CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 2: Computational Problems and their Complexity

Harvard SEAS - Fall 2024 *2024-09-05*

1 Announcements

- Detailed Lecture 1 notes posted on course calendar
- Handout: Lecture notes 2 (PDF on course calendar)
- Sender–Receiver exercise at start of class on Tuesday; prepare and come on time!

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 Loose Ends from Lec 1

- Revisiting insertion sort and its proof of correctness.
- Discussion on merge sort.

2.1 Insertion sort

The insertion sort is an intuitive sorting algorithm, as discussed in Lecture 1.

```
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 $A[i][0] \le A[j][0]$;
    Insert $A[i]$ into position $j$ and shift $A[j \ldots i-1]$ to positions $j+1, \ldots, i$
return $A$
```

Algorithm 1: Insertion Sort

Example: $A = ((6, a), (2, b), (1, c), (4, d))$.

As our algorithm runs, we produce the following sorted sub-arrays:

Iteration i	Array contents after iteration i
0	
1	
2	
3	

Proof of correctness: Notation:

Proof strategy: We'll prove by induction on i (from $i = 0, \ldots, n$) the "loop invariant":

$P(i) =$

Base Case ($P(0)$):

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

2.2 Merge sort

```
1 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 $A$
2 if $n \leq 1$ then return $A$;
3 else
4     $i = \lceil n/2 \rceil$
5     $A_1 =$ MergeSort($((K_0, V_0), \ldots, (K_{i-1}, V_{i-1}))$)
6     $A_2 =$ MergeSort($((K_i, V_i), \ldots, (K_{n-1}, V_{n-1}))$)
7     return Merge ($A_1$,$A_2$)
```

Algorithm 2: Merge Sort

The proof of correctness uses strong induction on the statement $P(n) =$ "`MergeSort` correctly sorts arrays of size n." See Lecture 1 notes for details.

3 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 next week. All of this requires having an abstract definition of what is a computational problem, and what it means for algorithm to solve a computational problem.

Definition 3.1. 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: Sorting:

- $\mathcal{I} = \mathcal{O} =$
- $f(x) =$

Note that there can be multiple valid outputs, which is why $f(x)$ is a set.

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

Informal Definition 3.2. 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.

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

Definition 3.3. 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, we were able to fairly quickly describe the sorting algorithms in the lectures.

- Our proofs of correctness of sorting algorithms are exactly proofs that the algorithms fit Definition 3.3. 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 solves it (or even no algorithm that solves it!), and our focus will be on trying to identify the most efficient among these.

4 Measuring Efficiency

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 $\mathsf{size}(x) \geq 0$. For example, in sorting, we typically let $\mathsf{size}(x)$ 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 4.1 (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 $T : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ given by:

$$T(n) =$$

The definition of $T(n)$ seems somewhat unusual - being a maximum over inputs of size *at most* n, not just equal to n. However, it allows significant notational convenience since $T(n)$ is defined for all real numbers n, not just integers, and is a nondecreasing function (i.e. $T(x) \geq T(y)$ whenever $x \geq y$). For flexibility, we also introduce the definition that considers inputs of size equal to n.

Informal Definition 4.2 (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 sized inputs* of A is the function $T^{=} : \mathbb{N} \to \mathbb{R}^{\geq 0}$ given by:

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

Remarks.

- *Basic operations:* Basic operations can be viewed as arithmetic on individual numbers, manipulating pointers, stepping through a line of code, 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 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. This is where the asymptotic notations become highly relevant:

Definition 4.3. 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 of the 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.

4.1 Computational Complexity of Sorting

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

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 3: Exhaustive-Search Sort

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

$T_{exhaustsort}(n) =$

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 4.1. We can also apply asymptotic notation to best-case running time and average-case running time, but in this course, we will mainly focus on worst-case running time.

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

Algorithm 4: Insertion Sort

$T_{insertsort}(n) =$

For the input keys $K_0 = n-1, K_1 = n-2, \ldots K_{n-1} = 0$, Line 4 will have to make about i comparisons. Thus $T_{insertsort}(n) = \Omega(n^2)$, which means $T_{insertsort}(n) = \Theta(n^2)$.

Merge Sort:

```
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
    $i = \lceil n/2 \rceil$
    $A_1 =$ MergeSort($((K_0, V_0), \ldots, (K_{i-1}, V_{i-1}))$)
    $A_2 =$ MergeSort($((K_i, V_i), \ldots, (K_{n-1}, V_{n-1}))$)
    return Merge ($A_1$,$A_2$)
```

Algorithm 5: Merge Sort

In order to analyze the runtime of the algorithm, we introduce a recurrence relation. Definitions 4.1 and 4.2 imply

$$T_{mergesort}(n) = \max_{m \leq n} T^{=}_{mergesort}(m).$$

From the description of the Merge Sort algorithm, we find that

$$\begin{aligned} T^{=}_{mergesort}(m) &\leq T^{=}_{mergesort}(\lceil m/2 \rceil) + T^{=}_{mergesort}(\lfloor m/2 \rfloor) + T^{=}_{merge}(m) + \Theta(1) \\ &= T^{=}_{mergesort}(\lceil m/2 \rceil) + T^{=}_{mergesort}(\lfloor m/2 \rfloor) + \Theta(m). \end{aligned}$$

Here, $T_{merge}(m)$ is the runtime to merge two arrays of size at most m; we use without proof the fact that $T_{merge}(m) = \Theta(m)$. Since, $m \leq n$, we have $T^{=}_{mergesort}(\lceil m/2 \rceil) \leq T_{mergesort}(\lceil n/2 \rceil)$ and $T^{=}_{mergesort}(\lfloor m/2 \rfloor) \leq T_{mergesort}(\lfloor n/2 \rfloor)$. Thus, we obtain the following recurrence relation:

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

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

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

Exercise 4.4. 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 4.5. Which of the following correctly describe the asymptotic (worst-case) runtime of each of the three sorting algorithms? (Include all that apply.)

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

- $T_{exhaustsort}(n) =$
- $T_{insertsort}(n) =$
- $T_{mergesort}(n) =$

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)

4.2 Complexity of Comparison-based Sorting

It is recommended that you read this section through the statement of Theorem 4.6 and the paragraph immediately after it, but study the proof only if you are interested! You can also refer to the Section 1 notes for an intuitive explanation.

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 Tuesday'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 4.6. *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.