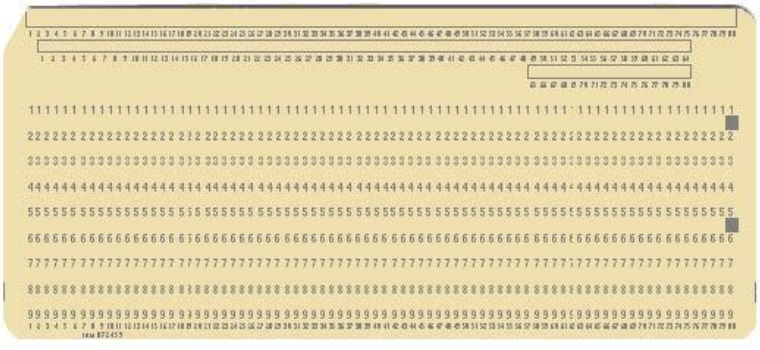
Worksheet 39: Radix Sorting

Group 11

In preparation: If you have not done so already, you should read chapter 12 to learn more about the concept of hashing.

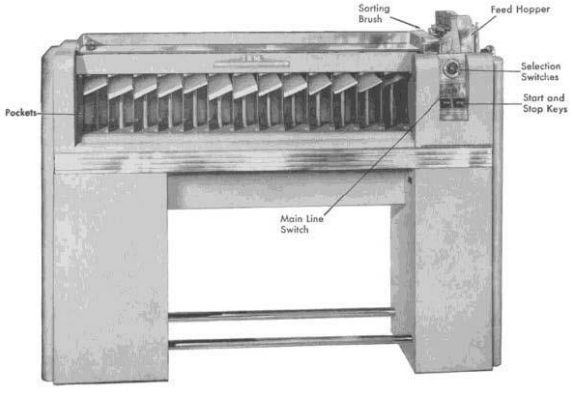


Although less ubiquitous today than they once were, computer punch cards were for many years commonly used to store information both for computer use and for other business applications. A punch card was made out of stiff paper, and holes in the card indicated a specific letter. A card could typically hold 80 columns of information.

Because it was far too easy to drop a tray of cards and get them out of order, a convention developed to place a *sequencing number* somewhere on the card, often in the last eight columns. A common task was to order a collection of cards according to their sequencing number.

A machine called a *sorter* would divide a deck of cards into buckets, based on their value in **one** specific column. This would seem to be of little use, however the technique described in this lesson, radix sorting, shows how this one column sorting could be extended to sort on a larger range of numbers.

Suppose, for example, that the sequencing numbers appeared in columns 78, 79 and 80. The deck would first be sorted on column 80. The person doing the sorting would then, by hand, gather the resulting decks together and sort them once again, this time on column 79. Gathering together for a third time, the deck would be sorted on column 78. After the third sort, the deck would be completely ordered.



Radix Sorting

Radix sorting is not a general purpose sorting technique. It works only on positive integer values. But in this limited domain of application it is very fast. It is also interesting because of its historical import (see above) and because of its novel use of both hash tables and queues.

Imagine a table of ten queues that can hold integer values. The elements are successively ordered on digit positions, from right to left. This is accomplished by copying the elements into the queues, where the index for the bucket is given by the position of the digit being sorted. Once all digit positions have been examined, the collection will be completely sorted.

To illustrate, assume the list to be sorted is as follows:

624 762 852 426 197 987 269 146 415 301 730 78 593

The following table shows the sequences of elements found in each bucket during the four steps involved in sorting the list. During pass 1 the one's place digits are ordered. Each value that ends in the same digit is placed into the same bucket. During pass 2, the ten's place digits are ordered. However, because we are using a queue the relative positions set by the earlier pass are maintained. Again all elements that are the same in the 10's place are placed in the same bucket. On pass 3, the hundred's place digits are ordered. After three passes, the result is a completely ordered collection.

As a function of d, the number of digits in each number, and n, the number of values being sorted, what is the algorithmic complexity of radix sorting? O(d*n)

Bucket	Pass 1	Pass 2	Pass 3	Pass 1	Pass 2	Pass 3
0	730	301	78	391		
1	301	415	146,197		616	142
2	762, 852	624, 426	269	742, 382,872, 732,142, 562	925	247
3	593	730	301	453	732	382,391
4	624	146	415, 426		742,142, 247	453
5	415	852	593	925	453	562
6	426, 146	762, 269	624	616	562	616
7	197, 987	78	730, 762	247	872	732,742
8	78	987	852		382	872
9	269	593, 197	987		391	925

How does this compare to merge sort, quick sort, or any of the other fast sorting algorithms we have seen?

Quick sort and merge sort arecomparison sorts (which can never have a worst-case running time less than O(N log N)), wherea radix sort is not a comparison sort.

Merge Sort:

- recursively sort the first N/2 items
  recursively sort the last N/2 items
  merge (using an auxiliary array)
- always O(N log N)

Quick Sort:

- choose a pivot value
  partition the array:

    left part has items <= pivot
    right part has items >= pivot

  recursively sort the left part
  recursively sort the right part

- worst-case $O(N^2)$

- expected O(N log N)

Radix Sort:

- make *len* passes through the N sequences to be sorted, right-to-left
  on each pass, put the values into the queue in position p of the auxiliary array, where p is the value of the current "digit"
  then put the values back from the auxiliary array into the original array

- no comparisons of values are done (i.e., radix sort is not a comparison sort).

- always O(N + *range*) * *len*)

reference: http://pages.cs.wisc.edu/~skrentny/cs367-common/readings/Sorting/#radix

In the space in the right of the table above perform a radix sort on the following values, showing the result after each pass: 742 247 391 382 616 872 453 925 732 142 562.