


Chapter 8-2: Sorting in Linear Time

About this lecture

- Sorting algorithms we studied so far
 - Insertion, Merge, Heapsort, Quicksort
 - ➔ determine sorted order by **comparison**
- We will look at 3 new sorting algorithms
 - Counting Sort, Radix Sort, Bucket Sort
 - ➔ assume some properties on the input, and determine the sorted order by **counting**

Counting Sort

extra info
on values



- Input: Array $A[1..n]$ of n integers, each has value from $[0,k]$
- Output: Sorted array of the n integers
- Idea 1: Create $B[1..n]$ to store the output
- Idea 2: Process $A[1..n]$ from right to left
 - ✓ Use $k + 2$ counters:
 - One for "which element to process"
 - $k + 1$ for "where to place"

Counting Sort (Step 1)

1. Initialize $c[0], c[1], \dots, c[k]$ to 0
2. /* First, set $c[j] = \#$ elements with value j */
 - (1.1) For $x = 1, 2, \dots, n$, increase $c[A[x]]$ by 1
3. /* Set $c[j] =$ the number of elements less than or equal to j (iteratively) */
 - (1.2) For $y = 1, 2, \dots, k$, $c[y] = c[y-1] + c[y]$

Time for Step 1 = $O(n + k)$

Counting Sort (Step 2)

```
/* Process A from right to left */  
For  $x = n, n-1, \dots, 2, 1$   
{  /* Process next element */  
     $B[c[A[x]]] = A[x];$   
    /* Update counter */  
    Decrease  $c[A[x]]$  by 1;  
}
```

Time for Step 2 = $O(n)$

Counting Sort (Details)

Before Running

A

2	1	2	5	3	3	1	2
---	---	---	---	---	---	---	---

next element

k+1 counters

$c[0] = 0$, $c[1] = 0$, $c[2] = 0$,

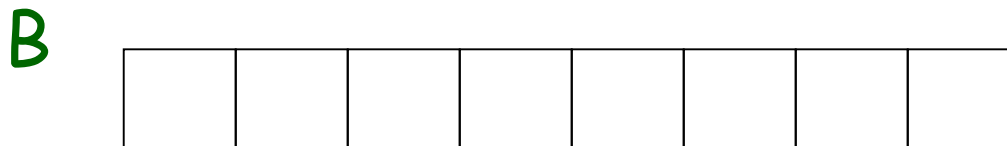
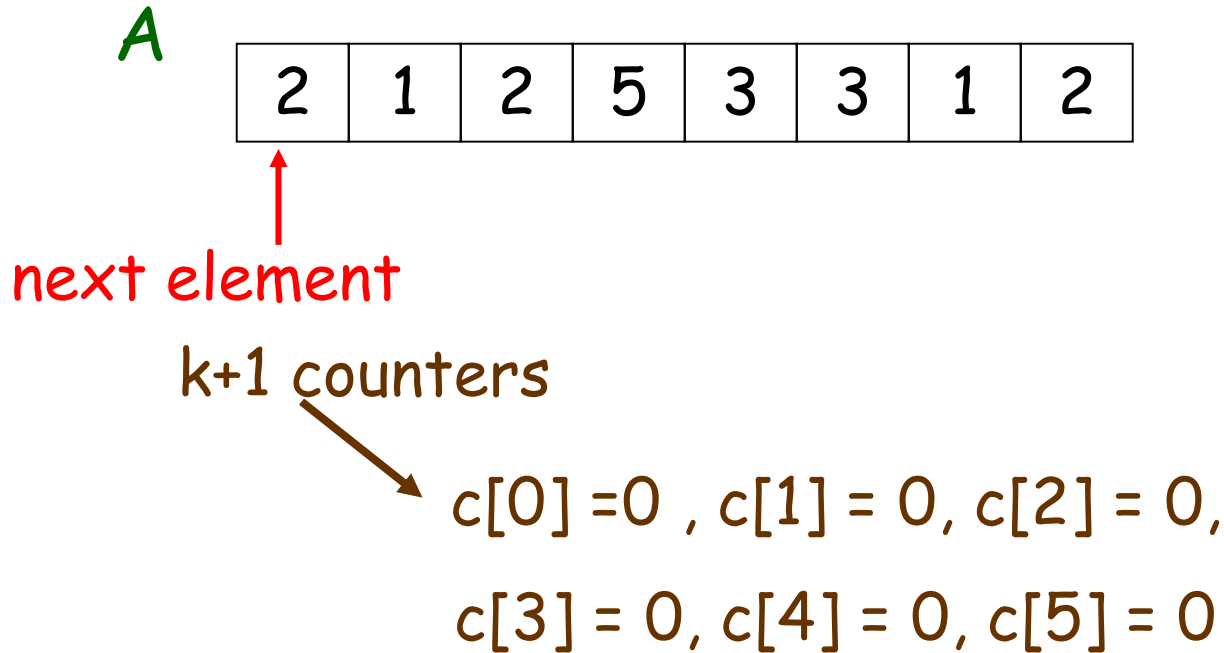
$c[3] = 0$, $c[4] = 0$, $c[5] = 0$

B

--	--	--	--	--	--	--	--

Counting Sort (Details)

Step 1.1: For $x = 1, 2, \dots, n$, increase $c[A[x]]$ by 1



Counting Sort (Details)

Step 1.1: For $x = 1, 2, \dots, n$, increase $c[A[x]]$ by 1

A

2	1	2	5	3	3	1	2
---	---	---	---	---	---	---	---



next element

k+1 counters



$c[0] = 0$, $c[1] = 0$, $c[2] = 1$,

$c[3] = 0$, $c[4] = 0$, $c[5] = 0$

B

--	--	--	--	--	--	--	--

Counting Sort (Details)

Step 1.1: For $x = 1, 2, \dots, n$, increase $c[A[x]]$ by 1

A

2	1	2	5	3	3	1	2
---	---	---	---	---	---	---	---



next element

k+1 counters



$c[0] = 0$, $c[1] = 1$, $c[2] = 1$,

$c[3] = 0$, $c[4] = 0$, $c[5] = 0$

B

--	--	--	--	--	--	--	--

Counting Sort (Details)

