Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

Non-comparison sorting

- Non-comparison sorts use any arbitrary operations on the input elements (such as subtract, divide, …)

- These sorting algorithms assume that $\boldsymbol{n}$ input elements are in the range $[\mathbf{1}, \boldsymbol{k}]$

- Non-comparison sorts can beat the lower bound of $\Omega(n \log n)$ and sort in linear time (i.e. $O(n)$) **if** k is not too large.

Counting-Sort

- Assume input numbers are in range $[1, k]$
- We make array $C[1..k]$, and use it for counting.
- Example for $k = 8$:

A

4	3	1	3	5	3	7

C

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

Counting-Sort

- Assume input numbers are in range $[1, k]$
- We make array $C[1..k]$, and use it for counting.
- Example:

A

4	3	1	3	5	3	7

C

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

Counting-Sort

- Assume input numbers are in range $[1, k]$
- We make array $C[1..k]$, and use it for counting.
- Example:

A

4	3	1	3	5	3	7

C

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

Counting-Sort

- Assume input numbers are in range $[1, k]$
- We make array $C[1..k]$, and use it for counting.
- Example:

A

4	3	1	3	5	3	7

C

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

Counting-Sort

- Assume input numbers are in range $[1, k]$
- We make array $C[1..k]$, and use it for counting.
- Example:

A	4	3	1	3	5	3	7

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

COUNTING-SORT

- Assume input numbers are in range $[1, k]$
- We make array $C[1..k]$, and use it for counting.
- Example:

A

4	3	1	3	5	3	7

C

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

Counting-Sort

- Assume input numbers are in range $[1, k]$
- We make array $C[1..k]$, and use it for counting.
- Example:

A

4	3	1	3	5	3	7

C

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

Counting-Sort

- Assume input numbers are in range $[1, k]$
- We make array $C[1..k]$, and use it for counting.
- Example:

A

4	3	1	3	5	3	7

C

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

Counting-Sort

- In order to sort, we traverse array C and duplicate each number in A, as many times as it was counted.

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

sort A	1	3	3	3	4	5	7

Counting-Sort

```
Basic-Counting-Sort(A, k)
    Allocate array C[1..k]
    for i = 1 to A.length
        C[A[i]] = C[A[i]] + 1
    h = 1
    for i = 1 to k
        for j = 1 to C[i]
            A[h] = i
            h = h+1
```

Counting-Sort

- **Question:** what is the time and space complexity of this algorithm?

Counting-Sort

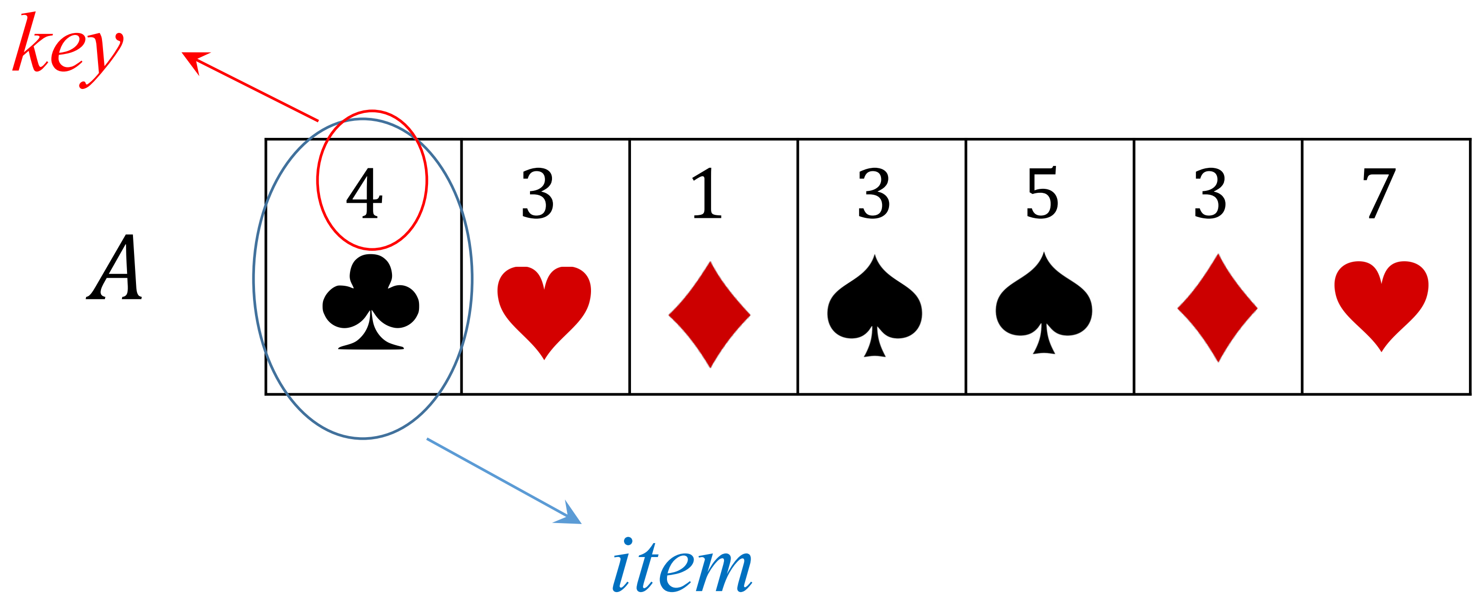
- **Question:** what is the time and space complexity of this algorithm?
- **Answer:** We need $O(k)$ additional memory for C. And the time to scan the array C and putting the elements back into A is $O\left(k + \sum_{i=1}^{k} C[i]\right) = O(n + k)$.

- Usually, we use this algorithm if $k = O(n)$ which results in $O(n)$ running time.

