CSE214 – Analysis of Algorithms

PhD Furkan Gözükara, Toros University

https://github.com/FurkanGozukara/Analysis-of-Algorithms-2019

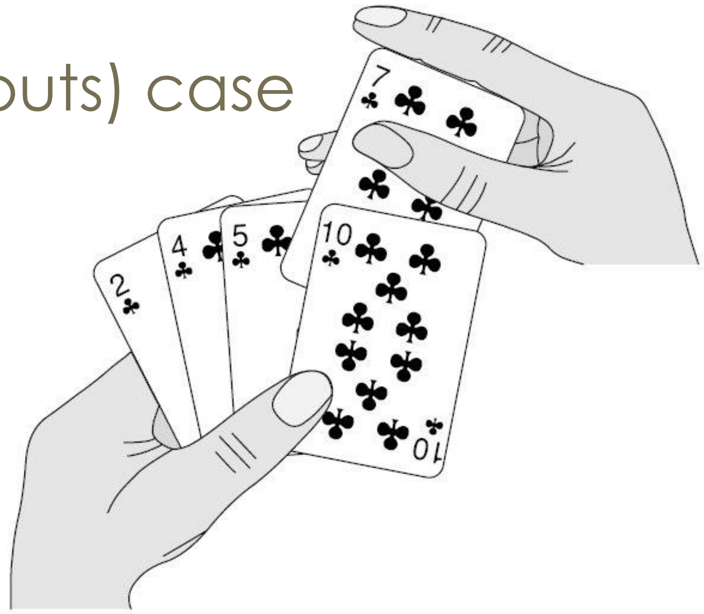
Lecture 6

Linear Sorting

Based on Andrew Davison's Lecture Notes

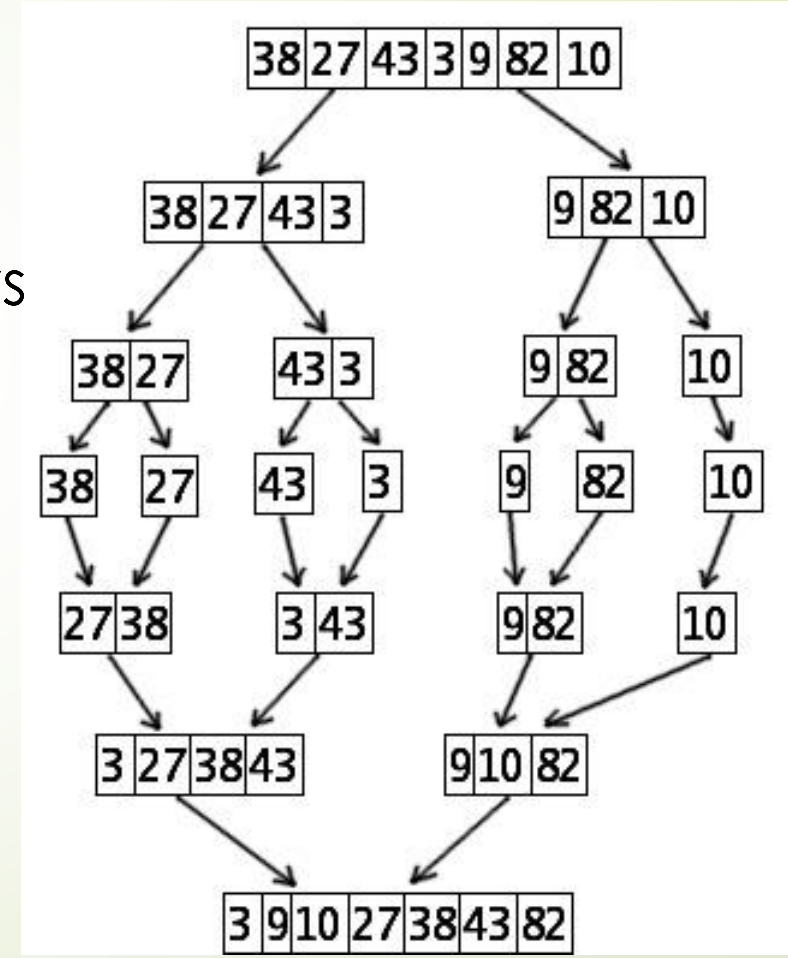
Sorting So Far

- **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



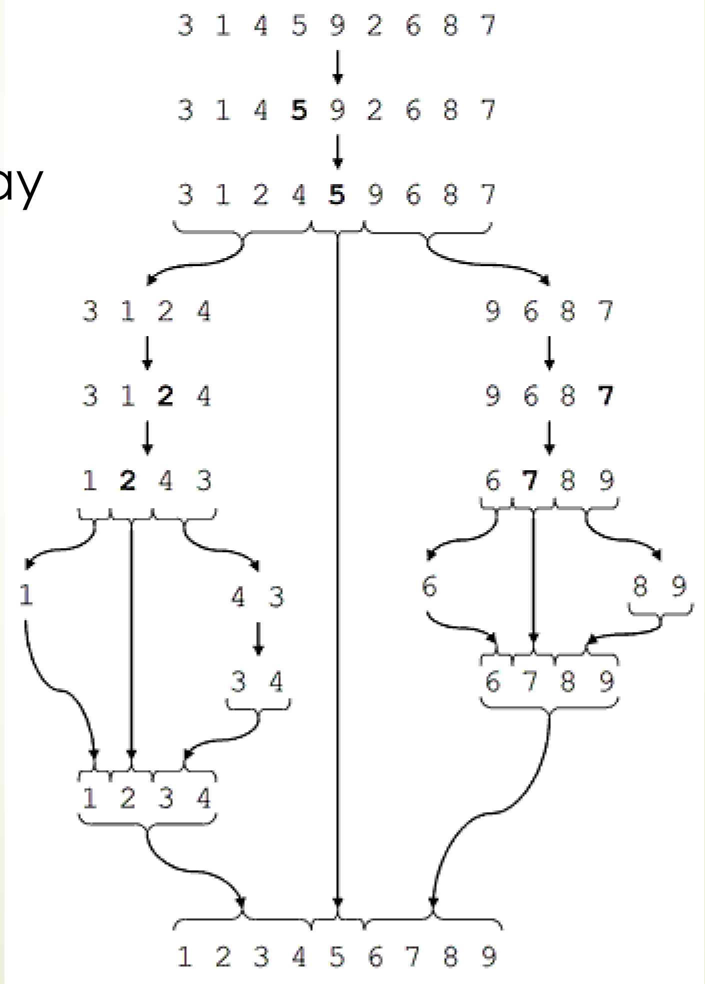
Sorting So Far

- **Merge sort**:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - Doesn't sort in place



Sorting So Far

- **Quicksort**:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of 1st subarray < all of 2nd subarray
    - No merge step needed!
  - $O(n \lg n)$ average case
  - Fast in practice
  - $O(n^2)$ worst case
    - for sorted input
    - this is avoided by using a randomized pivot



How Fast Can We Sort?

- Selection Sort, Bubble Sort, Insertion Sort: $O(n^2)$
- Heap Sort, Merge sort: $O(nlgn)$
- Quicksort: $O(nlgn)$ - *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 $a_i > a_j$

How Fast Can We Sort?

- All these sorting algorithms are *comparison sorts*
  - gain ordering information about a sequence using the comparison of two elements (=, <, >)
  - Theorem: all **comparison sorts** are **O(n lg n)** or slower
  - The **best speed for sorting is O(n)**
    - we must look at all the data
    - for that we must use sorting algorithms which don't require comparisons of all the data

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”

Non-Comparison Based Sorting

- Many times we have restrictions on our keys
  - Deck of cards: Ace->King and four suites
  - Social Security Numbers
  - Employee ID's

- We will examine three algorithms which under certain conditions can run in **O(*n*)** time.
  - Counting sort
  - Radix sort
  - Bucket sort

Counting Sort

- Does no comparisons between the array elements!
- This depends on an assumption about the numbers being sorted
  - We assume numbers are integers in the range *1.. k*
  - *running time depends on k, so might be slower than comparison sorts*
- The algorithm:
  - Input: A[1 .. n], where A[j] $\in$ {1, 2, 3, …, k}
  - Output: B[1 .. n], sorted (notice: not sorting in place)
  - Also: Array C[1 .. $\boldsymbol{k}$] for auxiliary storage

Pseudocode

```
CountingSort(int[] A, int[] B, int k)    // sort A into B; elems range: 1..k
for i ← 1 to k    // initialization count occurrences array
    do C[i] ←0

for j ← 1 to n     // counting
    do C[A[j]] ← C[A[j]] + 1    //  C[i] = |{key == i}|

for i ← 2 to k    // summing
    do C[i] ←C[i] + C[i–1]       //  C[i] = |{key ≤ i}|

for j ← n down to 1     // create output array (distribution)
    do   B[ C[ A[j] ] ]  ← A[j]
         C[ A[j] ]  ←  C[ A[j] ] –1
```

