Sorting in linear time

(CLRS 8.2, 8.3)

We know that it is not possible to sort n elements faster than $\Omega(n \lg n)$ in the worst case **when using only comparisons** (i.e. in the decision tree model). A question we can ask is: Are there other ways to sort? What else could we use besides comparisons?

Let's think about the following problem:

Exercise: Give an $O(n)$ algorithm to sort n integers in the range 0..999.

As we can see from this example, non-comparison sorting algorithm exist, but they crucially exploit special properties of the input (such as integers in a small range). The most common ones are bucket sort, counting sort and radix sort, which we discuss below.

Bucket sort

Input: n integers in the range $\{0, .., N-1\}$, for some $N \geq 2$.

```
BUCKET-SORT($A$)
1  Create an array $B[0..N-1]$
2  For $i = 0$ to $n-1$
3      insert $A[i]$ into $B[A[i]]$ at the end
4  For $i = 0$ to $N-1$
5      traverse $B[i]$
```

Note that here we denote n the size of the input, and N the range of the input.

Analysis: $O(n+N)$ time and $O(N+n)$ extra space.

How does $\Theta(n+N)$ compare with $\Theta(n \lg n)$? Put differently, when is Bucket-sort efficient?

- When N is small. If $N = O(n)$, then Bucket-sort runs in $O(n)$ time.

Counting sort

Input: integers in the range $\{0, .., N-1\}$, for some $N \geq 2$.

Counting-sort uses the same idea as bucket sort, except that, instead of creating buckets (with linked lists), it stores everything in an array. It uses less space (and is more elegant).

```
COUNTING-SORT($A$)
1   Create an auxiliary array $C[0..N-1]$
2   For $i$ = 0 **to** $N-1$
3        C[i] = 0
4   For $i$ = 0 **to** $n-1$
5        $C[A[i]]++$
6   For $i$ = 1 **to** $N-1$
7        C[i] = C[i] + C[i-1]
8   For $i = n-1$ down to 0
9        B[C[A[i]]] = A[i]
10       C[A[i]]–
```

Analysis: $O(n+N)$ time and $O(N+n)$ extra space.
Counting-sort is stable.

A sorting algorithm is called *stable* if it leaves equal elements in the same order that it found them.

A sorting algorithm that's stable is important when doing multiple passes of sorting. For example, we have a set of names which are tuples: name = [LastName, FirstName], and we want to sort them ordered by last name, and, for people with same last name, ordered by their first name (lexicographic order). We can implement this order by doing two passes of [name your favorite sorting algorithm here] sort.

- We first sort by FirstName;
- Then we sort by LastName.

This works assuming that we use a *stable* sorting algorithm to sort by LastName. In this case, it would keep all the people with same first name in the order they were in the input, which means that people with same last name would appear in the order of their first names.

Radix sort

Input: A set of n items, each represented on d-digits.

The idea of RADIX-SORT is to sort them looking at one digit at a time.

MSD Radix sort

Sort by the most-significant digit (MSD) first.

```
MSD-RADIX-SORT($A$)
1   Sort items by most significant digit and place the items in buckets
2   Sort each bucket recursively
3   Combine the buckets in order.
```

Example: Sort the following 3-digit numbers: 329, 457, 657, 839, 436, 720, 355.

Cons: generates many intermediate buckets; tail-recursion.
Pros: Does fewer than d passes if inputs have a small *distinguishing prefix.*

LSD Radix sort

Sort by the least-significant digit (LSD) first.

```
RADIX-SORT(A)
1  For i=1 to $d$
2      sort A on digit $i$
```

In order for this to work correctly, the intermediate sorting algorithm used must be *stable.*

Running time: If each digit is in range $[0..k]$, we can use Counting-Sort to sort in $O(n+k)$. Overall RADIX-SORT will take $O(d \cdot (n+k))$.

What's a digit/How many digits? By default we think of numbers in base-10. A base-10 representation of a number x will have $\log_{10} x$ digits, and a base-10 digit is one of $0,1,2,3,4,5,6,7,8,9$. Therefore we think of a base-10 digit as an integer in the range $[0..9]$. Representing in other bases is similar.

Representing a value x:

- base-10: $\lceil \log_{10} x \rceil$ digits, digit in $[0,9]$
- base-2: $\lceil \log_2 x \rceil$ digits, digit in $[0,1]$
- if one digit = 2 bits in the base-2 representation: $\log_2 x/2$ digits, each in range $[0,2^2)$
- if one digit = r bits in the base-2 representation: $\log_2 x$/r digits, each in range $[0,2^r)$

Radix-sorting a set of values in the range $[0..x]$ will take $O(\lg x \cdot O(n+1))$ if they are represented in base-2; and $O(\lg x/r \cdot O(n+2^r)$ if we chose r bits to represent a digit. We want to choose r in order to optimize $O(\lg x/r \cdot O(n+2^r)$.

Example: Show how to sort n values in the range $[0..n^3]$ in $O(n)$ time.

Answer: How to choose the right basis? If we choose base-2, each value will have $O(\lg n^3) = O(3 \lg n)$ bits, and radix-sort will take $O(3 \lg n \cdot O(n+1))$ time. Choose $r = \lg n$, which is same as saying represent the values in base-n.

Radix-sort, Counting-sort or Quicksort?

Assume we want to sort a set of integers. Which sort is better? Integers are represented on 4 bytes=32 bits, and the range of values of an integer on 4 bytes is $-2^{31}, 2^{31}$ (more or less).

If there are no duplicate values, n has to be smaller than the range, therefore $\lg n < 32$. Put differently, sorting up to 4 billion integers means $\lg n < 32$, which means Quicksort runs in $O(32n)$ on the average. Counting-sort will do one ass but require additional storage of an array of 4 billion integers, or 16GB. Radix-sort will do 32 passes of counting-sort n values in range $[0..1]$, which runs in $O(32 \cdot O(n+1))$. Similar.