CSC 225

Algorithms and Data Structures I

Rich Little
rlittle@uvic.ca
ECS 516

Linear Sorting

- If you know something about the values of the input set, you can potentially do better than the $\Omega(n \log n)$ sorting lower bound

- For example, if the values of the input set are in a certain given small range

Bucket or Bin Sort

- Bucket sort or bin sort partitions a set of elements into a finite number of buckets or bins
- Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm
- Bucketsort may run in linear time ($\Theta(n)$)
- Each bucket must contain only a single element or it incurs a cost for additional sorts on the buckets themselves
- Since bucket sort is not a comparison sort, it is not subject to the $\Omega(n \log n)$ lower bound

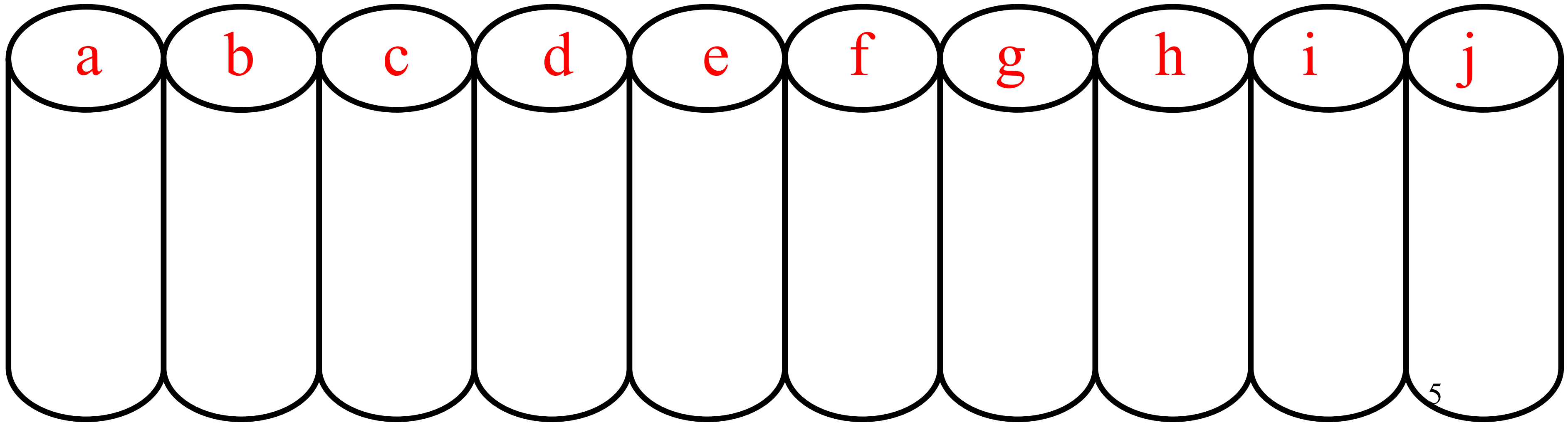
Bucket Sort

- Given $\underline{n}$ elements (e.g., words) to sort into $\underline{N}$ categories (e.g., letters of the alphabet)

- Input set
  - bucketsort insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort

