CS4311
Design and Analysis of Algorithms

Lecture 6: Sorting in Linear Time

About this lecture

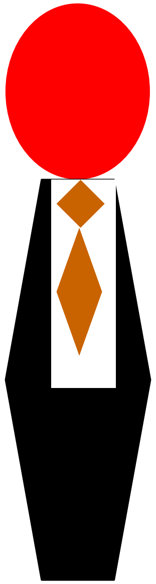
- Sorting algorithms we studied so far
  - Insertion, Selection, Merge, Quicksort

  ➔ 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 distribution

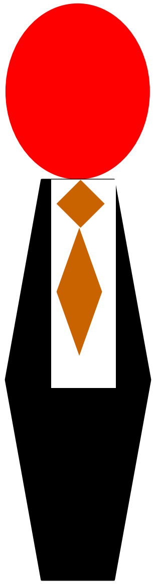
Helping the Billionaire



- Your boss, Bill, is a billionaire
- Inside his BIG wallet, there are a lot of bills, say, n bills
- Nine kinds of bills:

  \$1, \$5, \$10, \$20, \$50, \$100, \$200, \$500, \$1000

Helping the Billionaire

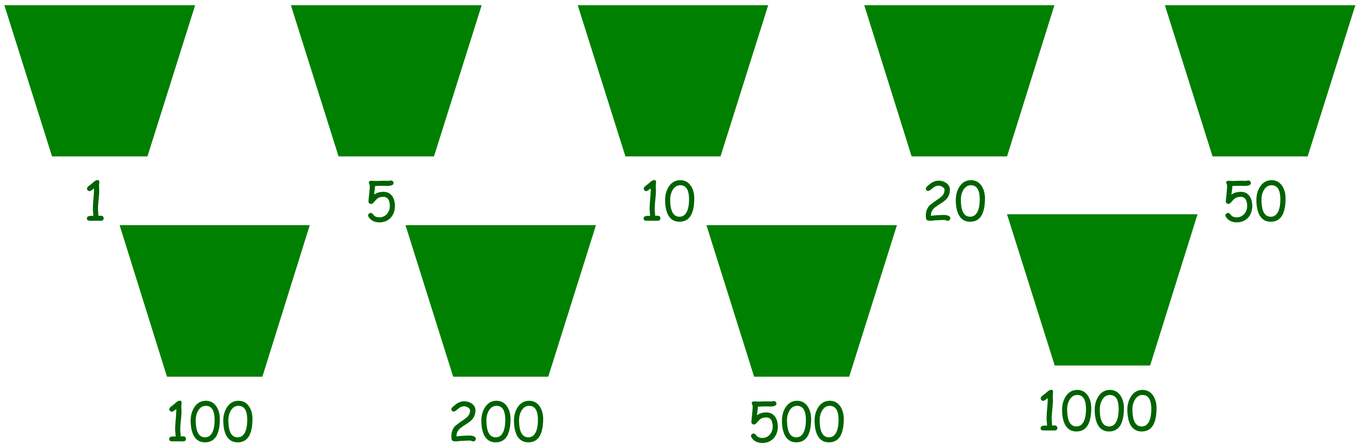


- He did not care about the ordering of the bills before
- But then, he has taken the Algorithm course, and learnt that if things are sorted, we can search faster

The horoscope says I should use only $500 notes today … Do I have enough in the wallet?

A Proposal

- Create a bin for each kind of bill
- Look at his bill one by one, and place the bill in the corresponding bin
- Finally, collect bills in each bin, starting from \$1-bin, \$5-bin, …, to \$1000-bin



A Proposal

- In the previous algorithm, there is no comparison between the items …
  - But we can still sort correctly… WHY?
- Each step looks at the value of an item, and distribute the item to the correct bin
  - So, in the end, when a bill is collected, its value must be larger than or equal to all bills collected before ➔ sorted

Sorting by Distribution

- Previous algorithm sorts the bills based on distribution operations
- It works because:
  - we have information about the values of the input items → we can create bins
- We will look at more algorithms which are based on the same distribution idea

