Learn Bhagawad Gita: From B K Sharma

It Is Better To Carry Out One's Own Duties A Little Imperfectly.

Rather Than To Faultlessly Perform Another's Duties.

-Bhagawad Gita

TCS-503: Design and Analysis of Algorithms

Linear-Time Sorting Algorithms

Unit I: Syllabus

- **Introduction:**
  - Algorithms
  - Analysis of Algorithms
  - Growth of Functions
  - Master's Theorem
  - Designing of Algorithms

Unit I: Syllabus

- Sorting and Order Statistics
  - Heap Sort
  - Quick Sort
  - Sorting in Linear Time
    - Counting Sort
    - Bucket Sort
    - Radix Sort
  - Medians and Order Statistics

Sorting So Far

Insertion sort:

- Easy to code
- Fast on small inputs (less than ~50 elements)
- Fast on nearly-sorted inputs
- $O(n^2)$ worst case
- $O(n^2)$ average (equally-likely inputs) case
- $O(n^2)$ reverse-sorted case

Merge sort:

- Divide-and-conquer:
  - Split array in half
  - Recursively sort subarrays
  - Linear-time merge step
- $O(n \lg n)$ worst case

Sorting So Far

Heap sort:

- Uses the very useful heap data structure
  - Complete binary tree
  - Heap property: parent key ≥ children's keys
  - $O(n \lg n)$ worst case
  - Sorts in place
  - Fair amount of shuffling memory around

Quick sort:

- Divide-and-conquer:
  - Partition array into two subarrays, recursively sort
  - All of first subarray ≤ all of second subarray
  - No merge step needed!
- $O(n \lg n)$ average case $O(n^2)$ worst case

What is common to all these algorithms?

These algorithms sort by making comparisons between the input elements.

Comparison sorts use **comparisons between elements** to **gain information** about an **input sequence** $\langle a_1, a_2, \ldots, a_n \rangle$

Perform tests:

$a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, or $a_i > a_j$

to **determine the relative order** of a_i and a_j

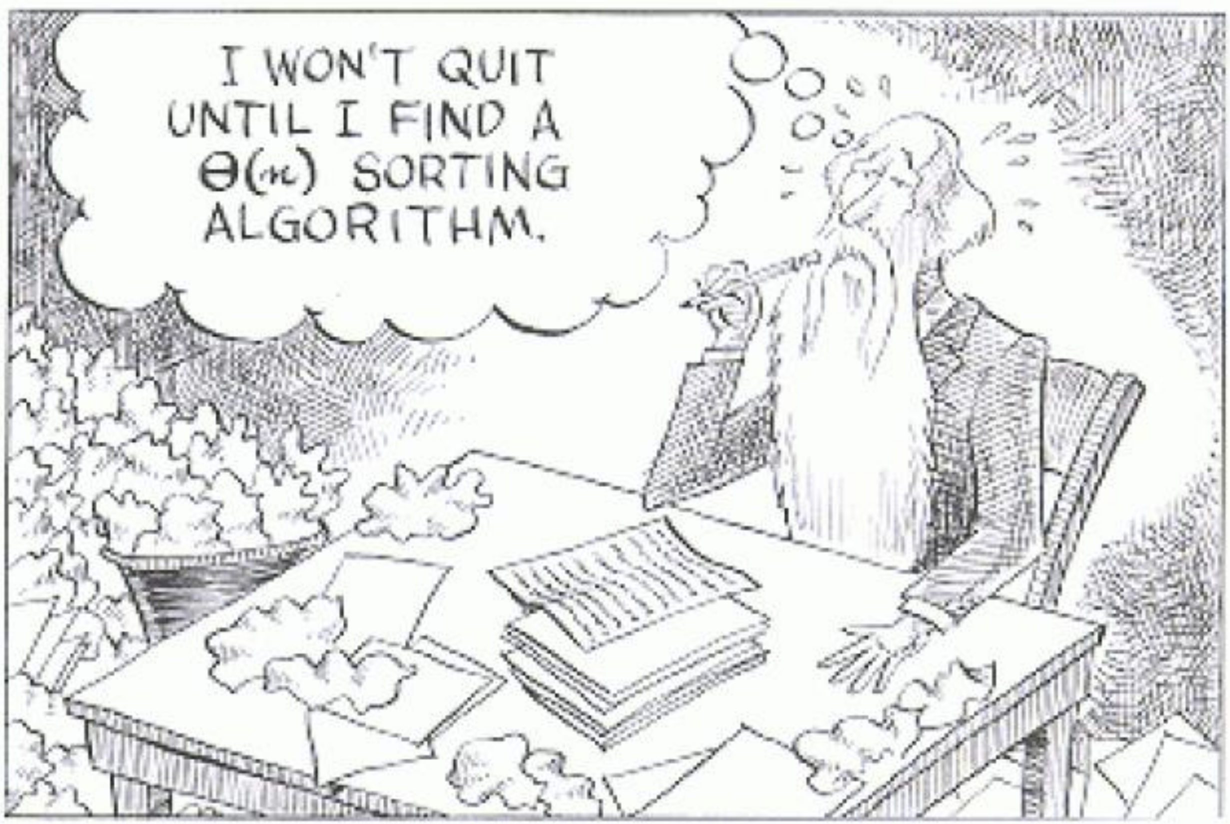
Lower Bound: least time complexity

The time to comparison sort n elements is $\Omega(n \lg n)$

How can we do better than $\Omega(n \lg n)$?