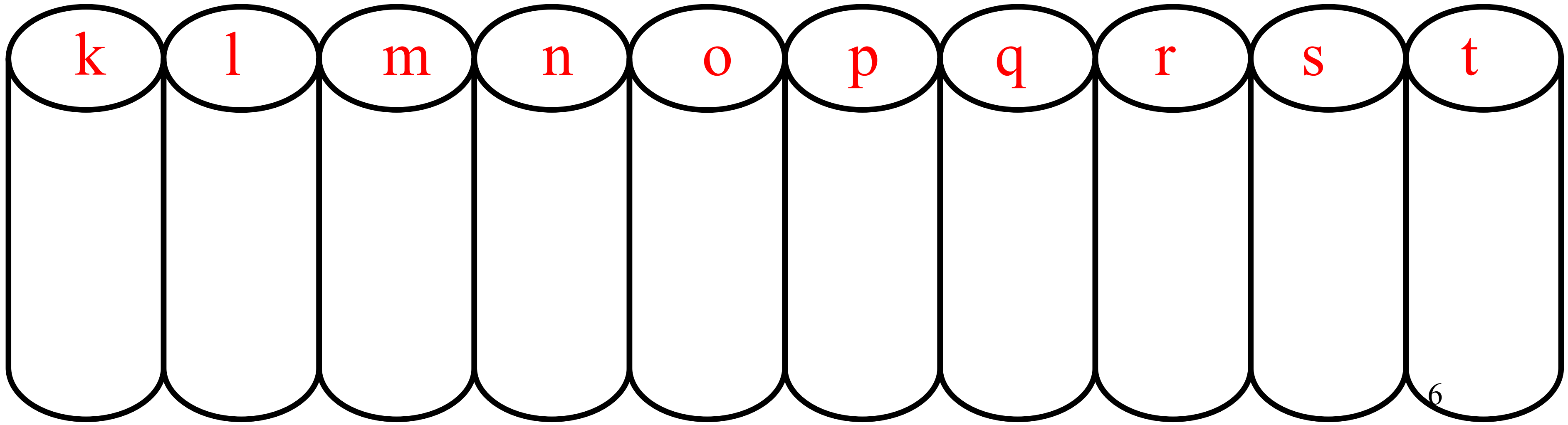
Bucket Sort

- bucketsort insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort



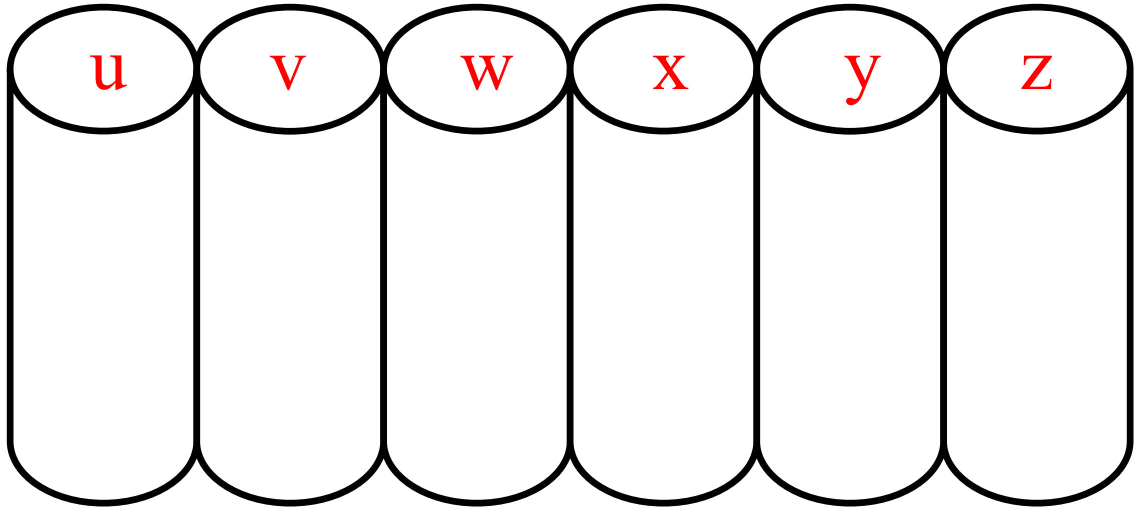
Bucket Sort

- bucketsort insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort



Bucket Sort

- bucketsort insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort

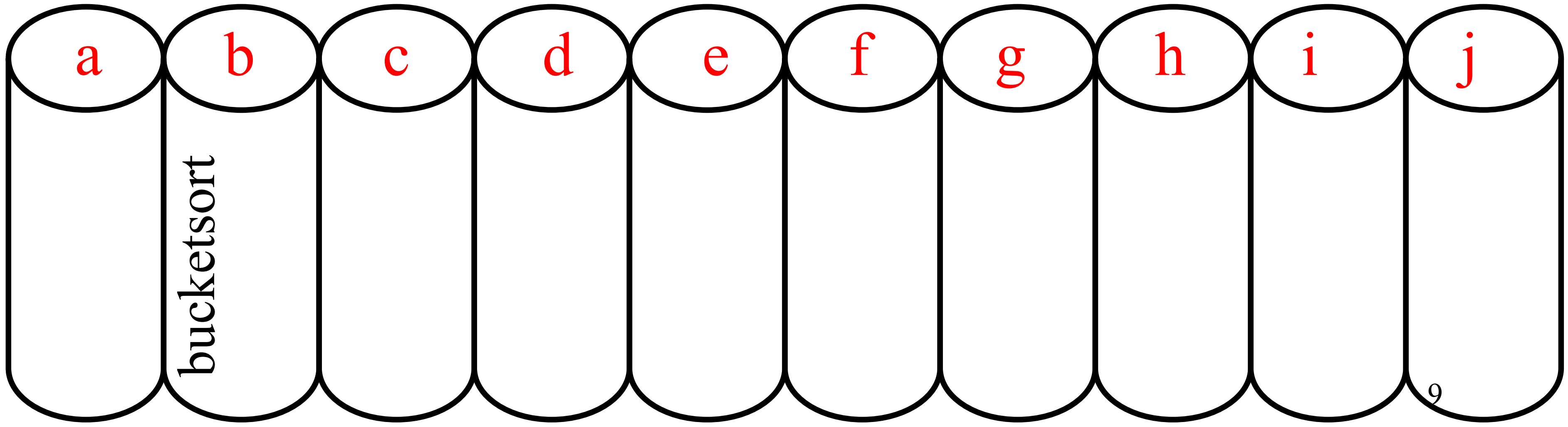


Bucket Sort

- bucketsort insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort

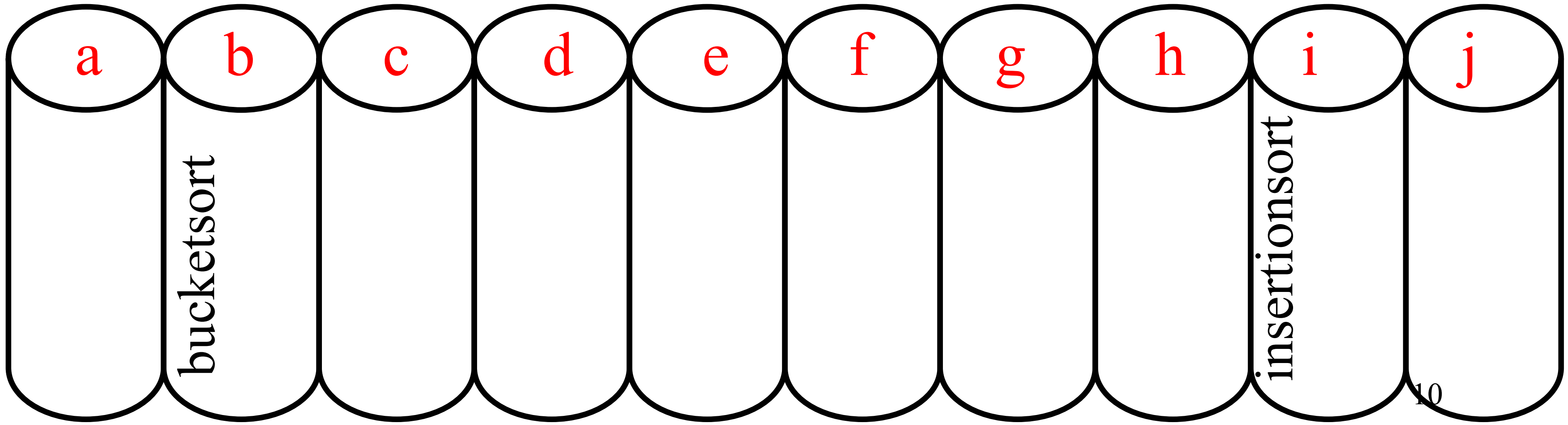
Bucket Sort

- **bucketsort** insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort



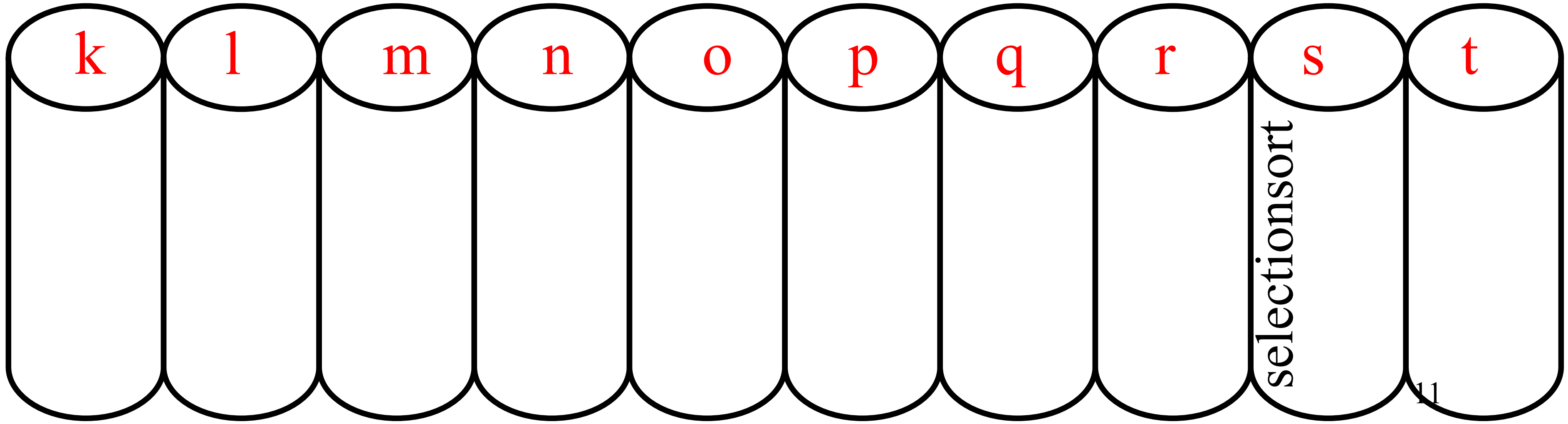
Bucket Sort

- **insertionsort** selectionsort quicksort mergesort shellsort treeselection heapsort



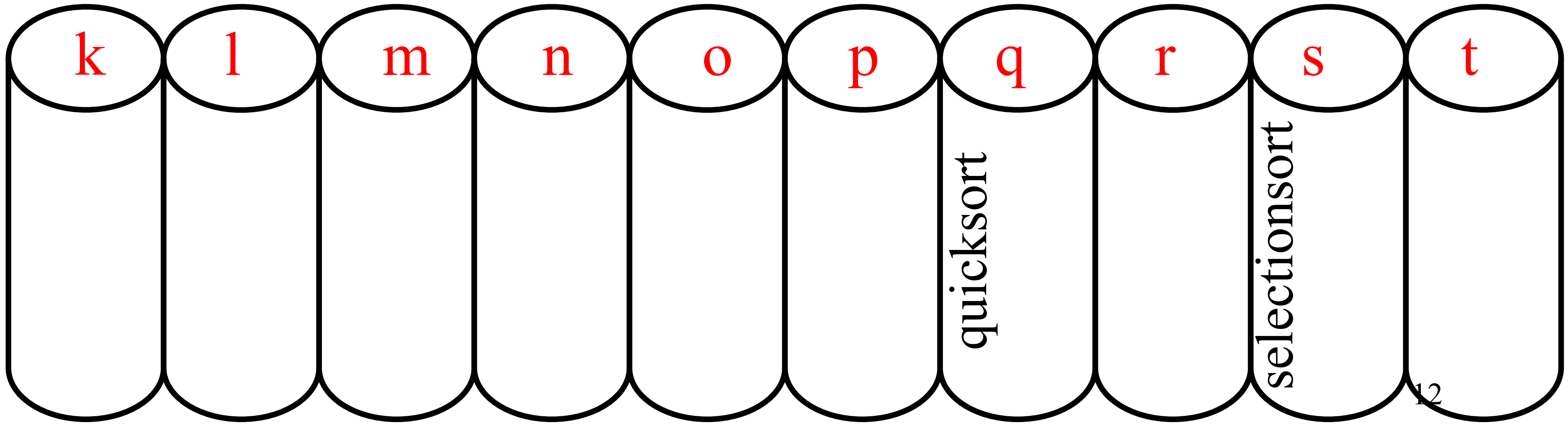
Bucket Sort

- **selectionsort** quicksort mergesort shellsort treeselection heapsort



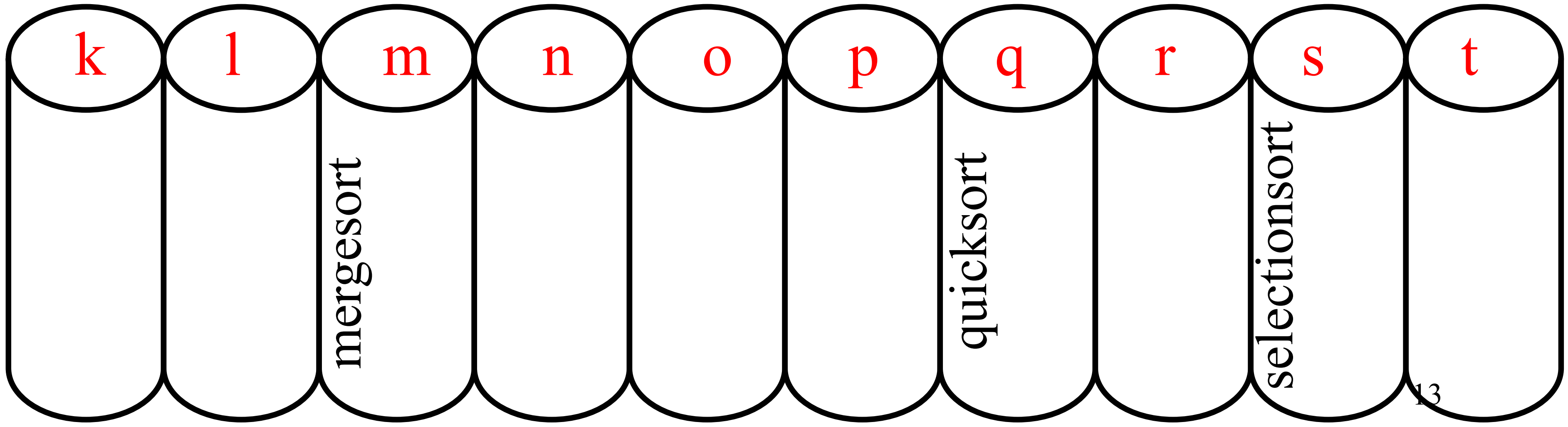
Bucket-Sort

- **quicksort** mergesort shellsort treeselection heapsort



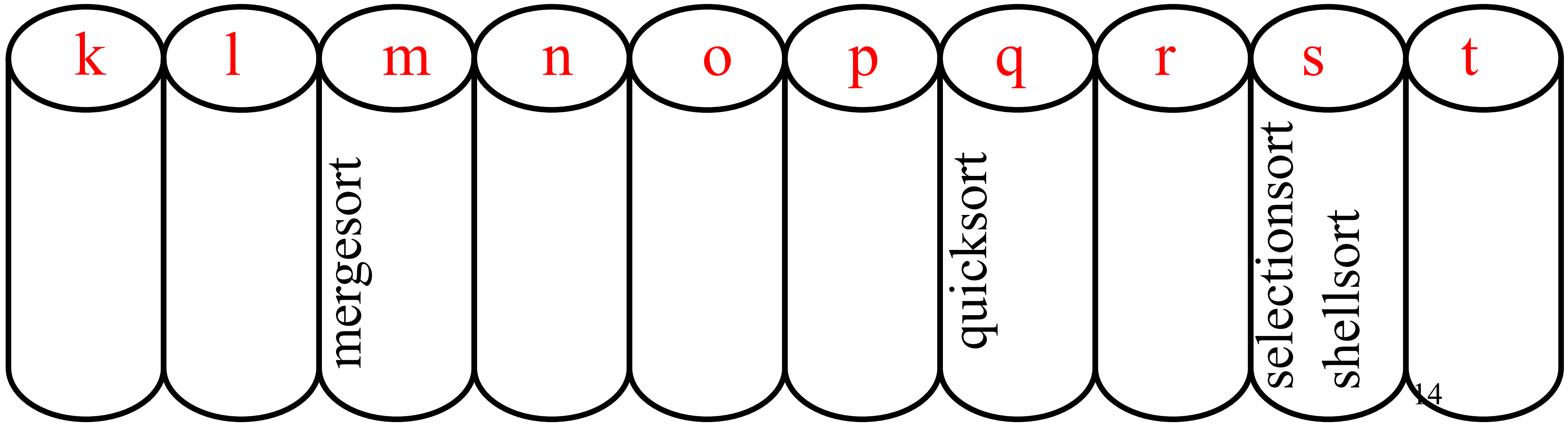
Bucket Sort

- **mergesort** shellsort treeselection heapsort



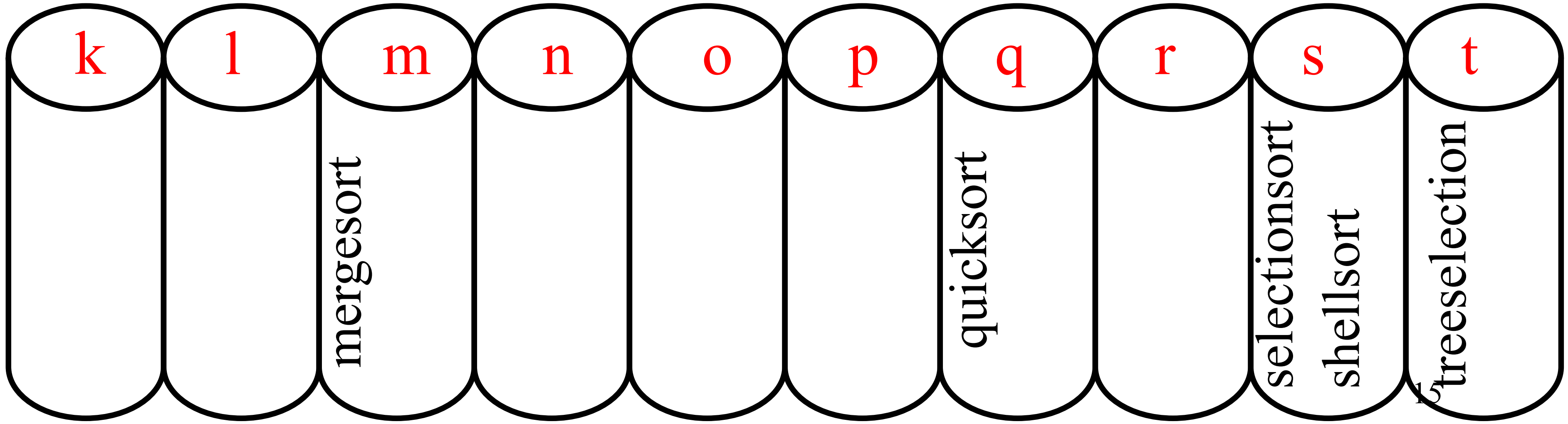
Bucket Sort

- **shellsort** treeselection heapsort



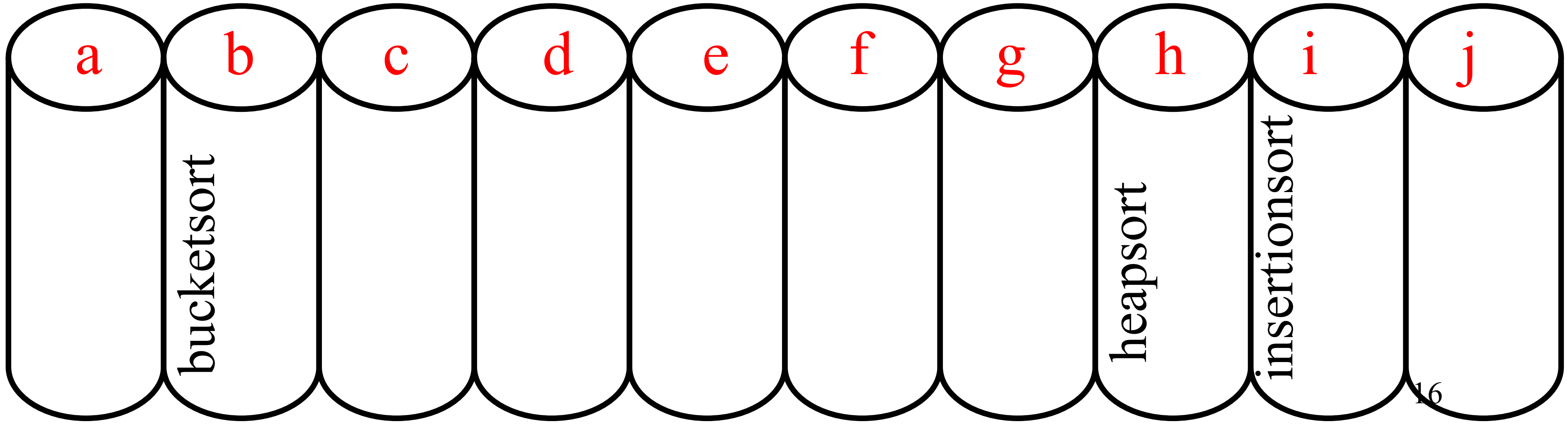
Bucket Sort

- **treeselection** heapsort



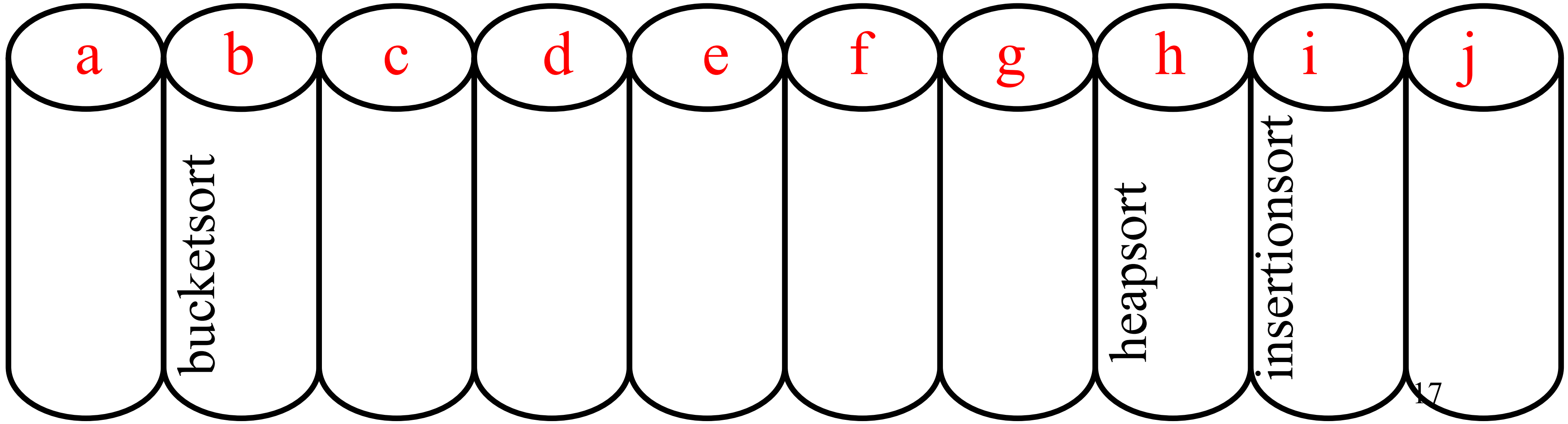
Bucket Sort

- heapsort



Bucket Sort

- Concatenate buckets
  - bucketsort heapsort insertionsort mergesort quicksort selectionsort shellsort treeselection



Bucket Sort

```
Algorithm bucketSort($S$)
    Input: Sequence $S$ of items with integer keys in the range [0, $N$–1]
    Output: Sequence $S$ sorted in nondecreasing order of the keys
    Let $B$ be an array of $N$ lists, each of which is
    initially empty
    for each item $x$ in $S$ do
        Let $k$ be the key of $x$
        Remove $x$ from $S$
        insert $x$ at the end of bucket $B[k]$          $O(n)$
    end
    for $i \leftarrow 0$ to $N-1$ do
        for each item $x$ in list $B[i]$ do
            remove $x$ from $B[i]$
            insert $x$ at the end of $S$               $O(n+N)$
        end
    end
```

Running Time of Bucket Sort

- First loop
  - ➢ Iterates n times
  - ➢ n removes from sequence S
  - ➢ n inserts into buckets B
- Second loop
  - ➢ Iterates N times
  - ➢ n removes from buckets B
  - ➢ n inserts into sequence S

Note that Bucket sort only sorts by one component of the key (e.g., first letter)

➔ **The time complexity of bucket sort is $O(n+N)$ and uses $O(n+N)$ space**

- ➢ Usually the range of N is small compared to n
- ➢ The second loop deals with the same elements as the first loop

➔ **The time complexity of bucket sort is $O(n)$ and uses $O(n)$ space**

Radix Sort

- Apply bucket sort multiple times to the components of a key or multiple keys.
- Integer representations can be used to represent things such as strings of characters (e.g., names of people, places).
- Suppose. For example, that keys are a pair (k, l) where k and l are integers in range $[0, N - 1]$.

- We define the lexicographical order as $(k_1, l_1) < (k_2, l_2)$ if
  - $k_1 < k_2$
  - $k_1 = k_2$ and $l_1 < l_2$

- Here, radix-sort applies bucket-sort twice, once on each component of the pair.

Radix Sort

- Two classifications of radix sorts are
  - ➢ Least significant digit (LSD) radix sorts (i.e., usually right most digit)
  - ➢ Most significant digit (MSD) radix sorts (i.e., usually left most digit)

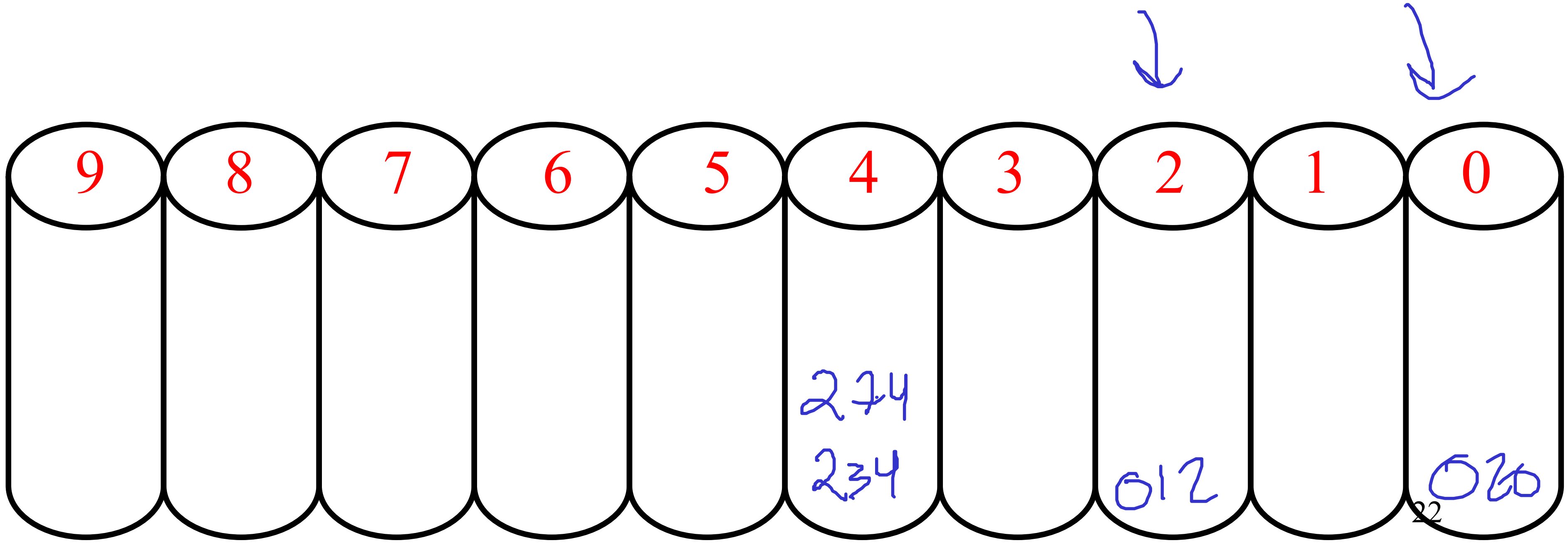
- LSD radix sorts process the integer representations starting from the <u>least significant digit</u> and move the processing towards the <u>most significant digit</u>

- MSD radix sorts process the integer representations starting from the <u>most significant digit</u> and move the processing towards the <u>least significant digit</u>

