Analysis of Algorithms I: Counting and Radix Sort

Xi Chen

Columbia University

In the last class, we showed that any comparison sorting algorithm takes time $\Omega(n \lg n)$ in the worst case and thus, Merge sort and Heapsort are asymptotically optimal comparison sorting algorithms. Now consider the following problem: If you are told in advance that the input sequence is a permutation of $\{1, 2, \ldots, n\}$, how much time do we need to sort it? still $\Omega(n \lg n)$?

Linear! Because you don't even need to examine the input, and the only thing to do is write down $(1, 2, \ldots, n)$ as output. The lesson here is that any knowledge about the input sequences may provide insight towards sorting algorithms that are more efficient than those optimal comparison-based sorting algorithms. Here we focus on the case when the input elements fall in a reasonably small range. We will see that making comparisons is not the most efficient way to obtain order information from the input sequence.

Counting Sort $(A[1 \ldots n])$, where $A[i] \in \{1, 2, \ldots, k\}$ for all i

1. Make a pass and compute, for each $j \in [k]$, the number of j's in A. Store it in $C[j]$, $j \in \{1, 2, \ldots, k\}$.
2. Create an empty sequence $B[1 \ldots n]$. Starting from $B[1]$, write $C[1]$ many 1's, $C[2]$ many 2's, ..., and $C[k]$ many k's.

Counting sort is clearly correct, and it does not use any comparison at all. Its running time is $O(k + n)$ and when k is $O(n)$, it is a linear-time sorting algorithm and beats every comparison-based algorithm. Counting sort gains order information from the values of the input elements (the reason why the comparison lower bound does not apply), which is way more efficient than comparisons when the range is reasonably small.

But can we do better? What if $k = n^2$? Yes, we can, by using Radix sort. Before that we give an alternative and seemingly unnecessarily complicated implementation of Counting sort.

Counting Sort $(A[1 \ldots n])$, where $A[i] \in \{1, 2, \ldots, k\}$ for all i

1. Make a pass and compute, for each $j \in [k]$, the number of j's in A. Store it in $C[j]$, $j \in \{1, 2, \ldots, k\}$.
2. Use $C[1 \ldots k]$ to get, for each $j \in [k]$, the number of elements in A that are $\leq j$. Store it in $C'[j]$ (How to do this in $O(k)$?)
3. For i from n down to 1 do
4. move $A[j]$ to $B[C'[A[j]]]$; and $C'[A[j]] \leftarrow C'[A[j]] - 1$

The operation of Counting sort is demonstrated in Figure 8.2 on Page 195. Basically, the new counters $C'[1 \ldots k]$ and the for loop place each element $A[j]$ into its "correct" sorted position so that

> *Equal elements appear in the output sequence in the same order as they do in the input sequence.*

A sorting algorithm with this property is called a stable sort. But why do we need this property? What is the difference between two 3's? It is important when each number being sorted by Counting sort is part of a bigger object. For example, we will use Counting sort as a subroutine in Radix sort (to sort on each digit).

Radix sort $(A[1 \ldots n])$, where $A[i]$ is a decimal number of d digits

1. For i from 1 to d do
2. use Counting sort (or any stable sort) to sort A on digit i

The operation of Radix sort is demonstrated in Figure 8.3 on Page 198. We use induction to prove the following statement:

Lemma

After the kth loop, where $k = 1, 2, \ldots, d$, the sequence is sorted on the lower k digits.

In the proof on the next slide, by the lower k digits of a, we mean the number represented by the lower k digits of a.

We use induction on k. The basis when $k = 1$ is trivial. Now assume (as the inductive hypothesis) that at the end of the $(k-1)$th loop, for some $k : 2 \leq k \leq d$, the sequence is sorted on the lower $(k-1)$ digits. We need to show that after applying a stable sort on digit k in the kth loop, the sequence is sorted on the lower k digits. To prove this, let $a_i, a_j \in A$ be any two elements from A with the lower k digits of a_i being strictly smaller than those of a_j. We show that after the kth loop, a_i must appear before a_j in the sequence.

1. If the kth digits of a_i and a_j are different, then the kth digit of a_i must be smaller by our assumption on a_i and a_j. Then after the kth loop, a_i must appear before a_j because the kth loop sorts on the kth digit.
2. If the kth digits of a_i and a_j are the same, then the lower $(k-1)$th digits of a_i must be strictly smaller than those of a_j, again, by the assumption on a_i and a_j. From the inductive hypothesis, we know a_i appears before a_j before we apply counting sort on digit k. Because they have the same digit k, they appear in the same order and thus, a_i still appears before a_j after the kth loop.

This finishes the proof of correctness.

But what is the running time of Radix sort? If every element lies in $\{0, 1, \ldots, n\}$, we need roughly $\log_{10} n$ digits in the decimal system. Each call to Counting sort takes time $O(n + 10) = O(n)$ because each digit has $k = 10$ possibilities. So the total running time is

$$O(n \log_{10} n) = O(n \lg n)!!!$$

even worse than Counting sort.

But ... it does not have to be the decimal system. What if we make each digit larger? Assume every element lies in the range of $\{0, 1, \ldots, 2^h - 1\}$ and is represented by $h = dr$ bits. We partition these h bits into d blocks of r bits each. We call each block a digit (or equivalently, we use the base-2^r system).

Now each call to Counting sort takes time $O(n + 2^r)$ because each digit lies in $\{0, 1, \ldots, 2^r - 1\}$. We make d calls so the total running time is $O(d(n + 2^r))$. Why is Radix sort better than Counting sort? It depends on how we pick r.

When $h = \ell \lg n$ for some constant $\ell > 0$, the range of the input elements is $\{0, 1, \ldots, n^\ell - 1\}$. If we set $r = \lg n$, then the total running time of Radix sort is

$$O\big(\ell(n + 2^r)\big) = O(\ell \cdot n) = O(n)$$

since we make ℓ (a constant) many calls to Counting sort and each call takes time $O(n + 2^r)$. As a result, for any positive constant $\ell > 0$, if we know in advance that the input elements fall in a polynomial-size range $\{0, 1, \ldots, n^\ell - 1\}$, then Radix sort runs in linear time by setting $r = \lg n$.