14 Lexical sort

Lexical or lexicographical or alphabetic sorting, also called binsort or bucket sort, is a fairly obvious way to sort an array of keys where each key is a structure or array of sortable items, called 'digits.' 'Mixed digit' keys, such as stone, pounds and ounces, are possible, but we are mostly concerned where each digit is in the range 0..radix-1 where 'radix' is a fixed integer ≥ 2.

To enable experiments, we shall assume that the keys are just nonnegative integers read from input, and the radix is an integer ≥ 2 read from the command line. That way one can experiment with different radixes.

Suppose that a and b are nonnegative integers and r the radix. Let $(a_{m-1} \ldots a_0)_r$ be a written to the base r and $(b_{m-1} \ldots b_0)_r$ be b written to the base r. If necessary, one of these should be padded with zeroes to produce representations of the same length. Write $\vec{a}$ for $(a_{m-1} \ldots a_0)_r$ and $\vec{b} = (b_{m-1} \ldots b_0)_r$.

These encodings are high-order leftmost, so

$$a = a_{m-1} r^{m-1} + \ldots + a_1 r + a_0$$

Define

$$\begin{aligned} \vec{a} = \vec{b} \quad & \text{if } a_j = b_j,\ 0 \leq j < m \\ \vec{a} < \vec{b} \quad & \text{if unequal, and for some } i \\ a_i < b_i, \quad & \text{but } a_j = b_j \quad \text{for } m > j > i \\ \vec{a} > \vec{b} \quad & \text{otherwise.} \end{aligned}$$

Of course, $\vec{a} = \vec{b} \iff a = b$, $\vec{a} < \vec{b} \iff a < b$, and $\vec{a} > \vec{b} \iff a > b$. The point is to focus on the digits.

Given an array $a^{(0)}, \ldots a^{(n-1)}$ of nonnegative integers — we use superscripts to avoid confusion with their breakdown into digits — define the *length* of a nonnegative number a, to base r, as the smallest $s > 0$ such that

$$a < r^s$$

Let m be the maximum length of all the numbers $a^{(j)}$.

For the purposes of the assignment, it is possible but unnecessary to convert the numbers $a^{(k)}$ into m-digit numbers. All one needs is the means to compute the i-th digit of a number a, and this is

$$\lfloor a/r^i \rfloor \mod r$$

This is counting the digits low-order first, with the zero-th digit just $a \mod r$.

14.1 Most significant digit

This is the most natural way to run lexical sort. It is recursive.

```
void msd ( int k, int r, int count, int number[] )
{
  sorts the numbers based on the k-th, (k-1)st, ...
```

```
  0th digit, as follows.

  distribute (somehow) the numbers into r
  buckets (linked lists) where r is the radix,
  based on the k-th digit in each number.
  If k > 0,
  recursively call msd ( k-1, r, ..., ... ) on
  each bucket;
  then combine the buckets into a single array.
}
Sorting is achieved by invoking msd ( m-1, r, count, number )
where m is the maximum length of the numbers.
```

MSD sort looks reasonable, but is actually very clumsy. The method of choice is much simpler: Least significant digit radix sort:

```
  for ( k = 0; k<m; ++k )
  {
    distribute the numbers into r buckets based
    on the k-th digit.

    This works only if the distribution is stable,
    i.e., it doesnt change the order of numbers with
    the same k-th digit.

    Redistribute the buckets into the array number.

    (m is the maximum length of the  numbers in radius r).
  }
```

Implementation. You will need some form of linking to chain together the numbers with the same k-th digit, rather like the 'chaining' hash-tables.

Every bucket must be managed using two arrays, indexed from 0 to $r-1$ — r is the radix. These arrays are `first[]` and `last[]`. You may choose pointers or you may choose to operate with indices into the array `number[]`. In any case you will need an array `next[]` so that `next[i]`, if not 'null,' gives the next number with the same k-th digit as `number[i]`.

Always extend a bucket by adding to the `last` in the bucket. This will ensure stable distribution.

Here are some examples of a working program which reads the numbers from standard input and the radix from the command line. The idea is that it should work with any radix.

```
Contents of file 5numbers:
83825 9448 6805 43217 438
% a.out 10 < 5numbers
```

```
sorted....
 00438 06805 09448 43217 83825
%
%
% a.out 2 < 5numbers
sorted....
 00438 06805 09448 43217 83825
%
% a.out 5 < 5numbers
sorted....
 00438 06805 09448 43217 83825
% a.out 15 < 5numbers
sorted....
 00438 06805 09448 43217 83825
% a.out 99 < 5numbers
sorted....
 00438 06805 09448 43217 83825
```