$ runtime.

However, we can get a faster algorithm by a **reduction** to sorting.

Proposition .4. *There is an algorithm that solves* IntervalScheduling–Decision *for n intervals in time $O(n \log n)$.*

Proof. We first describe the algorithm.

```
IntervalSchedulingViaSorting(C):
  Input  : A collection C of intervals [a_0, b_0], ..., [a_{n-1}, b_{n-1}], where each a_i, b_i ∈ ℝ
           and a_i ≤ b_i
  Output : YES if intervals are disjoint, NO otherwise.
0 Set A =
1 A' =;
2 foreach i = 1, ..., n − 1 do
  |
```

Algorithm .5: `IntervalSchedulingViaSorting()`

We now want to prove that `IntervalSchedulingViaSorting` has the claimed runtime of $O(n \log n)$ and is correct.

a: **Runtime analysis:**

b: **Proof of correctness:**

□

Question: Define IntervalScheduling–Decision–OnFiniteUniverse to be a variant of IntervalScheduling–Decision where we are also given a universe size $U \in \mathbb{N}$ and the interval endpoints a_i, b_i are constrained to lie in $[U]$. Can you think of an algorithm for solving IntervalScheduling–Decision–OnFiniteUniverse in time $O(n + U)$?

Answer: