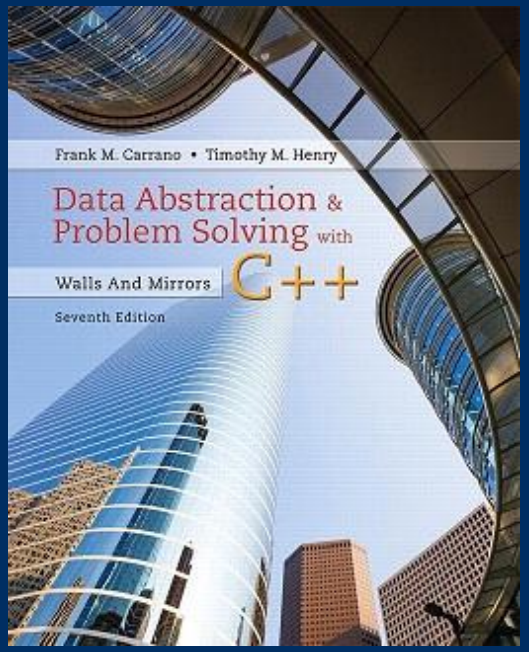
Chapter 11

Sorting Algorithms and their Efficiency

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors' slides

Linear Sorts

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

$a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$

Lower-Bound for Sorting

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

(see CS477 for a proof)

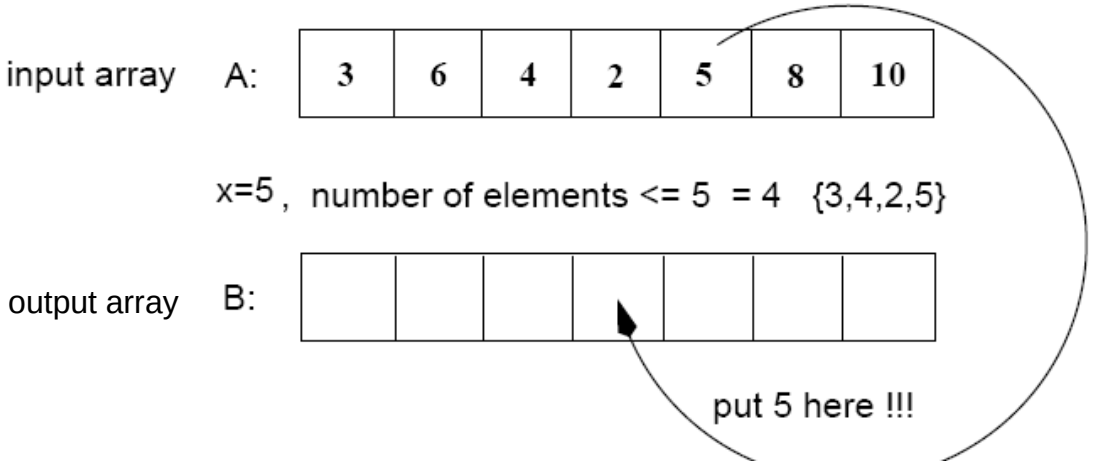
Can we do better?

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

Counting Sort

Counting Sort

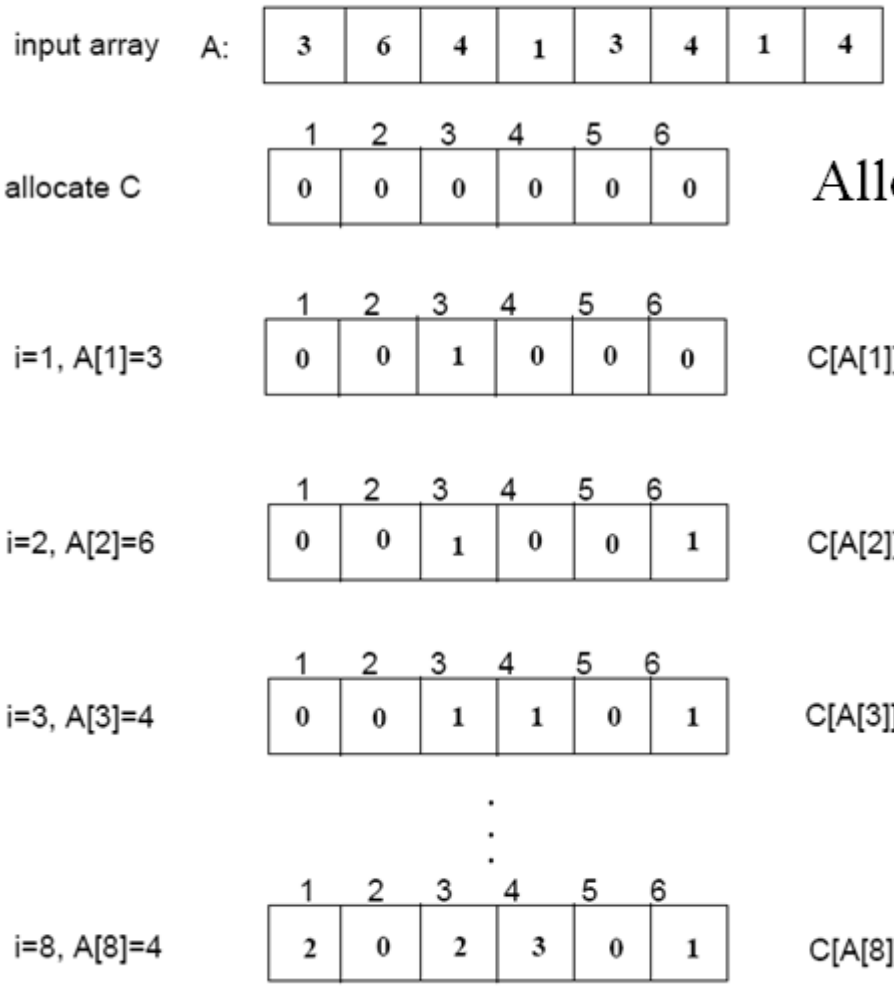
- Assumptions:
  - n integers which are in the range $[0 \ldots r]$
  - r is in the order of n, that is, $r=O(n)$
- Idea:
  - For each element x, find the number of elements $\leq x$
  - Place x into its correct position in the output array



