CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Sender–Receiver Exercise 0: Reading for Senders

Harvard SEAS - Fall 2022 *Sept. 8, 2022*

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 the (worst-case) computational complexity of sorting by comparison-based algorithms is $\Theta(n \log n)$. That is, there are sorting algorithms that have worst-case running time $O(n \log n)$, and every (correct) sorting algorithm has worst-case running time $\Omega(n \log n)$. This holds even when the keys are drawn from the universe $U = [n]$.

In our first active learning 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).

More generally, we will show:

Theorem 0.1. *There is an algorithm for sorting an array of n key-value pairs where the keys are drawn from a known universe of 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 informal. 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.

Proof. We assume without loss of generality that keys come from $[U]$. (Since the universe of size U is known, we can map the keys to $[U]$ while preserving the order of elements.)

Our algorithm is a variant of "Counting Sort". Counting Sort is typically presented for a case where there are no values paired with the keys, and we are just sorting an array of keys from the universe $[U]$. In Counting Sort, 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 of values, yielding the following algorithm:

Input : 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

Initialize an array C of length U, such that each entry of C is the start of an empty linked list.
foreach $i = 0, \ldots, n-1$ **do**
  Append (K_i, V_i) to the linked list $C[K_i]$.
Form an array A that contains the elements of $C[0]$, followed by the elements of $C[1]$, followed by the elements of $C[3]$, $\ldots$.
return A

Algorithm 1: Counting Sort with Values

To show the correctness of Algorithm 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.

For the runtime analysis, initializing the array takes time $O(U)$. Each iteration of the loop takes time $O(1)$, for a total loop runtime of $O(n)$. And concatenating the elements of $C[j]$ into the array A takes time $O(1) + O(|C[j]|)$, where $|C[j]|$ is the length of the linked list $C[j]$. Thus, forming the array A takes time

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

$\square$