Useful Counting-Sort

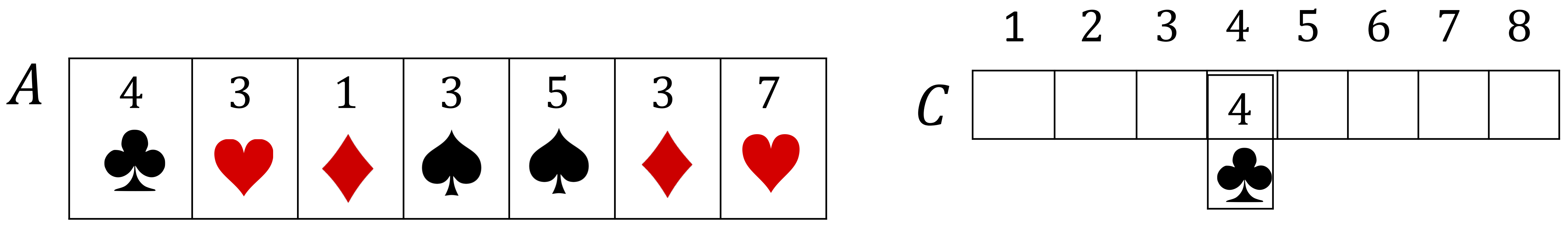
- In practice we are items that contain a lot of other information, along with a **key**.

- So, we should store different items with the **same key**.



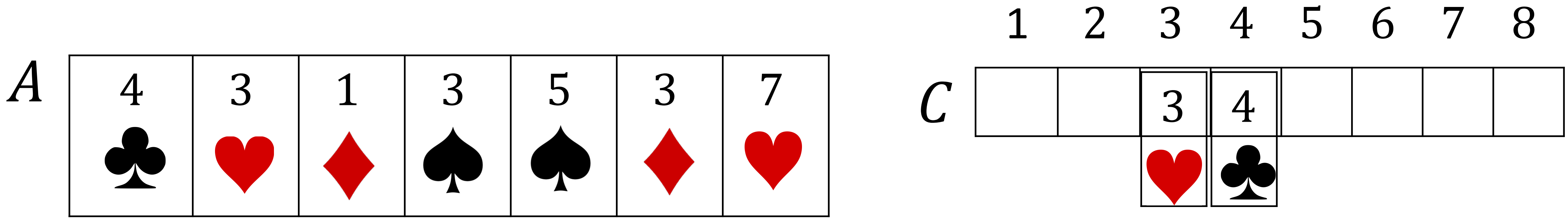
Useful Counting-Sort

- We define C to be an **array of lists**.
- Each time, we append an $item$ to the list at $C[item.key]$



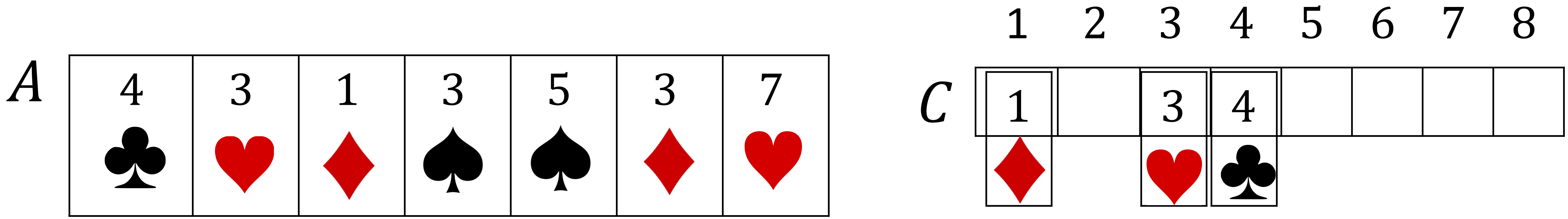
Useful Counting-Sort

- We define C to be an **array of lists**.
- Each time, we append an $item$ to the list at $C[item.key]$



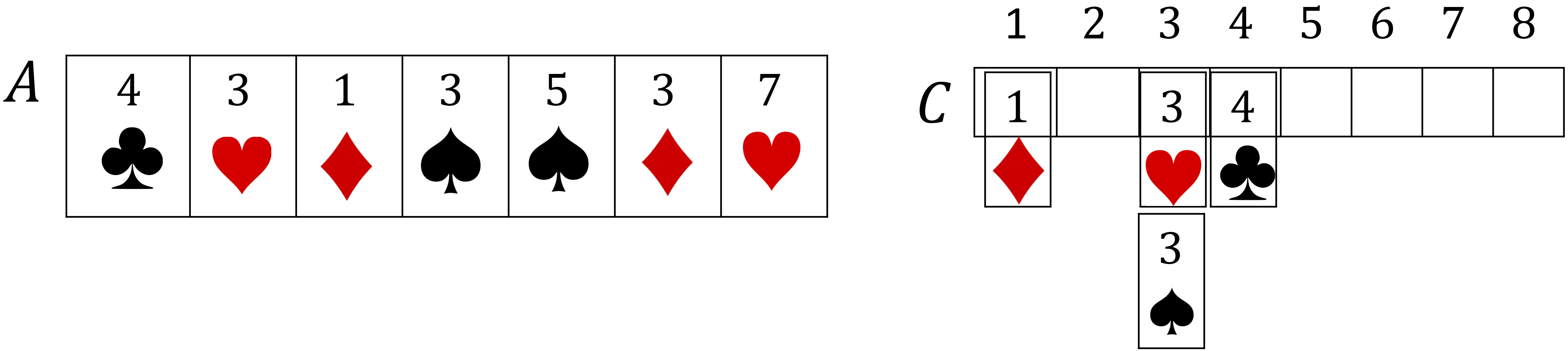
Useful COUNTING-SORT

- We define C to be an **array of lists**.
- Each time, we append an $item$ to the list at $C[item.key]$



