Linear Sort

"Our intuition about the future is linear. But the reality of information technology is exponential, and that makes a profound difference. If I take 30 steps linearly, I get to 30. If I take 30 steps exponentially, I get to a billion."

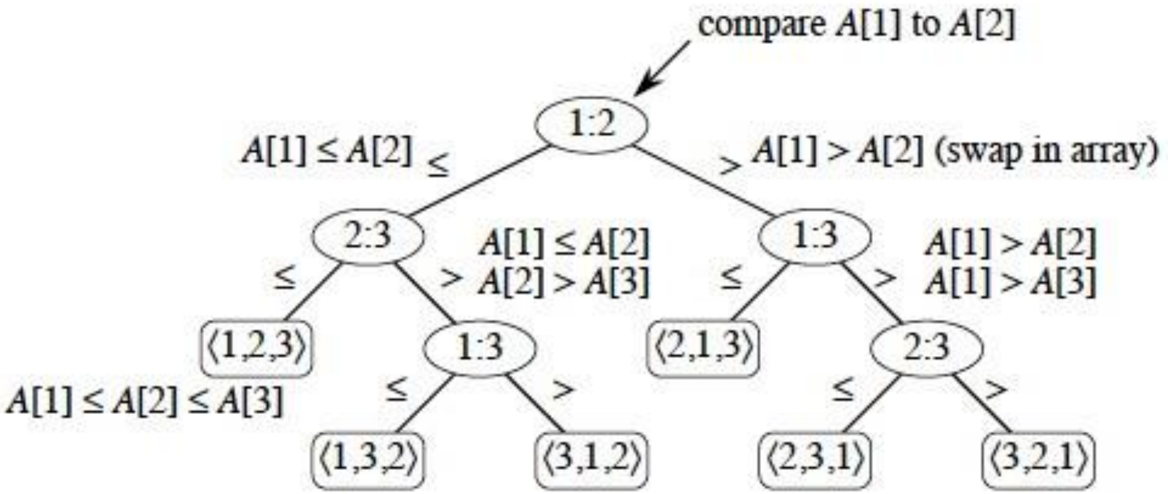
-Ray Kurzweil

Lower Bound for Sorting

- All the sorting algorithms seen so far are called *comparison sorts*: they sort by using comparison operations between the array elements.
- We can show that any comparison sort must make $\Omega(n*\log_2 n)$ comparisons in the <u>worst case</u> to sort `n` elements.

Lower Bound for Sorting

- Decision Tree Model: full binary tree representing comparison between array elements performed by a sorting algorithm
  - Internal nodes represent comparisons
  - Leaves represent outcomes
    - all array elements permutations
- Example: decision tree for insertion sort



Lower Bound for Sorting

- Worst-case number of comparisons = length of the longest simple path from the root to any leaves in the decision tree (i.e. tree height)
- Possible outcomes = total number of permutations = $n!$
- Therefore, the decision tree has at least $n!$ leaves
- In general, a decision tree of height h has 2^h leaves
- Thus, we have

$$n! \leq 2^h$$

$$h \geq \log_2(n!)$$

- Using Stirling's approximation:

$$n! > (n/e)^n$$

$$h \geq \log_2((n/e)^n) =$$

$$n*\log_2(n/e) = n*\log_2 n - n*\log_2 e$$

$$h = \Omega(n*\log_2 n)$$

Lower Bound for Sorting

Theorem: any comparison sort algorithm requires $\Omega(n*\log_2 n)$ comparisons in the worst case

- Logarithmic sorting algorithms, like heapsort, quicksort, and mergesort, have a worst-case running time of $O(n*\log_2 n)$

Corollary: Logarithmic sorting algorithms are asymptotically optimal for comparison sort.

Can we do better?

Sorting in Linear Time

- Comparison Sort:
  - Lower bound: $\Omega(n*\log_2 n)$

- Non-Comparison Sorts:
  - Possible to sort in linear time
    - under certain assumptions
  - Examples:
    - Counting sort
    - Bucket sort
    - Radix sort

Counting Sort

- Assumption:

  n input numbers are integers in range $[0, k]$, $k = O(n)$.

- Idea:
  – Determine the number of elements less than x, for each input x.
  – Place x directly in its position.

