CS1200: Intro. to Algorithms and their Limitations Vadhan

Lecture 2: Computational Problems and their Complexity

Harvard SEAS - Fall 2025 *2025-09-04*

1 Announcements

- Salil's upcoming OH: today 11:15-12:30 SEC 3.327; Mon 2-2:45 Zoom.
- First Sender–Receiver Exercise *on Tuesday*. Watch your email for your assignment and come prepared!

2 Recommended Reading

- Hesterberg–Vadhan, Sections 1.4–2.5.
- 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 Loose End: Correctness Proof for Insertion Sort

`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

`/* "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 among` $(A[0], \ldots, A[i-1])$`. */`
  Find the first index j such that $A[i][0] \leq A[j][0]$;
  Insert $A[i]$ into position j and shift $A[j \ldots i-1]$ to positions $j+1, \ldots, i$
return A

Algorithm .1: `InsertionSort()`

Theorem .1. `InsertionSort` *correctly solves the* Sorting *problem. That is, for every input array* A*,* `InsertionSort(A)` *returns a valid sorting of* A*.*

Note that, as a side-effect, here we will also prove that every array A has a valid sorting. This phenomenon, of proving that a mathematical object exists (e.g. a valid sorting of an arbitrary array A) by exhibiting and analyzing an algorithm to construct it, is a quite common and useful one. See Problem Set 0 for another example.

Proof. For $i = 0, \ldots, n$, let $A^{(i)}$ be the contents of the array A before iteration i and/or after iteration $i-1$ of the for-loop. (Both definitions of $A^{(i)}$ make sense and are equivalent when $0 < i < n$.) In particular, $A^{(0)}$ is the input array and $A^{(n)}$ is the output array.
Proof strategy: We'll prove by induction on i (from $i = 0, \ldots, n$) the following "loop invariant" $P(i)$:

"$A^{(i)}[0 \ldots i-1]$ is a valid sorting of $A^{(0)}[0 \ldots i-1]$ and $A^{(i)}[i \ldots n-1] = A^{(0)}[i \ldots n-1]$."

Notice that the statement $P(n)$ says that $A^{(n)}$, which is the output of `InsertionSort(`A`)`, is a valid sorting of A, as desired.

Base Case ($P(0)$):

Inductive Step ($P(i) \Rightarrow P(i+1)$):

$\square$

Remarks.

- Not all correctness proofs for algorithms use induction! (cf. Exhaustive-Search Sort.)
- This was not a fully formal proof. It is often necessary to skip steps to make such proofs manageable for humans, but you should be careful when do so. Be sure that (a) you are completely convinced of the correctness of your claims, and (b) you are not omitting the main point or idea of the argument.

4 Loose End: Merge Sort

Finally, you may have already seen the following even more efficient sorting algorithm, `MergeSort`. The idea is to recursively sort each half of the array and then efficiently "merge" the two sorted halves into a single, sorted array:

We may not cover this in lecture (depending on time), since most of you have already seen it in CS 50. But you should review it from the textbook on your own!

`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
    $i = \lceil n/2 \rceil$
    $B =$ `MergeSort`$(((K_0, V_0), \ldots, (K_{i-1}, V_{i-1})))$
    $B' =$ `MergeSort`$(((K_i, V_i), \ldots, (K_{n-1}, V_{n-1})))$
    return `Merge` (B, B')

Algorithm .2: `MergeSort()`

We omit the implementation of `Merge`, which you can find in the textbook.

Theorem .2. `MergeSort` *correctly solves the* Sorting *problem. That is, for every input array A,* `MergeSort`(A) *returns a valid sorting of A.*

Naturally, the proof of this theorem will rely on the correctness of `Merge`, whose proof we leave as an exercise:

Lemma .3. *If B and B' are sorted arrays, then* `Merge`(B, B') *is a valid sorting of $B \circ B'$.*

Proof Sketch of Theorem .2. Like with `InsertionSort`, this is a proof by induction, but we use *strong* induction. (Why?)