Counting-sort example

	1	2	3	4	5
A:	*4*	*1*	*3*	*4*	*3*

	1	2	3	4
C:				

B:					

n size

k size

Loop 1

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
0	0	0	0

B:

for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
0	0	0	1

B:

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

  do $C[A[j]] \leftarrow C[A[j]] + 1$

$// \; C[i] == |\{key = i\}|$

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	0	1

B:

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

  do $C[A[j]] \leftarrow C[A[j]] + 1$

$// \; C[i] = |\{key == i\}|$

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	1

B:					

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$
    $// \ C[i] = |\{key == i\}|$

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

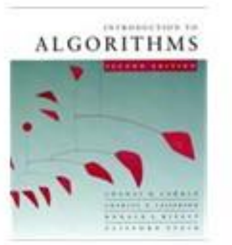
1	2	3	4
1	0	1	2

B:

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

  do $C[A[j]] \leftarrow C[A[j]] + 1$

// $C[i] = |\{key == i\}|$



Loop 2

C is the number of data occurences

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

B:

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$

// $C[i] = |\{key == i\}|$

Loop 3

C is changed to hold prefix sums

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	2	2

for $i \leftarrow 2$ **to** k

  do $C[i] \leftarrow C[i] + C[i\text{-}1]$

// $C[i] = |\{key \leq i\}|$

Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

B:

C':

1	1	3	2

for $i \leftarrow 2$ **to** k

  do $C[i] \leftarrow C[i] + C[i\text{-}1]$ // $C[i] = |\{key \leq i\}|$

Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

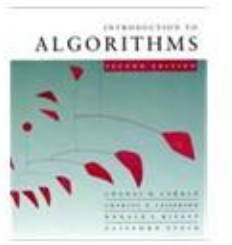
B:

C':

1	1	3	5

for $i \leftarrow 2$ **to** k

do $C[i] \leftarrow C[i] + C[i\text{-}1]$ // $C[i] = |\{key \leq i\}|$



Loop 4

B is will store the A data in its correct position (distribution) based on C

	1	2	3	4	5
A:	4	1	3	4	3
B:					

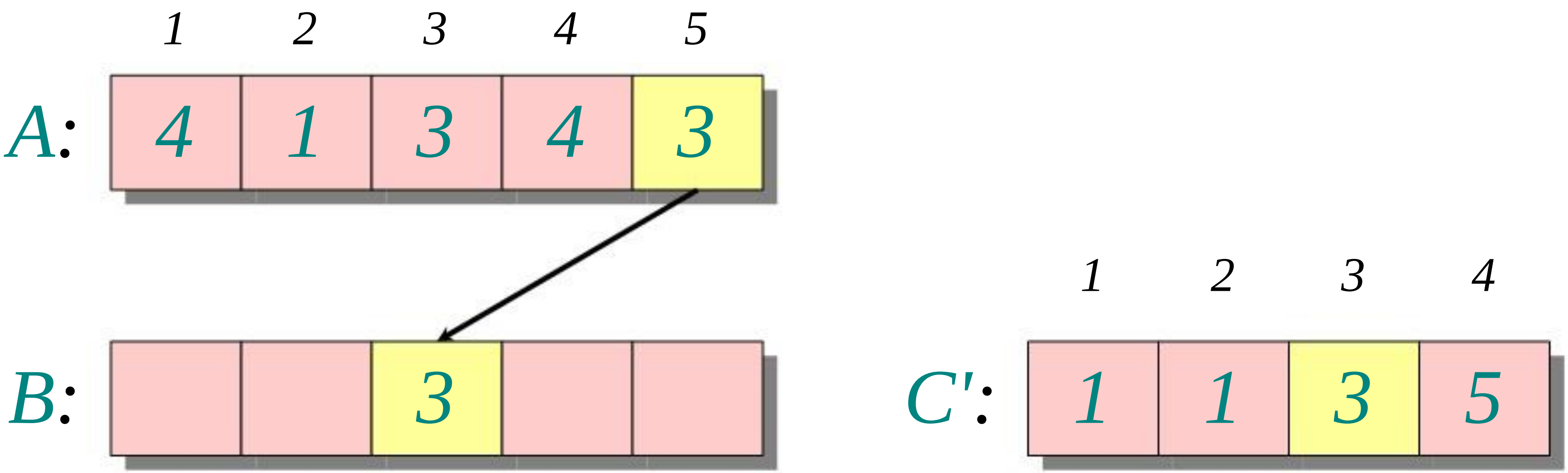
	1	2	3	4
C':	1	1	3	5

for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

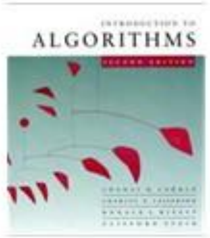
Loop 4



for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$



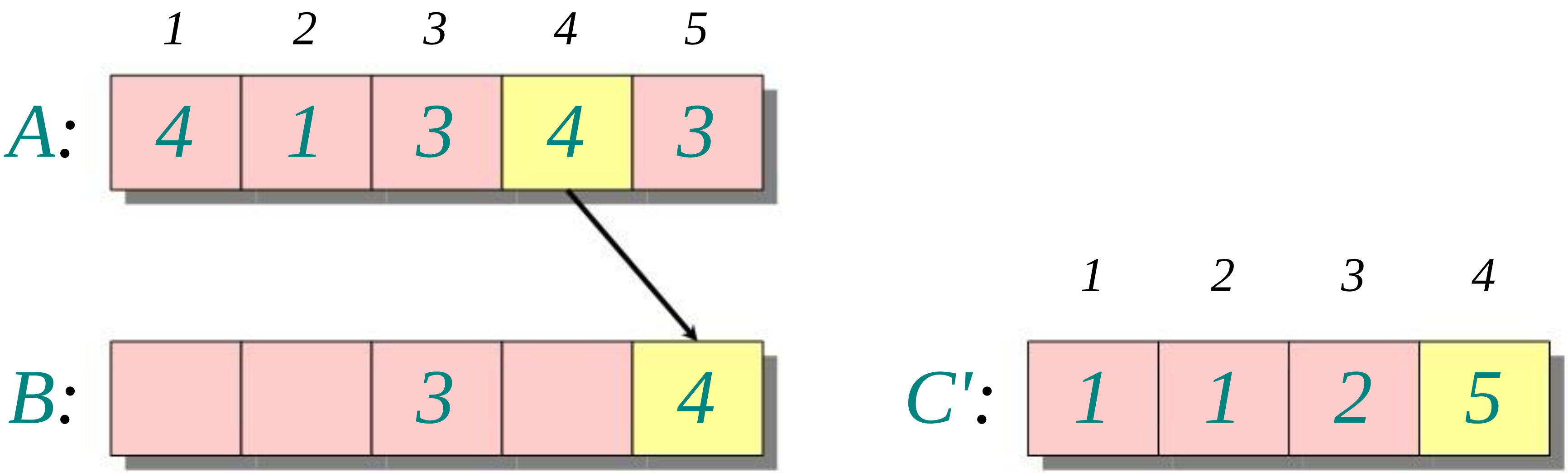
Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
B:			3		

	1	2	3	4
C':	1	1	2	5

for $j \leftarrow n$ **downto** 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$ // *decr counter*