LSD Radix Sort—insert into buckets by LSD

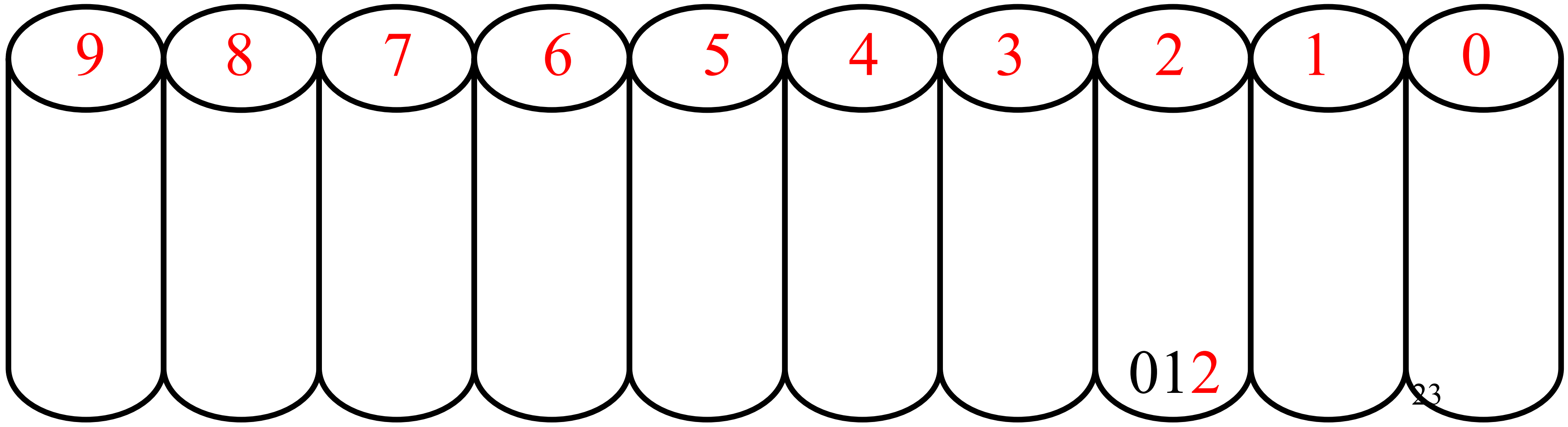
descending

~~012~~	~~234~~	~~274~~	020	001	111	002	034
009	029	199	109	005	203	123	401
568	073	193	122	033	120	040	081
006	221	032					



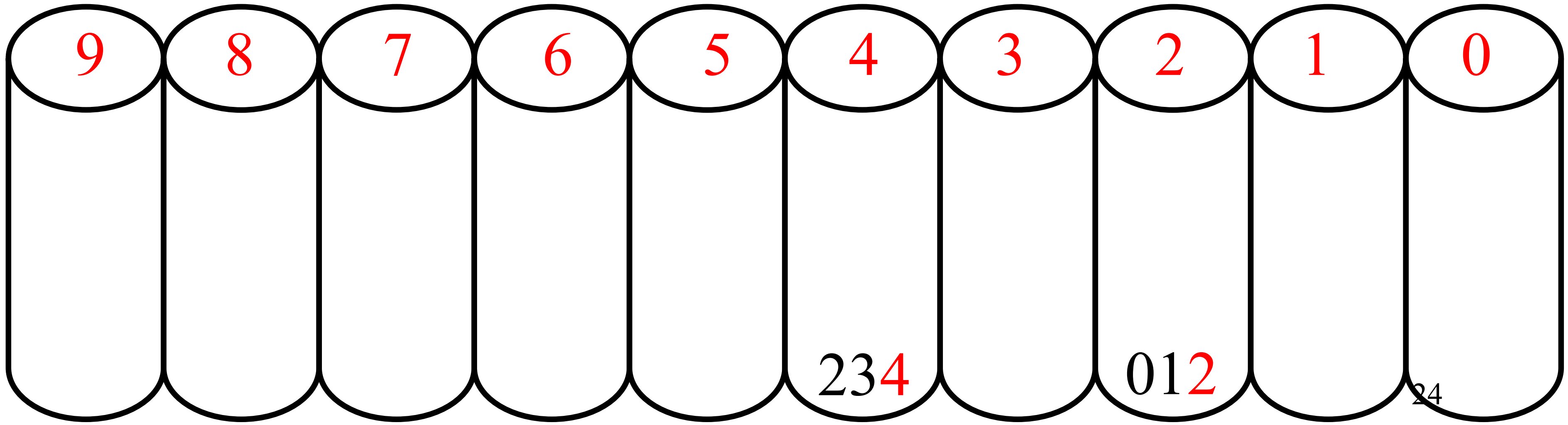
LSD Radix Sort—insert into buckets by LSD

	234	274	020	001	111	002	034
009	029	199	109	005	203	123	401
568	073	193	122	033	120	040	081
006	221	032					



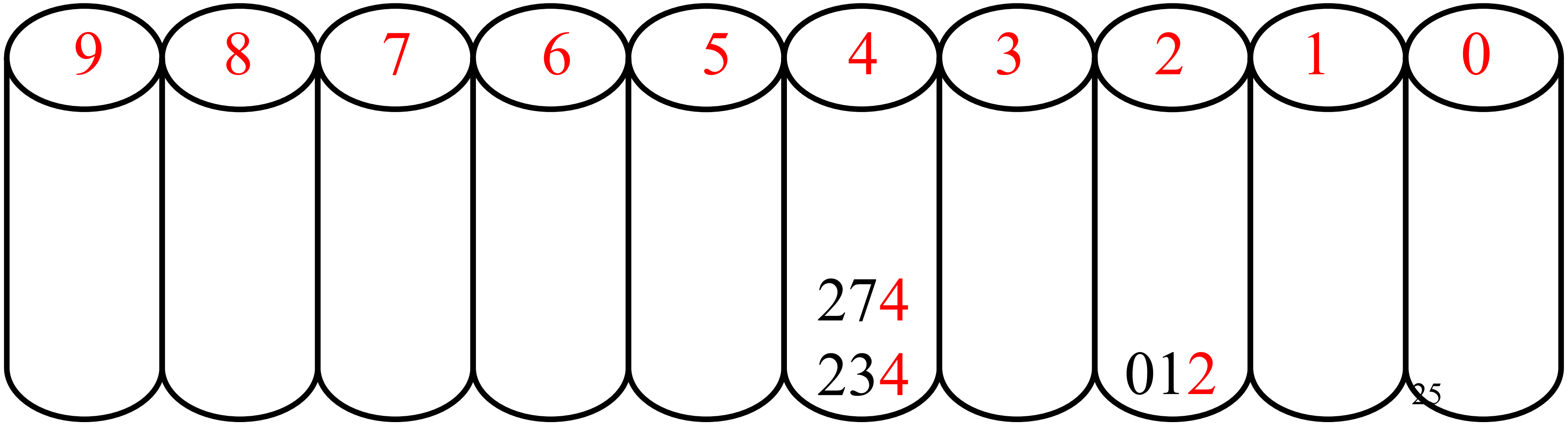
LSD Radix Sort—insert into buckets by LSD

		274	020	001	111	002	034
009	029	199	109	005	203	123	401
568	073	193	122	033	120	040	081
006	221	032					



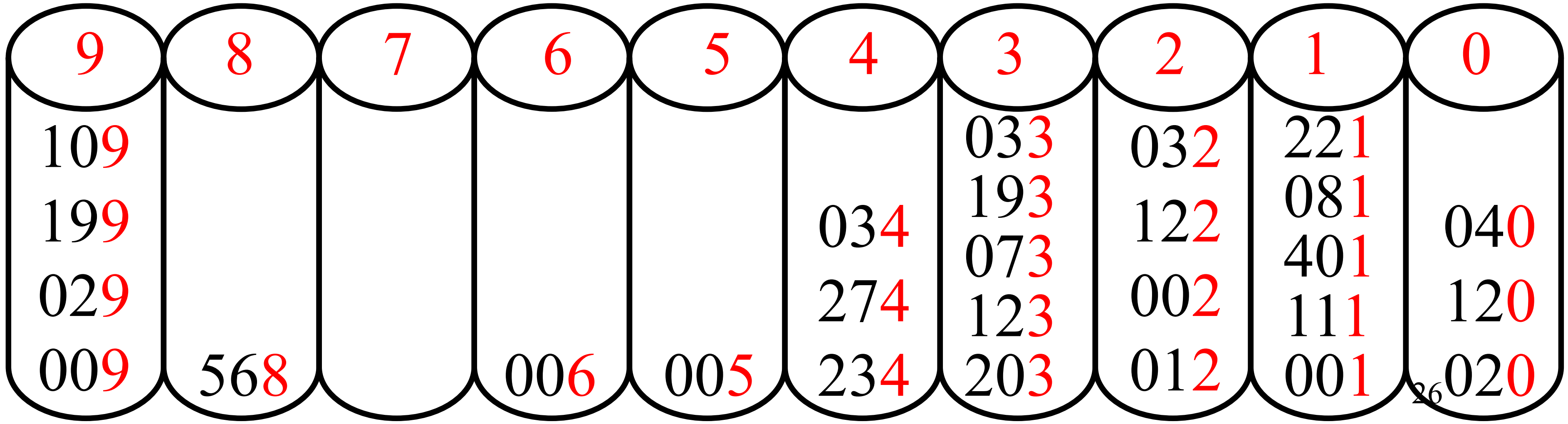
LSD Radix Sort—insert into buckets by LSD

			020	001	111	002	034
009	029	199	109	005	203	123	401
568	073	193	122	033	120	040	081
006	221	032					



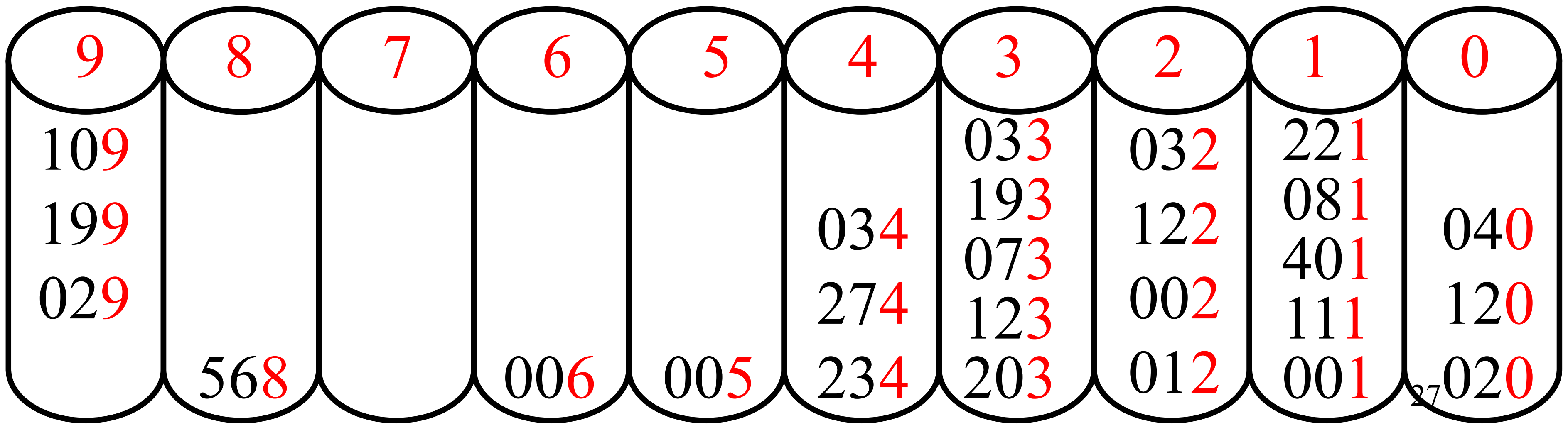
LSD Radix Sort—insert into buckets by LSD