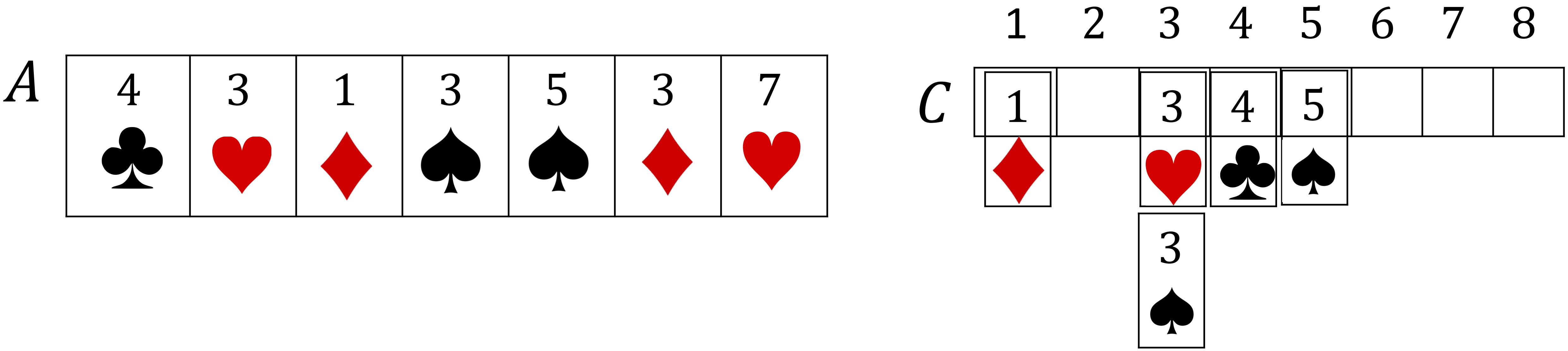
Useful Counting-Sort

- We define C to be an **array of lists**.
- Each time, we append an $item$ to the list at $C[item.key]$



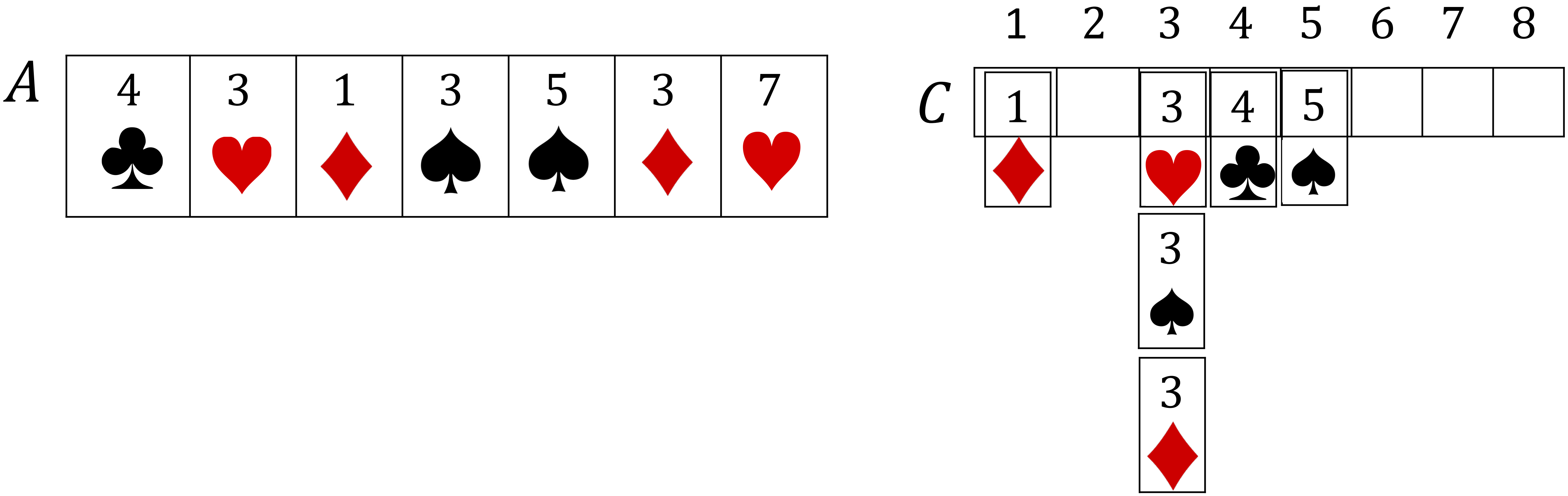
Useful Counting-Sort

- We define C to be an **array of lists**.
- Each time, we append an $item$ to the list at $C[item.key]$



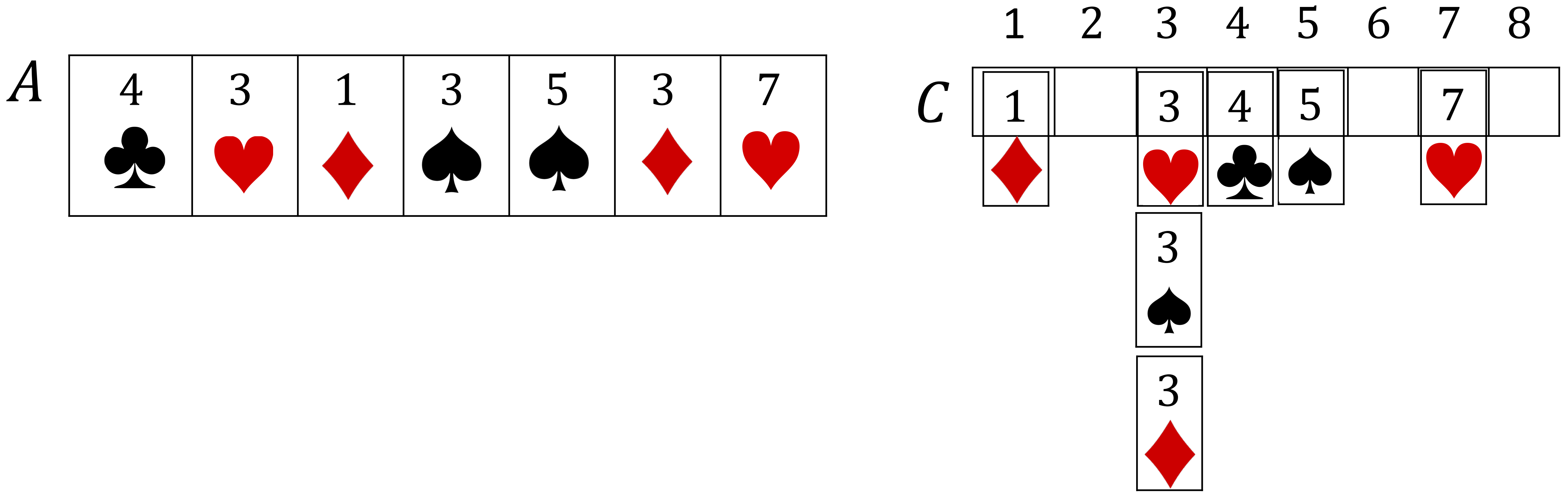
Useful Counting-Sort

- We define C to be an **array of lists**.
- Each time, we append an $item$ to the list at $C[item.key]$