Count Sort Algorithm

```
COUNTING-SORT(A, B, k)
for i← 0 to k
     do C[i] ←0
for j ← 1 to length[A]
     do C[A[j]] ← C[A[j]] + 1
// C[i] contains number of elements equal to i.
for i ← 1 to k
      do C[i] ← C[i] + C[i - 1]
// C[i] contains number of elements ≤ i.
for j ← length[A] downto 1
       do B[C[A[j]]] ← A[j]
            C[A[j]] ← C[A[j]] - 1
```

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for i ← 0 to k
      do C[i] ← 0
```

// initialize to 0

C:

0	1	2	3	4	5
0	0	0	0	0	0

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for j ← 1 to length[A]
     do C[A[j]] ← C[A[j]] + 1
```

```
// count elements
// equal to i
```

C:

0	1	2	3	4	5
0	0	0	0	0	0

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for j ← 1 to length[A]
     do C[A[j]] ← C[A[j]] + 1
```

j = 1
C[2] ← 0 + 1

C:

0	1	2	3	4	5
0	0	1	0	0	0

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for j ← 1 to length[A]
      do C[A[j]] ← C[A[j]] + 1
```

j = 2
C[5] ← 0 + 1

C:

0	1	2	3	4	5
0	0	1	0	0	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for j ← 1 to length[A]
      do C[A[j]] ← C[A[j]] + 1
```

j = 3
C[3] ← 0 + 1

C:

0	1	2	3	4	5
0	0	1	1	0	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for j ← 1 to length[A]
      do C[A[j]] ← C[A[j]] + 1
```

j = 4
C[0] ← 0 + 1

C:

0	1	2	3	4	5
1	0	1	1	0	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for j ← 1 to length[A]
      do C[A[j]] ← C[A[j]] + 1
```

j = 5
C[2] ← 1 + 1

C:

0	1	2	3	4	5
1	0	2	1	0	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for j ← 1 to length[A]
      do C[A[j]] ← C[A[j]] + 1
```

j = 6
C[3] ← 1 + 1

C:

0	1	2	3	4	5
1	0	2	2	0	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for j ← 1 to length[A]
      do C[A[j]] ← C[A[j]] + 1
```

j = 7
C[0] ← 1 + 1

C:

0	1	2	3	4	5
2	0	2	2	0	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for j ← 1 to length[A]
      do C[A[j]] ← C[A[j]] + 1
```

j = 8
C[0] ← 2 + 1

C:

0	1	2	3	4	5
2	0	2	3	0	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for i ← 1 to k
      do C[i] ← C[i] + C[i - 1]
```

// sum number of
// elements ≤ i

C:

0	1	2	3	4	5
2	0	2	3	0	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for i ← 1 to k
      do C[i] ← C[i] + C[i - 1]
```

i = 1
C[1] ← 0 + 2

C:

0	1	2	3	4	5
2	2	2	3	0	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for i ← 1 to k
      do C[i] ← C[i] + C[i - 1]
```

i = 2
C[2] ← 2 + 2

C:

0	1	2	3	4	5
2	2	4	3	0	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for i ← 1 to k
      do C[i] ← C[i] + C[i - 1]
```

i = 3

C[3] ← 3 + 4

C:

0	1	2	3	4	5
2	2	4	7	0	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for i ← 1 to k
      do C[i] ← C[i] + C[i - 1]
```

i = 4
C[4] ← 0 + 7

C:

0	1	2	3	4	5
2	2	4	7	7	1

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

```
for i ← 1 to k
      do C[i] ← C[i] + C[i - 1]
```

i = 5
C[5] ← 1 + 7

C:

0	1	2	3	4	5
2	2	4	7	7	8

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5
2	2	4	7	7	8

```
for j ← length[A] downto 1
        do B[C[A[j]]] ← A[j]
             C[A[j]] ← C[A[j]] - 1
```

// insert elements
// at final position

B:

1	2	3	4	5	6	7	8

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5
2	2	4	6	7	8

```
for j ← length[A] downto 1
       do B[C[A[j]]] ← A[j]
            C[A[j]] ← C[A[j]] - 1
```

```
j = 8
B[7] ← A[8]
C[3] ← 7 - 1
```

B:

1	2	3	4	5	6	7	8
						3	

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5
1	2	4	6	7	8

```
for j ← length[A] downto 1
      do B[C[A[j]]] ← A[j]
           C[A[j]] ← C[A[j]] - 1
```

```
j = 7
B[2] ← A[7]
C[0] ← 2 - 1
```

B:

1	2	3	4	5	6	7	8
	0					3	

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5
1	2	4	5	7	8