Loop 4

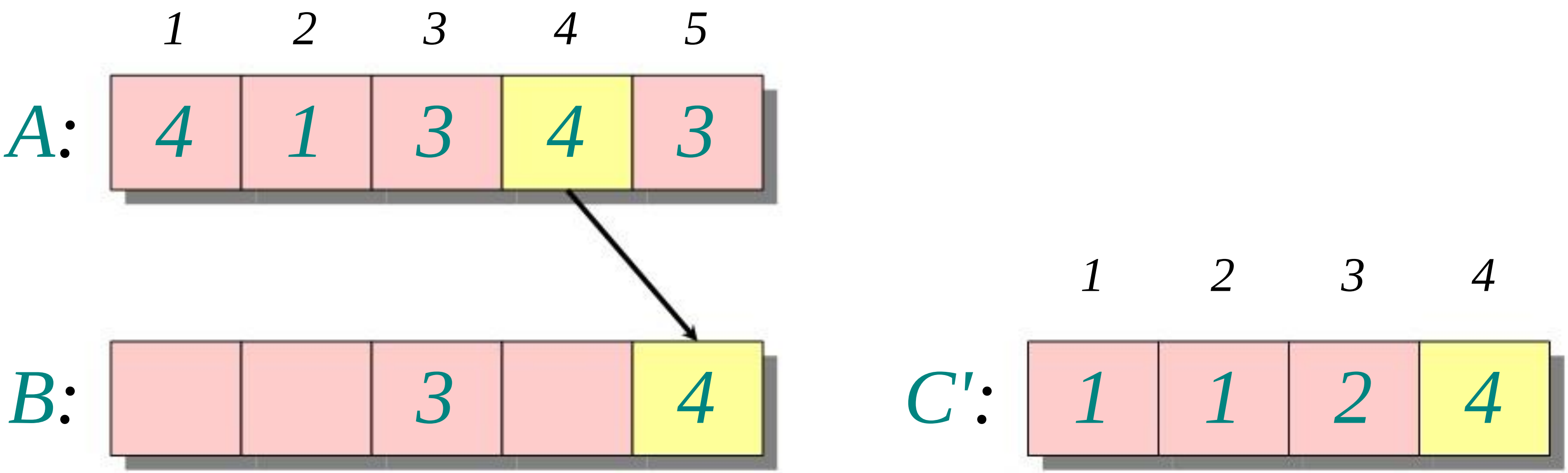


for $j \leftarrow n$ **downto** 1

    do $B[C[A[j]]] \leftarrow A[j]$

        $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$ // *decr counter*

Loop 4

the decrement means that this second 3 goes in the correct position

	1	2	3	4	5
A:	4	1	3	4	3

B:		3	3		4

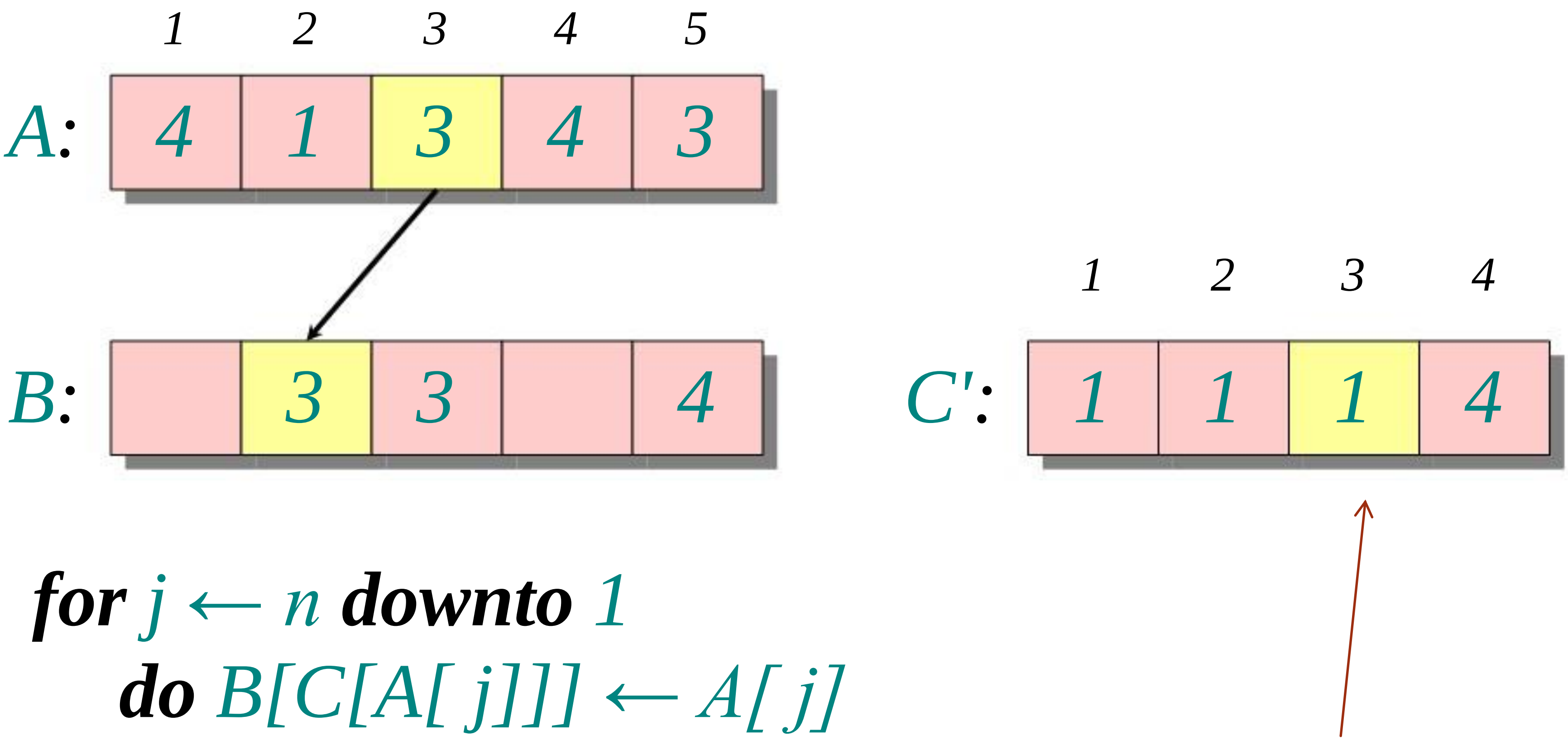
	1	2	3	4
C′:	1	1	2	4

for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

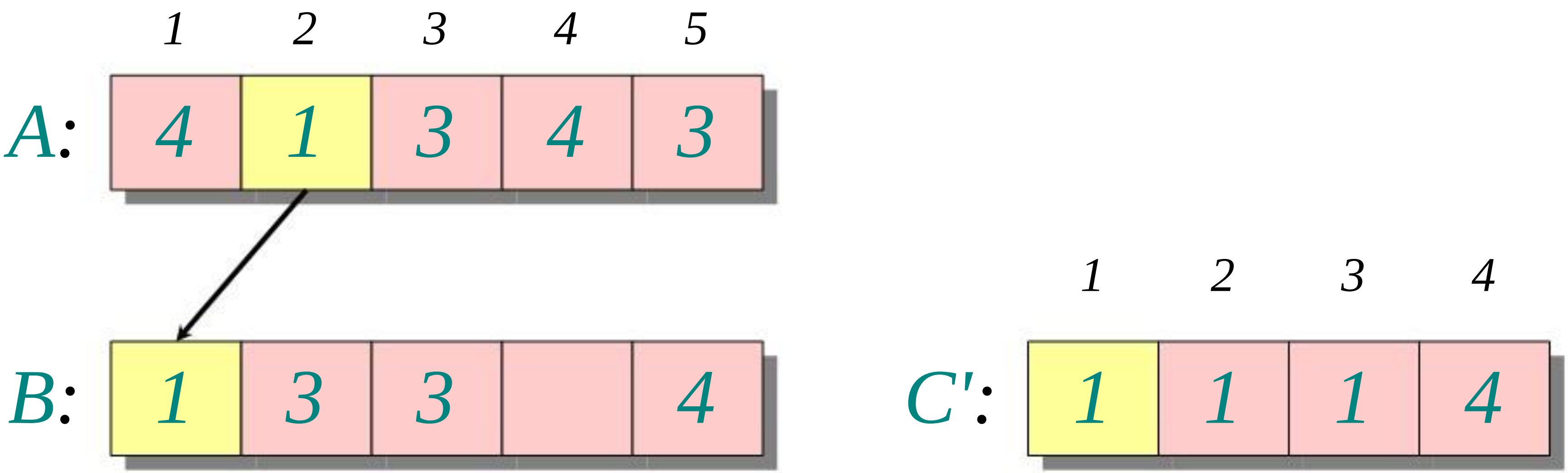
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



for $j \leftarrow n$ **downto** 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$ // *decr counter*

