Sorting in O(N) time

CS302

Data Structures

Section 10.4

How Fast Can We Sort?

- Selection Sort, Bubble Sort, Insertion Sort: $O(n^2)$
- Heap Sort, Merge sort: $O(n\lg n)$
- Quicksort: $O(n\lg n)$ - *average*
- What is common to all these algorithms?
  - Make **comparisons** between input elements

$$a_i < a_j, \quad a_i \leq a_j, \quad a_i = a_j, \quad a_i \geq a_j, \quad \text{or} \quad a_i > a_j$$

Lower-Bound for Sorting

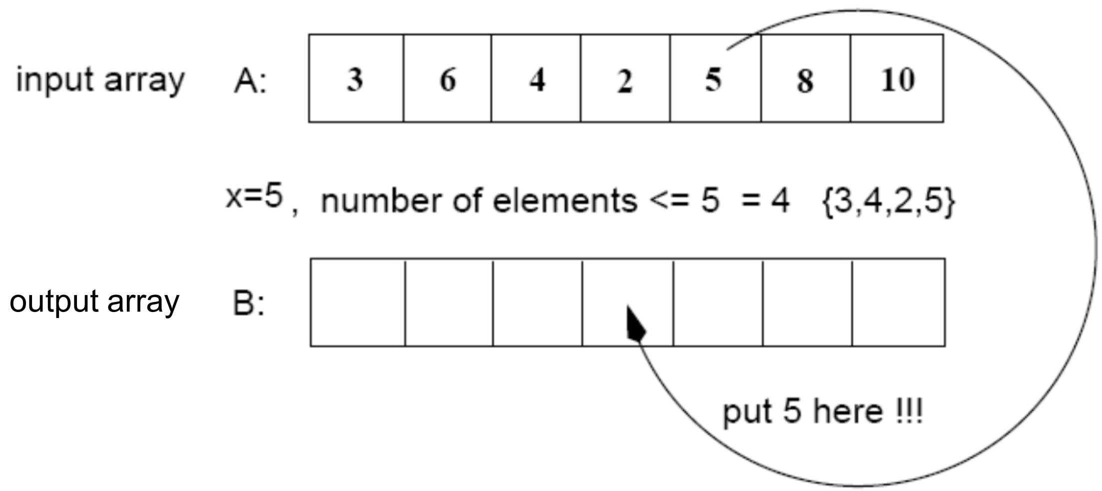
- Theorem: To sort n elements, comparison sorts **must** make $\Omega(n\lg n)$ comparisons in the worst case.

Can we do better?

- Linear sorting algorithms
  - Counting Sort
  - Radix Sort
  - Bucket sort
- Make certain assumptions about the data
- Linear sorts are NOT "comparison sorts"

Counting Sort

- Assumptions:
  - n integers which are in the range $[0 \ldots r]$
  - r is in the order of n, that is, $r=O(n)$
- Idea:
  - For each element x, find the number of elements $\leq x$
  - Place x into its correct position in the output array



Step 1

Find the number of times $A[i]$ appears in A

(i.e., frequencies)

input array A:

3	6	4	1	3	4	1	4

allocate C

1	2	3	4	5	6
0	0	0	0	0	0

Allocate $C[1..r]$ (r=6)