Counting Sort

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 (Details)

Before Running

A

2	1	2	5	3	3	1	2

next element

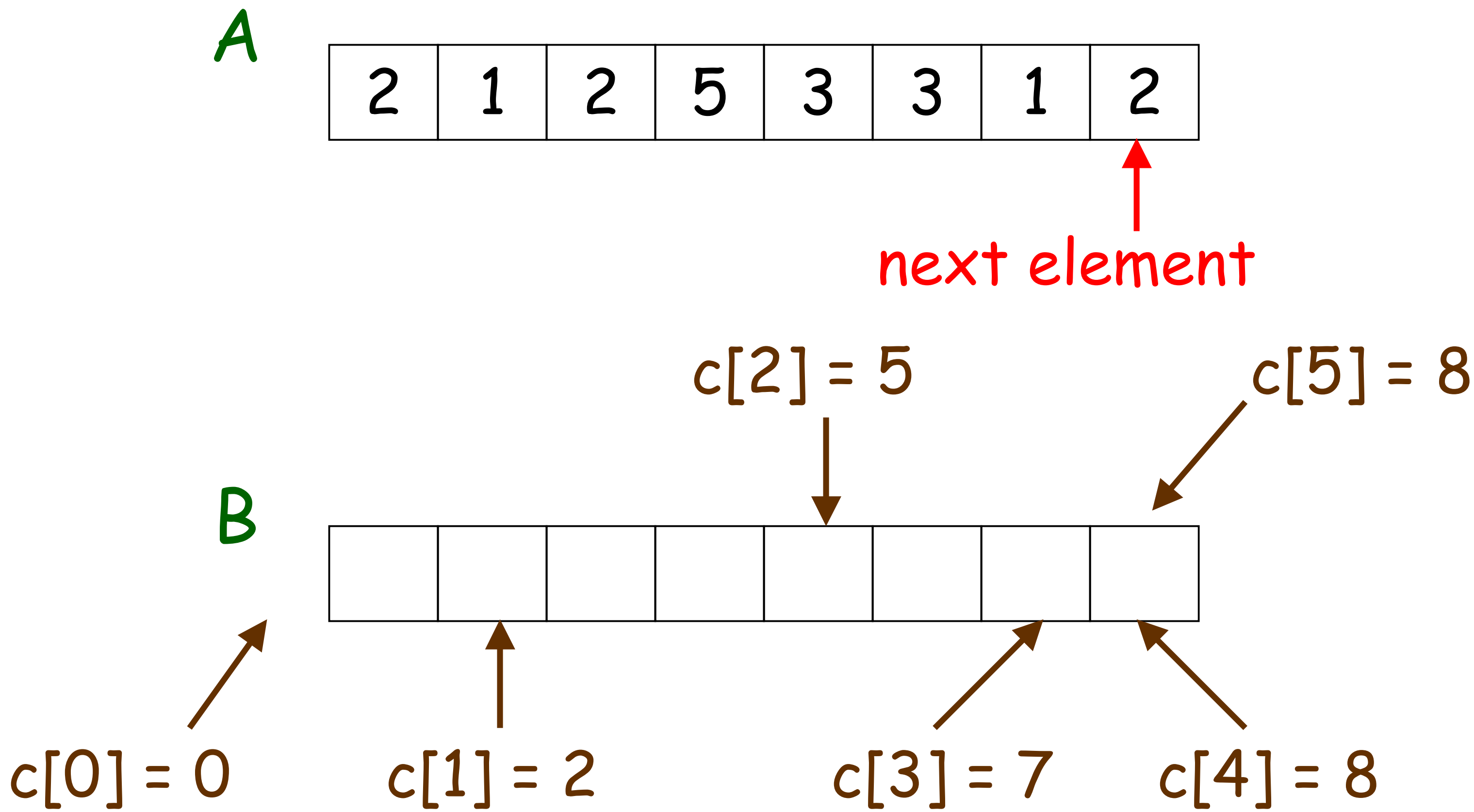
k+1 counters

