The Radix Sort

Two disclaimers:

1. This document uses a lot of LaTeX, and doesn't look very nice on GitHub. Look at the PDF version of it to get the full effect, or compile it with `pandoc`.
2. Those that are actually interested in a complete and in-depth discussion about the radix sort should go read *The Art of Computer Programming: Sorting and Searching*, section 5.2.5. Knuth gives a very nice treatment, although he does keep things lower-level than most.

With these out of the way, *here be dragons.*

The Rub

The big idea behind radix sorting is that you can see which of two integers is bigger just by looking at the digits in their places. As an example, we can compare 745 and 145 by noticing that $1 < 7$ and that the lower places don't matter. We can also compare 73591 and 73537 by noticing, from most to least significant, the digits of the two numbers match until the tens place is reached. The ones place is too small to matter, so because $3 < 9$, then $73537 < 73591$.

The usual definition for $m < n$ is that $n - m$ is positive. If we consider n and m as tuples of their digits, then we can get another way to define this. If each non-negative integer has p digits, then we could say

$$m = (a_p, a_{p-1}, \ldots, a_1).$$

and

$$n = (b_p, b_{p-1}, \ldots, b_1),$$

where $0 \leq a_k, b_k < 10$. It turns out that the usual definition for $m < n$ is equivalent to saying that there exists some $1 \leq j \leq p$ such that

$$a_i = b_i \qquad \text{for all } i > j, \qquad \text{but } a_j < b_j \tag{1}$$

That is, that the numbers differ at the kth place, but match everywhere above that. Note that the digits before the kth place do not matter, intuitively because they are "smaller" in value. Looking back at 73591 and 73537, we see that they differ at the tens place, but match above that. We know that $3 < 9$, so from the above definition, $73537 < 73591$.

The radix sort takes advantage of this definition. It keeps a predetermined ordering of the digits, e.g. that $8 < 9$, and for each digits place it places numbers

into "digit buckets." These buckets are joined so that the elements force them into their proper order for the current digits place. After every digits place is sorted, the list is sorted.

Implementation and Testing

The implementation of the radix sort given here follows TAOCP 5.2.5 as closely as possible, and uses C to get into some of the lower-level ideas in the algorithm.

The places where the C version differs from Knuth's assembly language version are places where types mess things up. For example, recall that queues can be defined in C as the following structs:

```
typedef struct NODE {
    unsigned int data;
    struct NODE *next;
} NODE;

typedef struct {
    NODE *rear;
    NODE *front;
} QUEUE;
```

To make a queue empty, the `rear` and `front` pointers are set to `NULL`. Inserting the `NODE *insert` then has a special case of setting up both pointers on empty queues, as opposed to saying that `rear->next = insert`.

Knuth is not burdened by types, and sets `rear = &front`, assuming that `(&front)->next = front`. Then inserting into a queue is *always* just `rear->next = insert`, because in an empty queue this becomes `front = insert`. Other differences can be found and explained in the source code.

For empirical testing, the program compares the radix sort to the C library's implementation of `qsort`. For small lists, it also prints out the result of the sort, just to make sure that everything is working the way that it should be.

The radix sort seems to perform better than `qsort` when the given radix is large enough. This is probably because, if the input has p digits in base-M, then there are only p passes made over the data. When `qsort` does better, it could be for lots of reasons, but the `qsort` implementation is probably heavily optimized, while the radix sort is not. Here are some timings sorting random data with everything the same except for the radices:

```
./radix
radix: 10
n: 100000
range: [0, 100000)
length in base-10: 5
```

```
radix sort time: 60.000000ms
qsort time: 38.000000ms

./radix -r 2
radix: 2
n: 100000
range: [0, 100000)
length in base-2: 17
radix sort time: 206.000000ms
qsort time: 38.000000ms

./radix -r 100
radix: 100
n: 100000
range: [0, 100000)
length in base-100: 3
radix sort time: 38.000000ms
qsort time: 38.000000ms

./radix -r 1000
radix: 1000
n: 100000
range: [0, 100000)
length in base-1000: 2
radix sort time: 26.000000ms
qsort time: 38.000000ms
```

And now with much shorter numbers:

```
./radix -r 100 -l 3
radix: 100
n: 100000
range: [0, 1000)
length in base-100: 2
radix sort time: 24.000000ms
qsort time: 37.000000ms

./radix -r 10 -l 3
radix: 10
n: 100000
range: [0, 1000)
length in base-10: 3
radix sort time: 35.000000ms
qsort time: 37.000000ms

./radix -r 2 -l 2
radix: 2
```

```
n: 100000
range: [0, 100)
length in base-2: 7
radix sort time: 65.000000ms
qsort time: 36.000000ms

./radix -r 10 -l 2
radix: 10
n: 100000
range: [0, 100)
length in base-10: 2
radix sort time: 22.000000ms
qsort time: 36.000000ms
```