Here, the statement we will prove by (strong) induction is simpler than for `InsertionSort`. It is simply

$$P(n) = \text{"}\texttt{MergeSort} \text{ correctly sorts arrays of size } n.\text{"}$$

□

5 Computational Problems

In the theory of algorithms, we want to not only study and compare a variety of different algorithms for a single computational problem like Sorting, but also study and compare a variety of different computational problems. We want to classify problems according to which ones have efficient algorithms, which ones only have inefficient algorithms, and which ones have no algorithms at all. We also want to be able to relate different computational problems to each other, via the concept of *reductions* that we will see in Chapter **??**. All of this requires having an abstract definition of what a computational problem is, and what it means for an algorithm to solve a computational problem.

Definition .4. A *computational problem* Π is a triple $(\mathcal{I}, \mathcal{O}, f)$ where:

- $\mathcal{I}$ is a (typically infinite) set of possible inputs (a.k.a. *instances*) x, and $\mathcal{O}$ is a (sometimes infinite) set of possible outputs y.
- For every input $x \in \mathcal{I}$, a *set* $f(x) \subseteq \mathcal{O}$ of *valid outputs* (a.k.a. *valid answers*).

Example .5. Let's put the SORTING problem described in Section **??** into the formalism of Definition .4.

- $\mathcal{I} = \mathcal{O} =$

- $f(x) =$

Note that there can be multiple valid outputs, which is why $f(x)$ is a set. For instance, in the example of SORTING above, if the input array x has pairs (K_i, V_i) and (K_j, V_j) with $K_i = K_j$ but $V_i \neq V_j$, then there are multiple valid answers.

The following definition is an informal way to describe an algorithm.

Informal Definition .6. An *algorithm* is a well-defined "procedure" A for "transforming" any input x into an output $A(x)$.

We will be more precise about this definition in a few weeks, but for now you can think of a "procedure" as something that you can write as a computer program or in pseudocode like we have seen for sorting algorithms.

The next definition tells us what it means to "solve" a computational problem.

Definition .7. Algorithm A *solves* computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ if the following holds:

Remarks.

- An algorithm A is supposed to have a fixed, finite description; and it should correctly solve the problem Π for *all* of the (infinitely many) inputs in the set $\mathcal{I}$. For instance, all the sorting algorithms discussed in Chapter **??** were described fairly concisely. In contrast, it would not qualify as an "algorithm" to list all (infinitely many) arrays of pairs and specify, for each of them, a sorting of it.

- Our proofs of correctness of sorting algorithms are exactly proofs that the algorithms satisfy Definition .7. This holds generally and we will frequently return to this definition throughout the course.

A fundamental point in the theory of algorithms is that we distinguish between computational problems and algorithms that solve them. A single computational problem may have many different algorithms that each solve it (or even no algorithm that solves it!), and our focus will be on trying to identify the most efficient among these.

6 Measuring Efficiency

To measure the efficiency of an algorithm, we consider how its computation time *scales* with the size of its input. To do so, we first define a size parameter, a function $\mathsf{size} : \mathcal{I} \to \mathbb{R}^{\geq 0}$. For example, in sorting, we typically let $\mathsf{size}(x)$ be the length n of the array x of key-value pairs. Sometimes we define (and measure the efficiency of algorithms in terms of) multiple size parameters: for instance, in the upcoming Sender-Receiver Exercise on `SingletonBucketSort`, we will measure the input size as a function of both the array length n and the size U of the universe of possible keys.

Informal Definition .8 (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 .9 (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, for which we review "asymptotic notation":

Definition .10. 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.

7 Computational Complexity of Sorting

Let's analyze the runtime of the sorting algorithms covered so far.

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

Algorithm .3: `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 .4: `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
if n ≤ 1 then return A;
else
    i = ⌈n/2⌉
    B = MergeSort(((K_0, V_0), ..., (K_{i-1}, V_{i-1})))
    B' = MergeSort(((K_i, V_i), ..., (K_{n-1}, V_{n-1})))
    return Merge (B,B')
```

Algorithm .5: `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 .11 (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)

[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.