i=1, A[1]=3

1	2	3	4	5	6
0	0	1	0	0	0

C[A[1]]=C[3]=1

For $1 \leq i \leq n, ++C[A[i]];$

i=2, A[2]=6

1	2	3	4	5	6
0	0	1	0	0	1

C[A[2]]=C[6]=1

i=3, A[3]=4

1	2	3	4	5	6
0	0	1	1	0	1

C[A[3]]=C[4]=1

⋮

i=8, A[8]=4

1	2	3	4	5	6
2	0	2	3	0	1

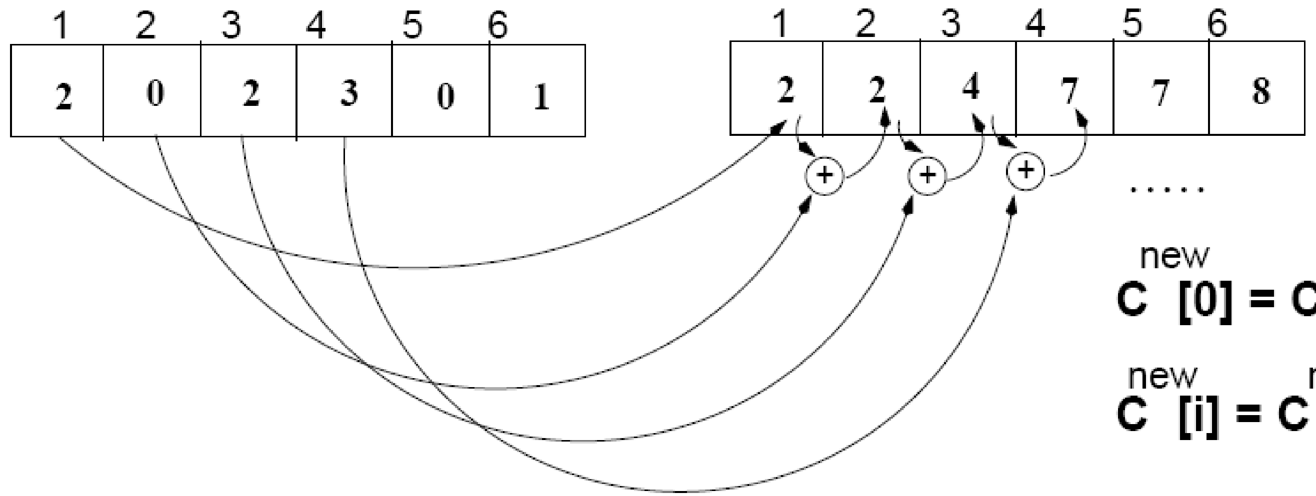
C[A[8]]=C[4]=3

C[i] = number of times element i appears in A

Step 2

Find the number of elements $\leq A[i]$,

C (frequencies) C^{new} (cumulative sums)



$$C^{new}[0] = C^{old}[0]$$

$$C^{new}[i] = C^{new}[i-1] + C^{old}[i]$$

C[i] = # elements <= i

Algorithm

- Start from the last element of A
- Place A[i] at its correct place in the output array
- Decrease C[A[i]] by one

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

	0	1	2	3	4	5
C^{new}	2	2	4	7	7	8

Example

A

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

C^{new}

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

B

1	2	3	4	5	6	7	8
						3	

C^{new}

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

B

1	2	3	4	5	6	7	8
	0					3	

C^{new}

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

B

1	2	3	4	5	6	7	8
	0				3	3	

C^{new}

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

B

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

C^{new}

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

Example (cont.)

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

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

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

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

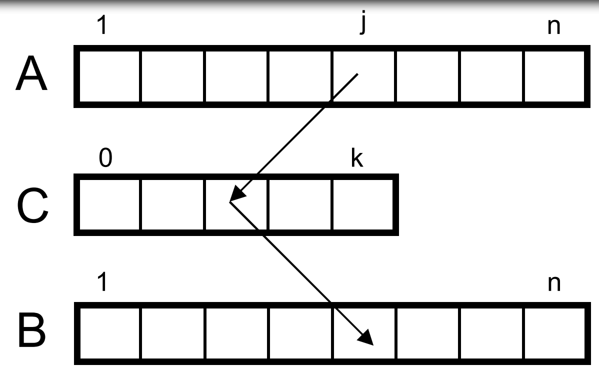
	0	1	2	3	4	5
C	0	2	3	4	7	7

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

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

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

COUNTING-SORT



Alg.: COUNTING-SORT(A, B, n, k)

```
for i ← 0 to r
    do C[ i ] ← 0
for j ← 1 to n
    do C[A[ j ]] ← C[A[ j ]] + 1
▷C[i] contains the number of elements equal to i
for i ← 1 to r
    do C[ i ] ← C[ i ] + C[i -1]
▷C[i] contains the number of elements ≤ i
for j ← n downto 1
    do B[C[A[ j ]]] ← A[ j ]
        C[A[ j ]] ← C[A[ j ]] - 1
```