Learn DAA: From B K Sharma

How Fast Can We Sort?



Linear Time Sorting Algorithms: Sorting in Linear Time

Counting Sort
Bucket Sort
Radix Sort

All Three Algorithms

Make some assumptions
About the inputs

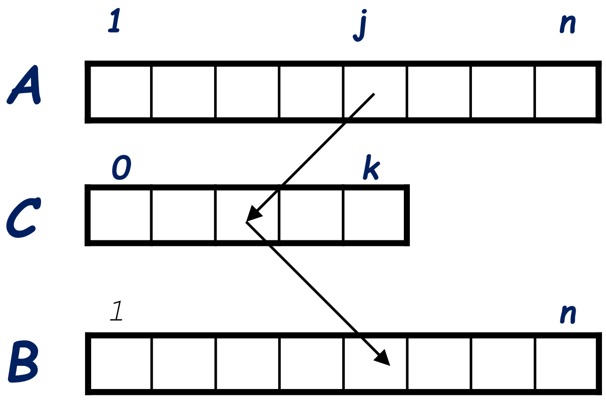
Sorting in Linear Time Algorithms

Counting Sort

Assumptions:

Sort n integers which are in the range [0 ... k]

k is in the order of n, that is, k=O(n)



Learn DAA: From B K Sharma

Sorting in Linear Time Algorithms

Counting Sort

Input Array:

A

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

Auxiliary Array C: Initialized to 0

C

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

Put in C[i] no. of elements equal to x

C

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

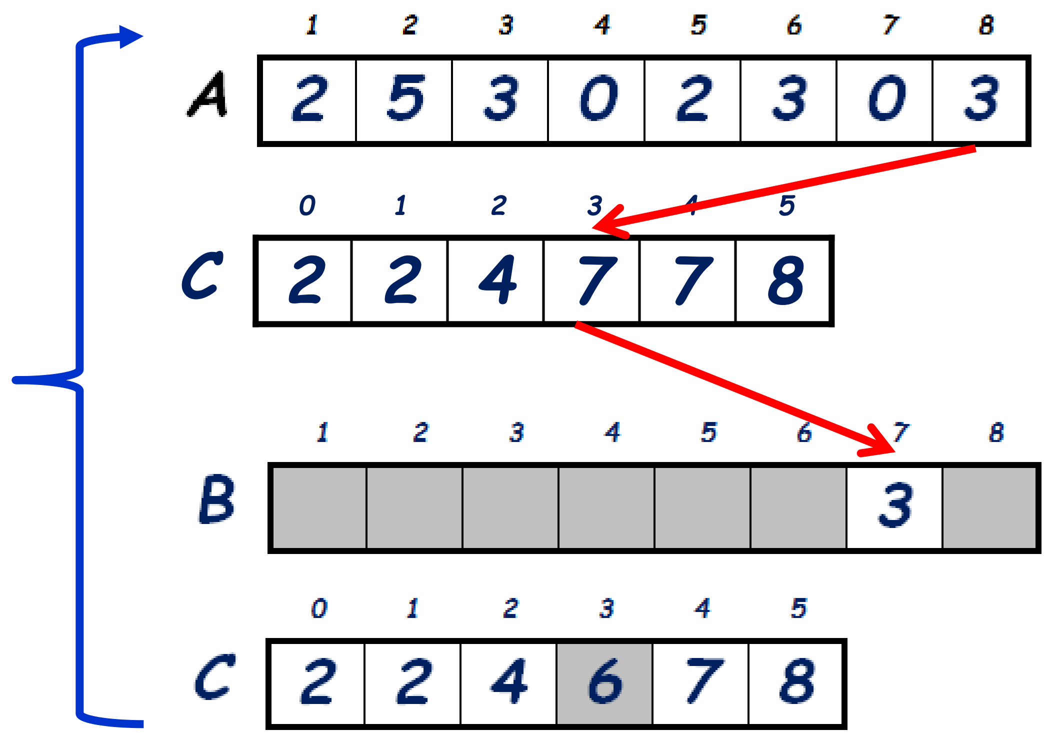
Put in C[i] no. of elements less than or equal to x