009 029 199 109 568



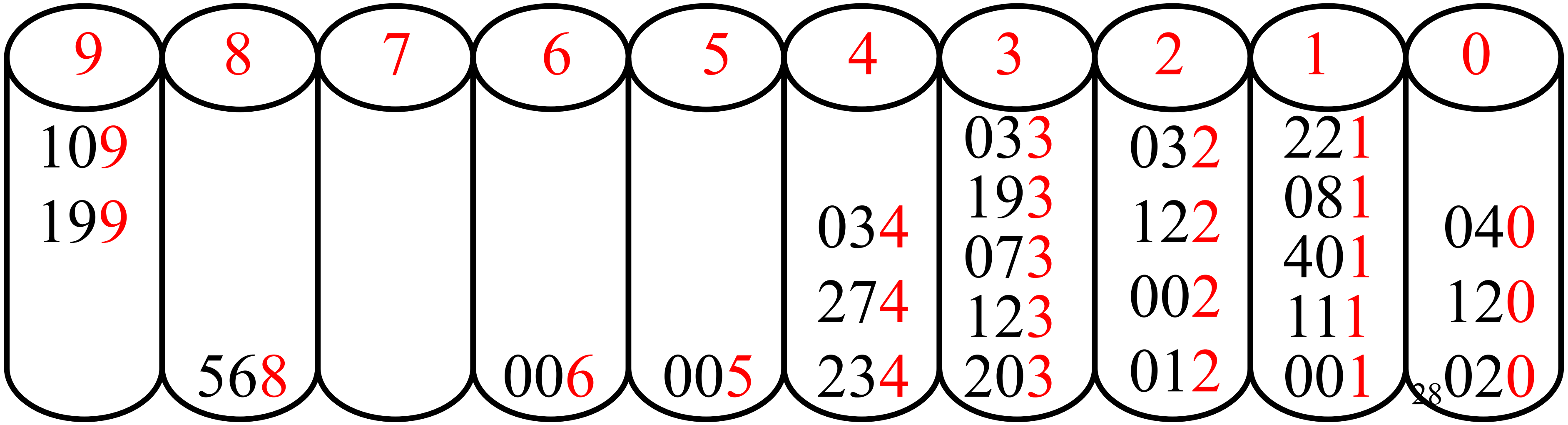
LSD Radix Sort—Concatenate

009



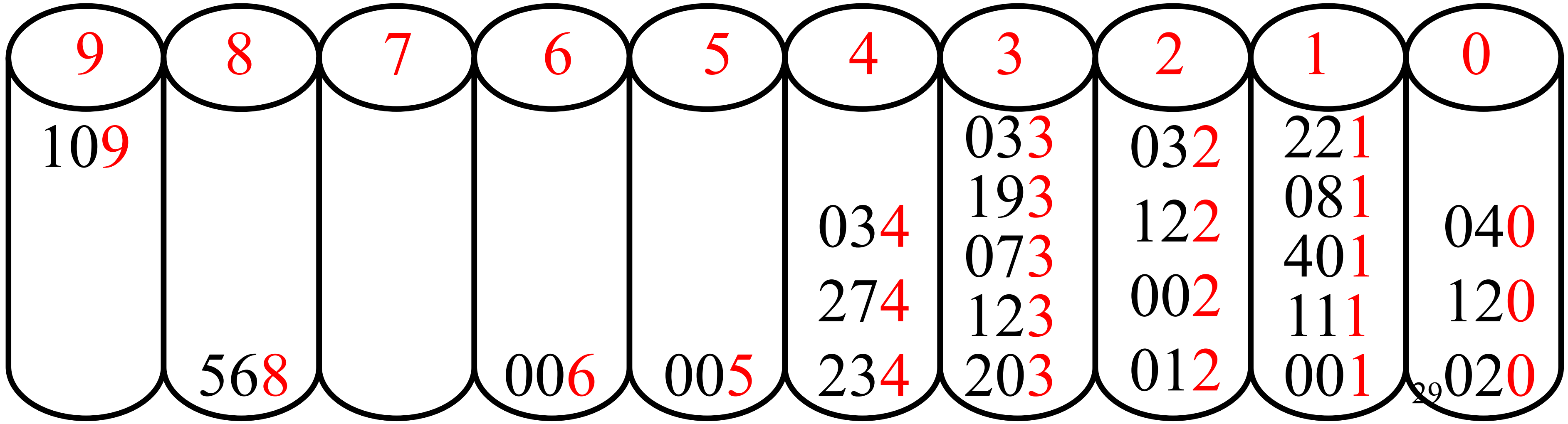
LSD Radix Sort—Concatenate

009 029



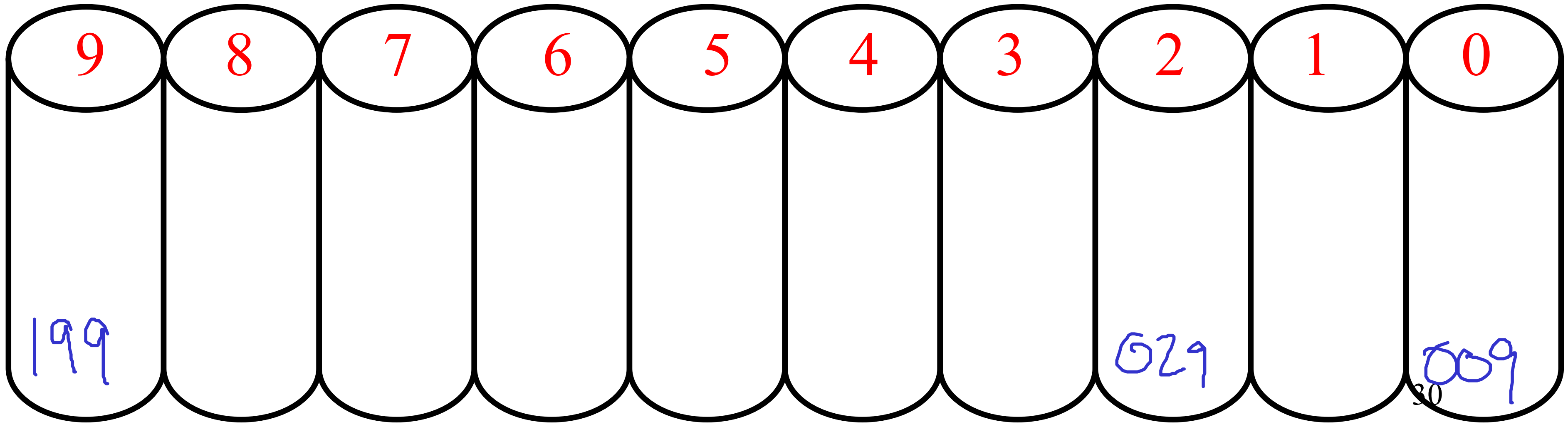
LSD Radix Sort—Concatenate

009 029 199



LSD Radix Sort—Concatenate

009	029	199	109	568	006	005	234
274	034	203	123	073	193	033	012
002	122	023	001	111	401	081	221
020	120	040					



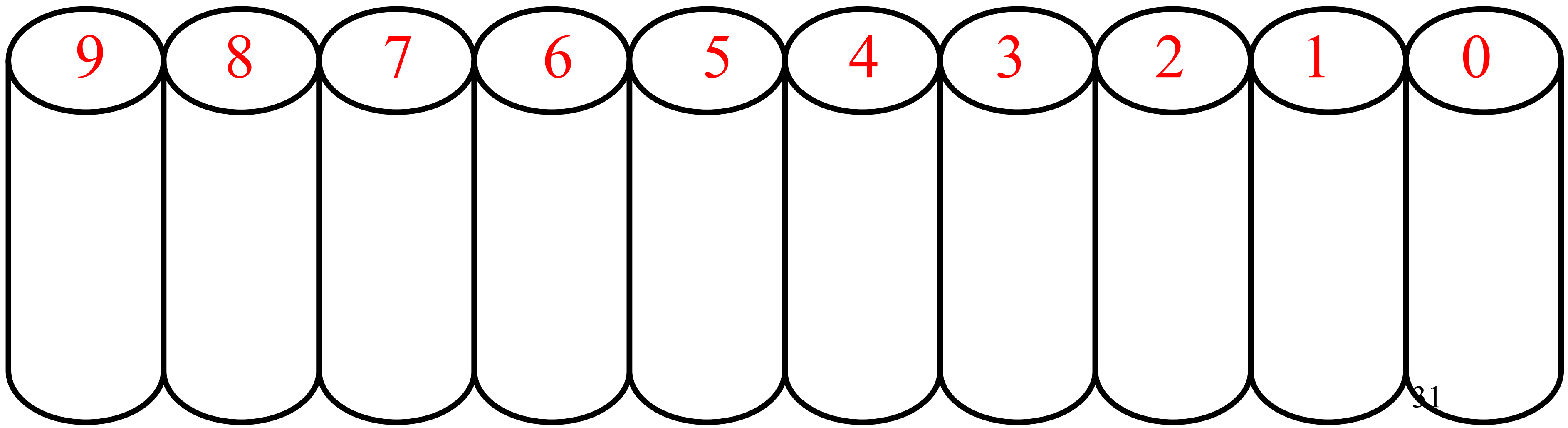
LSD Radix Sort—insert into buckets by 2nd digit

0$\overset{\downarrow}{0}$9   029   199   109   568   006   005   234

274   034   203   123   073   193   033   012

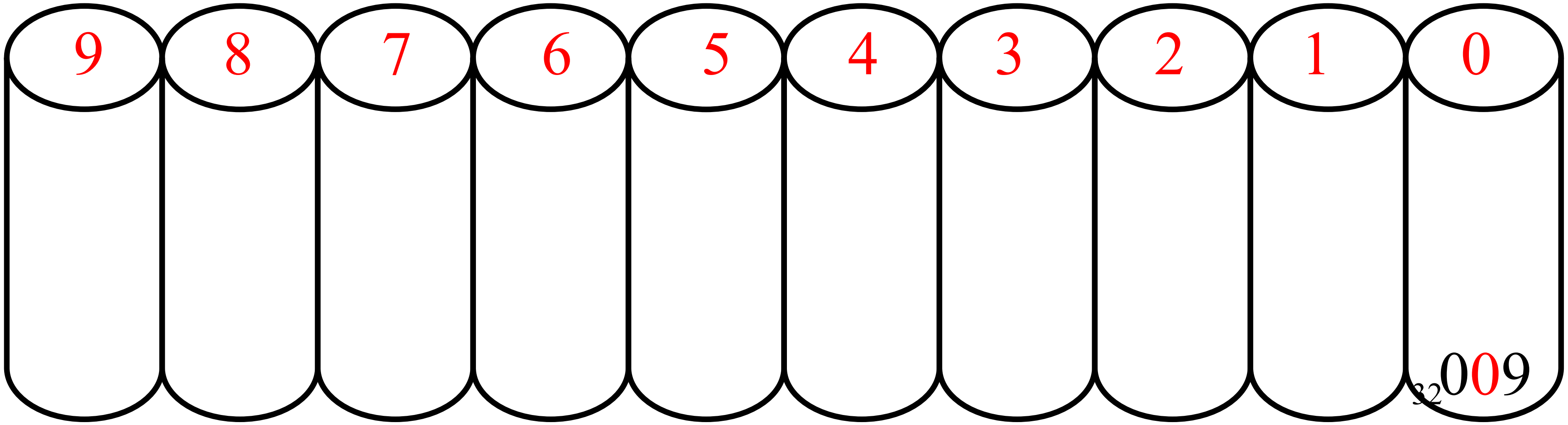
002   122   023   001   111   401   081   221