Analysis of Counting Sort

Alg.: COUNTING-SORT(A, B, n, k)

```
for i ← 0 to r                                   } O(r)
    do C[ i ] ← 0                                }
for j ← 1 to n                                   } O(n)
    do C[A[ j ]] ← C[A[ j ]] + 1                 }
▷C[i] contains the number of elements equal to i
for i ← 1 to r                                   } O(r)
    do C[ i ] ← C[ i ] + C[i -1]                 }
▷C[i] contains the number of elements ≤ i
for j ← n downto 1                               } O(n)
    do B[C[A[ j ]]] ← A[ j ]                     }
       C[A[ j ]] ← C[A[ j ]] - 1                 }
```

Overall time: $O(n + r)$

Analysis of Counting Sort

- Overall time: $O(n + r)$

- In practice we use COUNTING sort when $r = O(n)$

  $\Rightarrow$ running time is $O(n)$

Radix Sort

- Represents keys as d-digit numbers in some base-k

$$key = x_1 x_2 \ldots x_d \quad \text{where } 0 \le x_i \le k-1$$

- Example: key=15

$$key_{10} = 15,\ d=2,\ k=10 \quad \text{where } 0 \le x_i \le 9$$

$$key_2 = 1111,\ d=4,\ k=2 \quad \text{where } 0 \le x_i \le 1$$

Radix Sort

- Assumptions

  d=O(1)   and k =O(n)

- Sorting looks at one column at a time
  - For a d digit number, sort the least significant digit first
  - Continue sorting on the next least significant digit, until all digits have been sorted
  - Requires only d passes through the list

326
453
608
835
751
435
704
690

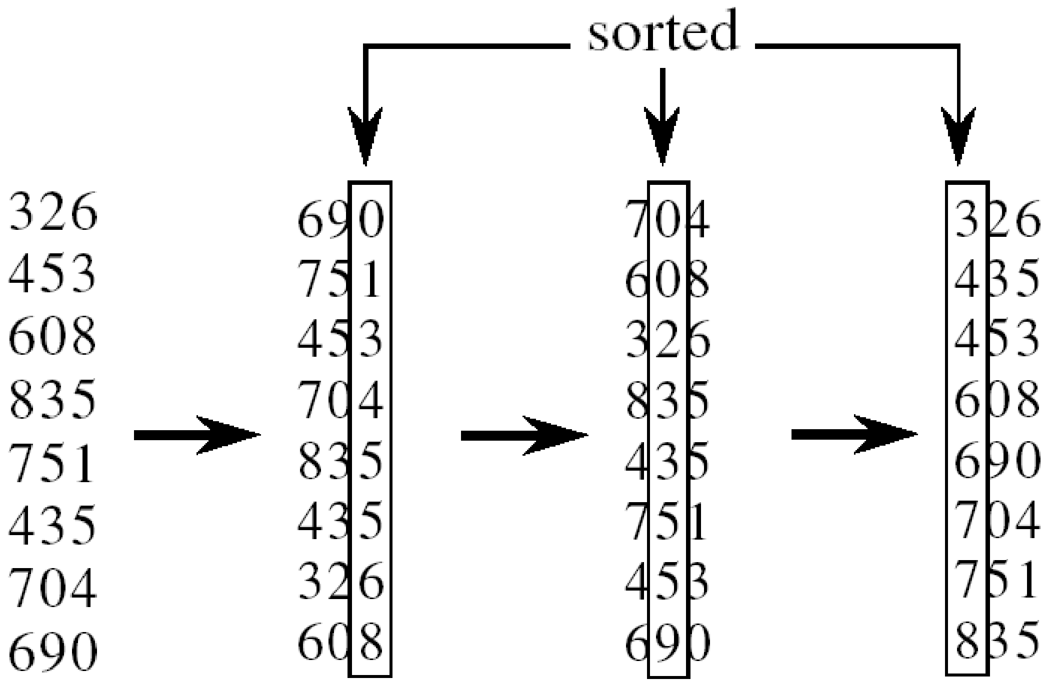
RADIX-SORT

Alg.: RADIX-SORT(A, d)

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

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

(stable sort: preserves order of identical elements)



Analysis of Radix Sort

- Given n numbers of d digits each, where each digit may take up to k possible values, RADIX-SORT correctly sorts the numbers in $O(d(n+k))$
  - One pass of sorting per digit takes $O(n+k)$ assuming that we use **counting sort**
  - There are d passes (for each digit)

Analysis of Radix Sort

- Given n numbers of d digits each, where each digit may take up to k possible values, RADIX-SORT correctly sorts the numbers in $O(d(n+k))$
  - Assuming d=O(1) and k=O(n), running time is O(n)

