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, ..., n − 1 do
    /* Insert A[i] into the correct place among (A[0], ..., A[i − 1]). */
    Find the first index j such that A[i][0] ≤ A[j][0];
    Insert A[i] into position j and shift A[j ... i − 1] to positions j + 1, ..., 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	((6,a),(2,b),(1,c),(4,d))
1	((2,b),(6,a),(1,c),(4,d))
2	((1,c),(2,b),(6,a),(4,d))
3	((1,c),(2,b),(4,d),(6,a))

See Lecture 1 notes for detailed proof of correctness, which uses induction on a "loop invariant" that roughly says that in the ith step of the loop, first i indices of A have been correctly sorted and the remaining indices are unchanged.

2.2 Merge sort

```
1 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
2 if n ≤ 1 then return A;
3 else
4 |  i = ⌈n/2⌉
5 |  A_1 = MergeSort(((K_0, V_0), ..., (K_{i-1}, V_{i-1})))
6 |  A_2 = MergeSort(((K_i, V_i), ..., (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:

- $\mathcal{I} = \mathcal{O} = \{$All arrays of key-value pairs with keys in $\mathbb{R}\}$

- $f(x) = \{\text{All valid sorts of } x\}$

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:

1. For every input $x \in \mathcal{I}$ with $f(x) \neq \emptyset$, we have $A(x) \in f(x)$.
2. There is a special output $\perp \notin \mathcal{O}$ such that for every input $x \in \mathcal{I}$ with $f(x) = \emptyset$, we have $A(x) = \perp$.

Condition 1 says that when there is a valid output on input x, the algorithm A must find one. This is the main condition. Condition 2 says that if there is no valid output, the Algorithm A must report so with the special output $\perp$ (a failure code).

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:

$$T(n) = \max_{x \in \mathcal{I} : \mathsf{size}(x) \leq n} (\#\text{ "basic operations" performed by } A \text{ on input } x)\,.$$

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:

$$T^{=}(n) = \max_{x : \mathsf{size}(x) = n} (\#\text{ "basic operations" performed by } A \text{ on input } x)\,.$$

Remarks.

- *Basic operations:* We will make this definition more formal in a couple of weeks, but the basic operations can be viewed as arithmetic on individual numbers, manipulating pointers, stepping through a line of code, and writing/reading individual numbers to/from memory. 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.

- *Worst-case runtime:* We are measuring the worst-case running time; $T(n)$ must be the maximum running time of A on *all* inputs of size at most n. 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).

- *Other notions of efficiency:* We will be focusing on runtime as the measure of efficiency of algorithms, as it is the most basic and natural measure in the context of computation. We note that in Computer Science, various other notions are of importance. For example, space complexity is a measure of how much space the algorithm needs to run and clearly important for hardware considerations. Energy efficiency turns out to be a crucial measure in large scale computation, and especially relevant these days due to the worldwide use of Generative AI.

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 there is a constant $c > 0$ such that $h(n) \leq c \cdot g(n)$ for all sufficiently large n.
  - **Example:** for $h(n) = 3n^2 + 2n + 5$, we can take $c = 3.1$ to show that $h(n) = O(n^2)$.
- $h = \Omega(g)$ if there is a constant $c > 0$ such that $h(n) \geq c \cdot g(n)$ for all sufficiently large n. Equivalently, $g = O(h)$.
- $h = \Theta(g)$ if $h = O(g)$ and $h = \Omega(g)$.
- $h = o(g)$ if for every constant $c > 0$, we have $h(n) \leq c \cdot g(n)$ for all sufficiently large n. Equivalently, $\lim_{n\to\infty} h(n)/g(n) = 0$.
- $h = \omega(g)$ if for every constant $c > 0$, we have $h(n) \geq c \cdot g(n)$ for all sufficiently large n. Equivalently, $\lim_{n\to\infty} h(n)/g(n) = \infty$. Equivalently, $g = o(h)$.

We also apply extend these definitions to functions $h, g : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ by interpreting "all sufficiently large n" to mean all sufficiently large real numbers n rather than all sufficiently large natural numbers n.

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. In the worst case, the loop in Line 1 will be executed $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$ times, because there are $n!$ permutations on n elements. Line 2 is not a "basic operation": it involves $n - 1$ comparisons and would be implemented with a loop from $i = 0$ to $n - 1$ that compares $K_{\pi(i)}$ to $K_{\pi(i+1)}$. Thus each loop iteration takes time $O(n)$ and

$$T_{exhaustsort}(n) = n! \cdot O(n) = O(n! \cdot n).$$

On the other hand, there are inputs on which one will have to go through all permutations before the condition on Line 2 is satisfied. Thus,

$$T_{exhaustsort}(n) = \Omega(n! \cdot n),$$

which collectively implies

$$T_{exhaustsort}(n) = \Theta(n! \cdot n).$$

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,...,n-1 do
3 |   /* Insert A[i] into the correct place among (A[0],...,A[i-1]).  */
4 |   Find the first index j such that A[i][0] ≤ A[j][0];
5 |   Insert A[i] into position j and shift A[j...i-1] to positions j+1,...,i
6 return A
```