```
for j ← length[A] downto 1
        do B[C[A[j]]] ← A[j]
             C[A[j]] ← C[A[j]] - 1
```

```
j = 6
B[6] ← A[6]
C[3] ← 6 - 1
```

B:

1	2	3	4	5	6	7	8
	0				3	3	

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5
1	2	3	5	7	8

```
for j ← length[A] downto 1
      do B[C[A[j]]] ← A[j]
           C[A[j]] ← C[A[j]] - 1
```

```
j = 5
B[4] ← A[5]
C[2] ← 4 - 1
```

B:

1	2	3	4	5	6	7	8
	0		2		3	3	

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5
0	2	3	5	7	8

```
for j ← length[A] downto 1
        do B[C[A[j]]] ← A[j]
             C[A[j]] ← C[A[j]] - 1
```

```
j = 4
B[1] ← A[4]
C[0] ← 1 - 1
```

B:

1	2	3	4	5	6	7	8
0	0		2		3	3	

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5
0	2	3	4	7	8

```
for j ← length[A] downto 1
       do B[C[A[j]]] ← A[j]
            C[A[j]] ← C[A[j]] - 1
```

```
j = 3
B[5] ← A[3]
C[3] ← 5 - 1
```

B:

1	2	3	4	5	6	7	8
0	0		2	3	3	3	

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5
0	2	3	4	7	7

```
for j ← length[A] downto 1
        do B[C[A[j]]] ← A[j]
             C[A[j]] ← C[A[j]] - 1
```

```
j = 2
B[8] ← A[2]
C[3] ← 8 - 1
```

B:

1	2	3	4	5	6	7	8
0	0		2	3	3	3	5

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5
0	2	2	4	7	7

```
for j ← length[A] downto 1
       do B[C[A[j]]] ← A[j]
          C[A[j]] ← C[A[j]] - 1
```

j = 1
B[3] ← A[1]
C[2] ← 2 - 1

B:

1	2	3	4	5	6	7	8
0	0	2	2	3	3	3	5

Example: Counting Sort

A:

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5
0	2	2	4	7	7

`Sorted`

B:

1	2	3	4	5	6	7	8
0	0	2	2	3	3	3	5

Analysis of Count Sort

```
COUNTING-SORT(A, B, k)
for i← 0 to k
      do C[i] ←0
for j ← 1 to length[A]
      do C[A[j]] ← C[A[j]] + 1
// C[i] contains number of elements equal to
i.
for i ← 1 to k
        do C[i] ← C[i] + C[i - 1]
// C[i] contains number of elements ≤ i.
for j ← length[A] downto 1
          do B[C[A[j]]] ← A[j]
               C[A[j]] ← C[A[j]] - 1
```

Loops 1 and 3 takes $\Theta(k)$ time → [Loop 1], [Loop 3]

Loops 2 and 4 takes $\Theta(n)$ time → [Loop 2], [Loop 4]

Total cost is $\Theta(k+n)$. If $k = O(n)$, then total cost is $\Theta(n)$.

Stable Sorting

- Counting sort is called a stable sort.
  - The same values appear in the output array in the same order as they do in the input array.

Bucket Sort

- Assumption: uniform distribution
  - Input numbers are *uniformly distributed* in `[0,1)`.
  - Suppose input size is n.
- Idea:
  - Divide `[0,1)` into n equal-sized buckets.
  - Distribute n numbers into buckets
  - Expect that each bucket contains a few numbers.
  - Sort numbers in each bucket
    - usually, insertion sort as default
  - Then go through buckets in order, listing elements.

Bucket Sort Algorithm