Step 1.1: For $x = 1, 2, \dots, n$, increase $c[A[x]]$ by 1

A

2	1	2	5	3	3	1	2
---	---	---	---	---	---	---	---

↑
next element

k+1 counters



$c[0] = 0$, $c[1] = 2$, $c[2] = 3$,

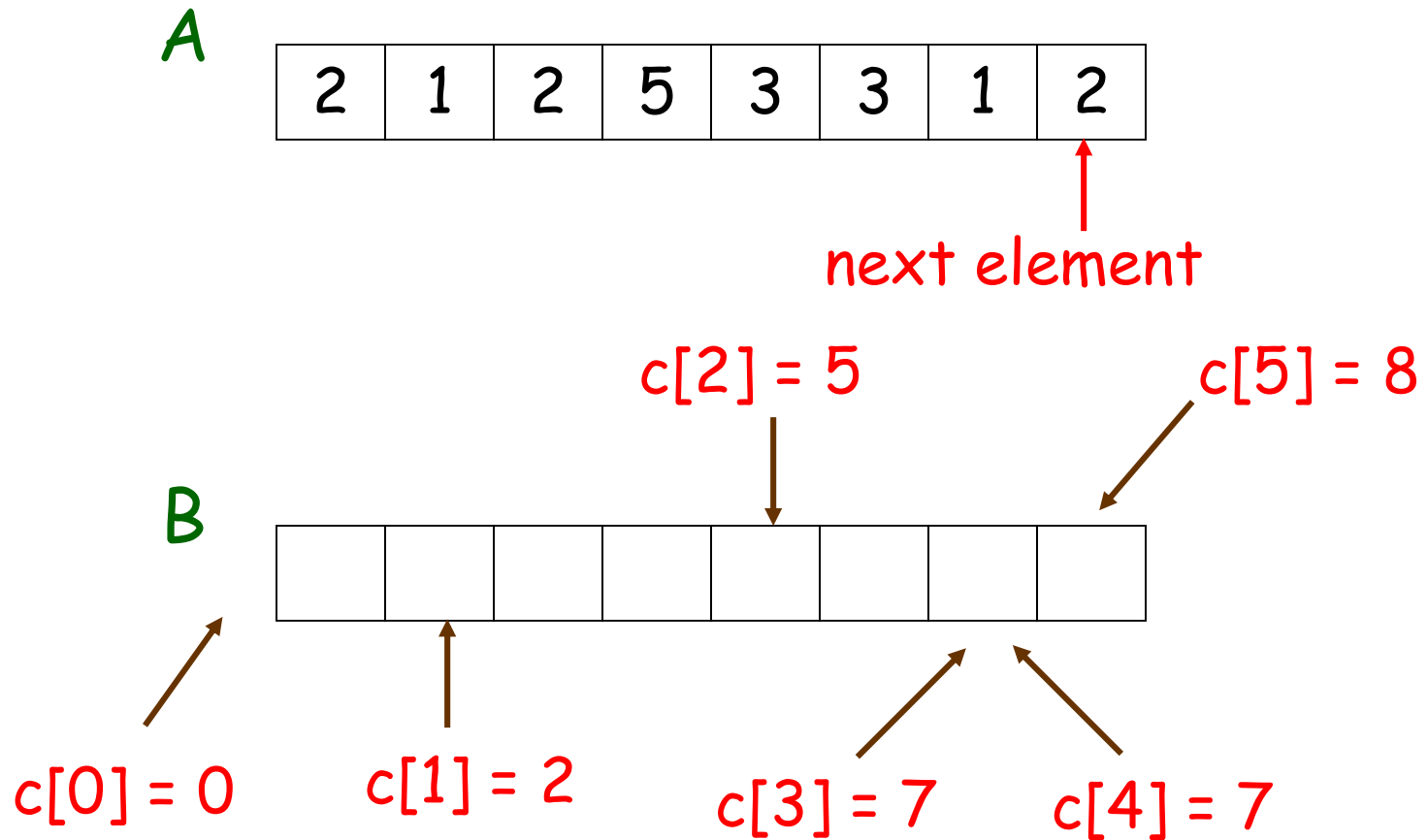
$c[3] = 2$, $c[4] = 0$, $c[5] = 1$

B

--	--	--	--	--	--	--	--

Counting Sort (Details)

Step 1.2: For $y = 1, 2, \dots, k$, $c[y] = c[y-1] + c[y]$



Counting Sort (Details)

Step 2: Process **next element** of **A** and update corresponding counter

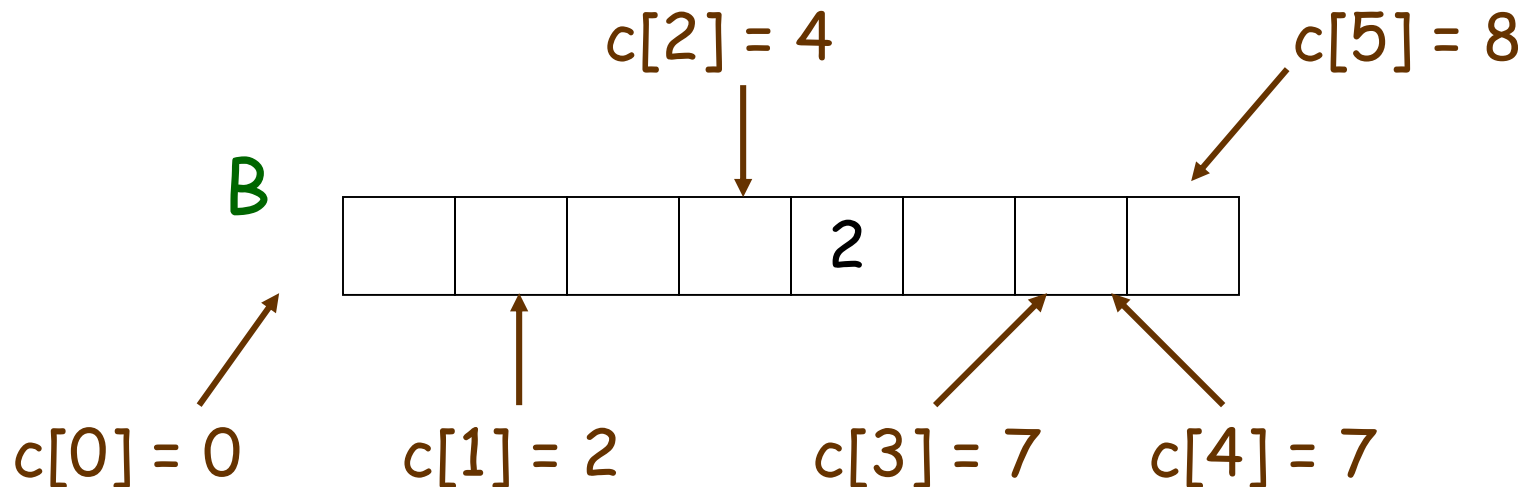
A

2	1	2	5	3	3	1	2
---	---	---	---	---	---	---	---

next element

B

				2			
--	--	--	--	---	--	--	--



Counting Sort (Details)