020   120   040



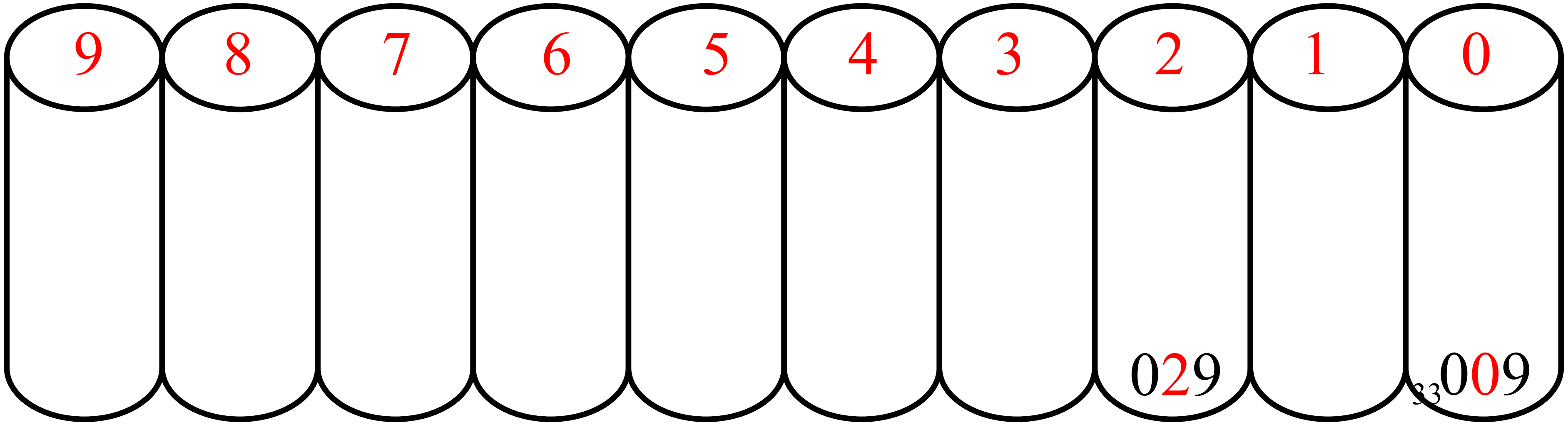
LSD Radix Sort—insert into buckets by 2nd digit

	029	199	109	568	006	005	234
274	034	203	123	073	193	033	012
002	122	023	001	111	401	081	221
020	120	040					



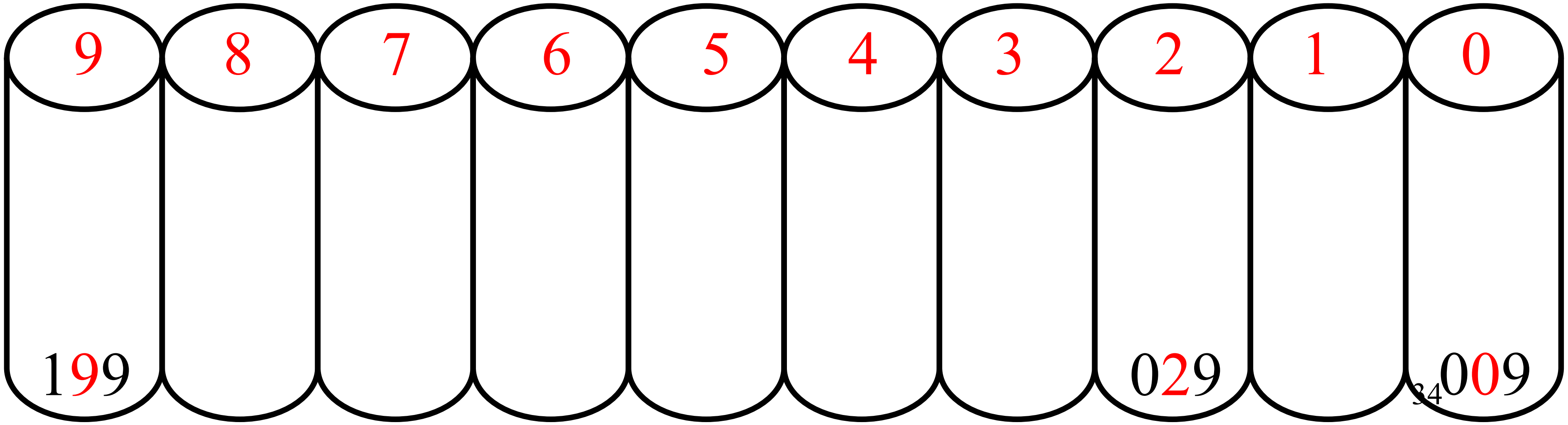
LSD Radix Sort—insert into buckets by 2nd digit

		199	109	568	006	005	234
274	034	203	123	073	193	033	012
002	122	023	001	111	401	081	221
020	120	040					



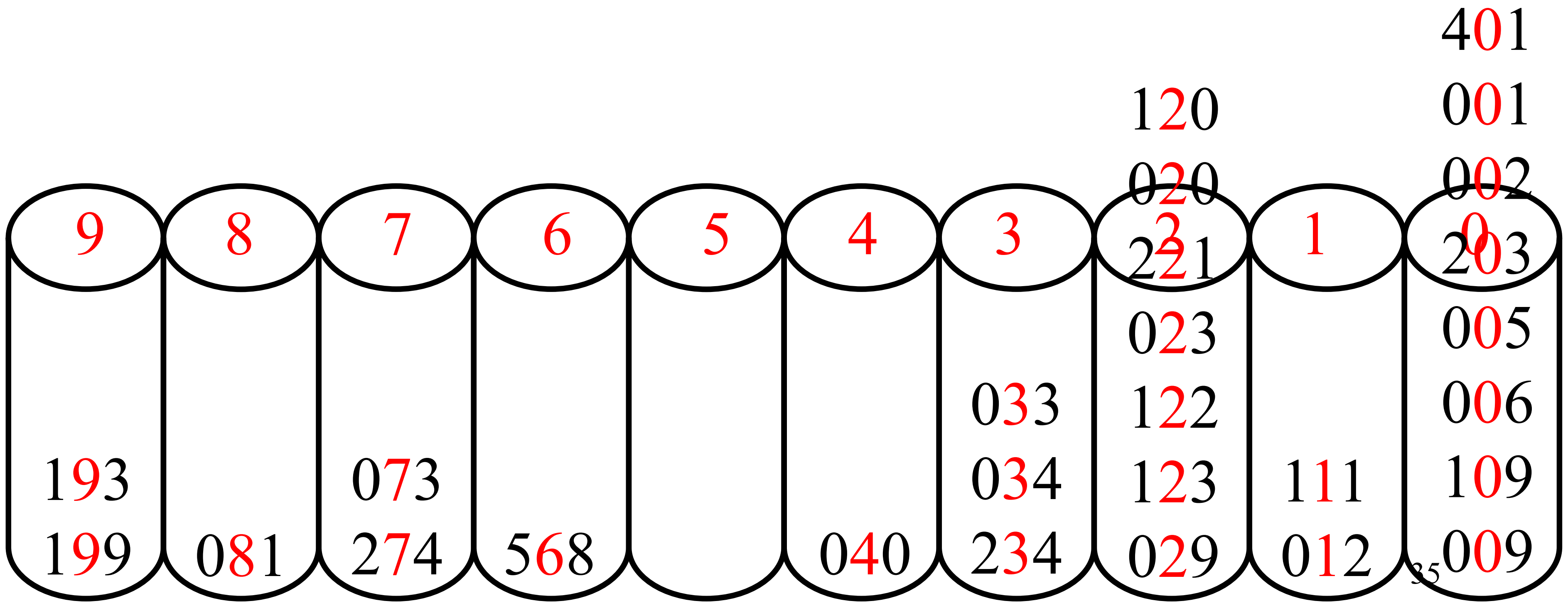
LSD Radix Sort—insert into buckets by 2nd digit

109 568 006 005 234
274 034 203 123 073 193 033 012
002 122 023 001 111 401 081 221
020 120 040



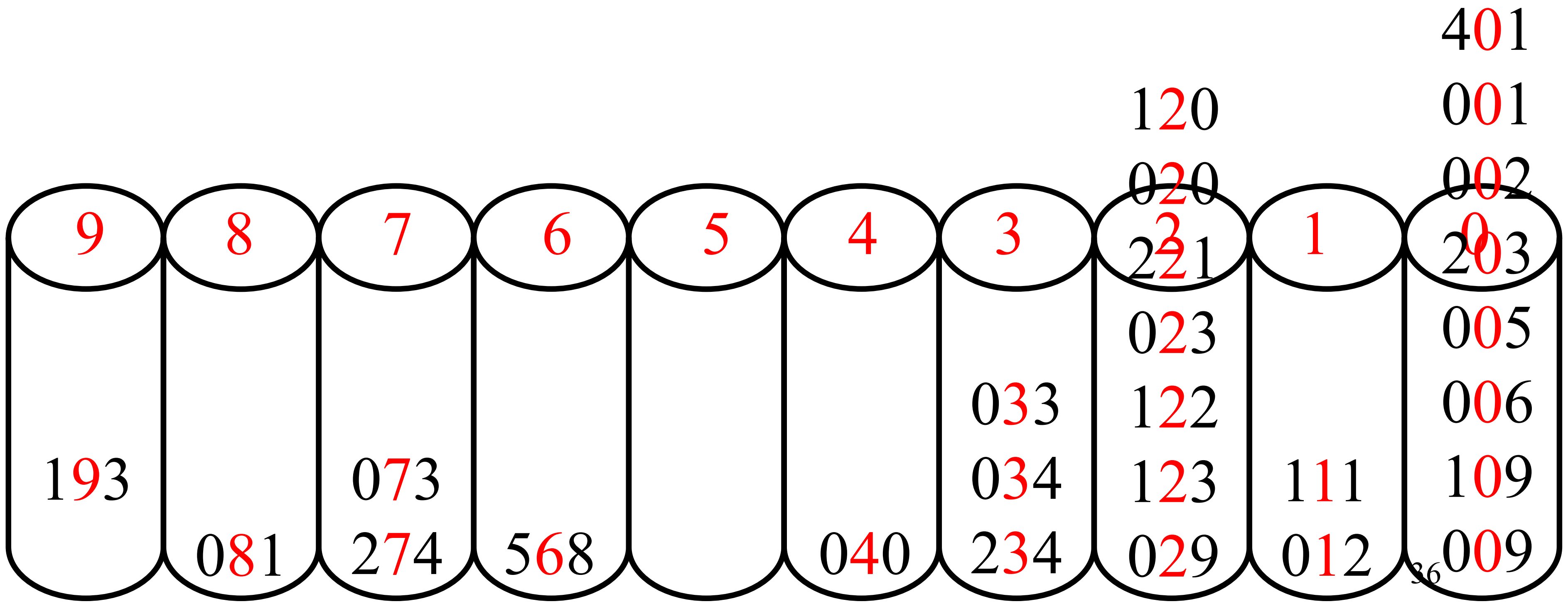
LSD Radix Sort—insert into buckets by 2nd digit

199 193 081 . . - .