Loop 4

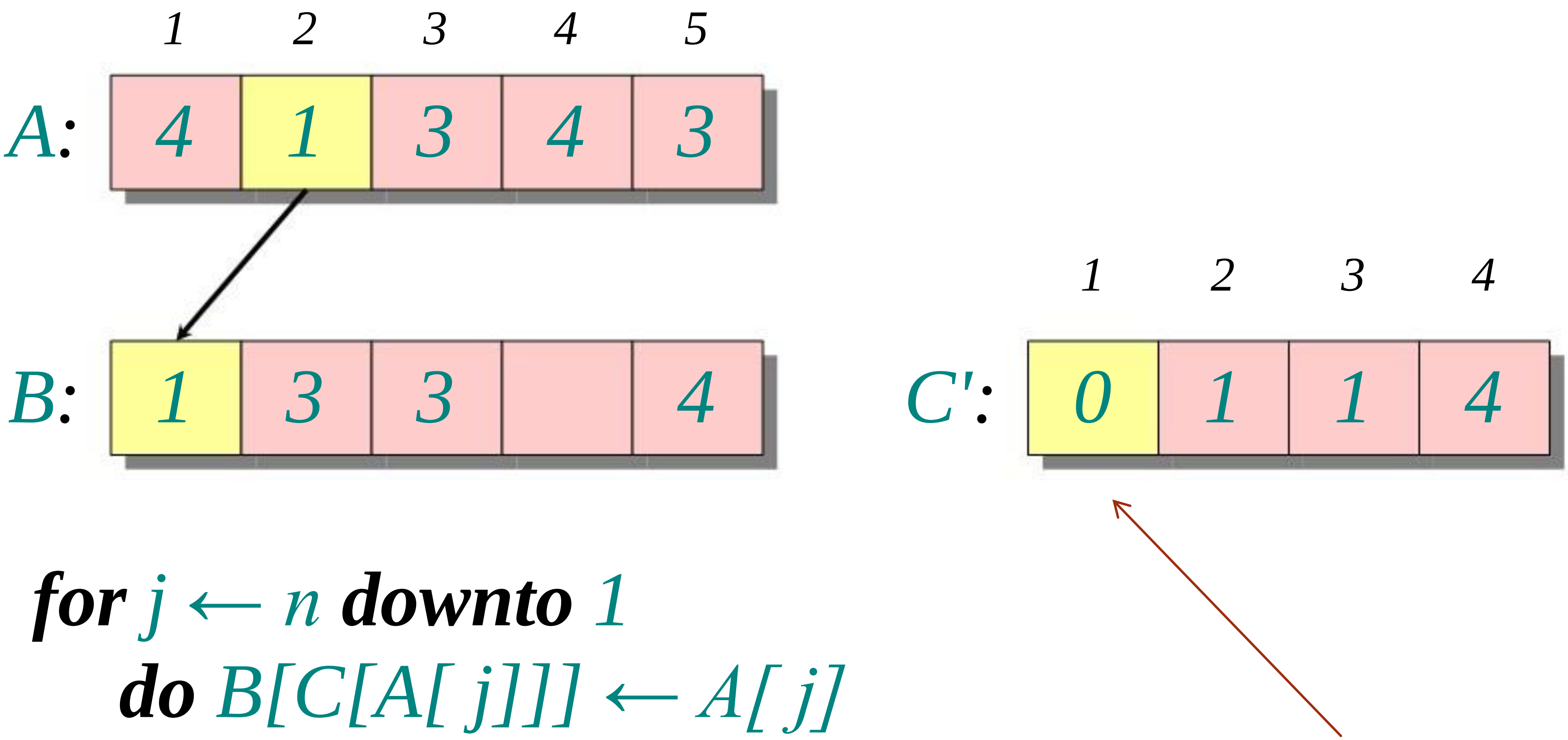


for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$ // *decr counter*

Loop 4

the decrement means that this second 4 goes in the correct position

A:

1	2	3	4	5
4	1	3	4	3

B:

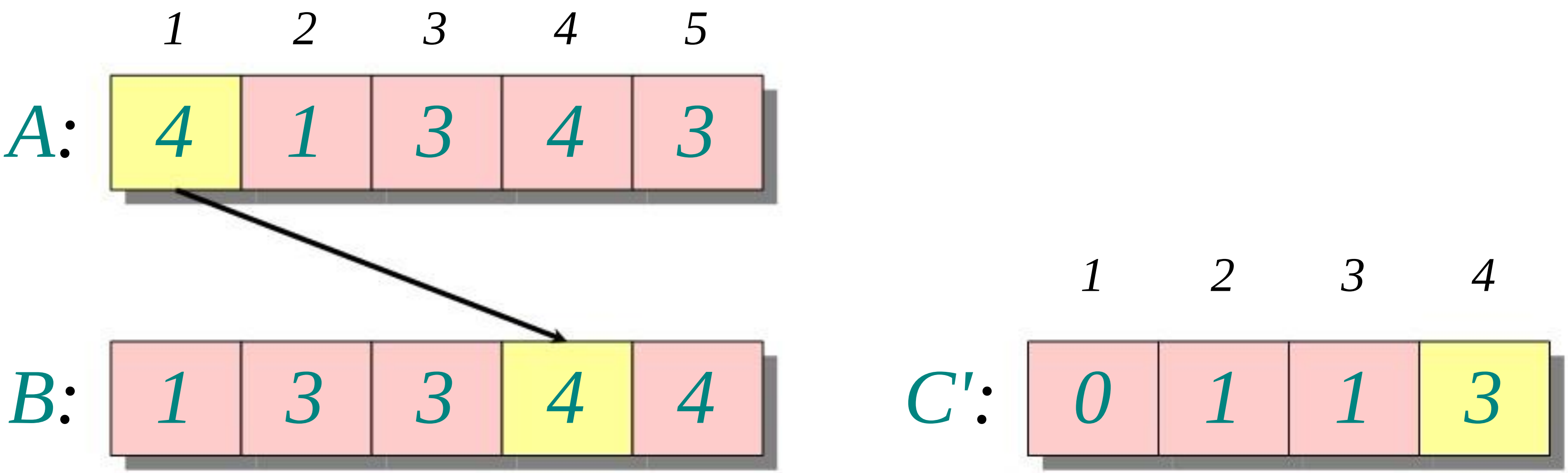
1	3	3	4	4

C':

1	2	3	4
0	1	1	4

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

Loop 4



for $j \leftarrow n$ **downto** 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$ // decr counter

B vs C

	1	2	3	4	5
B:	1	3	3	4	4

	1	2	3	4
C':	1	1	3	5

In the end, each element i occupies the range

B[C[i-1]+1 … C[i]]

Analysis

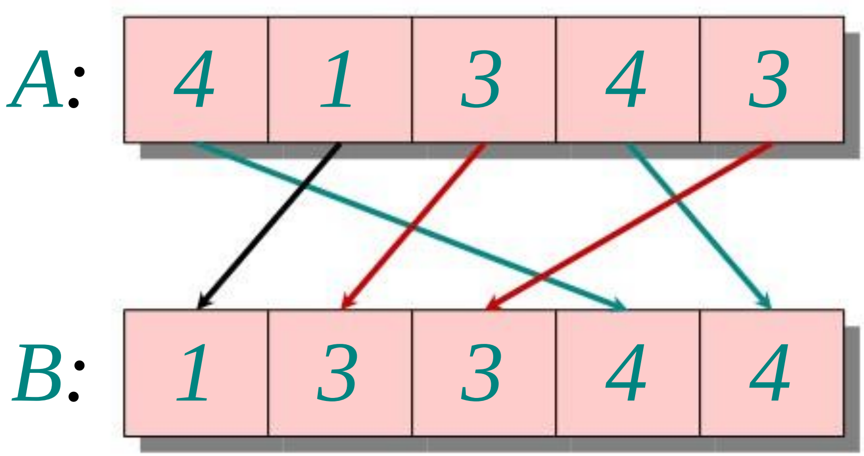
$O(k)$	**for** $i \leftarrow 1$ **to** k **do** $C[i] \leftarrow 0$
$O(n)$	**for** $j \leftarrow 1$ **to** n **do** $C[A[j]] \leftarrow C[A[j]] + 1$
$O(k)$	**for** $i \leftarrow 2$ **to** k **do** $C[i] \leftarrow C[i] + C[i-1]$
$O(n)$	**for** $j \leftarrow n$ **downto** 1 **do** $B[C[A[j]]] \leftarrow A[j]$ $C[A[j]] \leftarrow C[A[j]] - 1$
$O(n + k)$	

Running Time

- Total time: **$O(n + k)$**
  - Usually, $k = O(n)$
  - Thus counting sort runs in **$O(n)$** time
  - Notice that this algorithm is *stable*
    - A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted

Stable Sorting

Counting sort is a ***stable*** *sort: it preserves the input order among equal elements.*



