Radix sort: least significant digit first

Radix is a stuffy synonym for *base*; both words denote the number of unique digits used to represent numbers in our conventional positional number systems. For example, for the decimal number system, the *base* or *radix* is 10.

An *LSD* (**L**east **S**ignificant **D**igit first) *radix sort* sorts by first stably sorting the array based on its least significant digit, then on its second least significant digit, and so on up to its most significant digit.

For example, we use radix 10 and sort the array to the right. We have written in leading 0's to make all elements three digits long. Stably sorting the array on the least significant digit produces the second array. Stably sorting the second array on its middle digit produces the third array. Stably sorting the third array on its most significant digit produces the fourth array. You can verify that it is sorted.

(088, 074, 085, 320, 102, 004)
(320, 102, 074, 004, 085, 088)
(102, 004, 320, 074, 085, 088)
(004, 074, 085, 088, 102, 320)

Previously written method `countSort`, whose specification appears at the bottom of this page, stably sorts int array `c` based on a function `f(key)` where `key` is one of the array elements.

In our radix sort, function `f` has to return a digit of the `key`. Here, we make good use of Java's anonymous functions. For example,

To sort array `c` on its least-significant decimal digit, use the statement

```
c= countSort(c, key -> (key) % 10, 10);
```

To sort c on its second least-significant decimal digit, use the statement

```
c= countSort(c, key -> (key / 10) % 10, 10);
```

But any radix can be used. For example, use 8 instead of 10 and the sort processes octal digits.

Our algorithm needs to know how the maximum number of digits in any integer of the array, so we write the method as follows. Local variable `e1` is needed because its occurrence in the anonymous function requires that it appear to be final (meaning it won't be changed again). This is a weird thing about anonymous functions in Java.

```
// Sort int array c using radix sort with radix b
int max= maximum value in array c;
int k= 0;   int e= 1;
// invariant: c contains the initial array sorted on its k least significant digits   AND   e = b^k
while (e <= max) {
   int e1= e;
   countSort(c, key -> (key / e1) % b, b);
   e= 10 * e;
   k= k + 1;
}
```

```
/** Sort c according to function f. The sort is stable.
    * In this description, we use "key" for an element of c.

    *  Precondition: f(key) returns an integer in the range 0..b-1, i.e.:
    *  the keys c[i] satisfy 0 <= f(key) < b.

    *  The output array contains items with f(key) = 0, then items with f(key) = 1, and so on, i.e.
    *  Postcondition: c is changed to this:
    *      ---------------------------------------------------------------------------
    *   c | keys with f(key) = 0 | keys with f(key) = 1 | ... | keys with f(key) = k-1 |
    *      ---------------------------------------------------------------------------

    *  Using n for c.length, we give the space and time complexity:
    *  Space used: O(n + b). Worst-case and expected time: O(n + b). */
public static void countSort(int[] c, ToIntFunction<Integer> f, int b)
```

Discussion of the runtime of LSD radix sort

Assume the size of array `c` is `n`. Suppose the integers in `c` have at most `d` digits using base `b`. As shown above, radix sort calls `countSort` `d` times, and each call takes time O(`n+b`). Therefore, radix sort takes time O(`d`*(`n+b`)).

Suppose `v` is the largest value in array `c`. Then `d` is $\log_b v$. So the time is

$O((n+b) \log_b v)$.

Of course, v can be quite large. If the array being sorted contains a value $\geq n$, it's at least $O((n+b) \log_b n)$. This doesn't seem to be better than comparison-sort algorithms, like merge sort.

I thought radix sort was a linear-time algorithm

Sometimes, however, people claim that radix sort is a linear time algorithm. Here's an example.

Suppose we are sorting an **int** array `c` of nonnegative values. Therefore, the array values are in the range $0..2^{31}-1$, or 0.. 2147483647. Therefore, d is bounded above by 10, a constant. Thus, the time is O(`10`*(`n+b`)), which is O(`n+b`)). Since we are using a computer in which the array values are limited in size, we can claim that that radix sort is a linear-time algorithm.