C

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

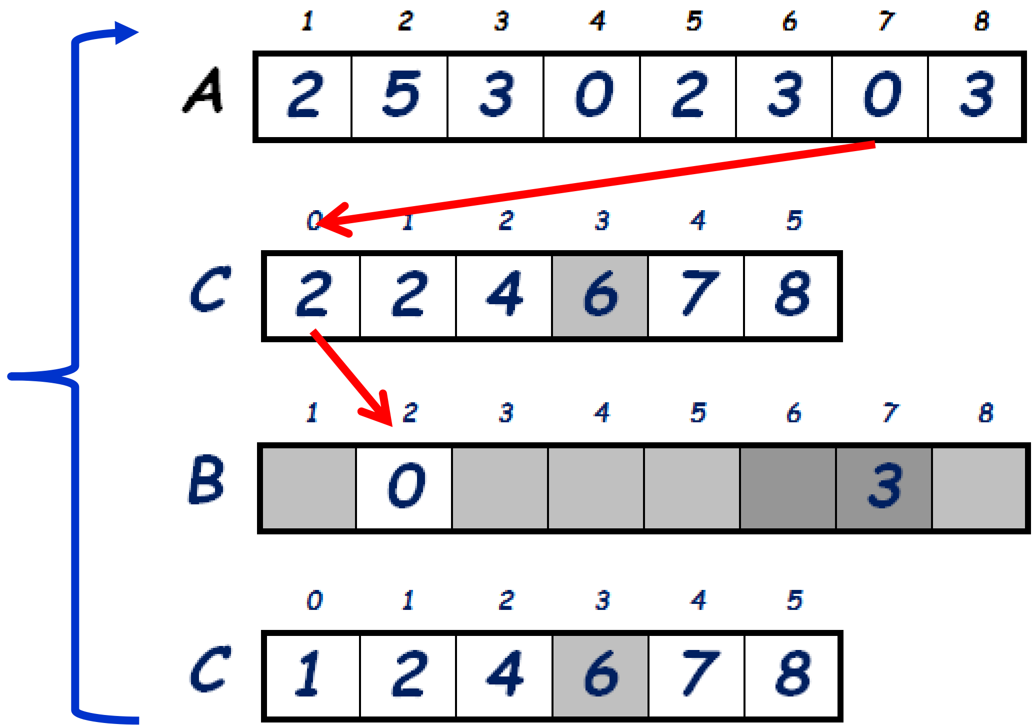
Sorting in Linear Time Algorithms

Counting Sort



Sorting in Linear Time Algorithms

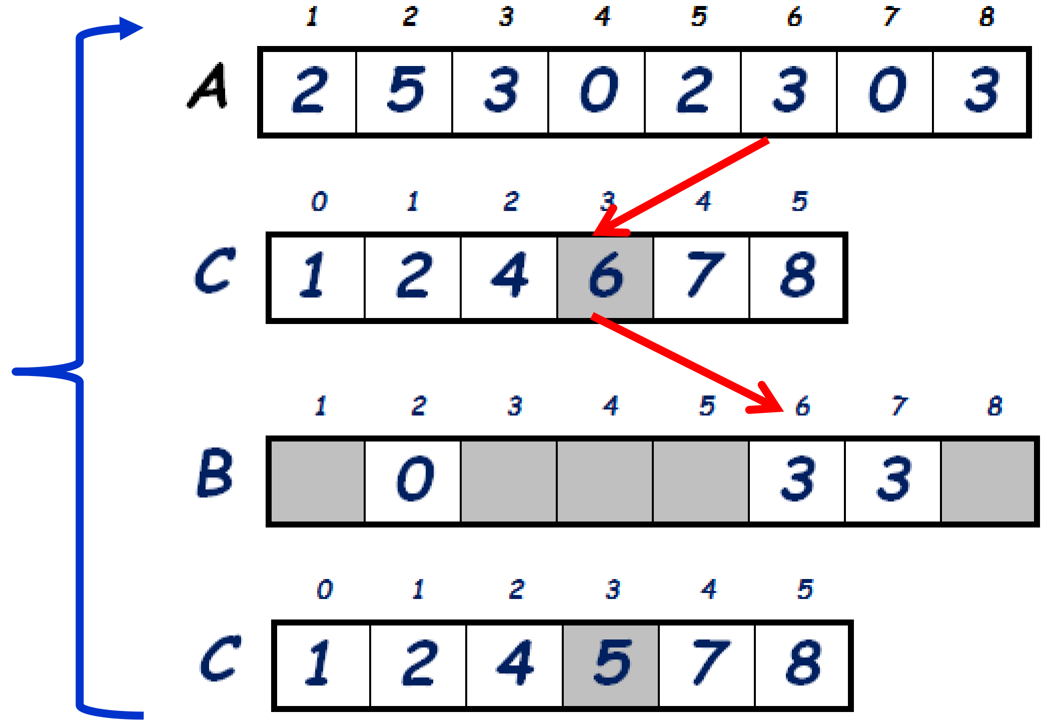
Counting Sort



Learn DAA: From B K Sharma

Sorting in Linear Time Algorithms

Counting Sort



Learn DAA: From B K Sharma

Sorting in Linear Time Algorithms

Counting Sort

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

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