Step 1

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

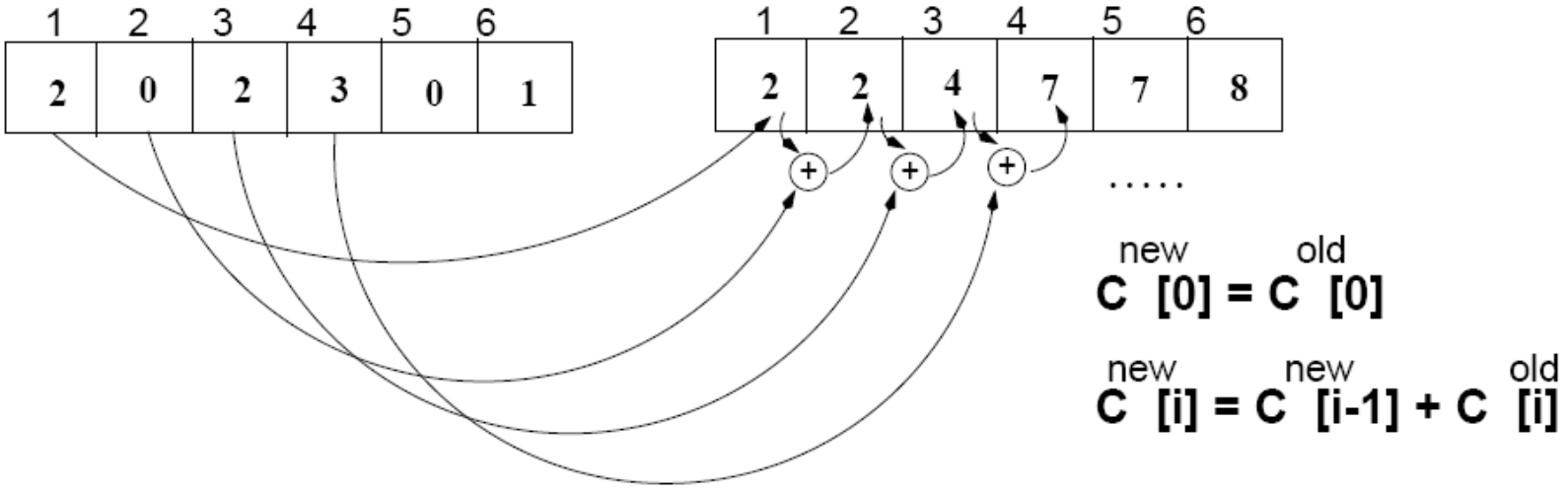
(i.e., frequencies)



Step 2

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

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



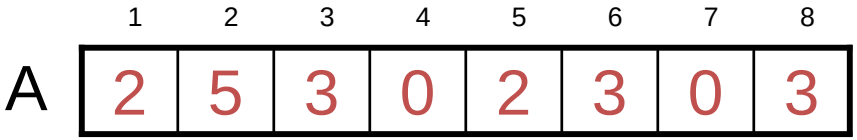
$C^{new}[0] = C^{old}[0]$

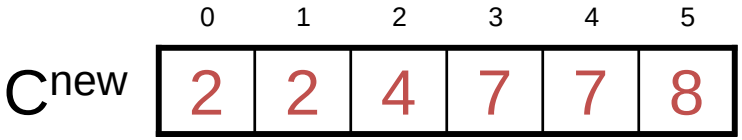
$C^{new}[i] = C^{new}[i-1] + C^{old}[i]$