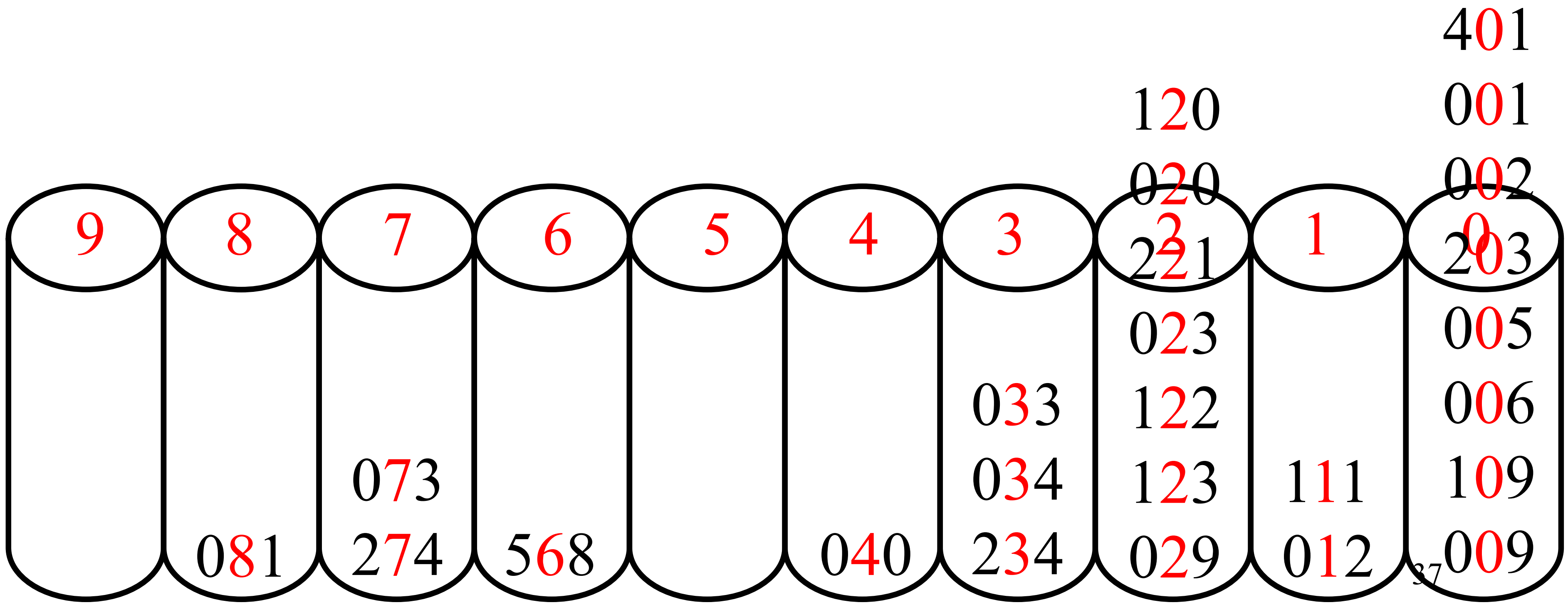
LSD Radix Sort—concatenate

199



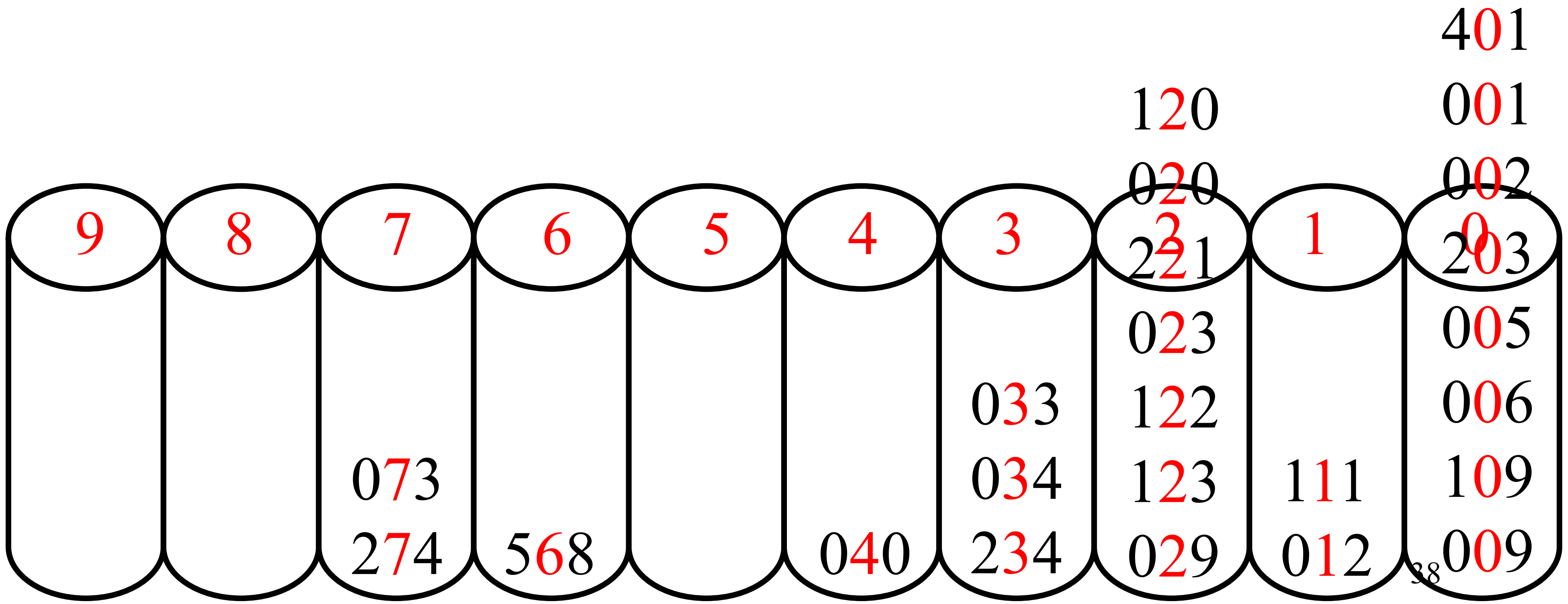
LSD Radix Sort—concatenate

199 193



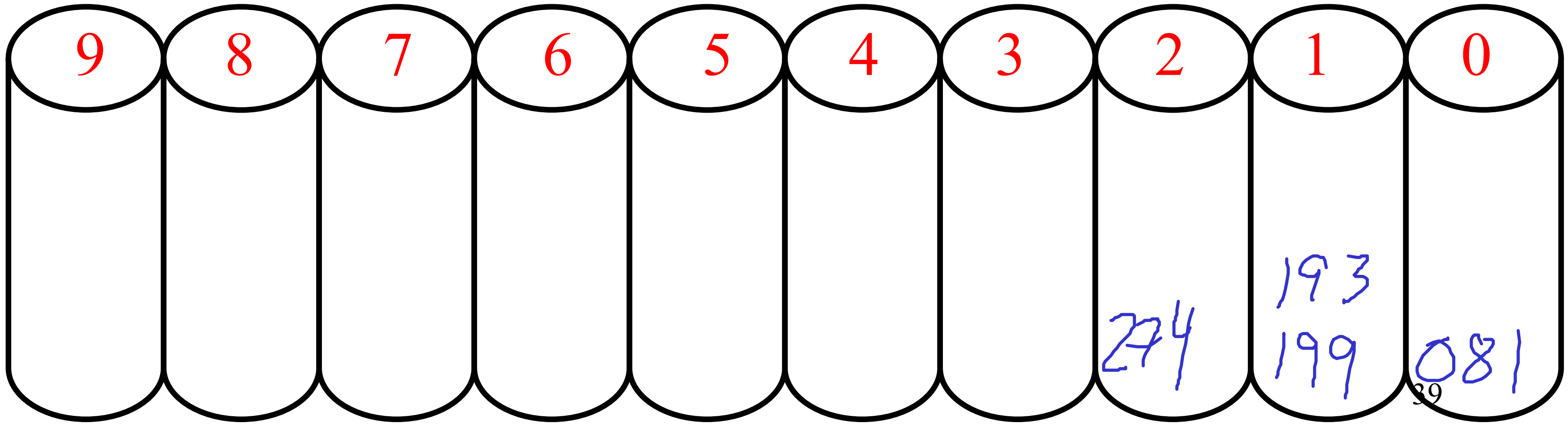
LSD Radix Sort—concatenate

199 193 081



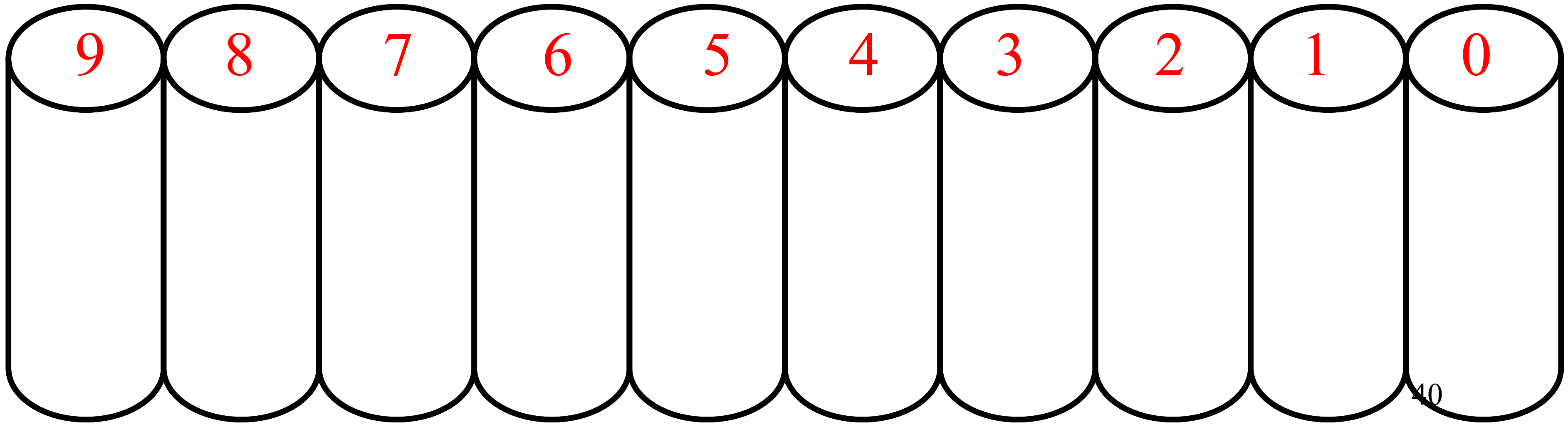
LSD Radix Sort—concatenate

199	193	081	274	073	568	040	234
034	033	029	123	122	023	221	020
120	012	111	009	109	006	005	203
002	001	401					