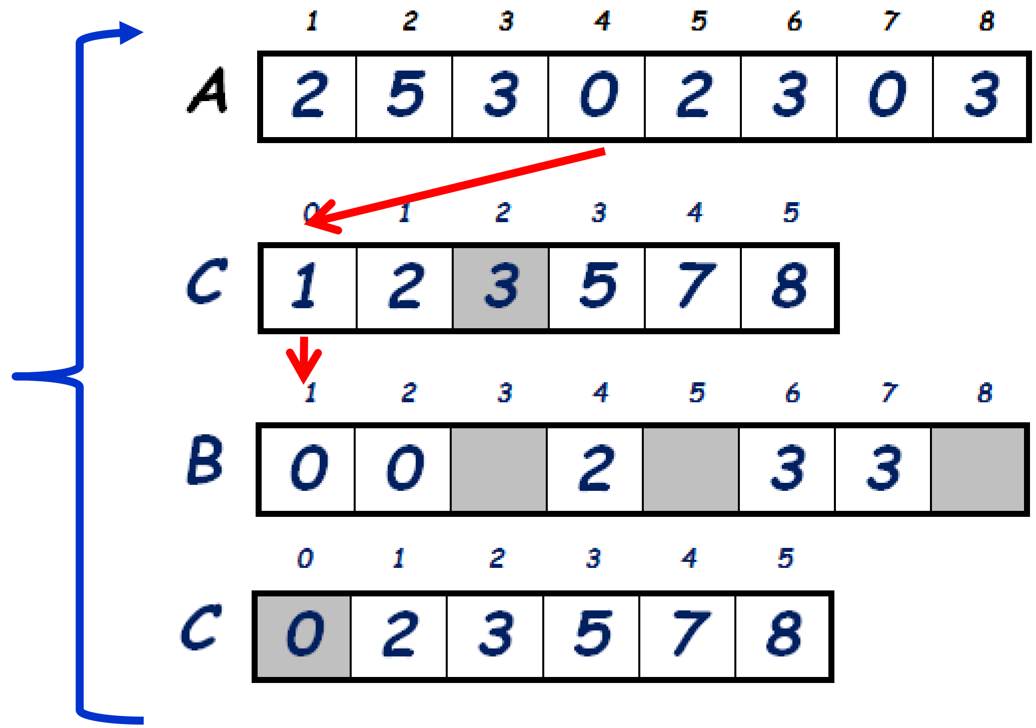
	1	2	3	4	5	6	7	8
B		0		2		3	3	

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

Learn DAA: From B K Sharma

Sorting in Linear Time Algorithms

Counting Sort



Learn DAA: From B K Sharma

Sorting in Linear Time Algorithms

Counting Sort

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

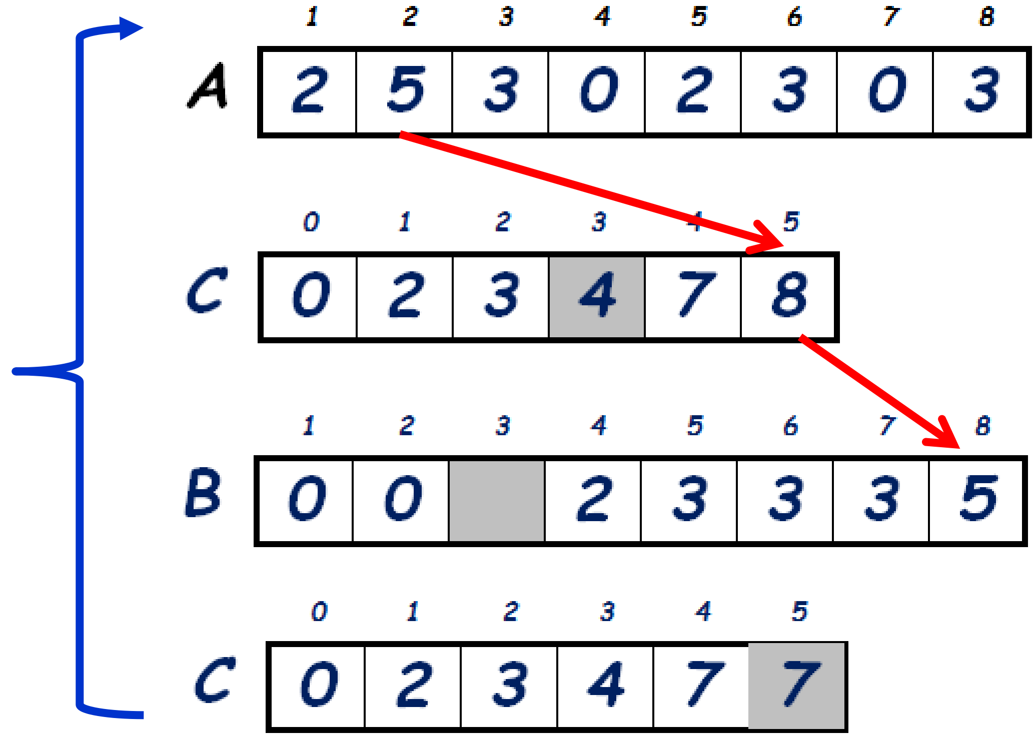
	0	1	2	3	4	5
C	0	2	3	5	7	8

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

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