C[i] = # elements <= i

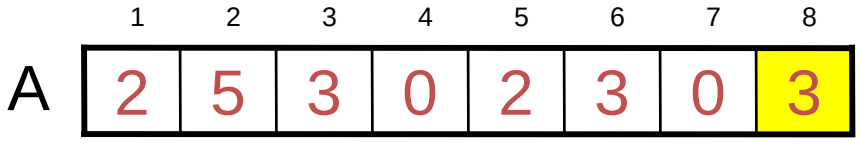
Algorithm

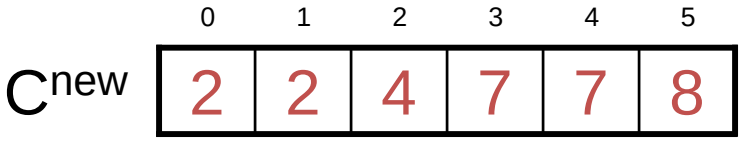
- Start from the last element of A
- Place A[i] at its correct place in the output array
- Decrease C[A[i]] by one

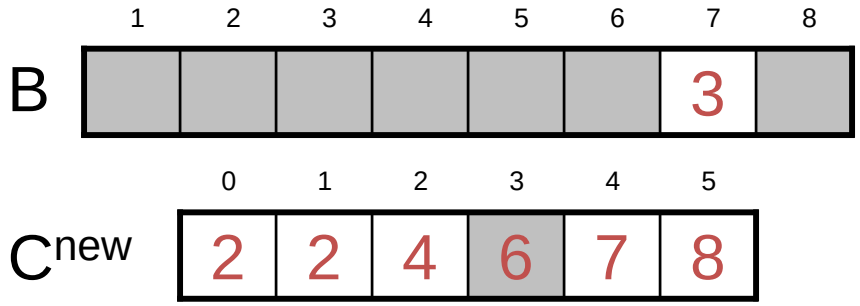


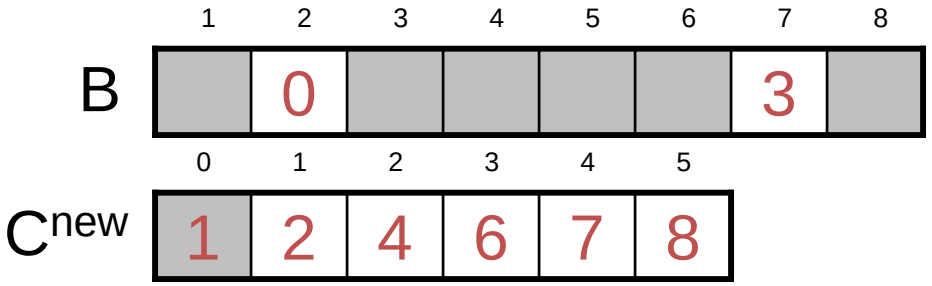


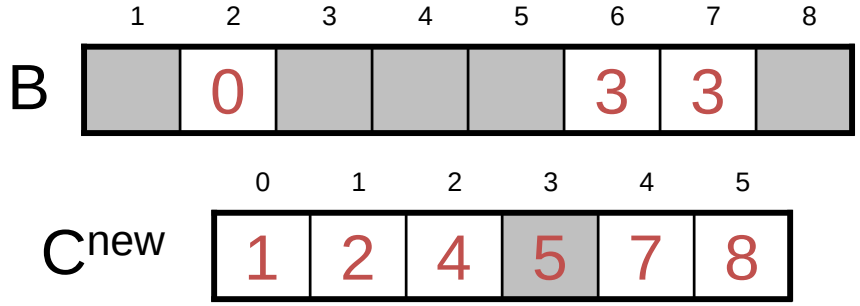
Example

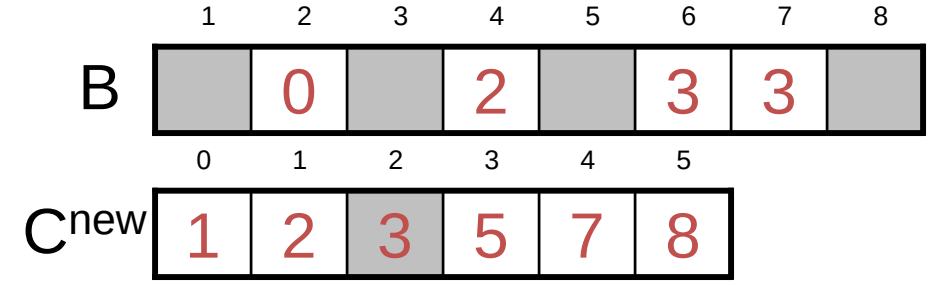

1 2 3 4 5 6 7 8
A
2 5 3 0 2 3 0 3


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

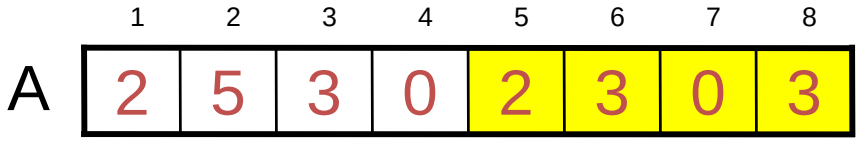

1 2 3 4 5 6 7 8
B
3
0 1 2 3 4 5
C new
2 2 4 6 7 8

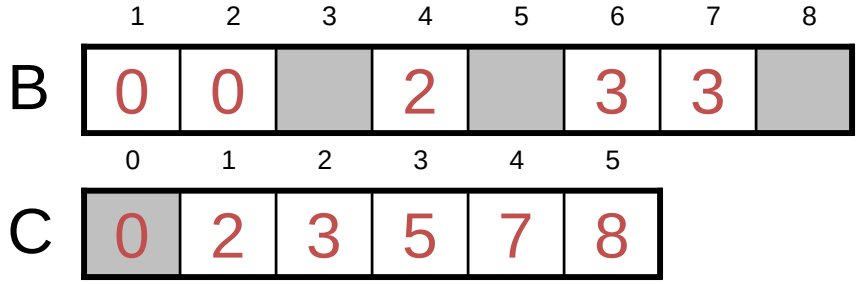

1 2 3 4 5 6 7 8
B
0 3
0 1 2 3 4 5
C new
1 2 4 6 7 8

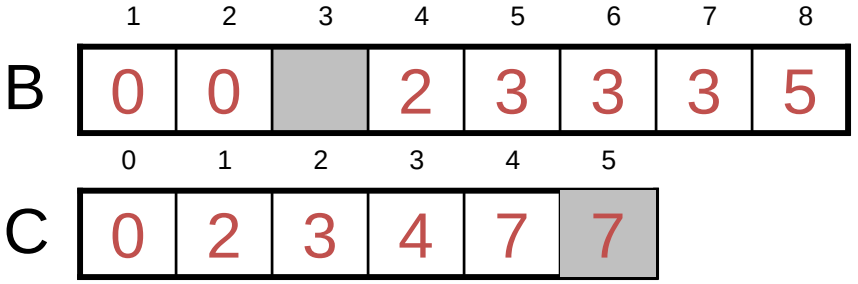

