The goals of this exercise are:

- to develop your skills at understanding, distilling, and communicating proofs and the conceptual ideas in them

- to practice reasoning about the running time of algorithms

- to see how the choice of a model of computation can affect the computational complexity of a problem

In the previous class, we have seen that there exists a sorting algorithm whose worst-case running time is $O(n \log n)$. In fact, the (worst-case) computational complexity of sorting by comparison-based algorithms is $\Theta(n \log n)$. That is, *every* (correct) comparison-based sorting algorithm (one that operates just by comparing keys to each other) has worst-case running time $\Omega(n \log n)$. This holds even when the keys are drawn from the universe $[n] = \{0, 1, \ldots, n-1\}$.

In our first sender-receiver exercise, you will see that for keys drawn from the universe $[n]$, it is actually possible to sort asymptotically faster — in time $O(n)$! How is this possible in light of the $\Omega(n \log n)$ lower bound? Well, the algorithm will not be a comparison-based one; it will directly access and manipulate the keys themselves (rather than just comparing them to each other).

1   The result

Let's precisely define the computational problem of sorting on a finite universe.

Input	: A universe size $U \in \mathbb{N}$ and an array A of key-value pairs $((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each key $K_i \in [U]$
Output	: An array A' of key-value pairs $((K'_0, V'_0), \ldots, (K'_{n-1}, V'_{n-1}))$ that is a valid sorting of A.

Computational Problem SortingOnFiniteUniverse

We will prove:

Theorem 1.1. *There is an algorithm for SortingOnFiniteUniverse on arrays of size n and key-universe size U with (worst-case) running time $O(n + U)$.*

Since we have not yet precisely defined our computational model or what constitutes a "basic operation," this theorem and its proof are still somewhat informal, but in a few weeks we will have the language to make it all completely precise. As you will see in Problem Set 2, it is possible to improve the dependence on U from linear to logarithmic with a more involved algorithm.

2　The Proof

1. Algorithm:

2. Correctness:

3. Runtime:

3 Food for Thought

If you and your partner complete the sender-receiver exercise early, here is an additional question to think about.

When there are multiple key-value pairs in A with the same key K, a direct implementation of Singleton Bucket Sort will produce an output in which those key-value pairs appear in the *opposite order* from how they appear in A. (Why?) As you will see on Problem Set 1, it is often useful to have *stable* sorting algorithms, which are guaranteed to maintain the input ordering for repeated keys.

Two approaches to make Singleton Bucket Sort stable are the following:

1. Append each new key value pair (K_i, V_i) as the *tail* of the linked list $C[K_i]$ rather than inserting it as the head, or

2. When constructing the array A', reverse the order of each linked list $C[K]$ before adding it to A'.

Which of these two approaches will maintain runtime $O(n + U)$?