Step 2: Process **next element** of **A** and update corresponding counter

A

2	1	2	5	3	3	1	2
---	---	---	---	---	---	---	---

next element

$c[2] = 4$

$c[5] = 8$

B

	1			2			
--	---	--	--	---	--	--	--

$c[0] = 0$ $c[1] = 1$

$c[3] = 7$ $c[4] = 7$

Counting Sort (Details)

Step 2: Process **next element** of **A** and update corresponding counter

A

2	1	2	5	3	3	1	2
---	---	---	---	---	---	---	---

next element

$c[2] = 4$

$c[5] = 8$

B

	1			2		3	
--	---	--	--	---	--	---	--

$c[0] = 0$ $c[1] = 1$

$c[3] = 6$ $c[4] = 7$

Counting Sort (Details)

Step 2: Process **next element** of **A** and update corresponding counter

A

2	1	2	5	3	3	1	2
---	---	---	---	---	---	---	---

next element

$c[2] = 4$

$c[5] = 8$

B

	1			2	3	3	
--	---	--	--	---	---	---	--

$c[0] = 0$

$c[1] = 1$

$c[3] = 5$

$c[4] = 7$

Counting Sort (Details)

Step 2: Process **next element** of **A** and update corresponding counter

A

2	1	2	5	3	3	1	2
---	---	---	---	---	---	---	---

↑
next element

$c[2] = 4$

$c[5] = 7$

B

	1			2	3	3	5
--	---	--	--	---	---	---	---

$c[0] = 0$ $c[1] = 1$ $c[3] = 5$ $c[4] = 7$

Counting Sort (Details)