1 2 3 4 5 6 7 8
B
0 3 3
0 1 2 3 4 5
C new
1 2 4 5 7 8

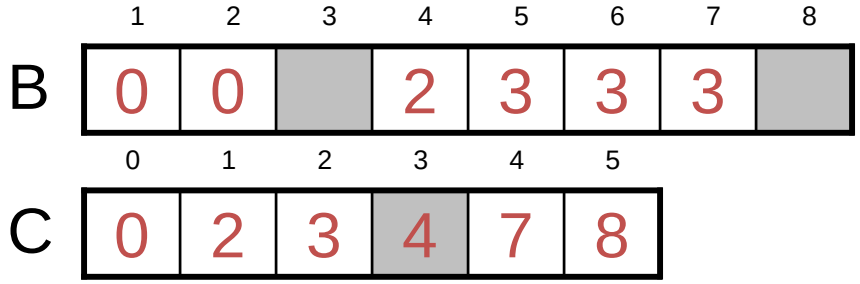

1 2 3 4 5 6 7 8
B
0 2 3 3
0 1 2 3 4 5
C new
1 2 3 5 7 8

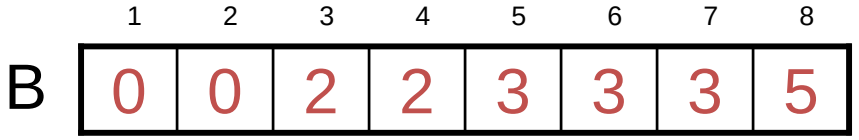
Example (cont.)



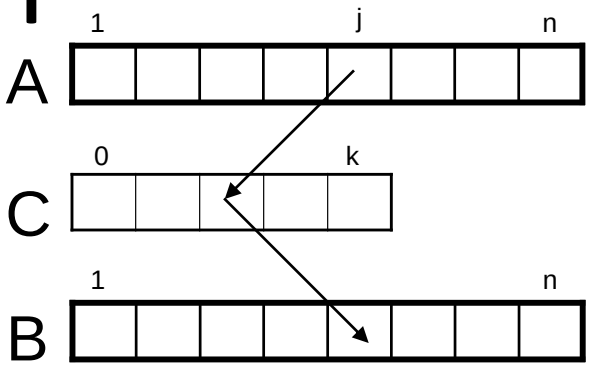








COUNTING-SORT



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

```
1.      for i ← 0 to r
2.          do C[ i ] ← 0
3.      for j ← 1 to n
4.          do C[A[ j ]] ← C[A[ j ]] + 1
5.   ▷  C[i] contains the number of elements equal to i
6.      for i ← 1 to r
7.          do C[ i ] ← C[ i ] + C[i -1]
8.   ▷  C[i] contains the number of elements ≤ i
9.      for j ← n downto 1
10.         do B[C[A[ j ]]] ← A[ j ]
11.            C[A[ j ]] ← C[A[ j ]] - 1
```