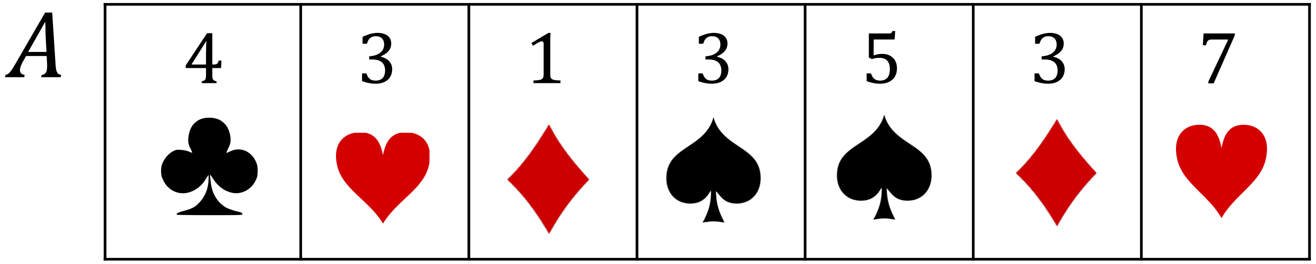
Useful COUNTING-SORT

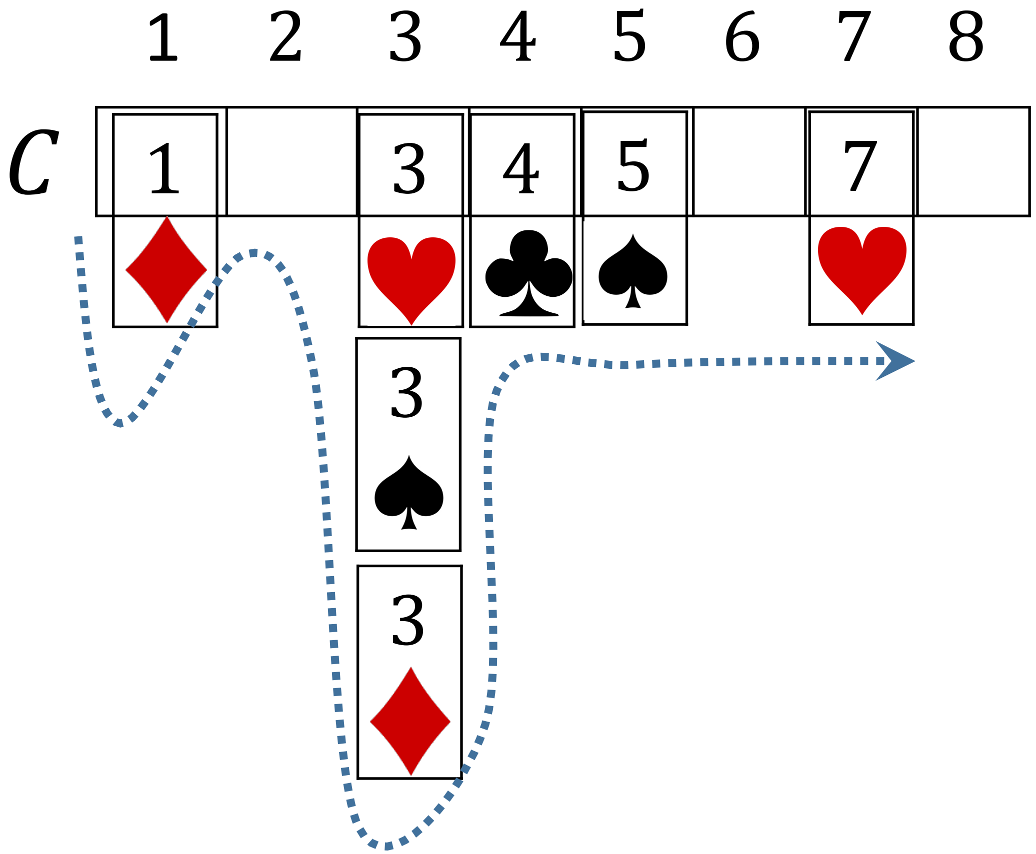
- We define C to be an **array of lists**.
- Each time, we append an $item$ to the list at $C[item.key]$



Useful COUNTING-SORT

- We define C to be an **array of lists**.
- Each time, we append an $item$ to the list at $C[item.key]$





- This is **stable** and still takes $O(n+k)$ time.
- But, the amount of additional memory is now $O(n+k)$.

Useful COUNTING-SORT

//Each element of A is an item with a field named *key*

COUNTING-SORT(A, k)

Allocate an array of lists $C[1..k]$
for $i = 1$ **to** $A.length$
    $key = A[i].key$
    append $A[i]$ to the end of the list $C[key]$ $O(1)$
$index = 1$
for $i = 1$ **to** k
    for $j = 1$ **to** $C[i].length$
        $A[index] = C[i][j]$ $O(n + k)$
        $index = index + 1$

RADIX-SORT

- Radix means **root**, or **base** in the mathematical sense.

- The idea is to use **each digit of the base** to sort rather than the whole key.

- RADIX-SORT can sort in linear time **even if** k is as large as $\boldsymbol{O(n^c)}$ for some constant c.