```
BUCKET-SORT(A)
n ← A.length
for i ←1 to n
    do insert A[i] into bucket B[⌊n*A[i]⌋]
for i ←0 to n-1
     do sort bucket B[i] using insertion sort
Concatenate bucket B[0],B[1],…,B[n-1]
```

Example of Bucket Sort

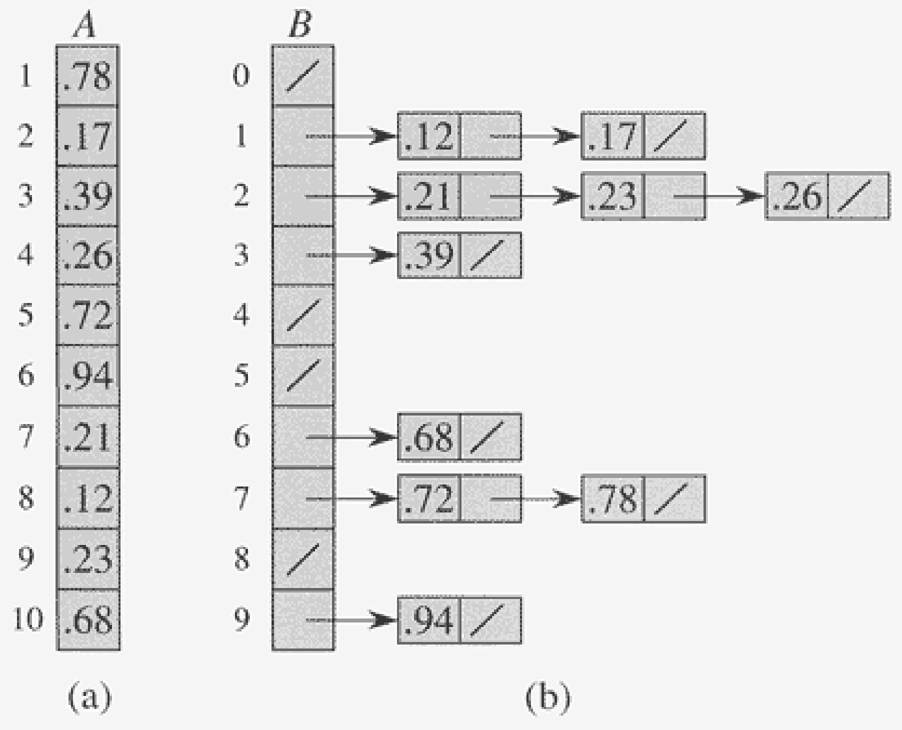


Figure 8.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Analysis of Bucket Sort

BUCKET-SORT(A)

```
n ←length[A]                                      Ω(1)
for i ←1 to n                                     O(n)
    do insert A[i] into bucket B[⌊nA[i]⌋]
for i ←0 to n-1                                   O(n)
    do sort bucket B[i] with insertion sort O(n_i^2)
Concatenate bucket B[0],B[1],…,B[n-1]             O(n)
```

Where n_i is the size of bucket B[*i*].

Thus, $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$

$= \Theta(n) + n*O(2-1/n) = \mathbf{\Theta(n)}$

Radix Sort

- Radix sort is a non-comparative sorting algorithm
  - Sorts data with integer keys with `d` digits
  - Sort *least* significant digits first, then sort the 2nd one, then the 3rd one, etc., up to `d` digits
- Radix sort dates back as far as 1887
  - Herman Hollerith used technique in tabulating machines
  - The1880 U.S. census took 7 years to complete
  - With Hollerith's "tabulating machines," the 1890 census took the Census Bureau six weeks

Radix Algorithm

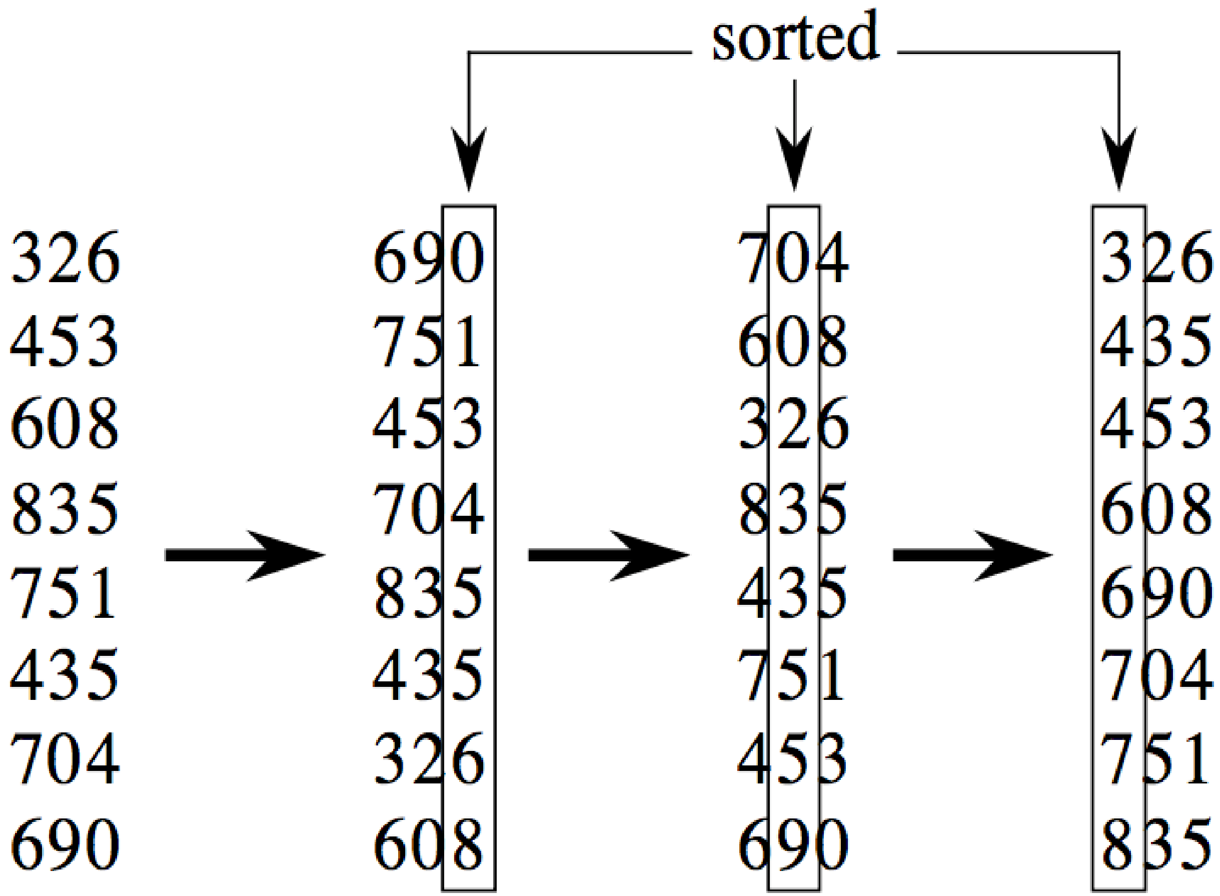
Given an array of integers `A` with up to `d` digits:

RADIX-SORT(A, d)

for $i \leftarrow 1$ **to** d

  do use a stable sort to sort array A on digit i

Example: Radix Sort



Input array: 3-digit elements

Sort the least significant digit

Sort the next significant digit

Sort the last digit

Stable Sort

What happens if we use a non-stable sorting algorithm?

