Adapted From Virginia Williams' lecture notes.

COMP 285 (NC A&T, Spr '22) Lecture 11

Radix Sort and Wrap-Up

Today, we will continue our discussion of non-comparison-based sorts, such as Counting Sort and Radix Sort. We will wrap-up our discussion by providing motivation for the study of specialized data structures, such as balanced binary trees.

1 Counting Sort

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, it's been proven 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 Counting 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 $\{0, 1, \cdots, r-1\}$, Counting 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: $A[0], \cdots, A[r-1]$.

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 Counting sort is $O(n + r)$ since it performs $O(1)$ passes over the n input elements and $O(1)$ passes over the r buckets of A. This is great if the size of the range r is small, but we pay dearly in the running time (and space) if r is large. The algorithm we discuss in the next section builds on Counting Sort and fixes this issue (that Counting Sort behaves poorly if the range of values r is very large).

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.

2 Radix Sort

Radix Sort is another non-comparison-based sorting algorithm that will use Counting Sort as a subroutine. Assuming that the input array contains d-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 $j = 1, \cdots, d$:
2. Apply Counting Sort to A using the jth digit as the key.

Note that we refer to the least significant digit as the first digit. Hence, the algorithm calls Counting 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.

Inductive Hypothesis: At 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: ($j = 1$) Radix Sort correctly sorts the numbers by the first digit as it uses Counting Sort to sort the numbers using the least significant digit as a key. (Note: we could have used $j = 0$as the base case as well).

/Inductive case: We will prove that, if the inductive hypothesis for $j = k-1$ is true, then the inductive hypothesis for $j = k$ is true as well. By the inductive hypothesis for $j = k-1$, by the end of iteration $k-1$, the input numbers have been sorted by the $k-1$ least significant digits. After we run Counting Sort on the elements using digit k as the key, the numbers are sorted by their j^{th} digit. Since Counting Sort is stable, the elements in each bucket keep their original order, and by the induction hypothesis, they are ordered by their $k-1$ least significant digits. Since **the elements are ordered first by their k^{th} digit, and then by their $k-1$ least significant digits**, we conclude that they are ordered by their k least significant digits.

To provide more detail for the bolded text in that last sentence, we could consider any two numbers x and y in the input and separately consider two cases as we did in lecture: case (i) where x and y have different kth least significant digits, and case (ii) where x and y have identical kth least significant digits. In both cases, we could establish that, in the output,x and y are ordered by their least k significant digits. Since this is true for every pair of numbers x and y in the input, this means that the output of this iteration is correctly sorted by the k least significant digits.

By induction, the inductive hypothesis is true for all $j \in \{1, \cdots, d\}$. Applying the inductive hypothesis for $j = d$, we conclude that at the end of iteration d, the numbers are in sorted order (since the input consists of d-digit numbers). In other words, Radix Sort correctly sorts the input.

The worst case running time of Radix Sort is $O(d(n+r))$ since we are calling Counting Sort

on n elements with r possible keys once for each digit in the input numbers. If $r = O(n)$ and $d = O(1)$, then this takes $O(n)$ time.

As we did in lecture, we could consider varying the base r in which we write our numbers and running radix sort (with the Counting Sort in the inner loop using r buckets). Now the maximum number of digits is a function of the maximum size of the numbers in the input and our choice of base r. If we have an input consisting of n numbers of value at most M, then the number of digits $d = \lfloor log_r(M) \rfloor + 1$. (verify this yourself!) Thus the running time for Radix Sort with base r is $O(d(n + r))$ which is $O((\lfloor \log_r(M) \rfloor + 1) \cdot (n + r))$.

How should we choose r? One reasonable choice is $r = n$ to balance the two values n and r in the term $(n+r)$ in the expression for the running time. For this choice of r, the running time for Radix sort is $O(n \cdot (\lfloor \log_r(M) \rfloor + 1))$. (Why didn't we drop the $+1$ term in the expression for the asymptotic running time?) Note that we might make a different choice of r if it was also important to optimize the space required to run Radix Sort.

So the running time of Radix Sort (in terms of n) depends on how large the numbers in the input are as a function of n. If the upper bound $M \leq n^c$ for some constant c, then the running time is $O(n)$. In this regime, Radix Sort beats MergeSort and Quicksort. On the other hand, if $M = 2^n$, then the running time is $O(n^2/\log n)$. In this regime, we would not use Radix Sort.

3 Data Structures

Thus far in this course we have mainly discussed algorithm design, and have specified algorithms at a relatively high level. For example, in describing sorting algorithms, we often assumed that we could insert numbers into a list in constant time; we didn't worry too much about the actual implementation of this, and took it as an assumption that this could actually be implemented (e.g. via a linked-list).

We will wrap-up this lecture by talking about the actual design and implementation of some useful datastructures. The purpose is mainly to give you a taste of data-structure design. We won't discuss this area too much more in the course, and will switch back to discussing higher-level algorithms.

To motivate the data structures that we will discuss in this course, consider the following table that lists a bunch of basic operations that we would like to perform on a set/list of numbers, together with the worst-case runtime of performing those operation for two of the data structures that you are (hopefully) familiar with: linked lists of n (unsorted) numbers, and an array of n sorted numbers. In this course, we will assume that all the numbers we store are unique.

Operation	Runtime with Unsorted Linked Lists	Runtime with Sorted Array
Search (ie, find 17)	$\Theta(n)$	$\Theta(\log n)$(via binary search)
Select (ie, find 12 smallest element)	$\Theta(n)$	$\Theta(1)$ (just look at the 12th element)
Rank (ie, # elt. less than 17)	$\Theta(n)$	$\Theta(\log n)$
Predecessor/Successor	$\Theta(n)$	$\Theta(1)$
Insert element	$\Theta(1)$	$\Theta(n)$
Delete element	$\Theta(1)$	$\Theta(n)$

The above table makes Sorted Arrays look like a pretty decent data structure for *static* data. In many applications, however, the data changes often, in which case the $\Theta(n)$ time it takes to insert or delete an element is prohibitive. This prompts the question: *Is it possible to have one data structure that is the best of both worlds (logarithmic or constant time for all these operations)?* The answer is, essentially, "Yes", provided we don't mind if some of the $\Theta(1)$'s turn into $\Theta(logn)$.

We will cover these data structures in more detail next lecture.