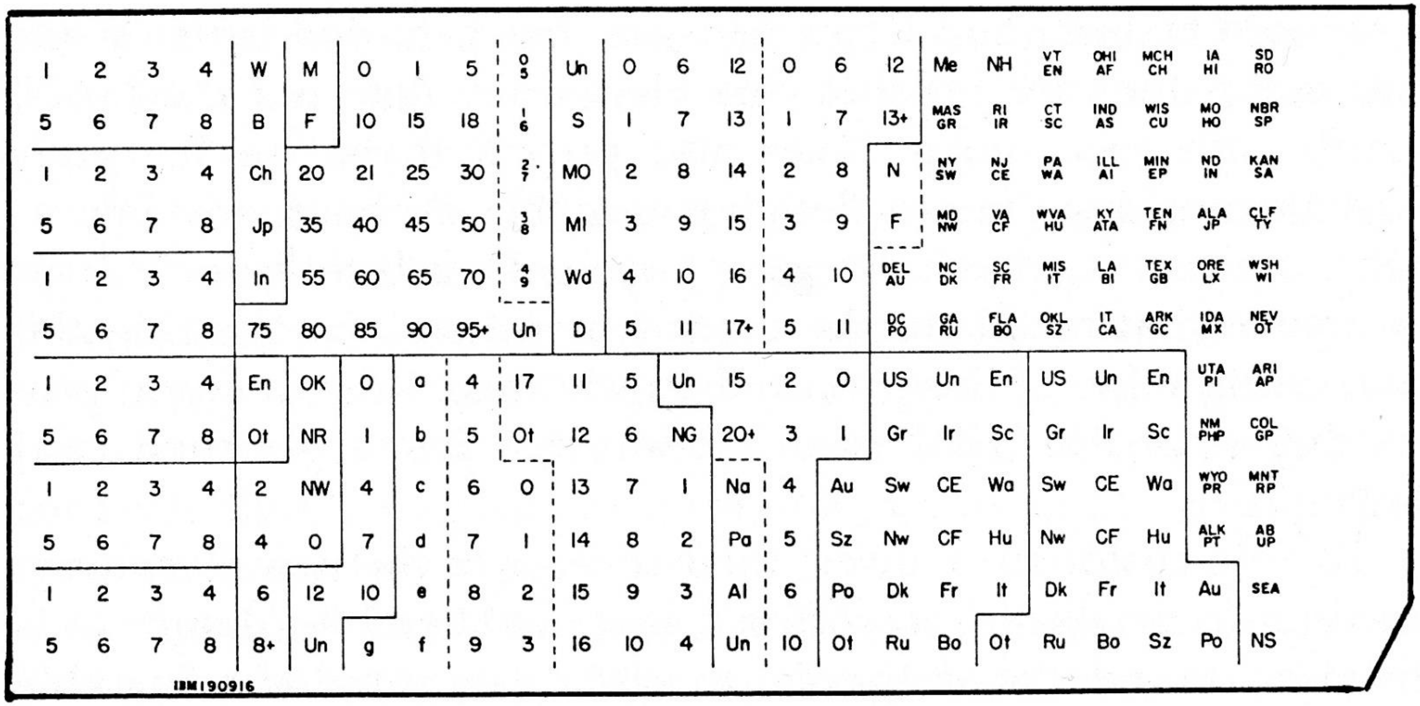
Drawbacks of Counting Sort

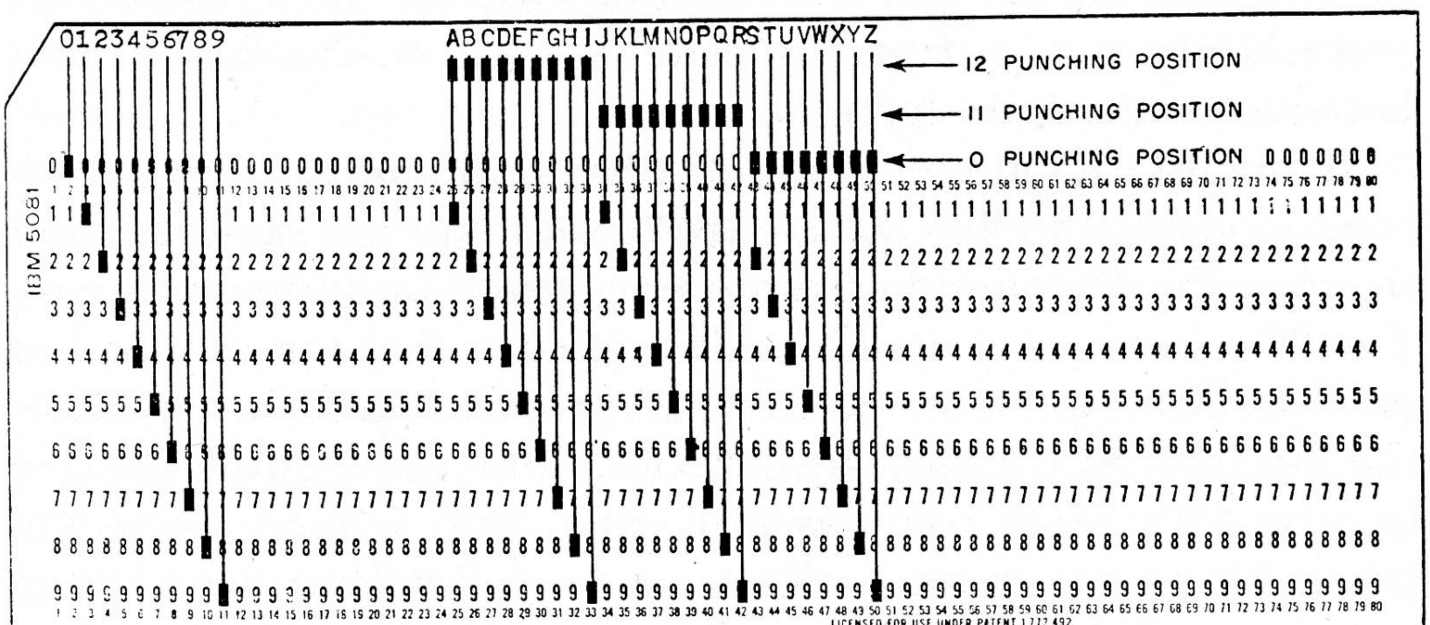
- *Why don't we always use counting sort?*
- Because it depends on the range *k* of the elements

- *Could we use counting sort to sort 32 bit integers?*
- Yes, on x64 and enough ram memory having computers (2^{32} = 4,294,967,296)
  - k is used for the size of the C array = 4 GB

Radix Sort

- **Origin**: Herman Hollerith's card-sorting machine for the 1890 U.S. census
  - probably the oldest implemented sorted algorithm
  - uses punch cards
- a digit-by-digit sort
- Hollerith's original (**bad**) idea: sort on the **most-significant digit** first
  - Problem: lots of intermediate piles of cards (i.e. extra temporary arrays) to keep track of the calculations


IBM190916


0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
12 PUNCHING POSITION
11 PUNCHING POSITION
0 PUNCHING POSITION
IBM 5081
LICENSED FOR USE UNDER PATENT 1,772,492

- **Good** idea: sort on the **least-significant digit** first with a *stable* sort
  - preserves the relative order of equal elements
  - this is a property of counting sorts, as seen earlier
- Code:

```
RadixSort(A, d){
    for i = 1 to d
        StableSort(A) on digit i
}
```

this can be any sorting algorithm, but must be stable

Operation of Radix Sort

n = size of array = 7
k = range = 0..9 = 10
d = no. digits = 3

d

n

k

3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

notice stability of 7's and 9's

Running Time

- *What StableSort() will we use to sort on digits?*
- Counting sort is a good choice:
  - Sort n numbers on digits that range from 0.. k
  - Time: $O(n + k)$
- Each pass over ***n* numbers** with ***d* digits** takes time $O(n+k)$, so total time $O(dn + dk)$
  - When d is constant and $k = O(n)$, takes **O(*n*)** time

Radix Sort Speed

- How do we sort 1 million 64-bit numbers?
  - Yes just ***d*** the number of digits thus the iteration count changes
- Compares well with typical O(n log n) comparison sort
  - Requires approximately log n = 20 operations per number being sorted

Radix Sort Speed

- In general, radix sort based on counting sort is
  - asymptotically fast (i.e., O(n))
  - simple to code
  - a good choice
- To think about: *can radix sort be used on floating-point numbers?*
  - *Yes it can be*

Bucket Sort

Bucket-Sort(int[] A, int x, int y)

1. divide interval [x, y) into n equal-sized subintervals (buckets)