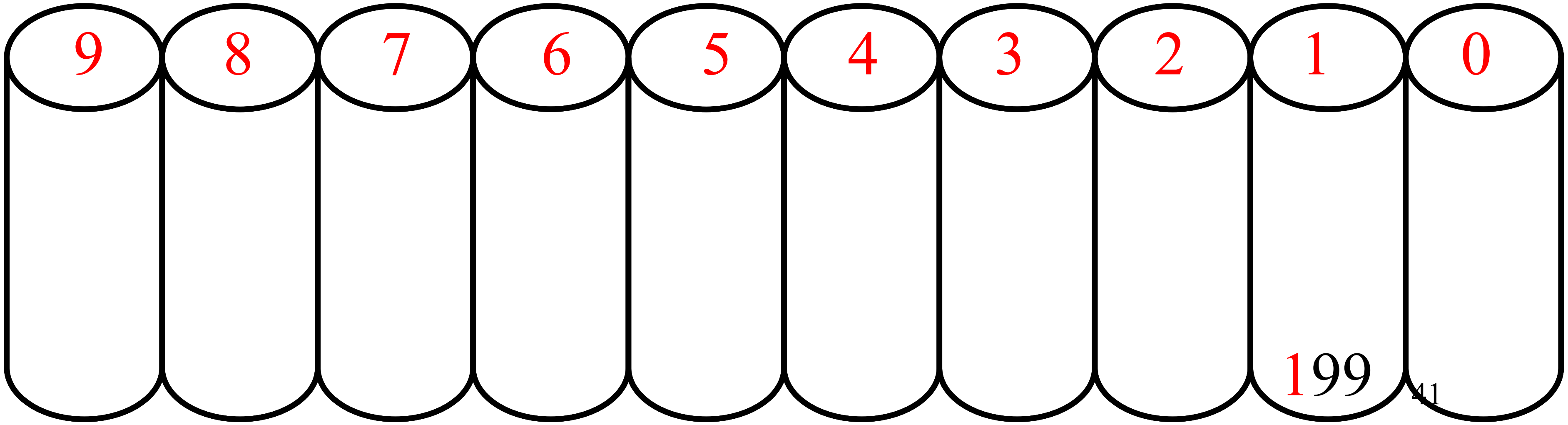
LSD Radix Sort

199	193	081	274	073	568	040	234
034	033	029	123	122	023	221	020
120	012	111	009	109	006	005	203
002	001	401					



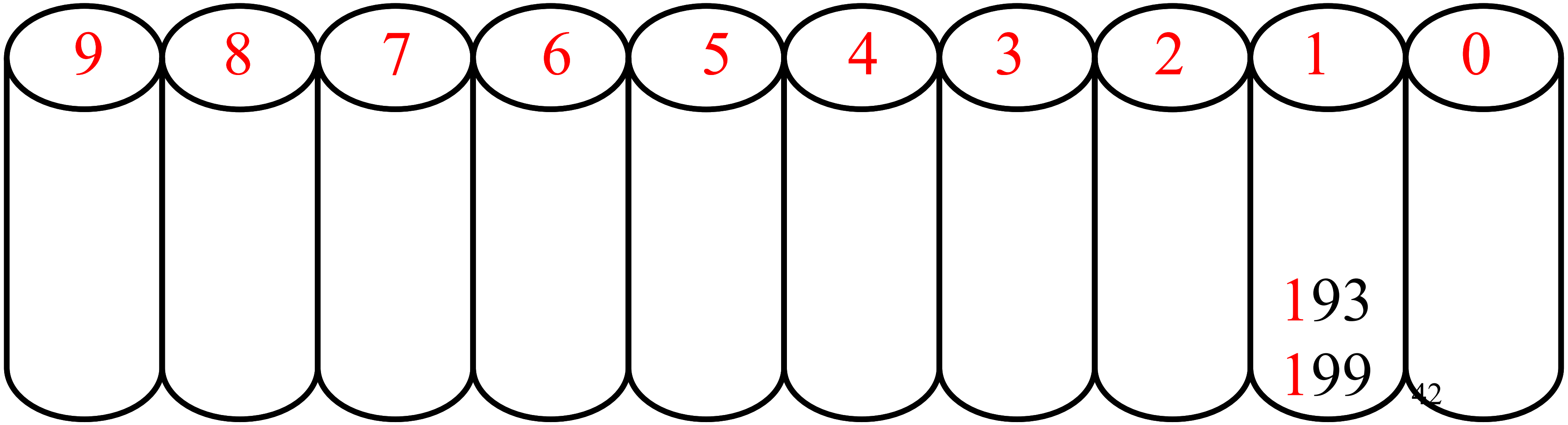
LSD Radix Sort

	193	081	274	073	568	040	234
034	033	029	123	122	023	221	020
120	012	111	009	109	006	005	203
002	001	401					



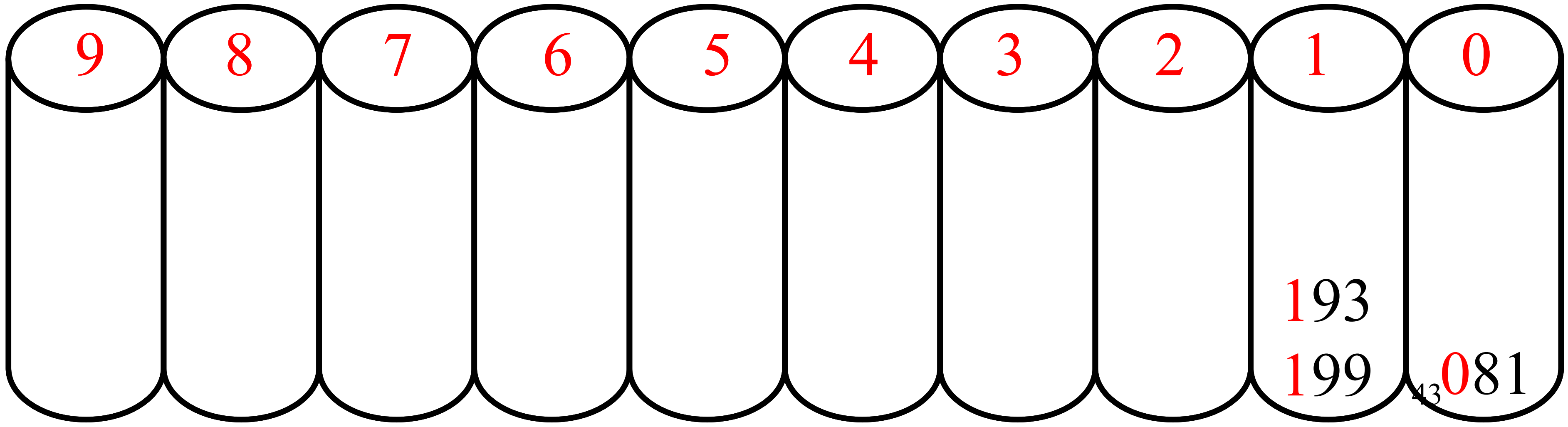
LSD Radix Sort

		081	274	073	568	040	234
034	033	029	123	122	023	221	020
120	012	111	009	109	006	005	203
002	001	401					



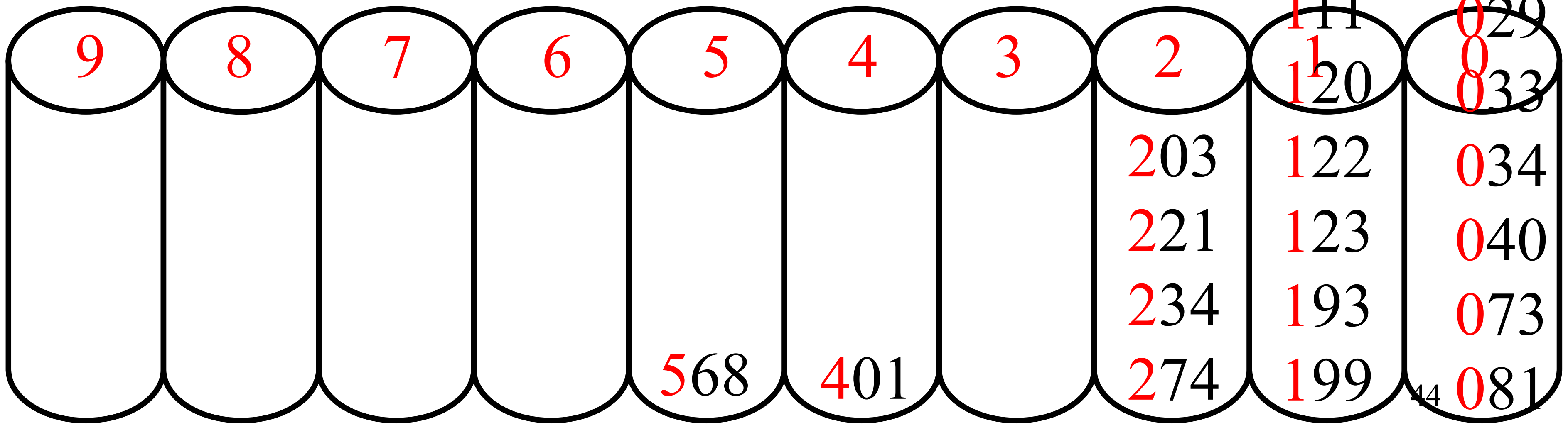
LSD Radix Sort

274 073 568 040 234
034 033 029 123 122 023 221 020
120 012 111 009 109 006 005 203
002 001 401



LSD Radix Sort

568 401 274 234 221 203

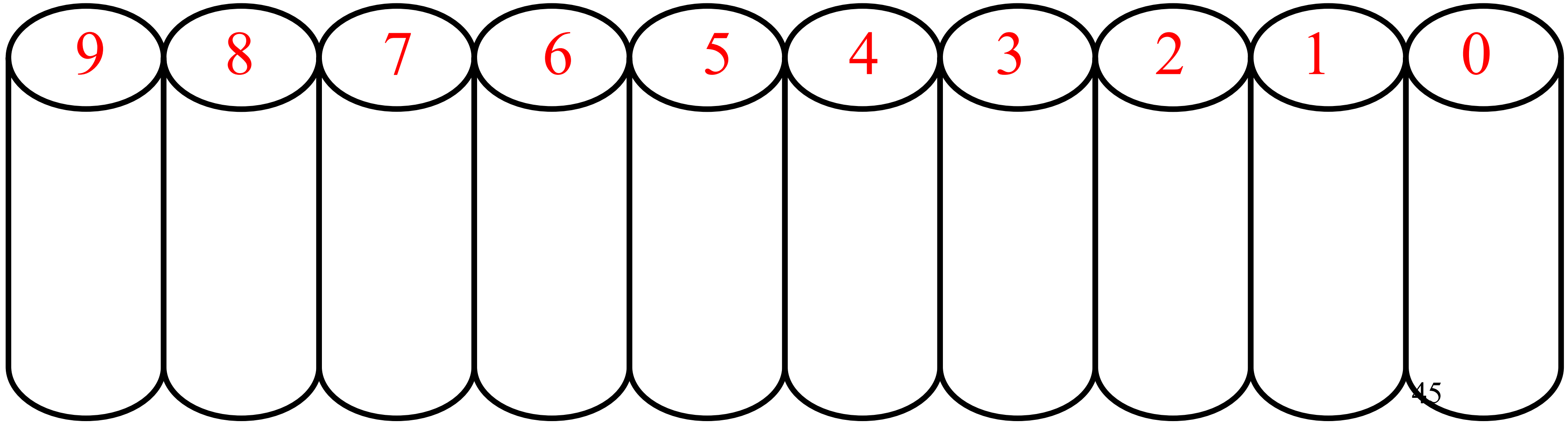


001
002
005
006
009
012
020
023
029
033
034
040
073
081

LSD Radix Sort—sorted

568	401	274	234	221	203	199	193
123	122	120	111	109	081	073	040
034	033	029	023	020	012	009	006
005	002	001					

$\underline{\underline{\underline{O(3n)}}}$



Radix Sort

- Repeated sorting by means of Bucket Sort
  - For each component of the key perform one Bucket Sort
- Start with the least significant component of the key and end with most significant component
- Implement buckets as queues
- Let the number of components per key be d

- **Theorem. The time complexity of Radix Sort is** $\boxed{O(d(n+N))}$ **or** $\boxed{O(dn)}$ **for large *n*.**