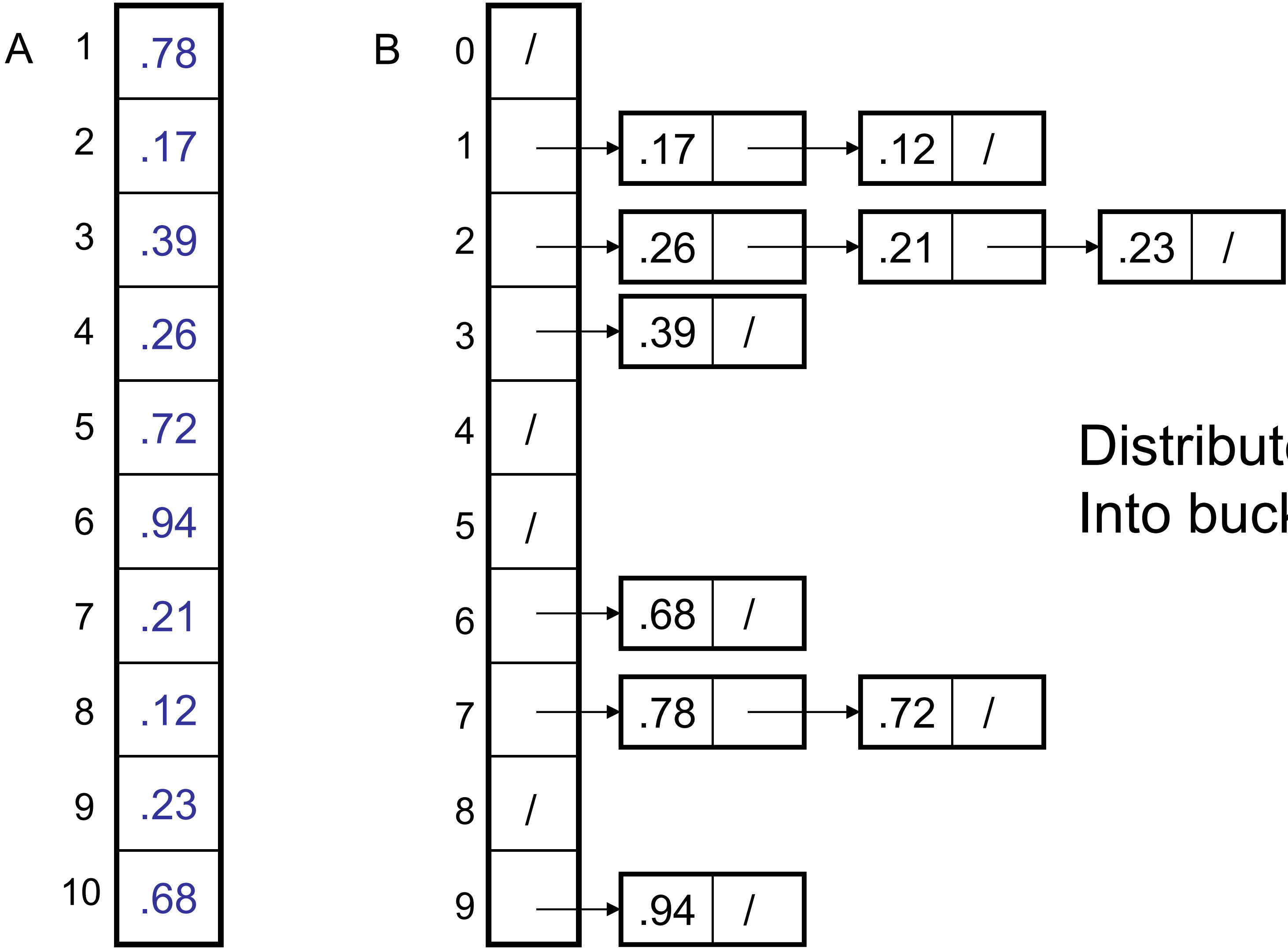
Bucket Sort

- Assumption:
  - the input is generated by a random process that distributes elements uniformly over [0, 1)
- Idea:
  - Divide [0, 1) into k equal-sized buckets ($k=\Theta(n)$)
  - Distribute the n input values into the buckets
  - Sort each bucket (e.g., using quicksort)
  - Go through the buckets in order, listing elements in each one