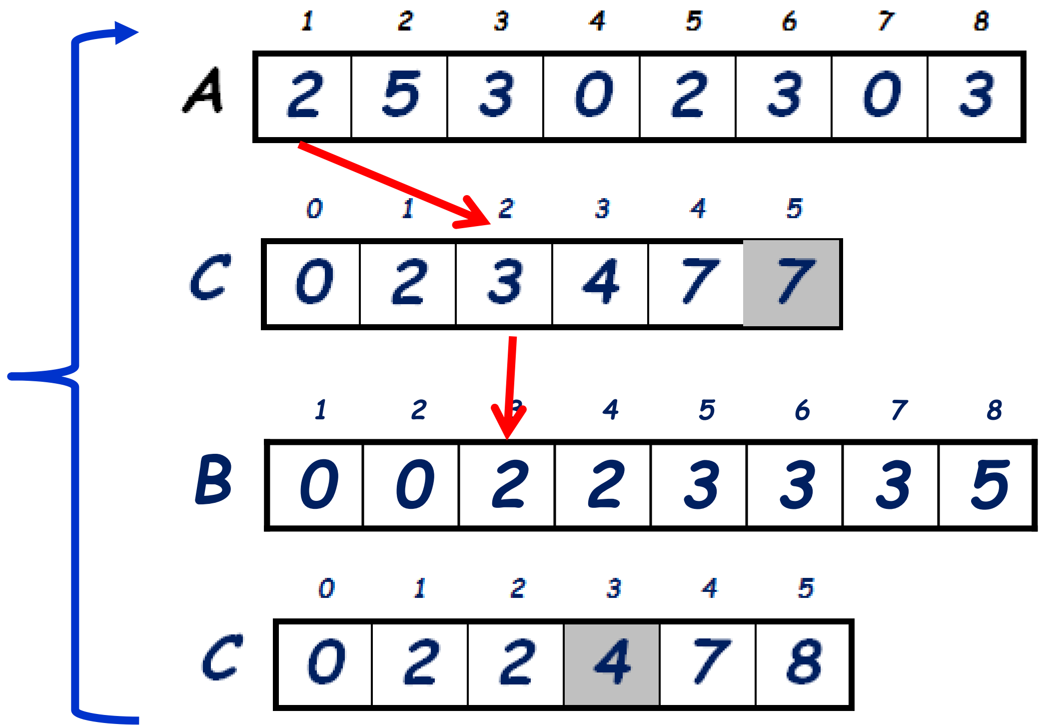
Sorting in Linear Time Algorithms

Counting Sort



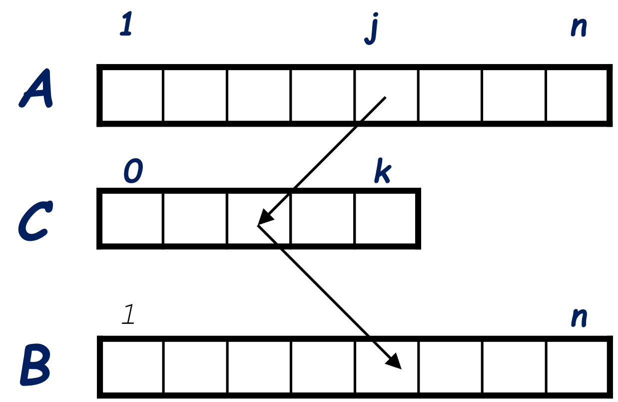
Sorting in Linear Time Algorithms

Counting Sort



Sorting in Linear Time Algorithms

Counting 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] now contains no. of elements equal to i

For i ← 1 to k do

  C[i] ←C[i] + C[i-1]

  ▷ C[i] now contains no. of elements less than or equal to i

For ← length[A] down to 1 do

  B[C[A[j]]] ← A[j]