```
213          321                312          123
312          312                213 <-       132
123          212                212 <-       213 <-
212  stable  132  not stable    321  stable  212 <-
321  ------> 213  ----------->  123  ------> 312
132          123                132          321
  ^           ^                 ^
```

Radix Sort and Stability

- Radix sort works as use a **stable** sort at each stage
  - Stability is a property of sorts:
  - A sort is stable if it guarantees the relative order of equal items stays the same
  - Given these numbers:

    [7_1, 6, 7_2, 5, 1, 2, 7_3, -5] (subscripts added for clarity)

  - A stable sort would be:

    [-5, 1, 2, 5, 6, 7_1, 7_2, 7_3]

Are Other Sorting Algorithms Stable?

- Counting sort?
  – Stable
- Insertion sort?
  – Stable
- Heapsort?
  – Not Stable - example input: $[5_1\ 5_2\ 5_3\ 3\ 4]$
    output: $[3\ 4\ 5_3\ 5_2\ 5_1]$
- Selection sort?
  – Not Stable - example input: $[5_1\ 5_2\ 5_3\ 3\ 4]$
    output: $[3\ 4\ 5_3\ 5_1\ 5_2]$
- Quicksort?
  – Not Stable - example input: $[5_1\ 5_2\ 5_3\ 3\ 4]$
    output: $[3\ 4\ 5_3\ 5_1\ 5_2]$

Radix Sort for Non-Integers

- Suppose a group of people, with last name, middle, and first name.
- Sort it by the last name, then by middle, finally by the first name
- Then after every pass of sort, the bins can be combined as one file and proceed to the next sort.

Analysis of Radix Sort

- Given all n numbers in the input array have d or fewer digits
- Suppose we use Counting Sort to sort each digit
- The running time for Radix Sort would be:

$$d * \Theta(n + k) = \Theta(dn + dk)$$

- If $k = O(n)$ and d is a constant, the running time is $\Theta(n)$.

Exercise

- How can we sort n integers in the range 0 to n^2-1 in time $\Theta(n)$?

Comparison of Various Sorts

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)