2. distribute the n input keys into the buckets

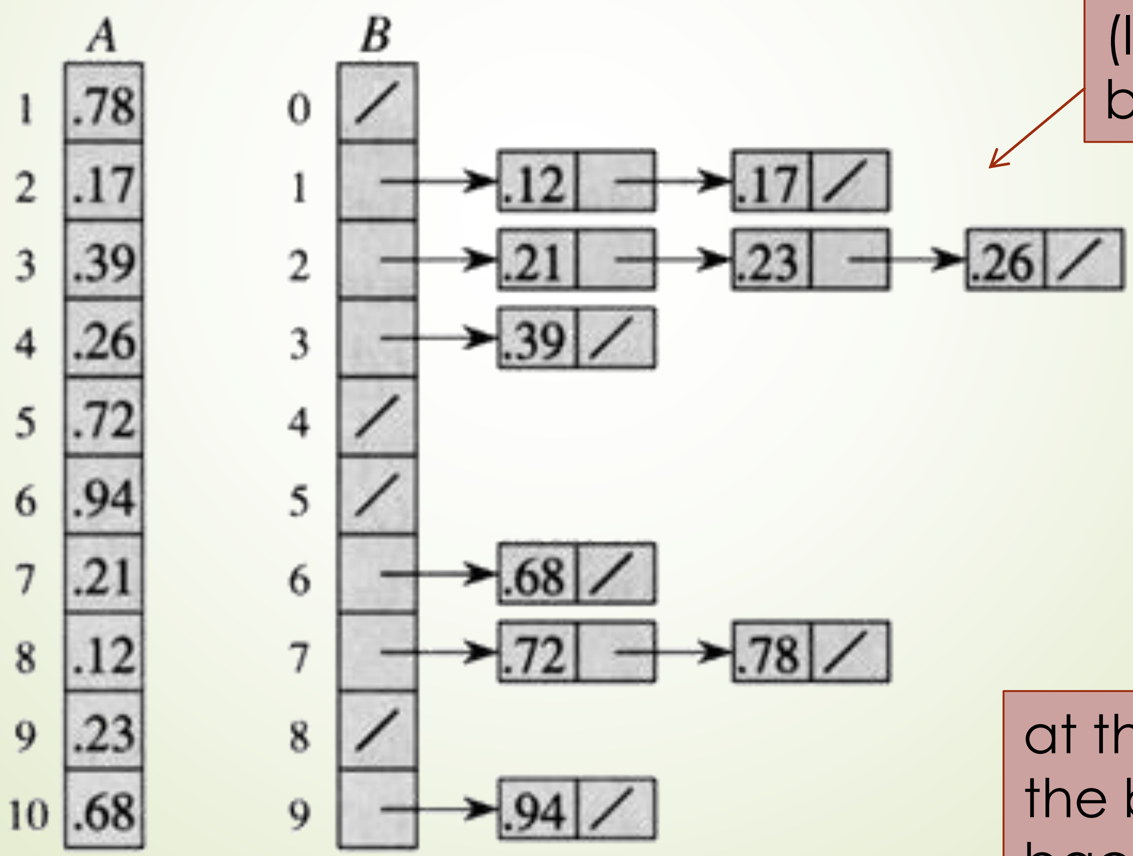
3. sort the numbers in each bucket (e.g., with insertion sort)

4. scan the (sorted) buckets in order and produce the output array (usually A)

Assumption: input elements are distributed uniformly over some known range, e.g., [0,1), so all the elements in A are greater than or equal to 0 but less than 1 .

Bucket Sort Example

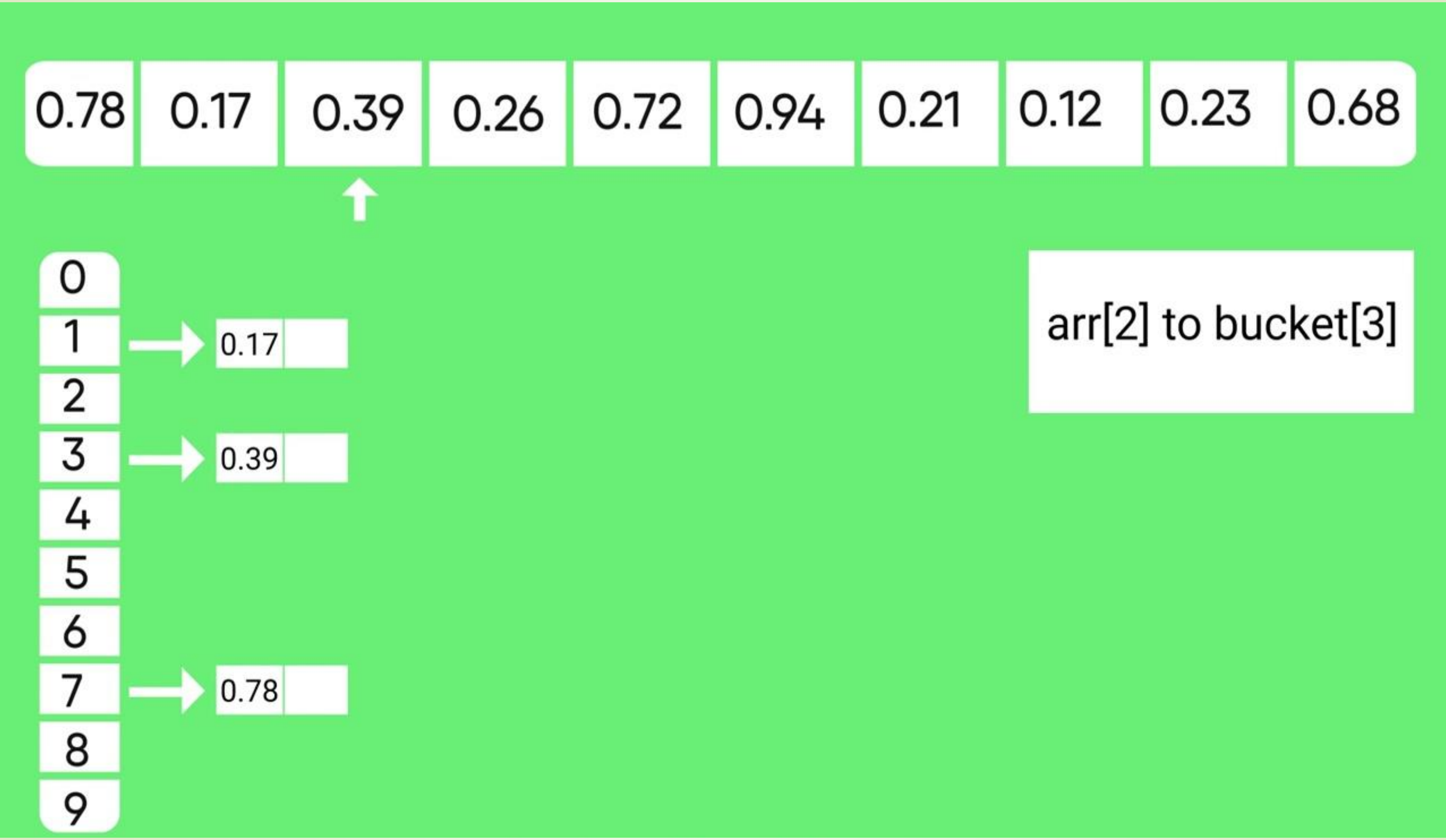
- Sort A[] into the ten 'buckets' in B[], assuming the data in A is in [0..1)

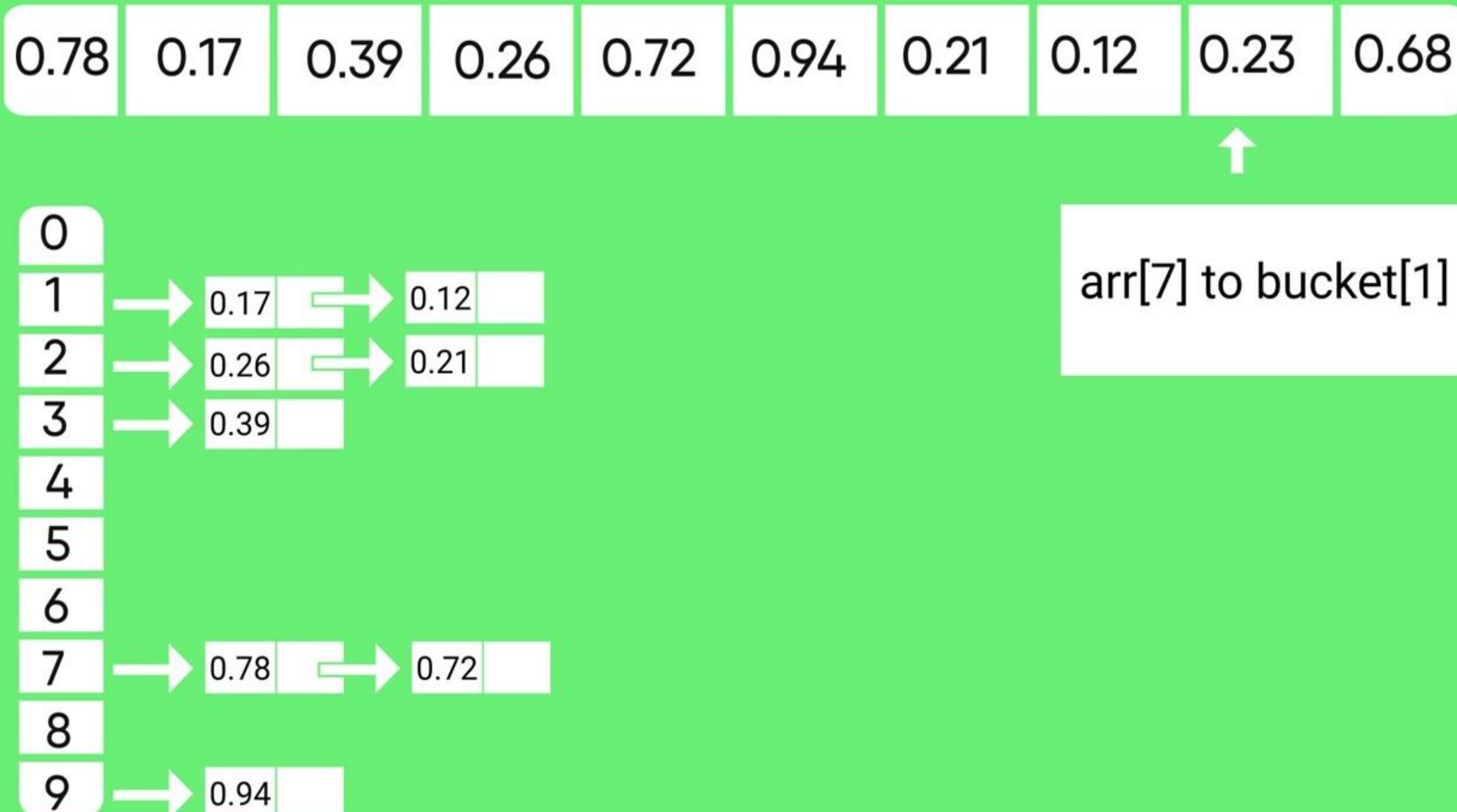


each bucket (list) must be sorted

at the end, copy the buckets (lists) back into A[]