Algorithm 4: Insertion Sort

Here the outer loop (Line 2) executes a fixed number of times (namely n times). Expanding Lines 4 and 5 into loops of their own, we see that they take $O(i)$ basic operations. Thus, the overall runtime is:

$$T_{insertsort}(n) = \sum_{i=0}^{n-1} O(i) = O\left(\sum_{i=0}^{n-1} i\right) = O(n^2).$$

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:

```
1 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
2 if n ≤ 1 then return A;
3 else
4 |   i = ⌈n/2⌉
5 |   A_1 = MergeSort((K_0,V_0),...,(K_{i-1},V_{i-1})))
6 |   A_2 = MergeSort((K_i,V_i),...,(K_{n-1},V_{n-1})))
7 |   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,

$$\begin{aligned} T_{mergesort}(n) &\leq 2 \cdot T_{mergesort}(n/2) + c \cdot n \\ &\leq 2 \cdot (2 \cdot T_{mergesort}(n/4) + c \cdot (n/2)) + c \cdot n \\ &= 4 \cdot T_{mergesort}(n/4) + 2c \cdot n \\ &\leq 4 \cdot (2 \cdot T_{mergesort}(n/8) + c \cdot (n/8)) + 2c \cdot n \\ &= 8 \cdot T_{mergesort}(n/8) + 3c \cdot n \\ &= \cdots \\ &\leq 2^{\ell} \cdot T_{mergesort}(n/2^{\ell}) + \ell c \cdot n \end{aligned}$$

for any natural number $\ell \leq \log n$ (here and throughout CS 1200 all logs are base 2 unless otherwise specified). Note that the constant c in the above equations upper bounds $T_{merge}(n) = \Theta(n) \leq cn$. Taking $\ell = \log n$, we get

$$T_{mergesort}(n) \leq n \cdot T_{mergesort}(1) + c(n \log n) = O(n \log n).$$

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

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

The first inequality follows from the fact that we are taking the maximum running time over inputs of length at most n, so increasing n can only increase the maximum. The last equality follows from the fact that $n \leq n' \leq 2n$. The analysis can be done more carefully in a manner that one gets the following bound $T_{mergesort}(n) = \Theta(n \log n)$ (for example, by using the fact that there is another constant c' such that $T_{merge}(n) \geq c'n$).

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.)

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

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

- $T_{insertsort}(n) = \Theta(n^2) \quad (\neq \Omega(n!), \Theta(n \log n))$
- $T_{mergesort}(n) = \Theta(n \log n) \quad (\neq \Omega(n!))$

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.

The key to the proof is the following lemma, which we state for input arrays consisting only of keys, since Theorem 4.6 holds even when the values are all empty.

Lemma 4.7. *If we feed a comparison-based algorithm A an input array $x = (K_0, K_1, \ldots, K_{n-1})$ consisting of elements of type `key` and the output $A(x)$ contains a variable K' of type `key`, then:*

1. *$K' = K_i$ for some $i = 0, \ldots, n-1$, and*
2. *The value of i depends only on the results of the boolean key comparisons that A makes on input x.*

Let's illustrate this with an example.

Example: insertion sort on the key array (K_0, K_1, K_2).

1. First comparison: $K_1 \leq K_0$?
2. If $K_1 \leq K_0$, check $K_2 \leq K_0$?
3. Else, check $K_1 \leq K_2$?

Proof Sketch of Lemma 4.7. Item 1 follows because a comparison-based algorithm does not have any way to create a variable of type `key` other than by copying (using the assignment operation $\texttt{var} = \texttt{var}'$).

For Item 2, the intuition is that A has access to no other information about the input other than the results of the comparisons it has made so far. We omit a formal proof. □

Proof of Theorem 4.6. For a permutation $\sigma : [n] \to [n]$, define the input array

$$x_\sigma = (K_0, \ldots, K_{n-1}) = (\sigma(0), \sigma(1), \ldots, \sigma(n-1)).$$

That is, the keys are the numbers $0, \ldots, n-1$ permuted by σ. Let's consider the behavior of A when run on x_σ for each of the $n!$ permutations σ. Since A is a correct sorting algorithm, its output $A(x_\sigma)$ satisfies