Step 2: Process **next element** of **A** and update corresponding counter

A

2	1	2	5	3	3	1	2
---	---	---	---	---	---	---	---

next element

$c[2] = 3$

$c[5] = 7$

B

	1		2	2	3	3	5
--	---	--	---	---	---	---	---

$c[0] = 0$

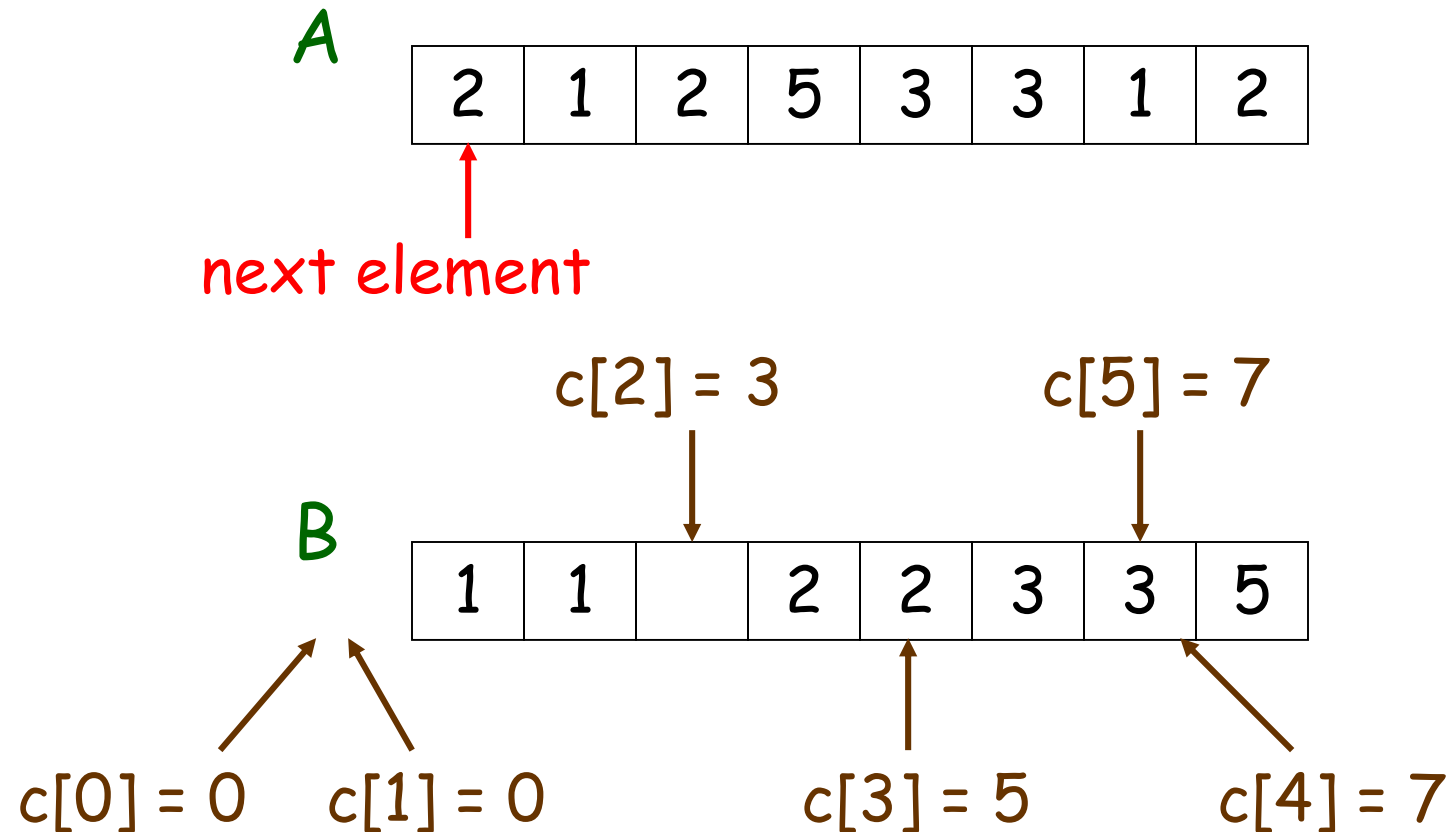
$c[1] = 1$

$c[3] = 5$

$c[4] = 7$

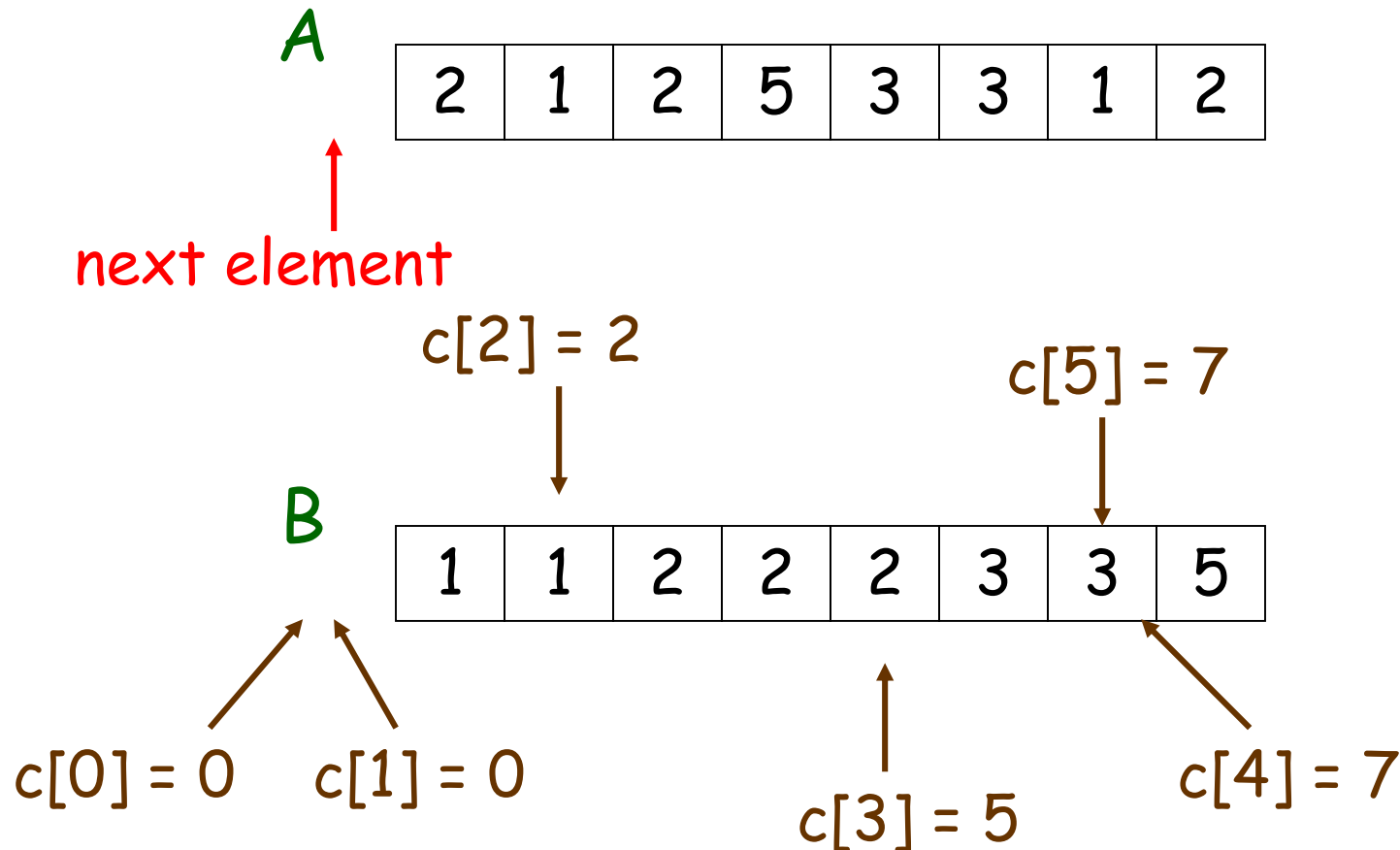
Counting Sort (Details)

Step 2: Process **next element** of **A** and update corresponding counter



Counting Sort (Details)

Step 2: Done when **all elements** of **A** are processed



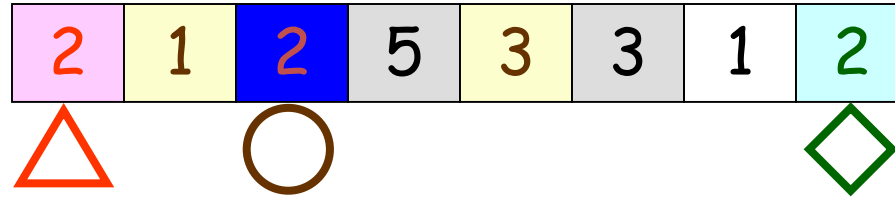
Counting Sort (Running Time)

Conclusion:

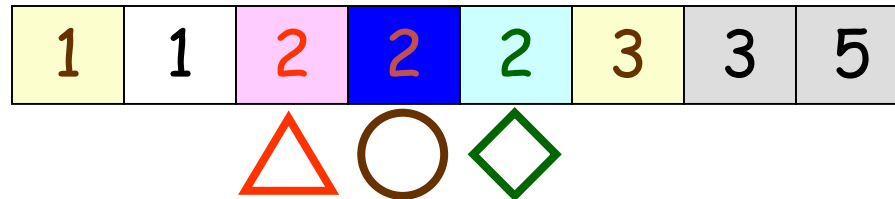
- Running time = $O(n + k)$
 - if $k = O(n)$, time is (asymptotically) optimal
- Counting sort is also **stable** :
 - elements with same value appear in same order in before and after sorting

Stable Sort

Before
Sorting




After
Sorting



Radix Sort

extra info
on values



- Input: Array $A[1..n]$ of n integers, each has d digits, and each digit has value from $[0, k]$
- Output: Sorted array of the n integers
- Idea: Sort in d rounds
 - At Round j , stable sort A on digit j (where rightmost digit = digit 1)

Radix Sort (Example Run)