- **Input:** $A[1 \,..\, n]$, where $0 \le A[i] < 1$ for all i

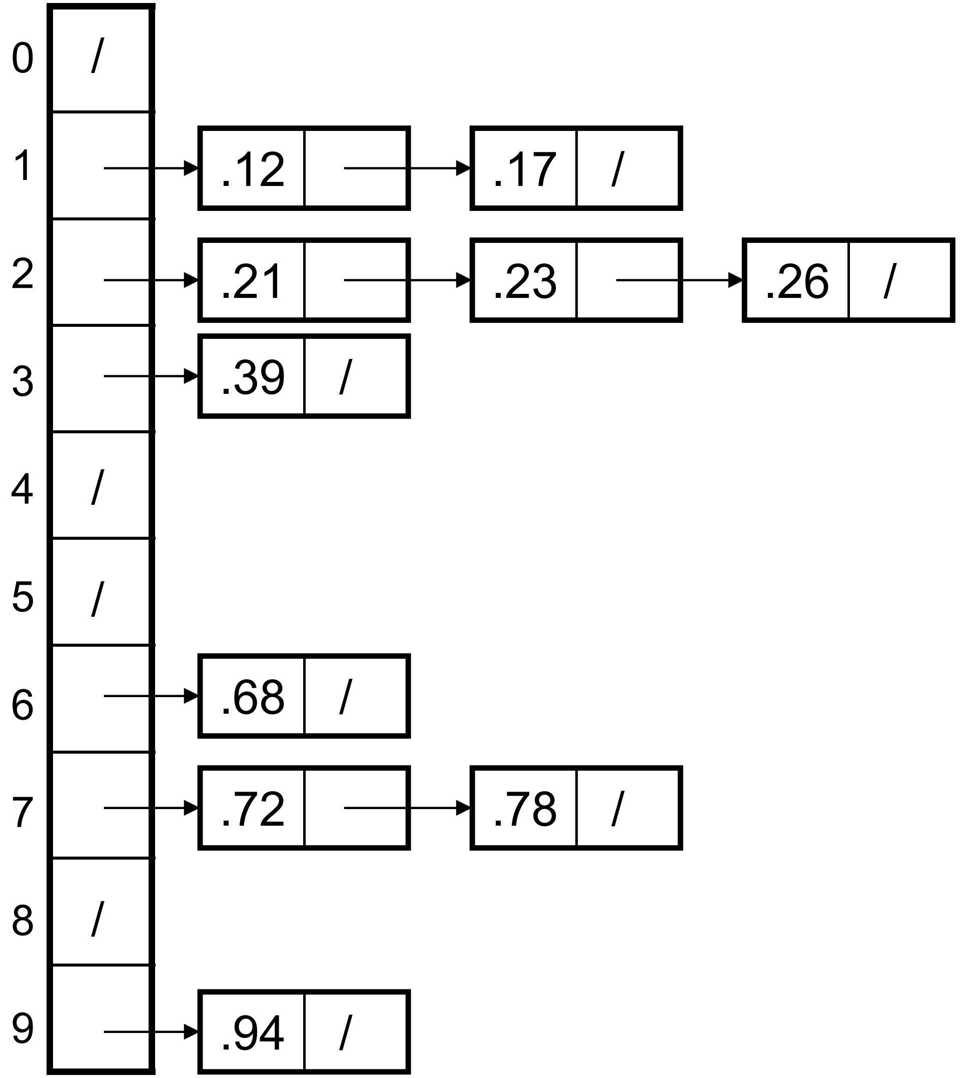
- **Output:** elements $A[i]$ sorted

Example - Bucket Sort



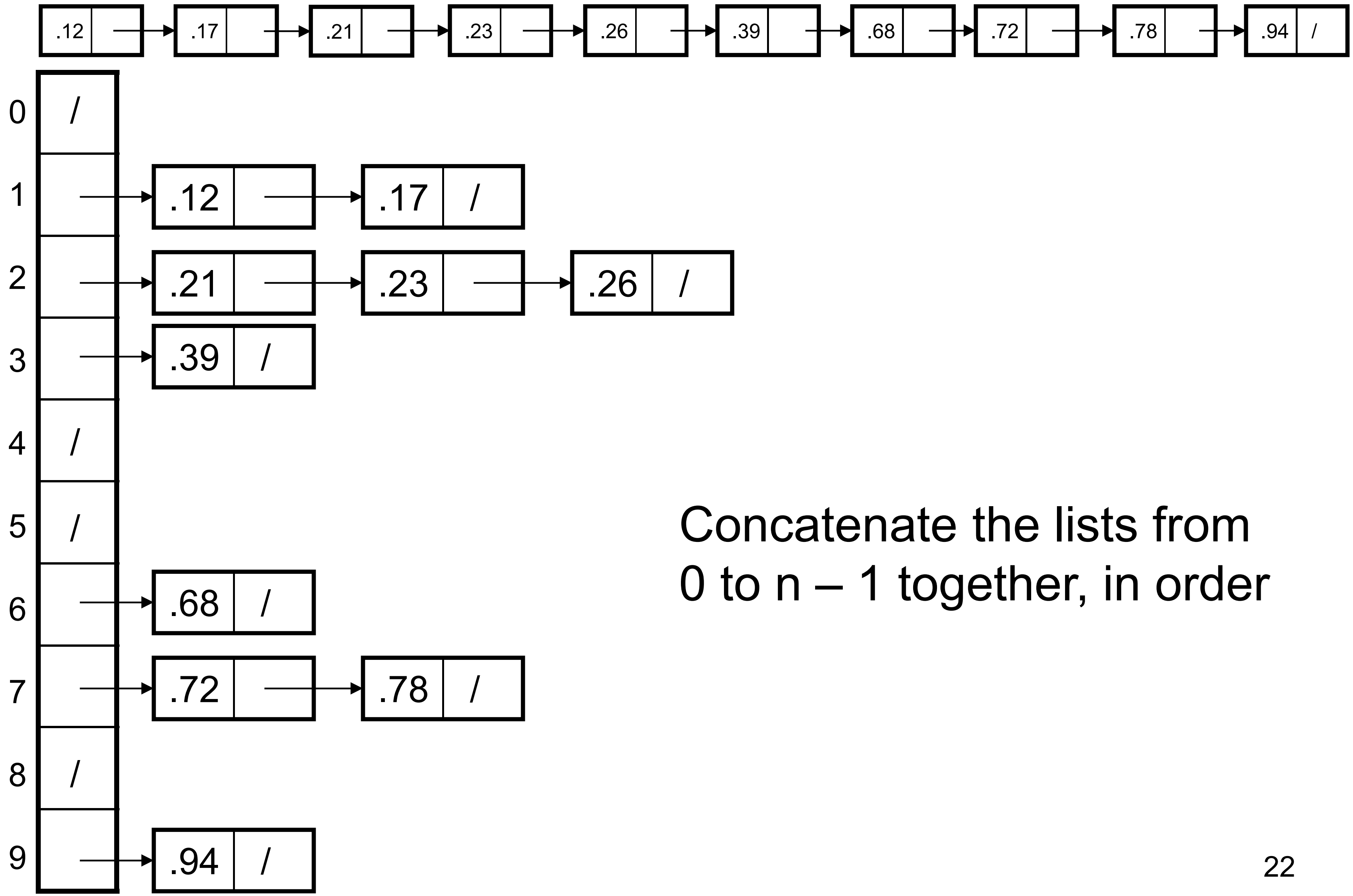
Distribute
Into buckets

Example - Bucket Sort



Sort within each bucket

Example - Bucket Sort



Concatenate the lists from 0 to n – 1 together, in order

Analysis of Bucket Sort

Alg.: BUCKET-SORT(A, n)

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

do insert A[i] into list $B[\lfloor nA[i] \rfloor]$

for $i \leftarrow 0$ **to** $k - 1$

do sort list $B[i]$ with quicksort sort

concatenate lists $B[0], B[1], \ldots, B[n-1]$

together in order

return the concatenated lists

O(n)

k O(n/k log(n/k))
=O(nlog(n/k)

O(k)

O(n) (if k=Θ(n))

Radix Sort as a Bucket Sort

- **Counting sort**: for integers with limited range
- **Radix sort**: when log(N) is significantly larger than the number of digits K
- **Bucket sort**: when the input is approximately uniformly distributed