Digits:

3 2 1

329
457
657
839
436
720
355

Radix-Sort

- We show each key as a number in **base *b***.

- Example: $\mathrm{b} = 10$, and numbers in range $[1, 999]$

- Each number has $\lceil \log_{10} 999 \rceil = 3$ digits.

Digits:

3 2 1

329
457
657
839
436
720
355

Radix-Sort

- Radix-sort is based on two facts:

1. If the keys are in range $[1 \,..\, k]$, each key has $\lceil \log_b k \rceil =$ **$O(\log_b k)$ digits** in base b.

2. And each digit is **in range $[0, b-1]$.**

Digits:

3 2 1

329
457
657
839
436
720
355

RADIX-SORT(A, k)

for $i = 1$ **to** $\lceil \log_b k \rceil$

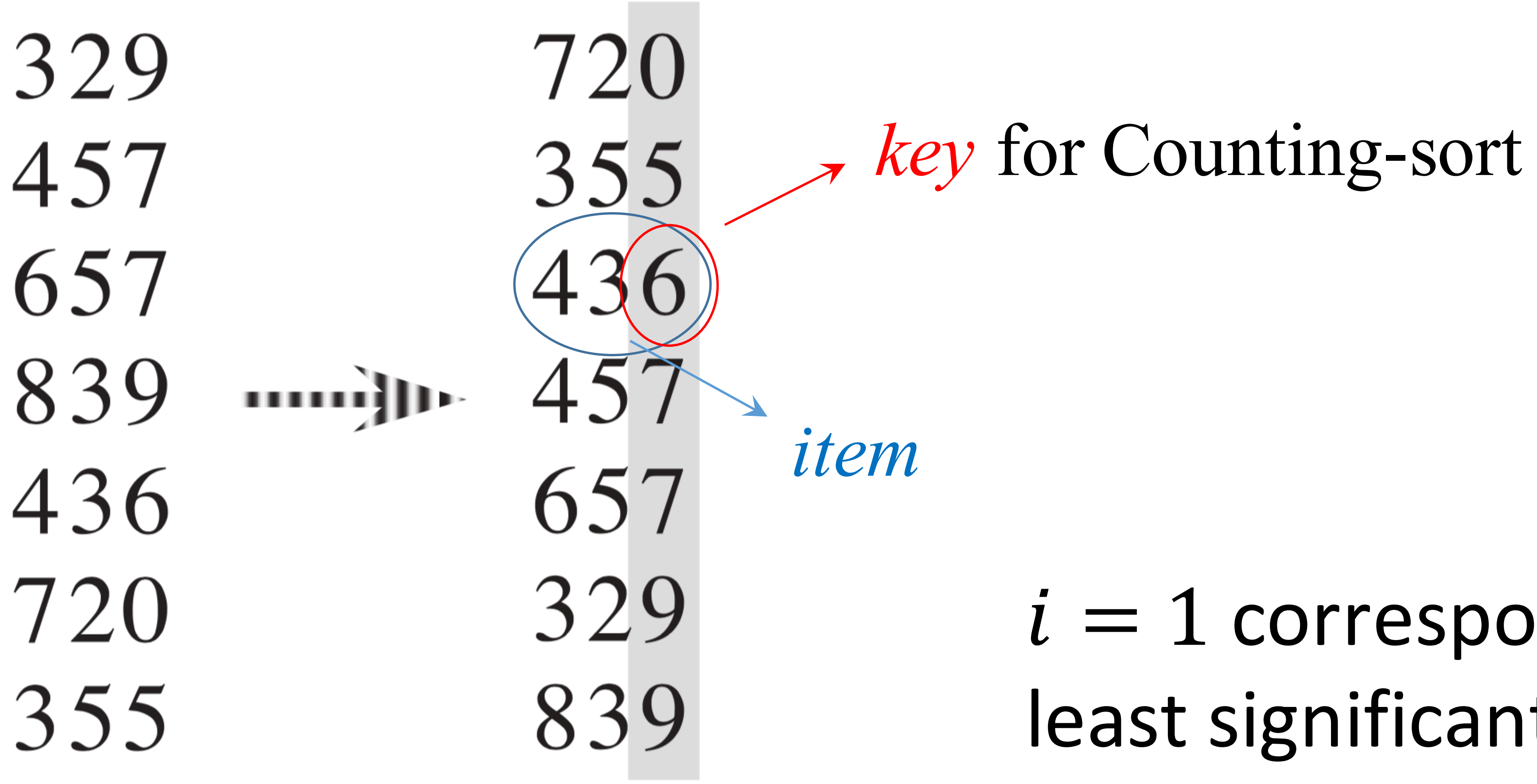
  use COUNTING-SORT to sort A $\overbrace{\text{on digit } i}^{\text{The key for sorting}}$

329
457
657
839
436
720
355

RADIX-SORT(A, k)

for $i = 1$ **to** $\lceil \log_b k \rceil$

  use COUNTING-SORT to sort A on digit i



$i = 1$ corresponds to the least significant digit

RADIX-SORT(A, k)

for $i = 1$ **to** $\lceil \log_b k \rceil$
    use COUNTING-SORT to sort A on digit i

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

RADIX-SORT(A, k)

```
for i = 1 to ⌈log_b k⌉
    use COUNTING-SORT to sort A on digit i
```

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

Radix-Sort

- Radix-sort works correctly **only if** the sort used inside the for loop is **stable**.

- Radix-sort is also stable itself.

- **Exercise:** Give an input of 2 integers for which if we **don't** use a stable sort inside Radix-Sort, the sorting goes wrong.

RADIX-SORT

- **Question:** What's the time complexity of RADIX-SORT if $A.length = n$?

Radix-Sort

- **Question:** What's the time complexity of Radix-Sort if $A.length = n$?

- **Answer:** $O\big((n+b)\log_b k\big)$:

1. The for loop has $O(\log_b k)$ iterations
2. For each digit i the key used for counting-sort is in range $[0, b-1]$. Assuming that we can allocate array $C[0\,..\,b-1]$ (from 0), counting-sort takes $O(n+b)$.