Analysis of Counting Sort

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

```
1.  for i ← 0 to r                                  } O(r)
2.      do C[ i ] ← 0
3.  for j ← 1 to n                                  } O(n)
4.      do C[A[ j ]] ← C[A[ j ]] + 1
5.  C[i] contains the number of elements equal to i
6.  for i ← 1 to r                                  } O(r)
7.      do C[ i ] ← C[ i ] + C[i -1]
8.  C[i] contains the number of elements ≤ i
9.  for j ← n downto 1                              } O(n)
10.     do B[C[A[ j ]]] ← A[ j ]
11.        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 $r = O(n)$

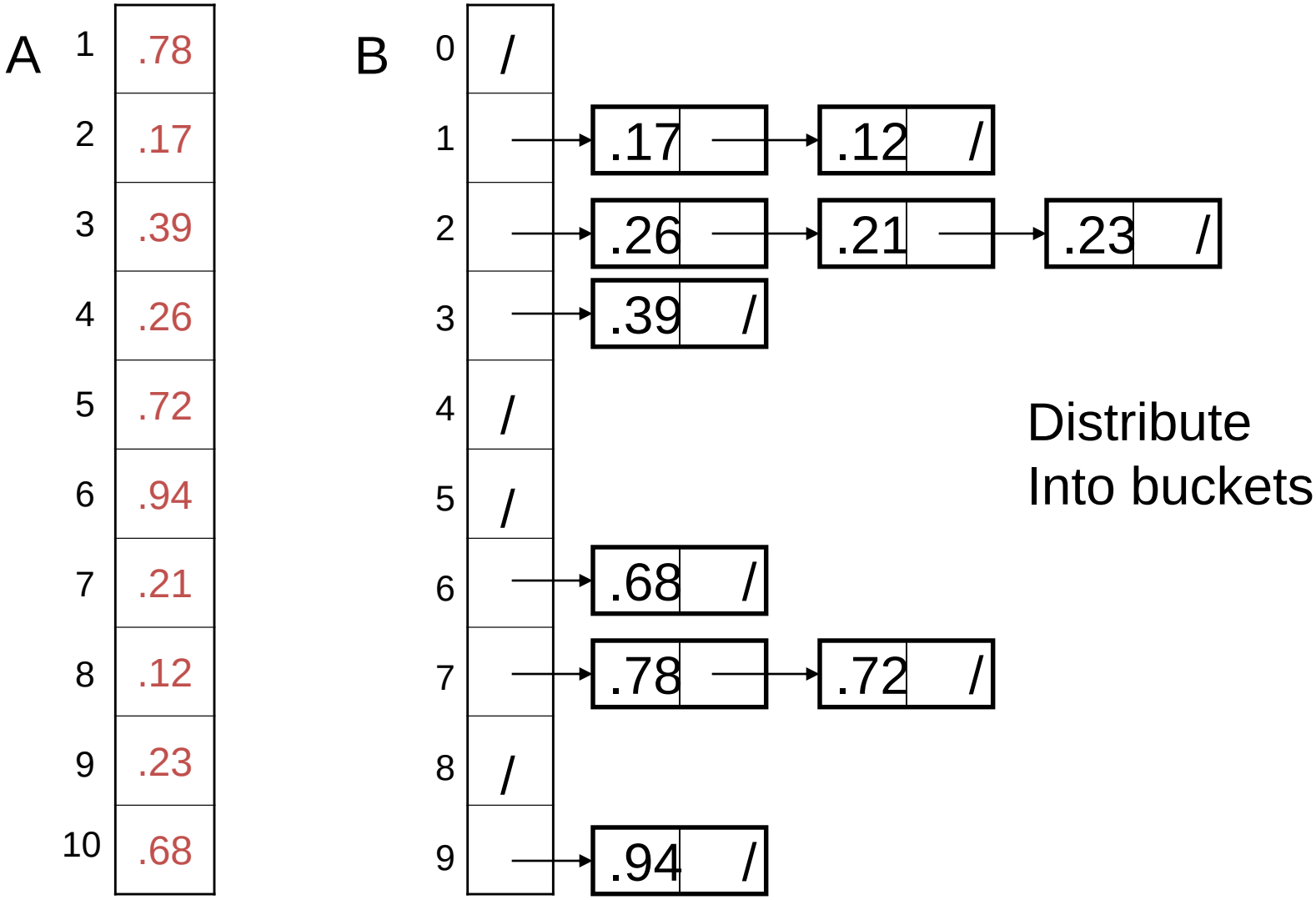
  $\Rightarrow$ running time is $O(n)$

Bucket Sort

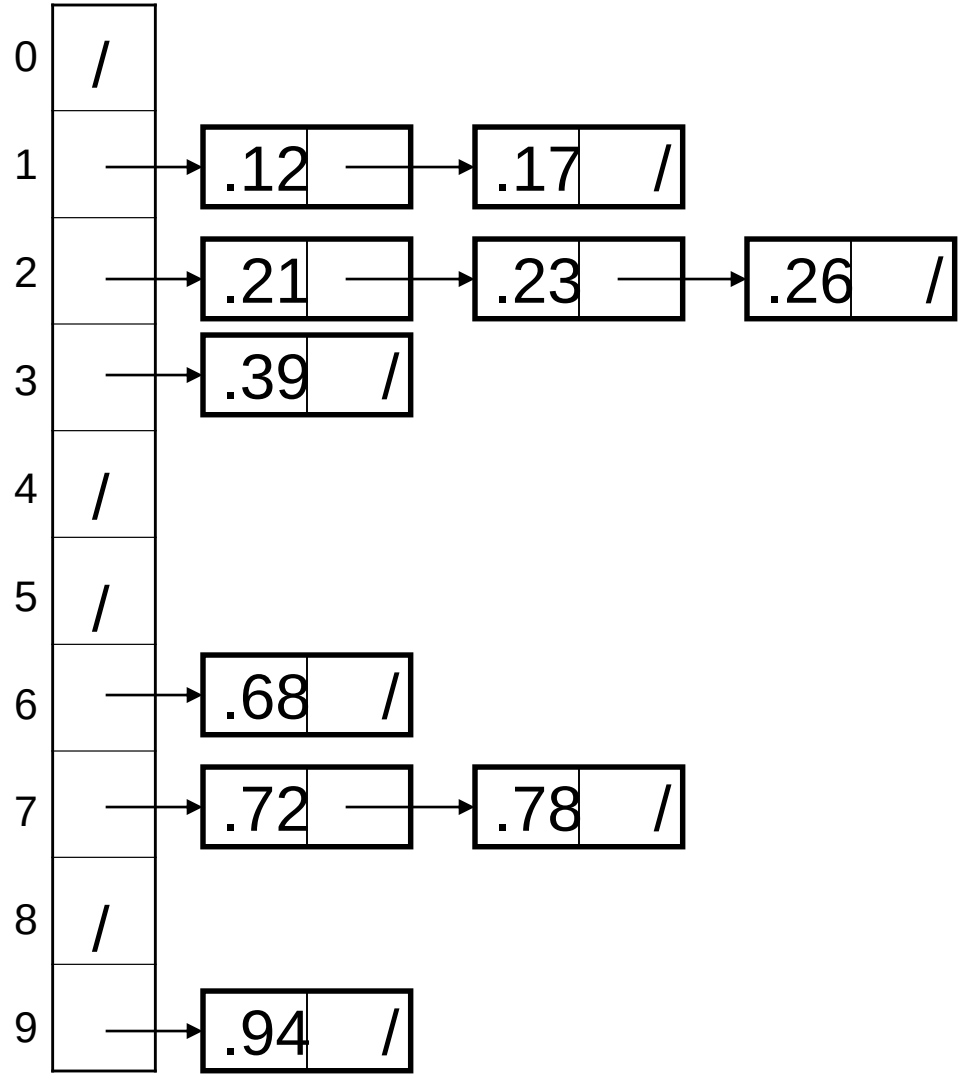
Bucket Sort

- 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=Θ(n)**)
  - Distribute the n input values into the buckets
  - Sort each bucket (e.g., using mergesort)
  - Go through the buckets in order, listing elements in each one
- **Input:** $A[1 \ldots n]$, where $0 \le A[i] < 1$ for all i
- **Output:** elements $A[i]$ sorted

Example - Bucket Sort

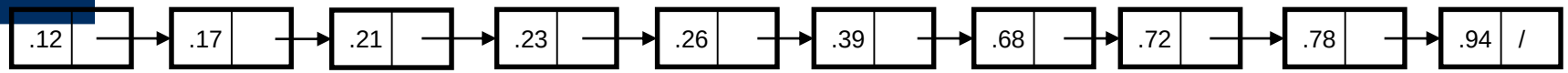


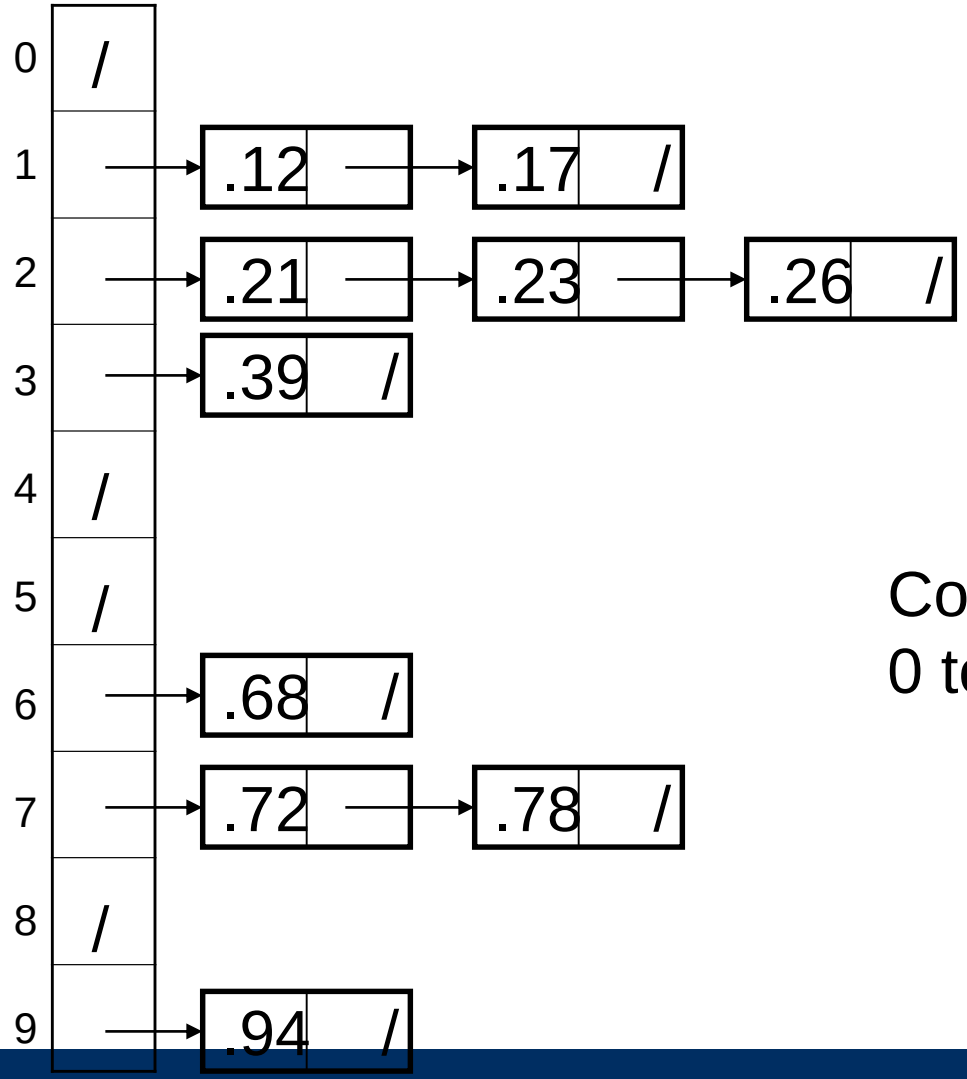
Example - Bucket Sort