  C[A[j]] ← C[A[j]] -1

Sorting in Linear Time Algorithms

Counting Sort

Step 1

for i=0 to k
C[i]=0

Input Array 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	0	0	0	0	0

Learn DAA: From B K Sharma

Sorting in Linear Time Algorithms

Counting Sort

Step 2

Find No. of Elements equal to x

For j=1 to n

$C[A[j]]=C[A[j]]+1$

A

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

j=1, A[1] =2 C

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

$C[A[1]]=C[2]=C[2]+1=0+1=1$

j=2, A[2] =5 C

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

$C[A[2]]=C[5]=C[5]+1=0+1=1$

C

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

Sorting in Linear Time Algorithms

Counting Sort

Step 2

At the end of loop, we have

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

Learn DAA: From B K Sharma

Sorting in Linear Time Algorithms

Counting Sort

Step 3

Find No. of Elements less than or equal to x

For i=1 to k

$$C[i]=C[i] + C[i-1]$$

A

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

i=1, C[1] =0

C

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

C[1]=C[1] + C[1-1] =C[1] + C[0]=2

i=2, C[2] =2

C

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

C[2]=C[2] + C[2-1] =C[2] + C[1]=4

C

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

Sorting in Linear Time Algorithms

Counting Sort

Step 3

At the end of loop, we have

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

Learn DAA: From B K Sharma

Sorting in Linear Time Algorithms

Counting Sort

Step 4

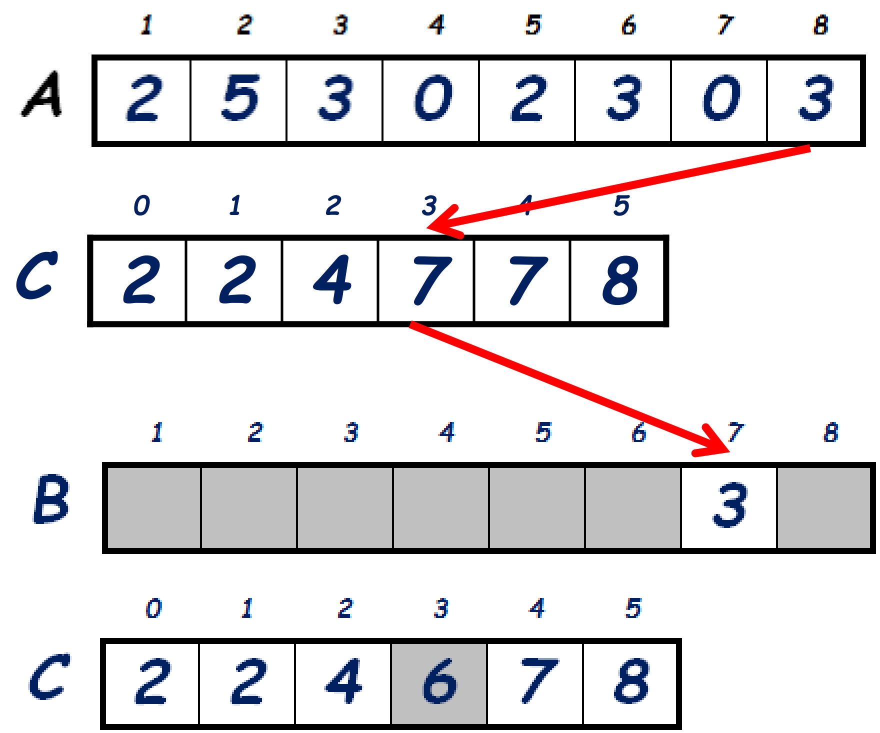
For j=n to 1
    B[C[A[j]]=A[j]
    C[A[j]]= C[A[j]] -1

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



j=8, A[8]=3, B[C[A[8]]]=B[C[3]]=B[7]=3

C[A[8]]=C[3]=C[3] -1 =7-1 =6

Learn DAA: From B K Sharma

Sorting in Linear Time Algorithms

Counting Sort

Step 4

At the end of loop, we have

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

	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	2	3	3	3	5

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

Learn DAA: From B K Sharma

Sorting in Linear Time Algorithms

Analysis of Counting Sort

COUNTING - SORT (A, B, k)

For i←0 to k do
  C[i] ← 0

$\Theta(k)$

For j ←1 to length[A] do
  C[A[j]] ← C[A[j]] + 1

$\Theta(n)$

For i ← 1 to k do
  C[i] ←C[i] + C[i-1]

$\Theta(k)$

For ← length[A] down to 1 do
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] -1

$\Theta(n)$

Overall time: $\Theta(2n + 2k)$

$T(n) = \Theta(n + k)$

Assumptions:

$k = O(n)$

$T(n) = \Theta(n + n)$

$T(n) = \Theta(2n)$

$T(n) = \Theta(n)$

Counting sort is stable

Counting sort is **not** in place sort

Learn DAA: From B K Sharma

Sorting in Linear Time Algorithms

Analysis of Counting Sort

Be Cool! And Answer the following Answer the following questions?

Question: *Why don't we always use counting sort?*

Answer: *Because it depends on range k of elements.*

Question: *Could we use counting sort to sort 32 bit integers? Why or why not?*

Answer: Number , k too large (2^{32} = 4,294,967,296)

Radix Sort

Assumptions:

All numbers are d-digit numbers.

e.g. $d=\Theta(1)$

k is in the order of n, that is, $k=O(n)$

Where k is the base.

Where each digit may take k possible values

326
453
608
835
751
435
704
690

Radix Sort

Sorting looks at one column at a time

- For a d digit number, sort the <u>least significant</u> digit first.
- Continue sorting on the <u>next least significant</u> digit, until all digits have been sorted.

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Requires only d passes through the list.

RADIX-SORT(A, d)

for i←1 to d do

Use a stable sorting algorithm to sort array A on digit i

Analysis of Radix Sort

```
RADIX-SORT(A, d)
  for i←1 to d do
    Use a stable sorting algorithm to sort array A on digit i
```

Assume that we use counting sort.