Before Running

1 9 0 4

2 5 7 9

1 8 7 4

6 3 5 5

4 4 3 2

8 3 1 8

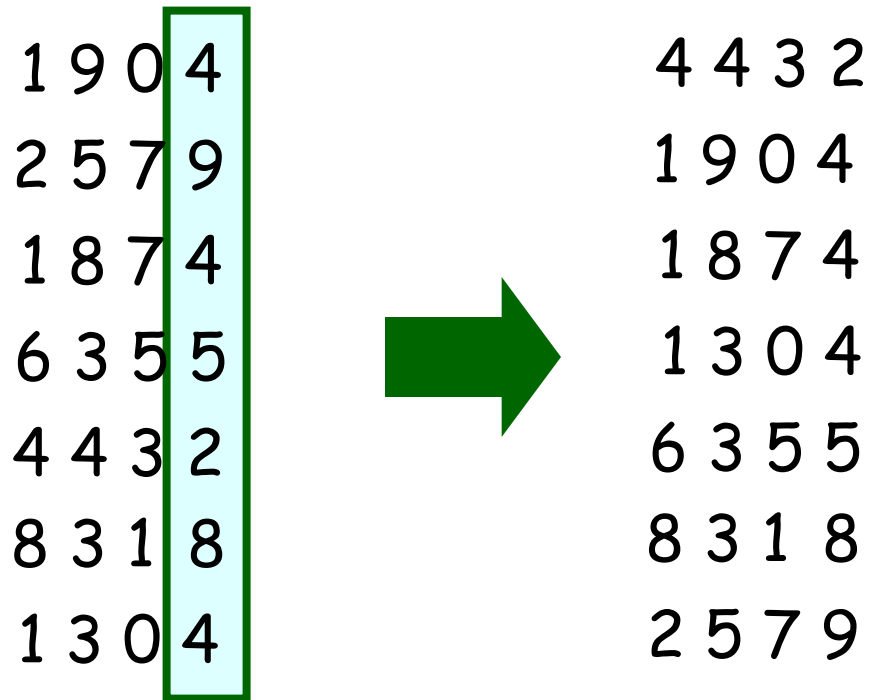
1 3 0 4



4 digits

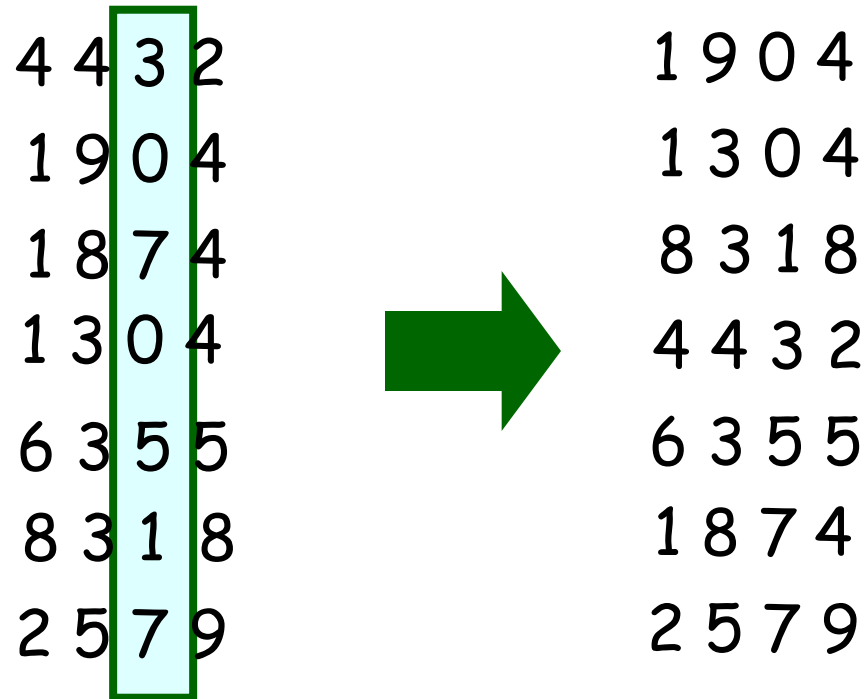
Radix Sort (Example Run)

Round 1: Stable sort digit 1



Radix Sort (Example Run)

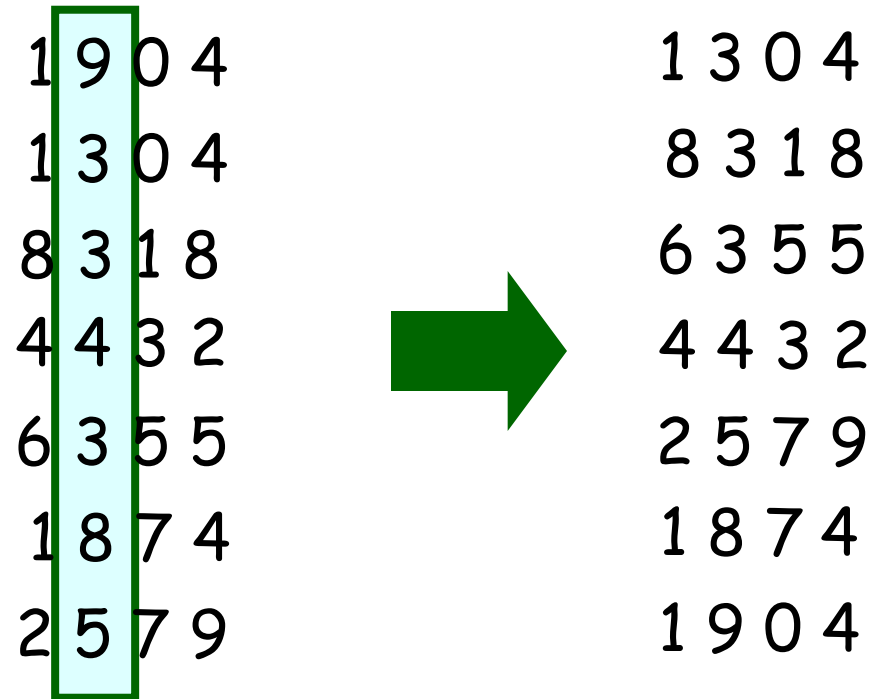
Round 2: Stable sort digit 2



After Round 2, last 2 digits
are sorted (why?)