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 ← 1 **to** n
    do insert A[i] into list B[⌊nA[i]⌋] — O(n)

for i ← 0 **to** k - 1
    do sort list B[i] with mergesort sort — k O(n/k log(n/k)) = O(nlog(n/k))

concatenate lists B[0], B[1], . . . , B[n -1] together in order — O(k)

return the concatenated lists

O(n) (if k=Θ(n))

Radix Sort

The Radix Sort

- Different from other sorts
  - Does not compare entries in an array
- Begins by organizing data (say strings) according to least significant letters
  - Then combine the groups
- Next form groups using next least significant letter

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

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

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

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

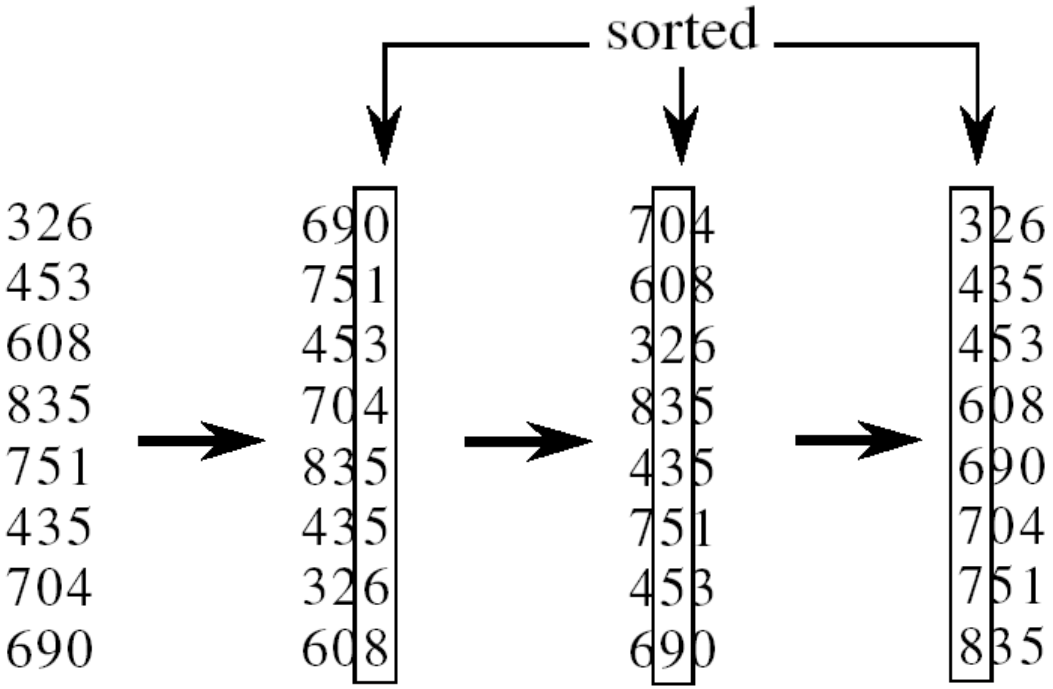
RADIX-SORT

Alg.: RADIX-SORT(*A*, d)

for i ← 1 **to** d

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

(stable sort: preserves order of identical elements)



The Radix Sort

- Uses the idea of forming groups, then combining them to sort a collection of data

- Consider collection of three letter groups

  ABC, XYZ, BWZ, AAC, RLT, JBX, RDT, KLT, AEO, TLJ

