Radix Sort

CS 240E

David Duan, Winter 2019

Contents

1 Overview

In mathematical numeral systems, the **radix** (aka *base*) is the number of unique digits, including the digit zero, used to represent numbers in a positional numeral system. For example, the binary system has a radix of 2 and the hexadecimal system has a radix of 16.

In computer science, **radix sort** is a *non-comparative*, *integer* sorting algorithm that sorts by grouping numbers by their individual digits (i.e., their radix). It uses each radix/digit as a key, and implements *counting sort* or *bucket sort* under the hood in order to do the work of sorting.

2 Single-Digit Bucket Sort

2.1 Strategy

Let A be an array of integers that we want to sort, B be an array of lists indexed from 0 to $R-1$ ($R=4$ in the following example). We sort elements in array A by their last digit:

```
          Put numbers into buckets   Concatenate them back into A
 A   ====================>   B   ====================>    A
123                          B[0] : [230, 320, 210]       230
230                          B[1] : [021, 101]            320
021                          B[2] : [232]                 210
320                          B[3] : [123]                 021
210                                                       101
232                                                       232
101                                                       132
```

Note that we append new items to the end of the list in buckets. This guarantees that sorting is stable, i.e., the relative order of records with equal keys is maintained.

2.2 Pseudocode

```
Bucket-sort(A, d)
A: array of size n, contains numbers with digits in {0, ..., R-1}
d: index of digit by which we wish to sort

1.  initialize an array B[0, ..., R-1] of empty lists
2.  for i <- 0 to n-1 do
3.    append A[i] at the end of B[dth digit of A[i]]
4.  i <- 0
5.  for j <- 0 to R-1 do
6.    while B[j] is non-empty do
7.      move first element of B[j] to A[i++]
```

3 Count Sort

3.1 Strategy

To avoid building new lists, observe the first item of $B[j]$ gets copied to index $|B[0]| + |B[1]| + \cdots + |B[j-1]|$ and the others follow. Thus, we don't need to build additional lists; it is enough to count how many there would be in it.

```
      Pre-compute Count                          Sort A
 A  ================>  Digit | Count | Index ================>   A
123                      0       3       0                   230  >  B[0]
230                      1       2       3                   320  >  B[0]
021                      2       1       5                   210  >  B[0]
320                      3       1       6                   021  >  B[1]
210                                                          101  >  B[1]
232                                                          232  >  B[2]
101                                                          123  >  B[3]
```

3.2 Pseudocode

```
key-indexed-count-sort(A, d)
A: array of size n, contains numbers with digits in {0, ..., R-1}
d: index of digit by which we wish to sort

// Populate `Count` column: count how many of each kind there are
1.  count <- array of size R, filled with zeros.
2.  for i <- 1 to n-1 do
3.    increment count[dth digit of A[i]]

// Populate `Index` column: find left boundary for each kind
4.  idx <- array of size R, idx[0] = 0
5.  for i <- 1 to R-1 do
6.    idx[i] <- idx[i-1] + count[i-1]

// Populate new array: move to new array in sorted order, then copy back
7.  aux = array of size n
8.  for i <- 0 to n-1 do
9.    aux[idx[A[i]]] <- A[i]
10.   increment idx[A[i]]
11. A <- copy(aux)
```

3.3 Analysis

3.3.1 Time and Space Complexity

- Runtime: $O(n+R)$
- Auxiliary Space: $\Theta(n)$

3.3.2 Decision Tree

- There are R children for each node because there are R buckets/choices.
- Given n digits from the range $\{0, \ldots, R-1\}$, we have R^n leaves in the decision tree.
- Let h be the height of the decision tree, then $h \geq \log_R(|\text{leaves}|)\Omega(\log_R(R^n)) = n$.
- As a conclusion, given n numbers ($\{0, \ldots, n-1\}$), you can sort them in $O(n)$.

4 Radix Sort

4.1 MSD-Radix-Sort

To sort multiple digits, MSD-Radix-Sort starts with the *most-significant-digit*.

```
MSD-Radix-Sort(A, l, r, d)
A: array of size n, contains m-digit radix-R numbers.
l, r, d: integers 0 <= l, r <= n-1 and 1 <= d <= m.

0.  i <- 0, r <- n-1, d <- 1
1.  if l < r
2.    partition A[l..r] into bins according to dth digit // use count sort here
3.    if d < m
4.      for i <- 0 to R-1 do
5.        let l_i and r_i be boundaries of ith bin // returned by count sort
6.        MSD-Radix-sort(A, l_1, r_1, d+1)
```

The main disadvantage of MSD Radix sorting is that when the data is cut up into small pieces, the overhead for additional recursive calls starts to dominate runtime, and this makes worst-case behavior substantially worse than $O(n \log(n))$.

4.2 LSD-Radix-Sort

Instead, we can start sorting from the last digit.

```
LSD-Radix-Sort(A)
A:array of size n, contains m-digit radix-R numbers.

1.  for d <- m down to 1 do
2.    key-indexed-count-sort(A, d)
```

4.2.1 Example

```
123              230              101              021
231              230              210              101
021              210              320              123
320  == d=3 ==>  021  == d=2 ==>  021  == d=1 ==>  210
210              101              123              230
232              232              230              232
101              123              232              320
```

4.2.2 Analysis

- *Loop Invariant*: A is sorted with respect to digit $d, \ldots, m$ of each entry, thanks to count sort being stable.
- *Time*: $\Theta(m(n + R))$.
- *Space*: $\Theta(n + R)$.