CS1200: Intro. to Algorithms and their Limitations Prof. Anurag Anshu

Sender–Receiver Exercise 1: Reading for Senders

Harvard SEAS - Spring 2026 *2026-02-02*

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 (and earlier today), we have seen that there exists a sorting algorithm (namely `MergeSort`) 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 Sorting On Finite Universe()

We will prove:

Theorem 1.1. *There is an algorithm for* Sorting On Finite Universe *on arrays of size n and key-universe size U with (worst-case) running time $O(n + U)$. One such algorithm is called* `SingletonBucketSort`.

The `SingletonBucketSort` algorithm is also known as "Pigenhole" sorting algorithm in some texts.

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 1, it is possible to improve the dependence on U from linear to logarithmic with a more involved algorithm.

Note that this is a case where we measure runtime as a function of two size parameters n and U. For convenience, we include the universe size U explicitly in the input to the Sorting On Finite Universe problem, but it is also reasonable to omit U, allow the keys to be arbitrary natural numbers, and modify the algorithm to start by calculating $U = \max_i K_i$.

1.1 The Proof

Our algorithm will be a special case of an algorithm called `BucketSort`, one where the buckets each correspond to only a single possible key (rather than a range of keys), so we will call it `SingletonBucketSort`. It is inspired by `CountingSort`, which applies when there are no values associated with the keys, and we are just sorting an array of keys from the universe $[U]$. In `CountingSort`, we initialize an array C of length U to have zeroes in every entry. Then we make a pass over the array A of keys, incrementing $C[A[i]]$ when we are at the i^{th} element of A. At the end of this pass, for each key $K \in [U]$, $C[K]$ will have a count of the number of elements of A that have key K. We now make a pass over C, filling in our output array A' from beginning to end with $C[K]$ elements of value K as we go.

To generalize this idea to sorting arrays of key-value pairs, we replace the counts in the array C with linked lists, like shown in Figure 1. If you are not familiar with linked lists, you can think of them as rooted *unary* trees, i.e. like the binary trees on Problem Set 0, but where each vertex can only have at most one child, rather than two. In a linked list, we refer to the root as the *head* and the (unique) leaf as the *tail*.

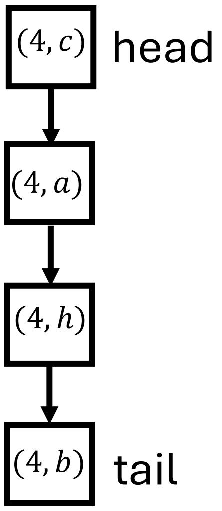

Figure 1: A linked list of key-value pairs, all with the key 4 and with values c, a, h, and b.

Now, the entries of an array are of fixed size, so we cannot put an entire variable-length linked list inside them. However, an entry of an array can hold a *pointer* to the head of a linked list (which is stored elesewhere in memory). This is explicit in C, where we can directly manipulate pointers to structs, but is also true in Python, where variables and array entries do not actually contain objects but only are references to the objects, which are stored elsewhere in memory.

This idea of using linked lists leads to the following algorithm:

```
SingletonBucketSort($U$,$A$):
Input   : A universe $U \in \mathbb{N}$ and an array $A = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where
          each $K_i \in [U]$
Output  : A valid sorting of $A$
0 Initialize an array $C$ of length $U$, such that each entry of $C$ is the start of an empty linked
    list.;
1 foreach $i = 0, \ldots, n-1$ do
2     Insert $(K_i, V_i)$ into the linked list $C[K_i]$ as the new head.
3 Form an array $A'$ that contains the elements of $C[0]$, followed by the elements of $C[1]$,
    followed by the elements of $C[3]$, ....
4 return $A'$
```

Algorithm 1.1: `SingletonBucketSort()`

To confirm your understanding of the algorithm, you may find it useful to run it on some example inputs, such as $U = 5$ and $A = ((2, a), (3, b), (0, c), (4, d), (3, e), (3, f), (0, g))$. You may notice that it's redundant to include the keys in the linked lists; it would be enough to put the values there. We included them only because it makes the algorithm slightly easier to describe in pseudocode.

To show the correctness of Algorithm 1.1, we observe that after the loop, for each $K \in [U]$, the linked list $C[K]$ contains exactly the key-value pairs whose key equals K. Thus concatenating these linked lists into a single array will be a valid sorting of the input array: it is a permutation of the original array because it contains each of the original key-value pairs exactly once (since each pair was inserted into exactly one of the linked lists), and nothing else; and its keys are in ascending order because of the previous observation and the fact that we iterate through C in order of $K = 0, \ldots, U - 1$.

For the runtime analysis, initializing the array takes time $O(U)$. Each iteration of the for loop takes time $O(1)$, enough for a constant number of steps to add (K_i, V_i) as the head of the linked list $C[K_i]$ and have it point to the original linked list as its descendants. Thus, the for loop has runtime $O(n)$. Concatenating the elements of $C[j]$ into the array A takes time $O(|C[j]|) + O(1) = O(|C[j]| + 1)$, where $|C[j]|$ is the length of the linked list $C[j]$ and the $O(1)$ is to account for the fact that we'll still use a positive number of steps to iterate through the (implicit) loop even if $|C[j]| = 0$. Thus, forming the array A takes time:

$$\sum_{j=1}^{U} O\left(|C[j]| + 1\right) = O\left(U + \sum_{j} |C[j]|\right) = O(U + n).$$

□

2 Food for Thought

If you and your partner have extra time after completing the exercise, 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 `SingletonBucketSort` will produce an output in which those key-value pairs appear in the *opposite order* from how they appear in A. (Why?) It is often useful to have *stable* sorting algorithms, which are guaranteed to maintain the input ordering for repeated keys. (See Problem 1.3 in the textbook for the stability of other sorting algorithms.)

Two approaches to make `SingletonBucketSort` 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 $O(n + U)$?