- Group strings by rightmost letter

  (AB**C**, AA**C**) (TL**J**) (AE**O**) (RL**T**, RD**T**, KL**T**) (JB**X**) (XY**Z**, BW**Z**)

- Combine groups

  ABC, AAC, TLJ, AEO, RLT, RDT, KLT, JBX, XYZ, BWZ

The Radix Sort

- Group strings by middle letter

  (A**A**C) (A **B** C, J **B** X) (R **D** T) (A **E** O) (T **L** J, R **L** T, K **L** T) (B **W** Z) (X **Y** Z)

- Combine groups

  AAC, ABC, JBX, RDT, AEO, TLJ, RLT, KLT, BWZ, XYZ

- Group by first letter, combine again

  (**A** AC, **A** BC, **A** EO) (**B** WZ) (**J** BX) (**K** LT) (**R** DT, **R** LT) (**T** LJ) (**X** YZ)

- Sorted strings

  AAC, ABC, AEO, BWZ, JBX, KLT, RDT, RLT, TLJ, XYZ

The Radix Sort

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(1560, 2150) (1061) (0222) (0123, 0283) (2154, 0004)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(0004) (0222, 0123) (2150, 2154) (1560, 1061) (0283)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0004, 1061) (0123, 2150, 2154) (0222, 0283) (1560)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(0004, 0123, 0222, 0283) (1061, 1560) (2150, 2154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

- A radix sort of eight integers

The Radix Sort