Radix Sort (Example Run)

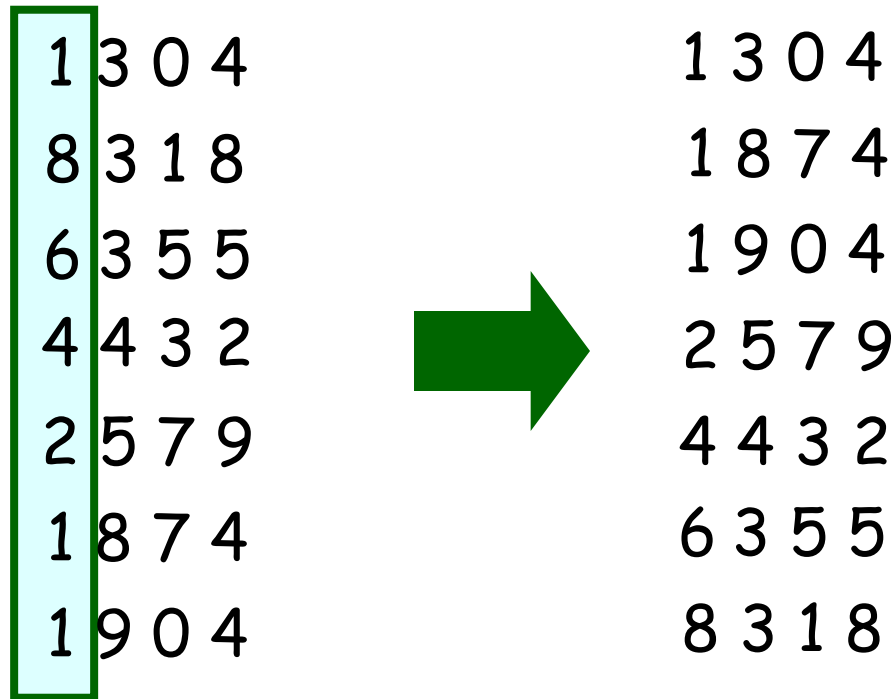
Round 3: Stable sort digit 3



After Round 3, last 3 digits
are sorted (why?)

Radix Sort (Example Run)

Round 4: Stable sort digit 4



After Round 4, last 4 digits
are sorted (why?)

Radix Sort (Example Run)

Done when all digits are processed

1 3 0 4

1 8 7 4

1 9 0 4

2 5 7 9

4 4 3 2

6 3 5 5

8 3 1 8

The array is sorted (why?)

Radix Sort (Correctness)

Question:

"After r rounds, last r digits are sorted"

Why ??

Answer:

This can be proved by induction :

The statement is true for $r = 1$

Assume the statement is true for $r = k$

Then ...

Radix Sort (Correctness)

At Round $k+1$,

- if two numbers differ in digit " $k+1$ ", their relative order [based on last $k+1$ digits] will be correct after sorting digit " $k+1$ "
- if two numbers match in digit " $k+1$ ", their relative order [based on last $k+1$ digits] will be correct after stable sorting digit " $k+1$ " (why?)

→ Last " $k+1$ " digits sorted after Round " $k+1$ "


Radix Sort (Summary)

Conclusion:

- After d rounds, last d digits are sorted, so that the numbers in $A[1..n]$ are sorted
- There are d rounds of stable sort, each can be done in $O(n + k)$ time
 - ➔ Running time = $O(d(n + k))$
 - if $d=O(1)$ and $k=O(n)$, asymptotically optimal

Bucket Sort

extra info
on values



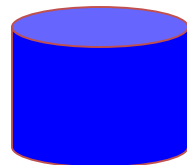
- Input: Array $A[1..n]$ of n elements, each is drawn uniformly at random from the interval $[0,1)$
- Output: Sorted array of the n elements
- Idea:
Distribute elements into n buckets, so that each bucket is likely to have fewer elements → easier to sort

Bucket Sort (Details)

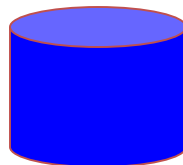
Before
Running

0.78, 0.17, 0.39, 0.26, 0.72,
0.94, 0.21, 0.12, 0.23, 0.68

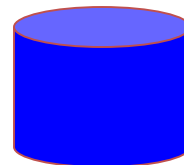
Step 1:
Create n
buckets



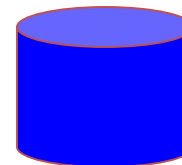
[0,0.1)



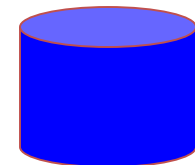
[0.1,0.2)



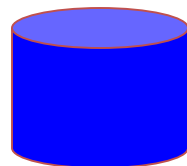
[0.2,0.3)



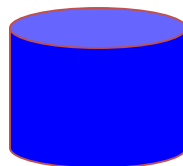
[0.3,0.4)



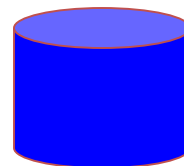
[0.4,0.5)



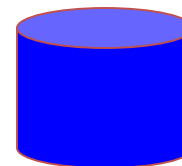
[0.5,0.6)



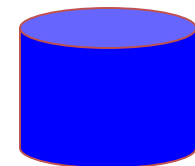
[0.6,0.7)



[0.7,0.8)



[0.8,0.9)



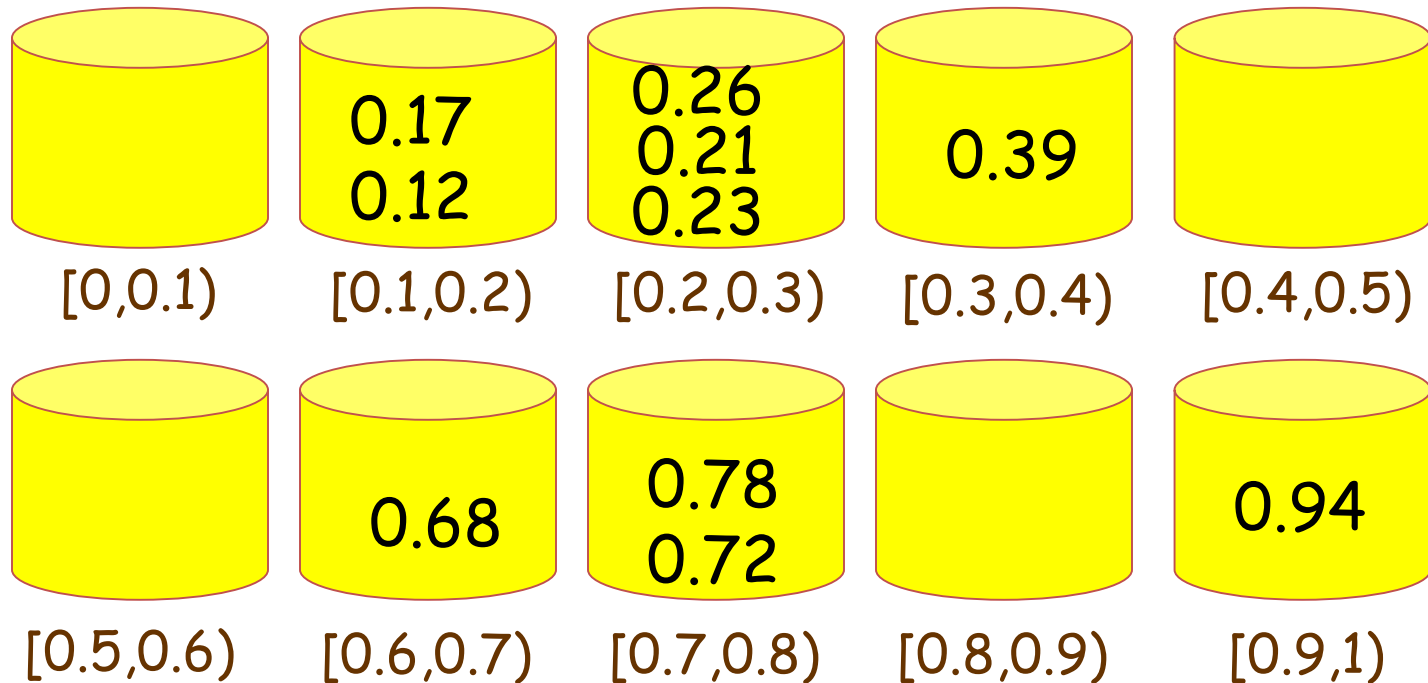
[0.9,1)