Radix-Sort

- We know that $\boldsymbol{k = O(n^c)}$ and $\boldsymbol{\log_b n^c = c \log_b n}$

$$O((n+b)\log_b k) = O((n+b)\cdot \log_b n)$$

Radix-Sort

- We know that $k = O(n^c)$ and $\log_b n^c = c \log_b n$

$$O((n+b)\log_b n^c) = O((n+b) \cdot \log_b n)$$

- **Question:** What choice of b makes $O\big((n+b)\log_b n\big)$ linear in n?

Radix-Sort

- We know that $k = O(n^c)$ and $\log_b n^c = c \log_b n$

$$O((n+b)\log_b n^c) = O((n+b) \cdot \log_b n)$$

- **Question:** What choice of b makes $O\big((n+b)\log_b n\big)$ linear in n?

- **Answer:** If we pick $b = n$, then $O\big((n+b)\log_b n\big) = O(n)$.

Disadvantages of Radix-Sort

1. It's not in-place since it is using $O(b) = O(n)$ auxiliary memory in the Counting-Sort

2. It's inflexible since it can only sort numbers. In a comparison sorting algorithm, however, you may easily sort strings, etc.

Bucket Sort

- Bucket sort is another sorting algorithm that works in **average-case** linear time.
- However, bucket sort assumes that each key in the input array is chosen **independently and uniformly at random** from the range $[\boldsymbol{a}, \boldsymbol{b})$.
- Note that a is inclusive and b is exclusive, and there are $b - a$ numbers in this range.

Bucket Sort

- This means that for any index i, $A[i]$ is **equally likely** to be any of the numbers $a, a+1, \dots, b-1$.

- Each number appears with probability of $\boldsymbol{\frac{1}{b-a}}$

- The **good thing** is that now we can sort in linear time even when the input range is **asymptotically bigger than $\boldsymbol{n^c}$**

Bucket Sort

- For example: say $n = 10$ numbers are chosen from $[100, 200)$

$$178, 117, 139, 126, 172, 194, 121, 112, 123, 168$$

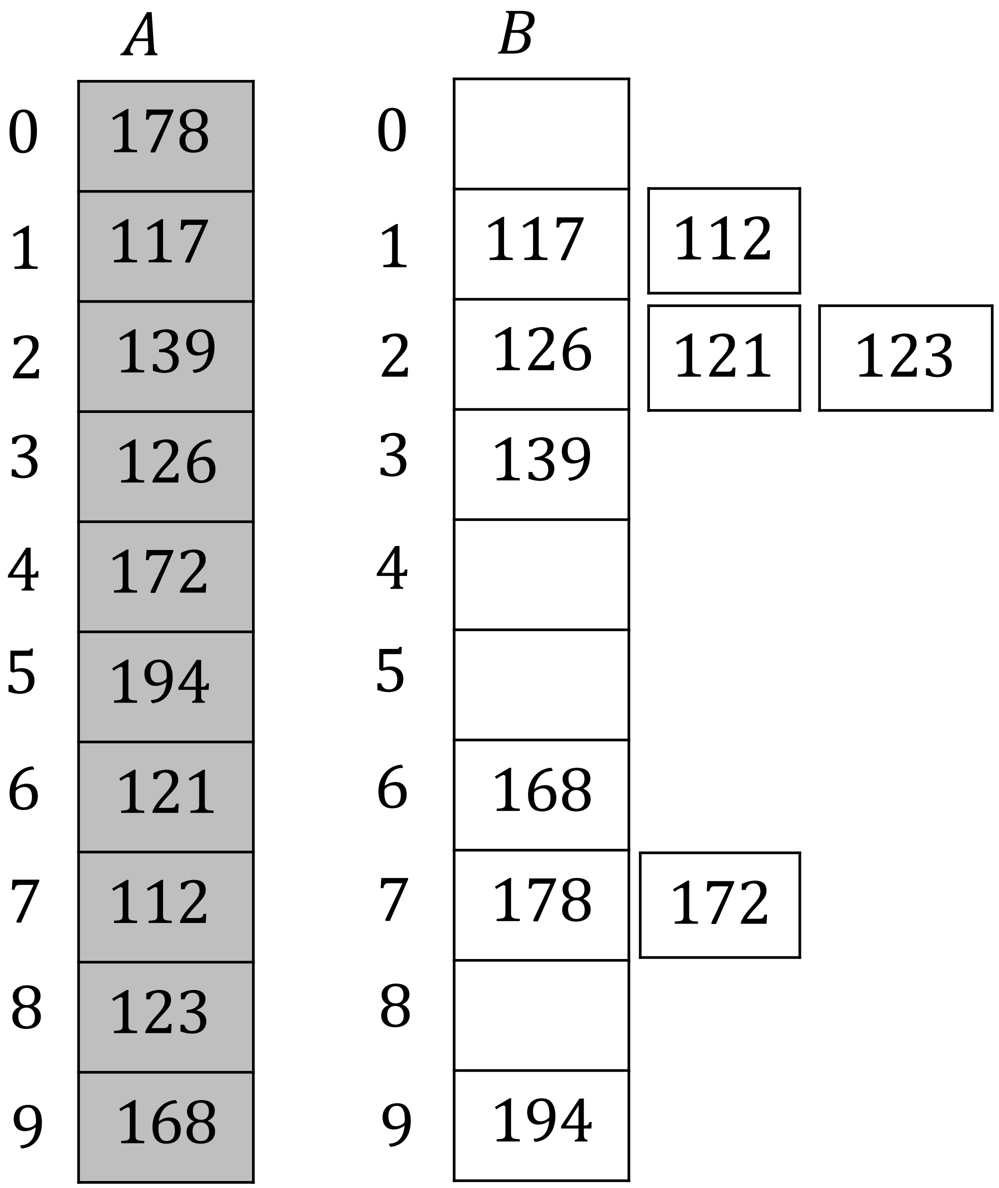
- From here, we switch to 0-based indices.