```
// Sorts n d-digit integers in the array theArray.
radixSort(theArray: ItemArray, n: integer, d: integer): void
{
    for (j = d down to 1)
    {
        Initialize 10 groups to empty
        Initialize a counter for each group to 0
        for (i = 0 through n - 1)
        {
            k = jth digit of theArray[i]
            Place theArray[i] at the end of group k
            Increase kth counter by 1
        }
        Replace the items in theArray with all the items in group 0,
          followed by all the items in group 1, and so on.
    }
}
```

- Pseudocode for algorithm for a radix sort of *n* decimal integers of *d* digits each:

The Radix Sort

- Analysis
  - Requires n moves each time it forms groups
  - n moves to combine again into one group
  - Performs these $2 \times n$ moves d times
  - Thus requires $2 \times n \times d$ moves
- Radix sort is of order $O(n)$

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)
  - Assuming d=O(1) and k=O(n), running time is O(n)

Stable Sorting Algorithms

- $O(N^2)$
  - Selection Sort, Insertion Sort, Bubble Sort
- O(N logN)
  - Merge Sort
- O(N)
  - Counting Sort, Bucket Sort, Radix Sort

A Comparison of Sorting Algorithms

	Best case	Average case	Worst case
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Buble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Counting sort	$O(n)$	$O(n)$	$O(n + r)$
Bucket sort	$O(n)$	$O(n)$	$O(n \log n)$
Radix sort	$O(n)$	$O(n)$	$O(n + k)$

- Approximate growth rates of time required for sorting algorithms

Heap Sort

Heap Sort

- https://www.studytonight.com/data-structures/heap-sort

- https://www.programiz.com/dsa/heap-sort