n = #buckets
= #elements

each bucket represents a
subinterval of size $1/n$

Bucket Sort (Details)

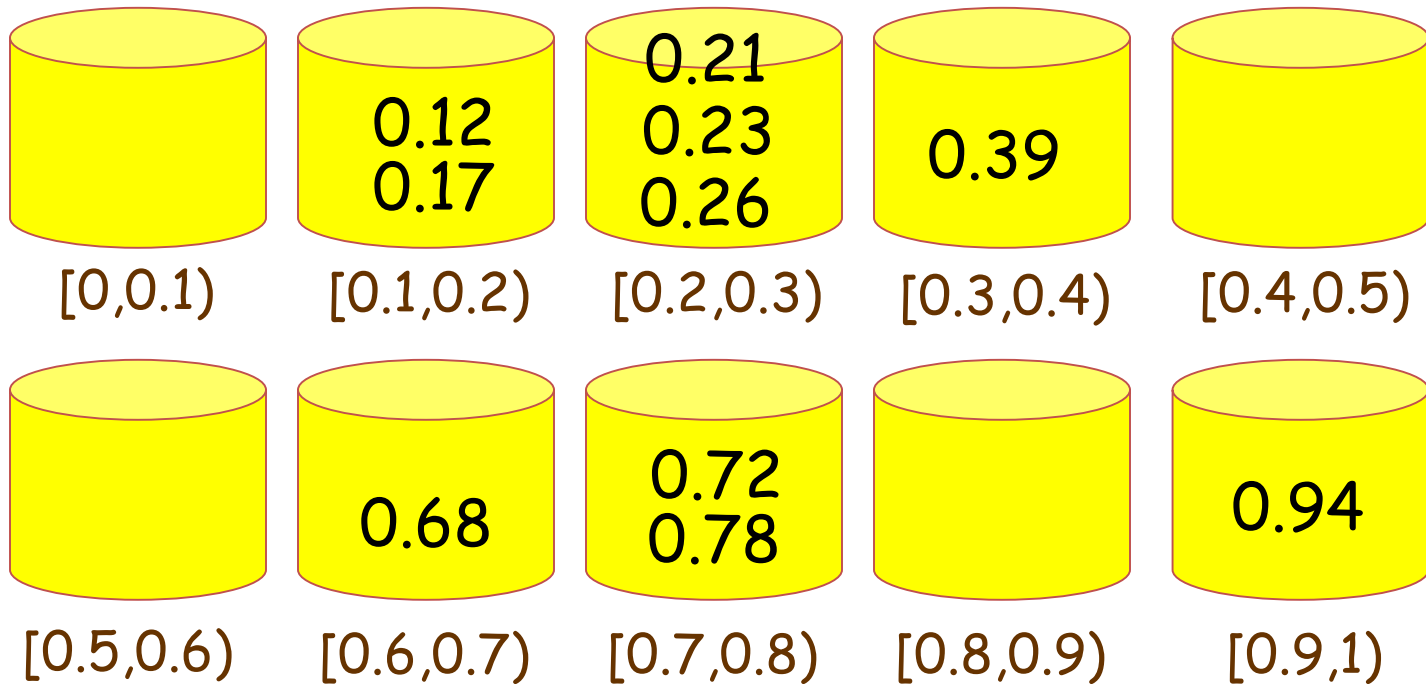
Step 2: Distribute each element to correct bucket



If Bucket j represents subinterval $[j/n, (j+1)/n)$,
element with value x should be in Bucket $\lfloor xn \rfloor$

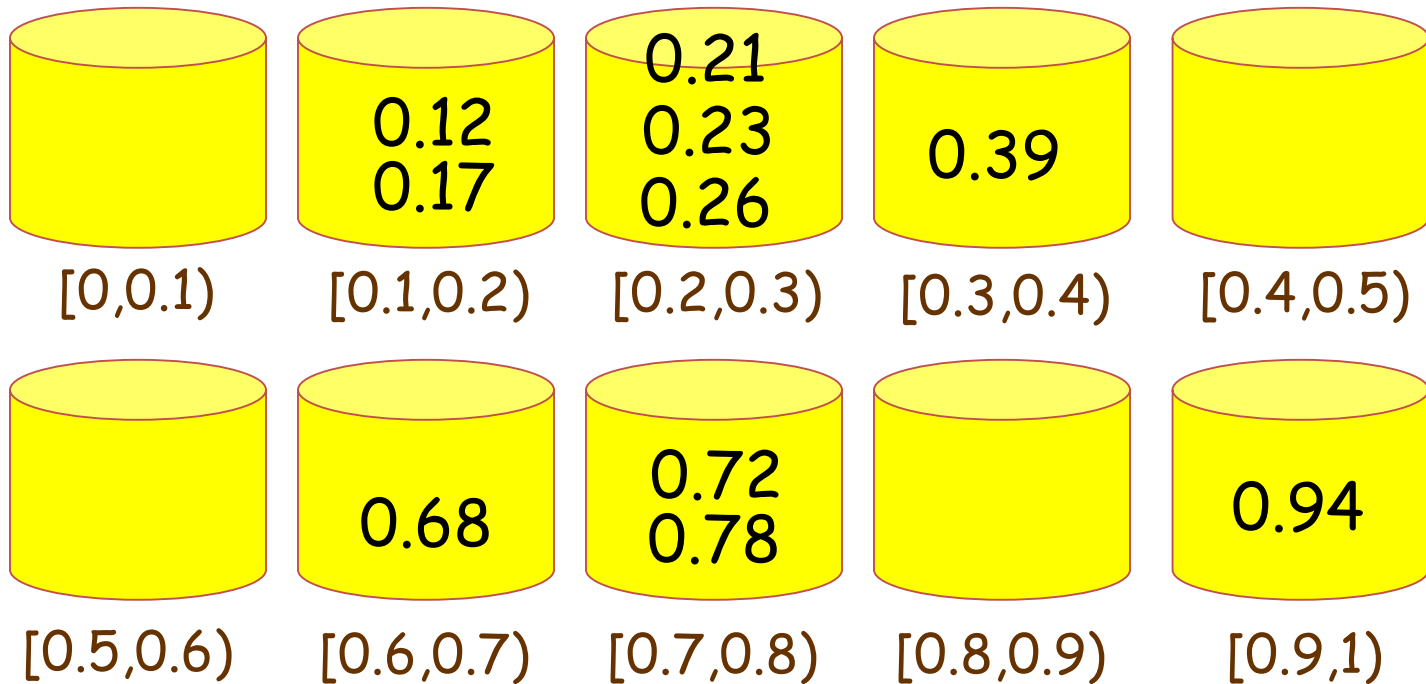
Bucket Sort (Details)