Bucket Sort

	A		B	
0	178	0		$[100, 110)$
1	117	1		$[110, 120)$
2	139	2		$[120, 130)$
3	126	3		
4	172	4		
5	194	5		
6	121	6		
7	112	7		
8	123	8		
9	168	9		$[190, 200)$

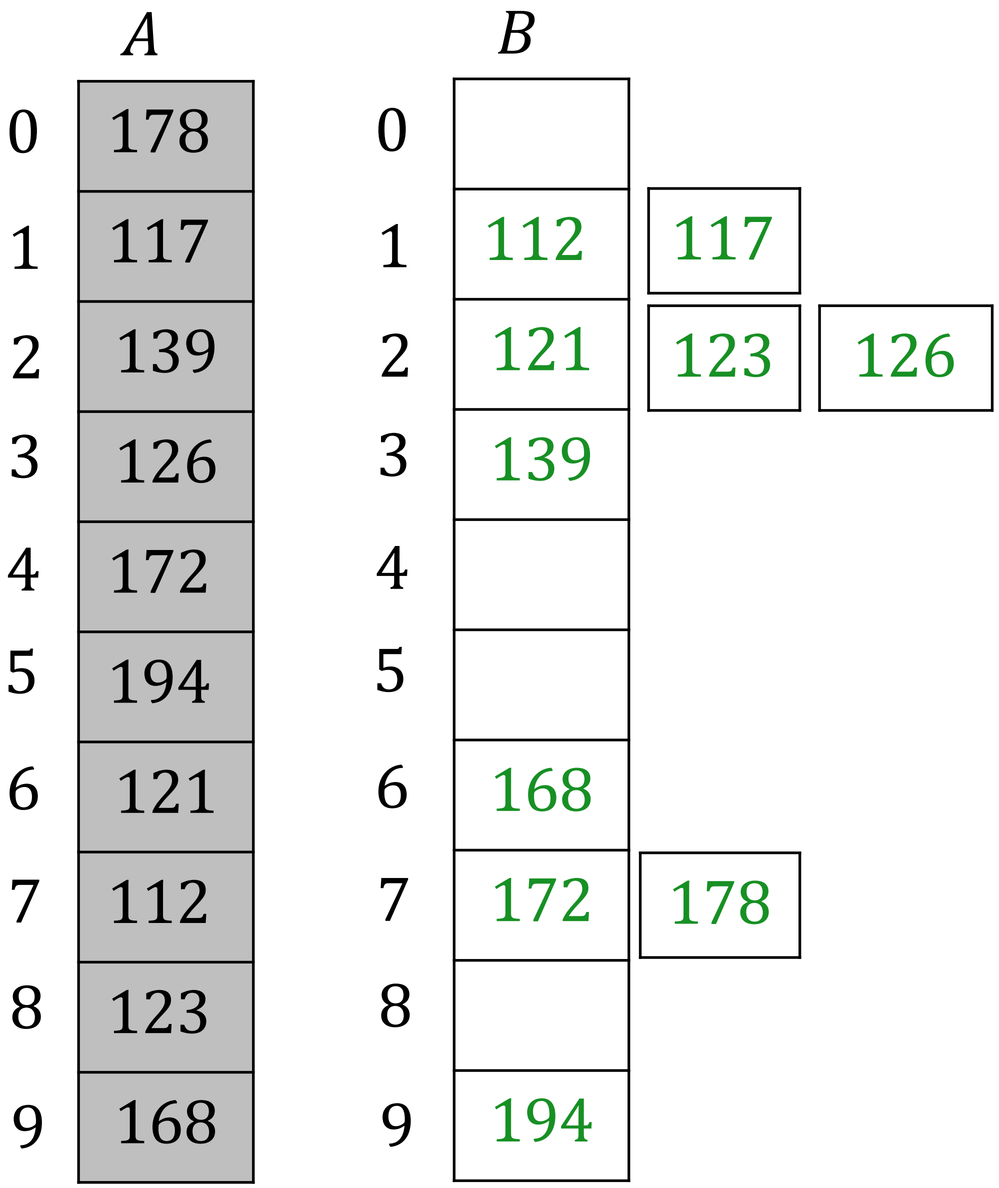
- We make n buckets where each bucket covers a range of $\frac{b-a}{n}$ numbers.
- Here, $\frac{b-a}{n} = 10$.
- $B[i]$ is a **list** of numbers.

Bucket Sort



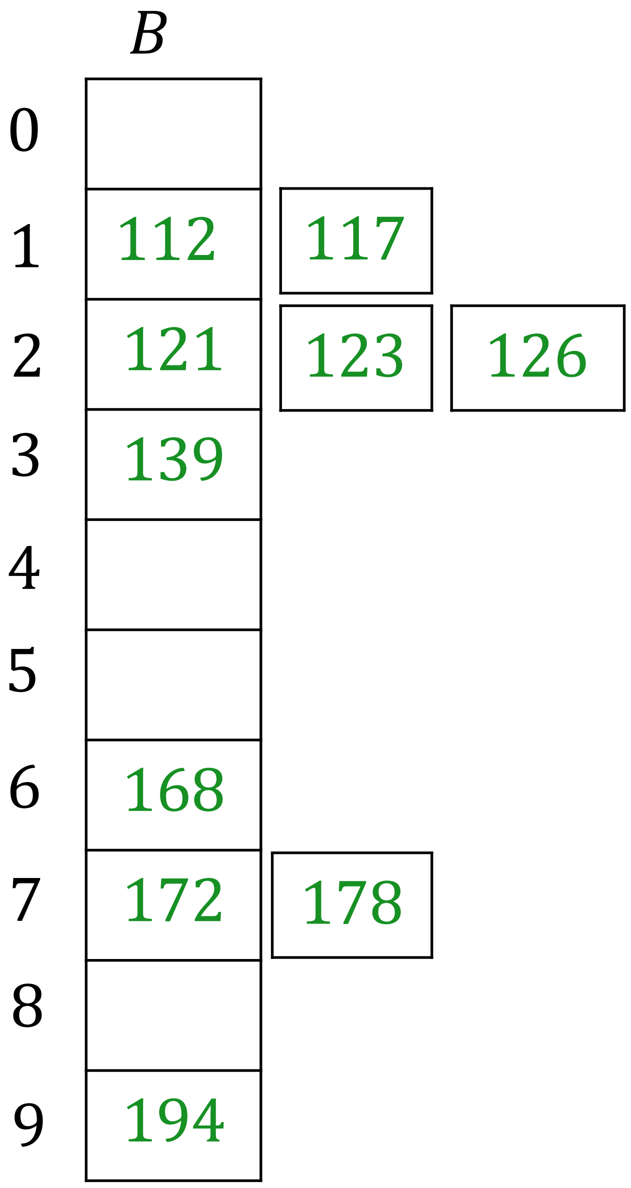
- The element with key k is appended to the list B $\left\lfloor n \cdot \frac{k-a}{b-a} \right\rfloor$
- This means that if $A[i]$ is random from $[a, b)$, it goes with equal probability to any of the n buckets.