Each pass over n numbers with d digits takes time: $\Theta(n+k)$

There are d passes (for each digit)

Given n numbers of digits, where each digit may take k possible values, RADIX-SORT correctly sorts the numbers in $\Theta(d(n+k))$

d is constant, $d=\Theta(1)$, $T(n) = \Theta(n+k))$

$k=O(n)$, $k \le n$, $T(n) = \Theta(n+n))=\Theta(2n)=\Theta(n)$

326
453
608
835
751
435
704
690

Bucket Sort

Assumptions:

The input is generated by a random process that distributes elements uniformly over [0, 1) means ≥ 0 and < 1

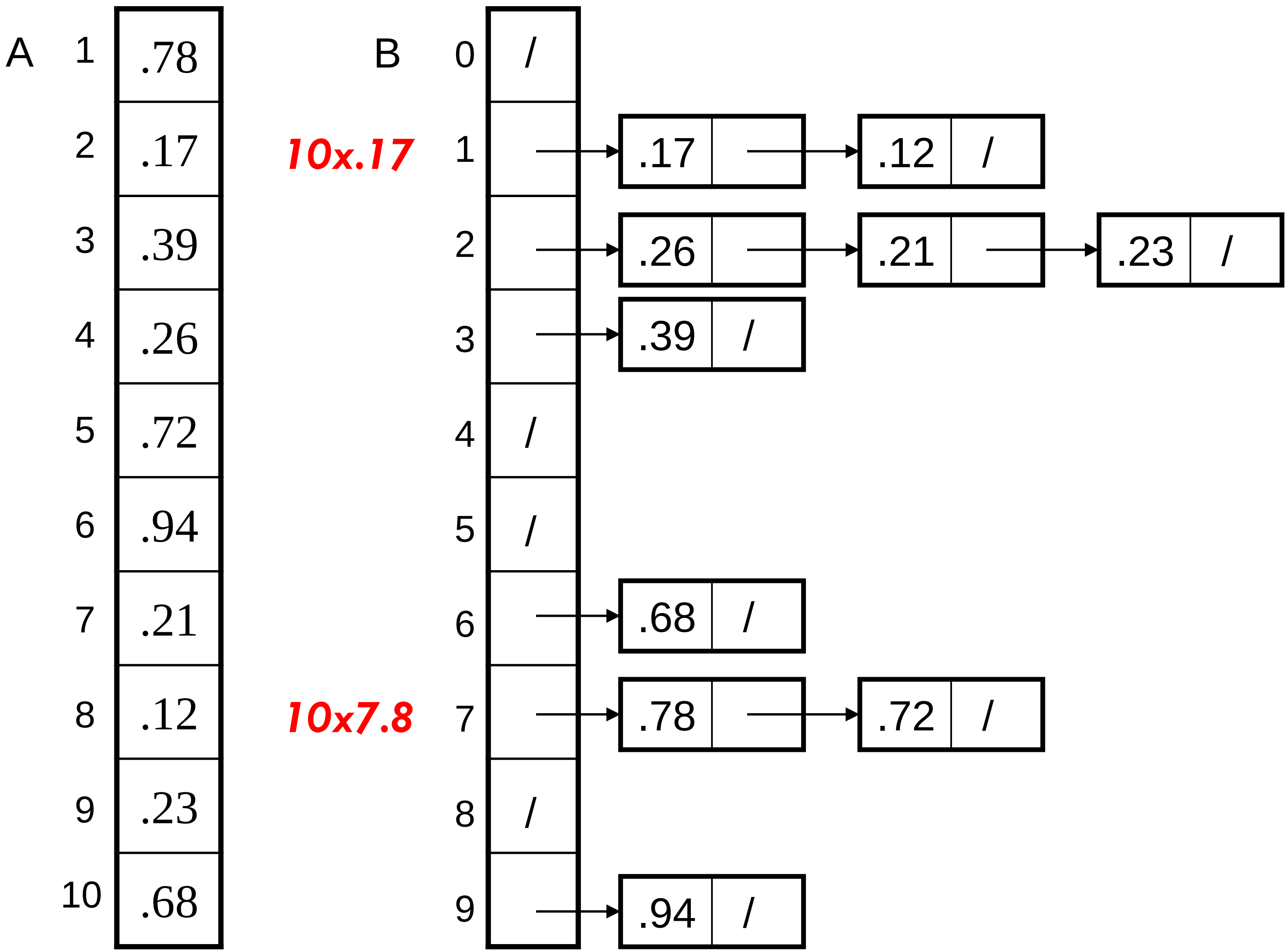
Number of Buckets $k=O(n)$

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

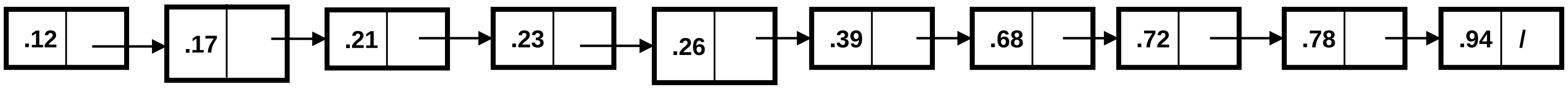
Output: *elements A[i] sorted*

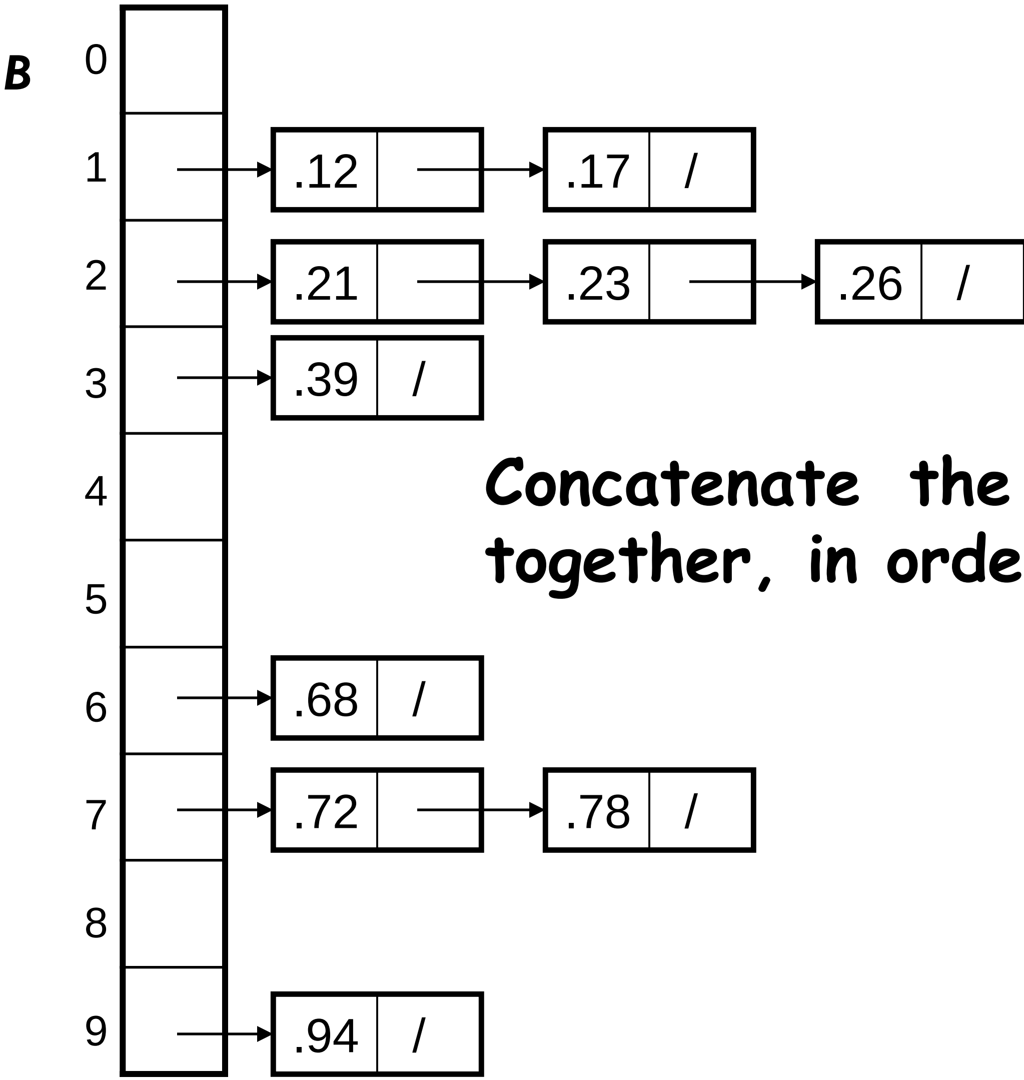
Auxiliary array: *B[0 . . n - 1] of linked lists, each list initially empty*

Bucket Sort: Example



Bucket Sort: Example





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

Bucket Sort

Alg.: BUCKET-SORT(A, n)

for i ← 1 to n do

  insert A[i] into list B[⌊nA[i]⌋]

for i ← 0 to n - 1 do

  sort list B[i] with Insertion Sort/Quick Sort

concatenate lists B[0], B[1], . . . , B[n -1] together in order

return the concatenated lists

Analysis of Bucket Sort

Alg.: BUCKET-SORT(A, n)

for i ← 1 to n

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

$\Big\}$ O(n)

for i ← 0 to n - 1

do sort list B[i] with insertion sort

$\Big\}$ O(n)

concatenate lists B[0], B[1], . . . , B[n -1]

together in order

$\Big\}$ O(n)

return the concatenated lists

O(n)

Comparison of Sorting Algorithms

Insertion sort: suitable only for small *n*.

Merge sort: guaranteed to be fast even in its worst case; stable.

Heapsort: requiring minimum memory and guaranteed to run fast; average and maximum time both roughly twice the average time of quicksort.

Quicksort: most useful general-purpose sorting for very little memory requirement and fastest average time. (choose the median of three elements as pivot in practice :-)

Counting sort: very useful when the keys have small range; stable; memory space for counters and for 2*n* records.

Radix sort: appropriate for keys either rather short or with a lexicographic collating sequence.

Bucket sort: assuming keys to have uniform distribution.