Scribe: William Chen (2016), G. Valiant (2017) **Date:** October 11, 2017

1 Lower Bounds for Comparison-Based Sorting Algorithms

See link on the course website.

2 Bucket Sort[1]

Recall that our sorting lower bounds applied to the class of algorithms that can only evaluate the values being sorted via *comparison queries*, namely via asking whether a given element is greater than, less than, or equal to some other element. For such algorithms, we proved that any correct algorithm (even a randomized one!) will require $\Omega(n \log n)$ such queries on some input. As with any lower-bound that we prove in a restricted model, it is fruitful to ask "is it possible to have an algorithm that does not fall in this class, and hence is not subject to the lowerbound?" For sorting, the answer is "yes".

We start by looking at a very simple non-comparison-based sorting algorithm called Bucket Sort. Since it is not comparison-based, it is not restricted by the $\Omega(n \log n)$ lower bound for sorting. For a given input of n objects, each with a corresponding key (or value) in the range $\{0, 1, ..., r-1\}$, Bucket Sort will sort the objects by their keys:

1. Create an array A of r buckets where each bucket contains a linked list.
2. For each element in the input array with key k, concatenate the element to the end of the linked list $A[k]$.
3. Concatenate all the linked lists: $A[0], \ldots, A[r-1]$.

The algorithm correctly sorts the n elements by their keys because the elements are placed into buckets by key where bucket i (containing elements with key $= i$) will come before bucket j (containing elements with key $= j$) in A for $i < j$. Therefore, when the algorithm concatenates the buckets, all elements with key $= i$ will come before elements with key $= j$.

The worst case run time of bucket sort is $\mathrm{O}(n + r)$ since it does $O(1)$ passes over the n input elements and $O(1)$ passes over the r buckets of A.

An important property (which we will use in the next section) is that the algorithm described above is *stable*: If two input elements x, y have the same key, and x appears before y in the input array, x will appear before y in the output.

3 Radix Sort

Radix Sort is another non-comparison-based sorting algorithm that will use Bucket Sort as a subroutine. Assuming that the input array contains L-digit numbers where each digit ranges from 0 to $r-1$, Radix Sort sorts the array digit-by-digit (or field-by-field for non-numerical inputs). The algorithm works on input array A as follows:

1. For $j = 1, \ldots L$:
2. Bucket Sort A using the j^{th} digit as the key.

[1] Note that this may not be the standard name used in CLRS. What we describe here is similar to Counting Sort from CLRS.

Note that we refer to the least significant digit as the first digit. Hence, the algorithm calls Bucket Sort first using the least significant digit as the key, then again using the second least significant digit, until the most significant digit.

We will show that Radix Sort correctly sorts an input list of n numbers via induction on the iterations of the loop. We prove that by the end of the j^{th} iteration, the elements in A are sorted when considering only the j least significant digits of each element.

Base case: Radix Sort correctly sorts the numbers by the first digit as it uses Bucket Sort to sort the numbers using the least significant digit as a key.

Inductive case: By the induction hypothesis, by the end of iteration $j-1$, the input numbers have been sorted by the $j-1$ least significant digits.

After we run Bucket Sort on the elements using digit j as the key, the numbers are sorted by their j^{th} digit. Since Bucket Sort is stable, the elements in each bucket keep their original order, and by the induction hypothesis, they are ordered by their $j-1$ least significant digits. Since the elements are ordered first by their j^{th} digit, and then by their $j-1$ least significant digits, we conclude that they are ordered by their j least significant digits.

By the end of iteration L, the numbers are in sorted order.

The worst case run time of Radix Sort is $\mathrm{O}(L(n+r))$ since we are calling Bucket Sort on n elements with r possible keys once for each digit in the input numbers. If $r = \mathrm{O}(n)$ and $L = \mathrm{O}(1)$, then this takes $\mathrm{O}(n)$ time.