Bucket Sort

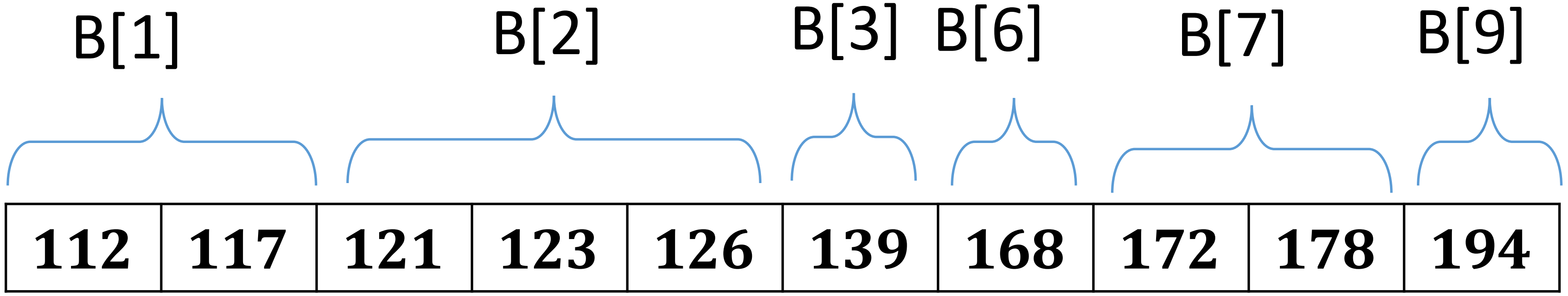


- Then, we sort each bucket using **insertion-sort**

Bucket Sort



- Finally, we concatenate all buckets in order



Bucket Sort

BUCKET-SORT(A, a, b)

$n = A.length$
//A and B start from 0
Allocate an array of lists $B[0..n\text{-}1]$
for $i = 0$ **to** n-1
    $k = A[i].key$
    $bucket = \left\lfloor n \cdot \frac{k-a}{b-a} \right\rfloor$
    append $A[i]$ to the list $B[bucket]$
for $i = 1$ **to** n
    sort $B[i]$ using insertion-sort
concatenate the lists $B[0]$, $B[1]$, …, $B[n\text{-}1]$

Bucket Sort

```
BUCKET-SORT(A, a, b)
n = A.length
//A and B start from 0
Allocate an array of lists B[0..n-1]
for i = 0 to n-1
    k = A[i].key
    bucket = ⌊n · (k−a)/(b−a)⌋
    append A[i] to the list B[bucket]
for i = 1 to n
    sort B[i] using insertion-sort
concatenate the lists B[0], B[1], …, B[n-1]
```

Question: What is the worst-case running time?

Bucket Sort

BUCKET-SORT(A, a, b)

$n = A.length$
//A and B start from 0
Allocate an array of lists $B[0..n\text{-}1]$
for $i = 0$ **to** $n\text{-}1$
    $k = A[i].key$
    $bucket = \left\lfloor n \cdot \frac{k-a}{b-a} \right\rfloor$
    append $A[i]$ to the list $B[bucket]$
for $i = 1$ **to** n
    sort $B[i]$ using insertion-sort
concatenate the lists $B[0]$, $B[1]$, …, $B[n\text{-}1]$

Question: What is the worst-case running time?

Answer: The worst-case is when all items go to the same bucket. It takes $O(n^2)$ using insertion-sort and $O(n \log n)$ using any other algorithm.

Analysis

- **Question:** What kind of analysis should I do for this:
  1. Worst-case expected running time?
  2. Average-case running time?
  3. Worst-case analysis?

Analysis

- **Question:** What kind of analysis should I do for this:
  1. Worst-case expected running time?
  2. Average-case running time?
  3. Worst-case analysis?

- **Answer:** This algorithm is not randomized so we don't do expected running time analysis. On the other hand, the worst-case input is too bad. Since we know that the elements of the input array are chosen from a uniform distribution, average-case analysis is our best bet.

Analysis

BUCKET-SORT(A, a, b)

$n = A.length$

//A and B start from 0

Allocate an array of lists $B[0..n\text{-}1]$

for $i = 0$ **to** n-1

  $k = A[i].key$

  $bucket = \left\lfloor n \cdot \frac{k-a}{b-a} \right\rfloor$

  append $A[i]$ to the list $B[bucket]$

$\Theta(n)$

for $i = 1$ **to** n

  sort $B[i]$ using insertion-sort

$\sum_{i=0}^{n-1} O(n_i^2)$

concatenate the lists $B[0]$, $B[1]$, …, $B[n\text{-}1]$ $\Theta(n)$

Analysis

- So, we have

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

- Here, n_i is random. So, we use expected value.
- **However,** n_i depends **on the input**, and the algorithm is not randomized.
- So, $E[T(n)]$ should be interpreted as the **average-case running time** rather than the expected running time.

Analysis

- So, we have

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

- It can be proved $E(T(n))$ that $E(T(n))$ is $\Theta(n)$.

- The proof is optional and can be found in section 8.4.