Step 3: Sort each bucket (by insertion sort)



Bucket Sort (Details)

Step 4: Collect elements from Bucket 0 to Bucket n-1



Sorted Output: 0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94

Bucket Sort (Running Time)

- Let X = # comparisons in all insertion sort

Running time = $\Theta(n + X)$ → varies on input

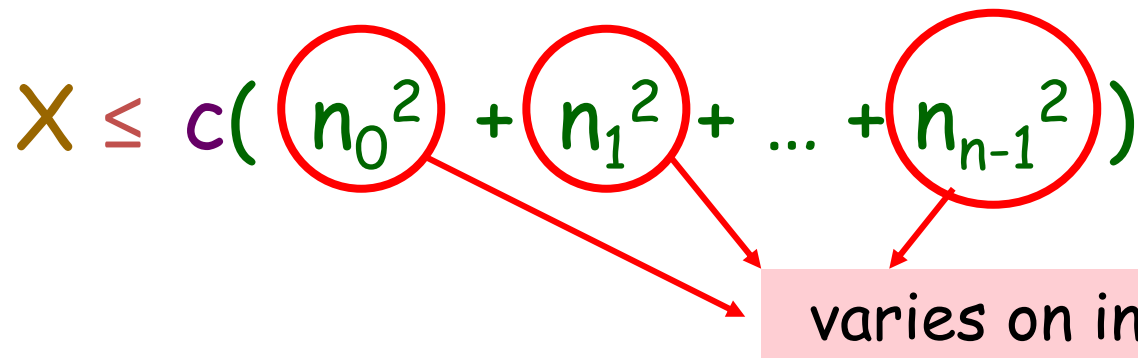
→ worst-case running time = $\Theta(n^2)$

- How about average running time?

Finding average of X (i.e. #comparisons)
gives average running time

Average Running Time

Let n_j = # elements in Bucket j

$$X \leq c(n_0^2 + n_1^2 + \dots + n_{n-1}^2)$$


varies on input

$$\begin{aligned} \text{So, } E[X] &\leq E[c(n_0^2 + n_1^2 + \dots + n_{n-1}^2)] \\ &= c E[n_0^2 + n_1^2 + \dots + n_{n-1}^2] \\ &= c (E[n_0^2] + E[n_1^2] + \dots + E[n_{n-1}^2]) \\ &= cn E[n_0^2] \quad (\text{by uniform distribution}) \end{aligned}$$

Average Running Time

Textbook (pages 202-203) shows that

$$E[n_0^2] = 2 - (1/n)$$

$$\rightarrow E[X] \leq cn E[n_0^2] = 2cn - c$$

In other words, $E[X] = O(n)$

$$\rightarrow \text{Average running time} = \Theta(n)$$

For Interested Classmates

The following is how we can show

$$E[n_0^2] = 2 - (1/n)$$

Recall that $n_0 = \#$ elements in Bucket 0

So, suppose we set

$A_k = 1$ if element k is in Bucket 0

$A_k = 0$ if element k not in Bucket 0

Then, $n_0 = A_1 + A_2 + \dots + A_n$

For Interested Classmates

Then,

$$\begin{aligned} E[n_0^2] &= E[(A_1 + A_2 + \dots + A_n)^2] \\ &= E[A_1^2 + A_2^2 + \dots + A_n^2 \\ &\quad + A_1A_2 + A_1A_3 + \dots + A_1A_n \\ &\quad + A_2A_1 + A_2A_3 + \dots + A_2A_n \\ &\quad + \dots \\ &\quad + A_nA_1 + A_nA_2 + \dots + A_nA_{n-1}] \end{aligned}$$

$$\begin{aligned}
&= E[A_1^2] + E[A_2^2] + \dots + E[A_n^2] \\
&\quad + E[A_1 A_2] + \dots + E[A_n A_{n-1}] \\
&= n E[A_1^2] + n(n-1) E[A_1 A_2]
\end{aligned}$$

The value of A_1^2 is either 1 (when $A_1 = 1$),
or 0 (when $A_1 = 0$)

The first case happens with $1/n$ chance
(when element 1 is in Bucket 0), so

$$E[A_1^2] = 1/n * 1 + (1 - 1/n) * 0 = 1/n$$

For A_1A_2 , it is either 1 (when $A_1=1$ and $A_2=1$),
or 0 (otherwise)

The first case happens with $1/n^2$ chance
(when both element 1 and element 2 are in
Bucket 0), so

$$E[A_1A_2] = 1/n^2 * 1 + (1 - 1/n^2) * 0 = 1/n^2$$

$$\begin{aligned}\text{Thus, } E[n_0^2] &= n E[A_1^2] + n(n-1) E[A_1A_2] \\ &= n (1/n) + n(n-1) (1/n^2) \\ &= 2 - 1/n\end{aligned}$$

Practice at Home

- Exercises 8.2-3, 8.2-4, 8.3-2, 8.3-4, 8.4-2
Problem 8-2: a, b
- It is known that $\Omega(n \log n)$ is a lower bound for sorting problem. However, we have seen algorithms like counting sort or radix sort which can sort n items in $O(n)$ time. Is there a contradiction? If not, why? Explain?