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

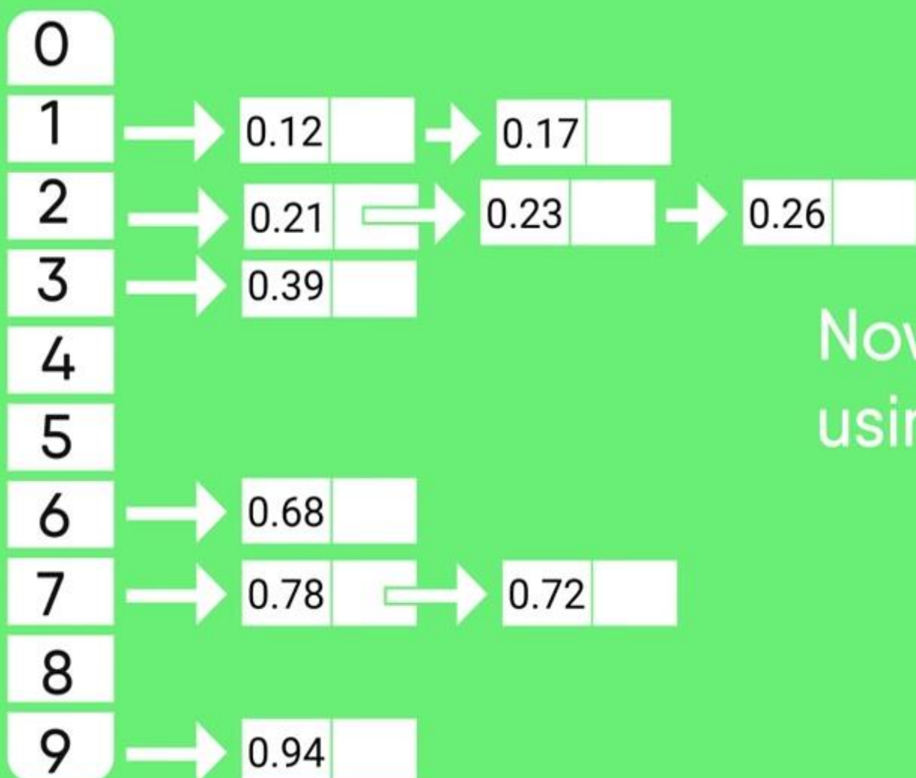

0.78
0.17
0.39
0.26
0.72
0.94
0.21
0.12
0.23
0.68
0
1
2
3
4
5
6
7
8
9
0.17
0.39
0.78
arr[2] to bucket[3]


0.78
0.17
0.39
0.26
0.72
0.94
0.21
0.12
0.23
0.68
arr[7] to bucket[1]
0
1
0.17
0.12
2
0.26
0.21
3
0.39
4
5
6
7
0.78
0.72
8
9
0.94

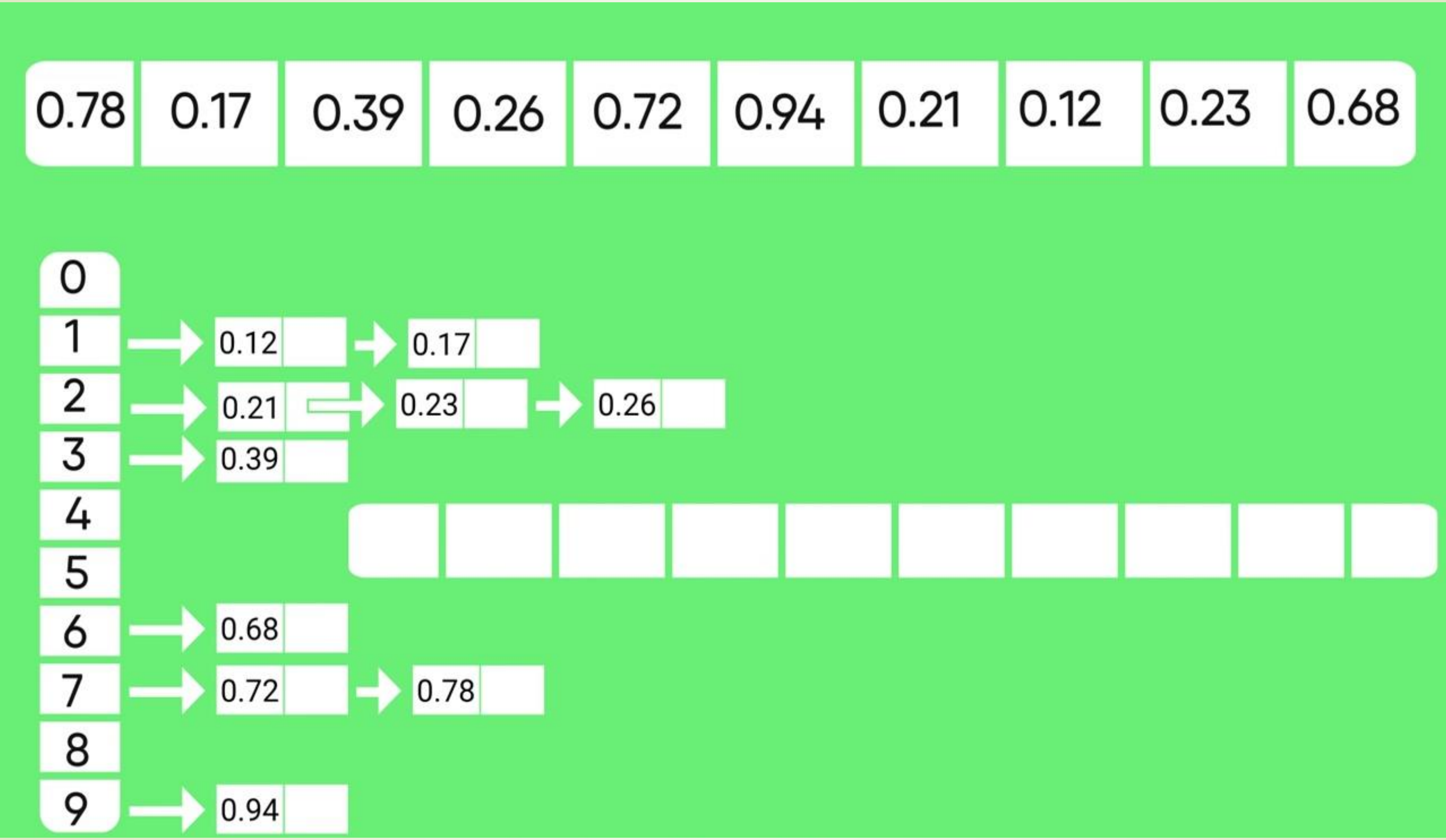
0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