$$A(x_\sigma) = (K_{\pi(0)}, K_{\pi(1)}, \ldots, K_{\pi(n-1)}),$$

for a permutation π such that $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$. The only permutation that satisfies this property is $\pi = \sigma^{-1}$ (i.e. the one that "undoes" the initial permutation σ).

By Lemma 4.7, the permutation π depends only on the the boolean results of the comparisons made by A on input x_σ. Since A can make at most $T(n)$ comparisons if it runs in time T and each comparison has only two possible results (`true` or `false`), there are at most $2^{T(n)}$ possibilities for the output permutation π.

But each of the $n!$ choices for σ must lead to a different output permutation π (namely $\pi = \sigma^{-1}$). Thus:

$$2^{T(n)} \geq n! \geq n \cdot (n-1) \cdot (n-2) \cdots (n/2) \geq \left(\frac{n}{2}\right)^{n/2}.$$

Taking logarithms, we have:

$$T(n) \geq \frac{n}{2} \log_2 \left(\frac{n}{2}\right) = \Omega\left(n \log n\right).$$

□

4.3 Considerations when Using Asymptotic Notation

As you may have noticed above, we often use asymptotic notation inside expressions. For example, when we analyzed the runtime of `MergeSort` we wrote

$$T^{=}_{mergesort}(n) \quad \leq \quad T^{=}_{mergesort}(\lceil n/2 \rceil) + T^{=}_{mergesort}(\lfloor n/2 \rfloor) + T^{=}_{merge}(n) + \Theta(1)$$

The interpretation of a $\Theta(g(n))$ in the right-hand side of an equation or inequality like this means that the equation or inequality holds with the $\Theta(g(n))$ replaced by some unspecified function $f(n)$ such that $f(n) = \Theta(g(n))$ (and similarly for other asymptotic notation).

What about when we use asymptotic notation on both sides of an equation or inequality, like the following?

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

When we have asymptotic notation in both the left-hand side and the right-hand side of an equality or inequality, we mean that for every unspecified function f_1 that could substitute into the asymptotic notation on the left-hand side, there is an unspecified function f_2 that could substitute into the asymptotic notation on the right-hand side such that the equation holds. For example, we can write $O(1/n) = o(1)$ because for every function $f_1 = O(1/n)$ we have $f_1 = o(1)$. But the converse is not true, so we cannot write $o(1) = O(1/n)$. As a more sophisticated example, we can write

$$n^2 + O(n) = (1 + o(1)) \cdot n^2$$

because for every function $f_1(n) = O(n)$, there is a function $f_2(n) = o(1)$ such that $n^2 + f_1(n) = (1 + f_2(n)) \cdot n^2$ (namely $f_2(n) = f_1(n)/n^2$). On the other hand, we *cannot* write $(1+o(1)) \cdot n^2 = n^2 + O(n)$, because $1/\sqrt{n} = o(1)$ but $(1 + 1/\sqrt{n}) \cdot n^2 \neq n^2 + O(n)$.

On Sender-Receiver Exercise 1, you will see an example where we use asymptotic notation for a function of two parameters, the array length n and the key universe U, bounding the runtime $T(n, U)$ as $T(n, U) = O(n+U)$. This means that there is a constant $c > 0$, such $T(n, U) \leq c \cdot (n+U)$ whenever n is sufficiently large *or* U is sufficiently large. In the midst of bounding $T(n, U)$ in SRE1, we refer to the time of the j'th loop iteration as $O(|C[j]| + 1)$, where $|C[j]|$ is the length of a linked list in the algorithm, which may be as small as 0 even when n and U are large. In such cases, it is safest to avoid the "sufficiently large n" aspect of $O(\cdot)$ notation, and use such notation to mean that there is a constant c such that the time of the j'th loop iteration is always bounded by $c \cdot (|C[j]| + 1)$, to avoid accidentally assuming that $|C[j]|$ is large. Indeed, that is why we include the $+1$ inside the expression (to handle the case when $|C[j]| = 0$). In particular, $o(\cdot)$ and $\omega(\cdot)$ should not be used this way, since taking limits as $n \to \infty$ is necessary to make sense of these notations.