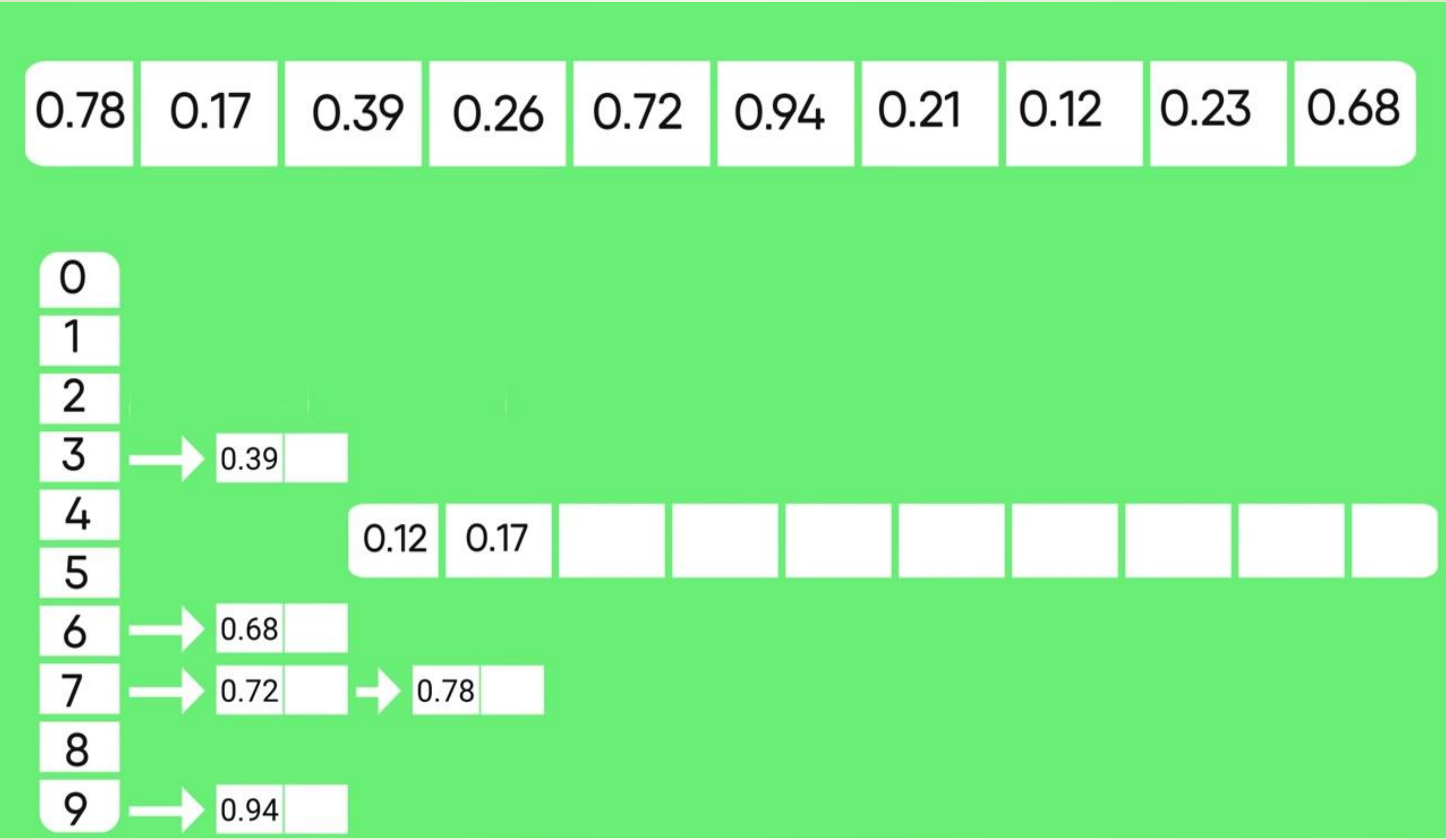
Now sort each bucket individually using insertion sort

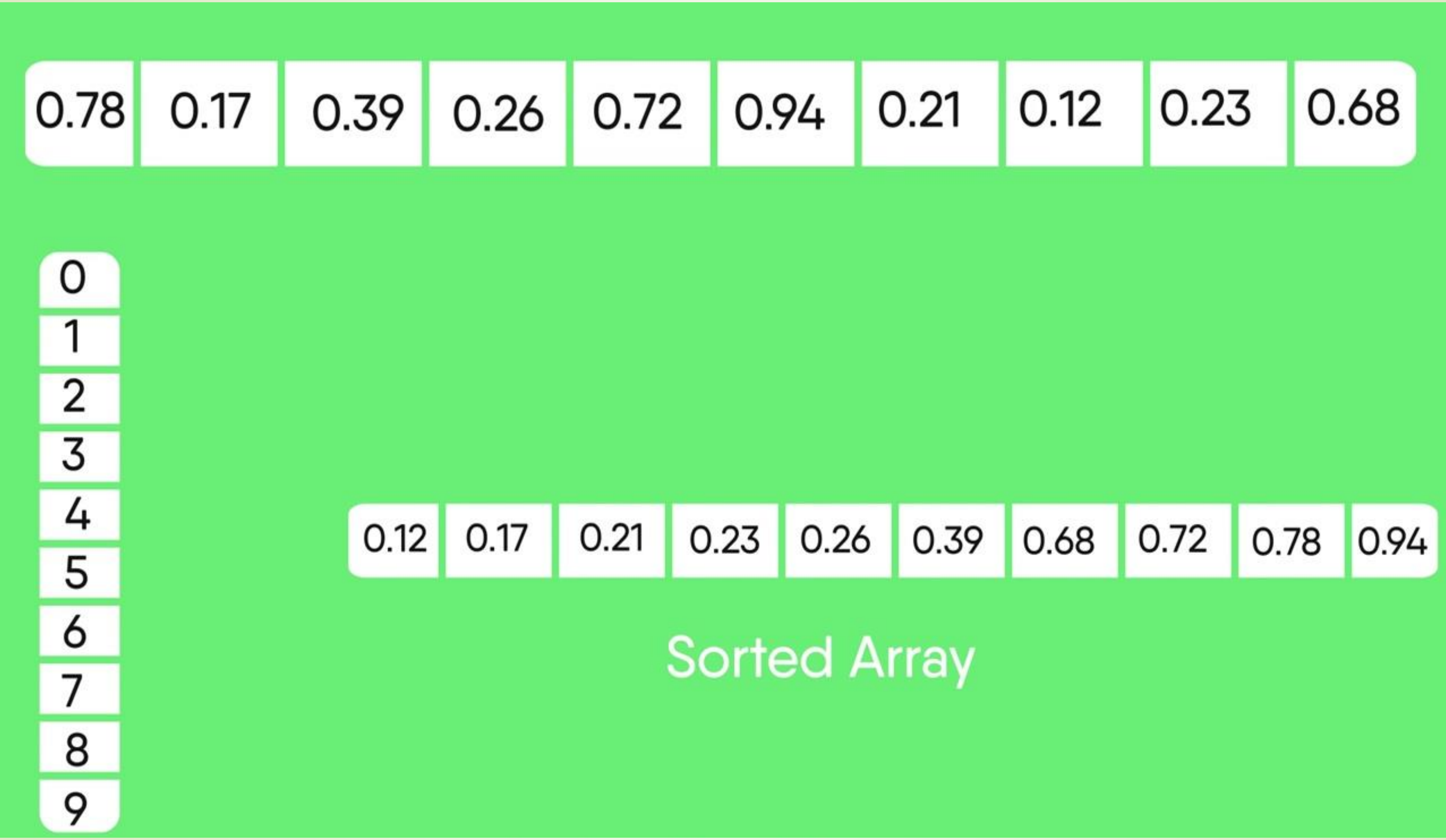
0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68



Now sort each bucket individually using insertion sort


0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
0
1
0.12
0.17
2
0.21
0.23
0.26
3
0.39
4
5
6
0.68
7
0.72
0.78
8
9
0.94


0.78
0.17
0.39
0.26
0.72
0.94
0.21
0.12
0.23
0.68
0
1
2
3
4
5
6
7
8
9
0.39
0.12
0.17
0.68
0.72
0.78
0.94


0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
0
1
2
3
4
5
6
7
8
9
0.12 0.17 0.21 0.23 0.26 0.39 0.68 0.72 0.78 0.94
Sorted Array

Bucket Sort Review

- Pro's:
  - Fast
  - asymptotically fast (i.e., **O(*n*)** when distribution is uniform)
  - simple to code
  - good for a rough sort
- Con's:
  - doesn't sort in place

Why use Bucketsort and not Quicksort?

- If we know **k** up front, and it is small (k << n) then bucket sort can efficiently run faster than Quicksort, since n*log(n), the average for Quicksort, will be more than (n + k), which is the average for bucket sort.
- I.e., **sortedList = (n*log(n) > n + k) ? bucketSort(list) : quicksort(list);**
- For example, bucketsort may be preferred for streams over quicksort because the max char to be ordered compared to the minimum value can be as small as the ASCII value of 0 to the ASCII value of z, hence reducing K to a number around 60

Summary of Linear Sorts
Non-Comparison Sorts

https://en.wikipedia.org/wiki/Sorting_algorithm

Name	Best	Average	Worst	Memory	Stable
Bucket sort (uniform keys)	—	$n+k$	$n^2 \cdot k$	$n \cdot k$	Yes
Bucket sort (integer keys)	—	$n+r$	$n+r$	$n+r$	Yes
Counting sort	—	$n+r$	$n+r$	$n+r$	Yes
LSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n+2^d$	Yes
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n+2^d$	Yes
MSD Radix Sort (in-place)	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	2^d	No