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

B

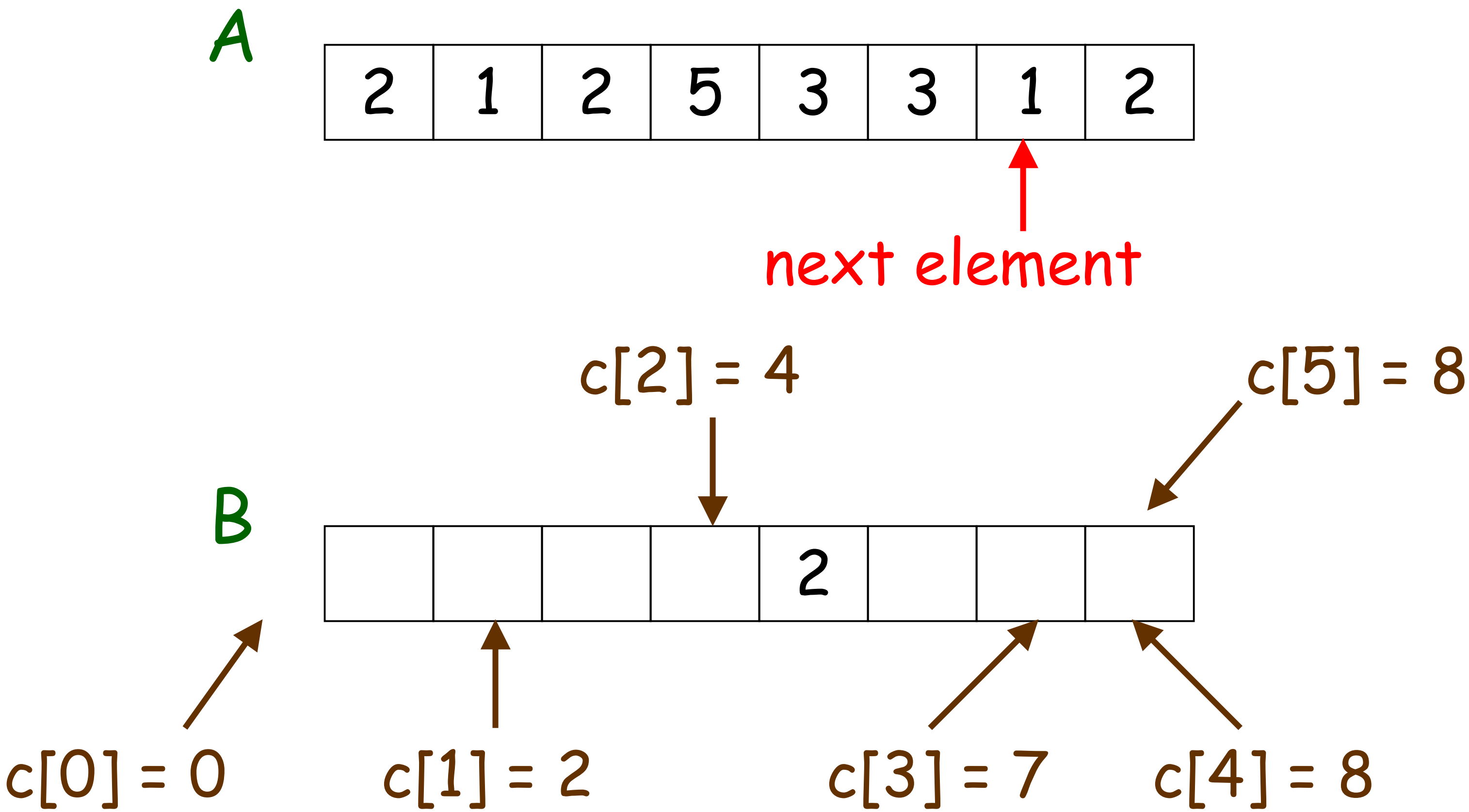
Counting Sort (Details)

Step 1: Set c[j] = location in B for placing the next element if it has value j



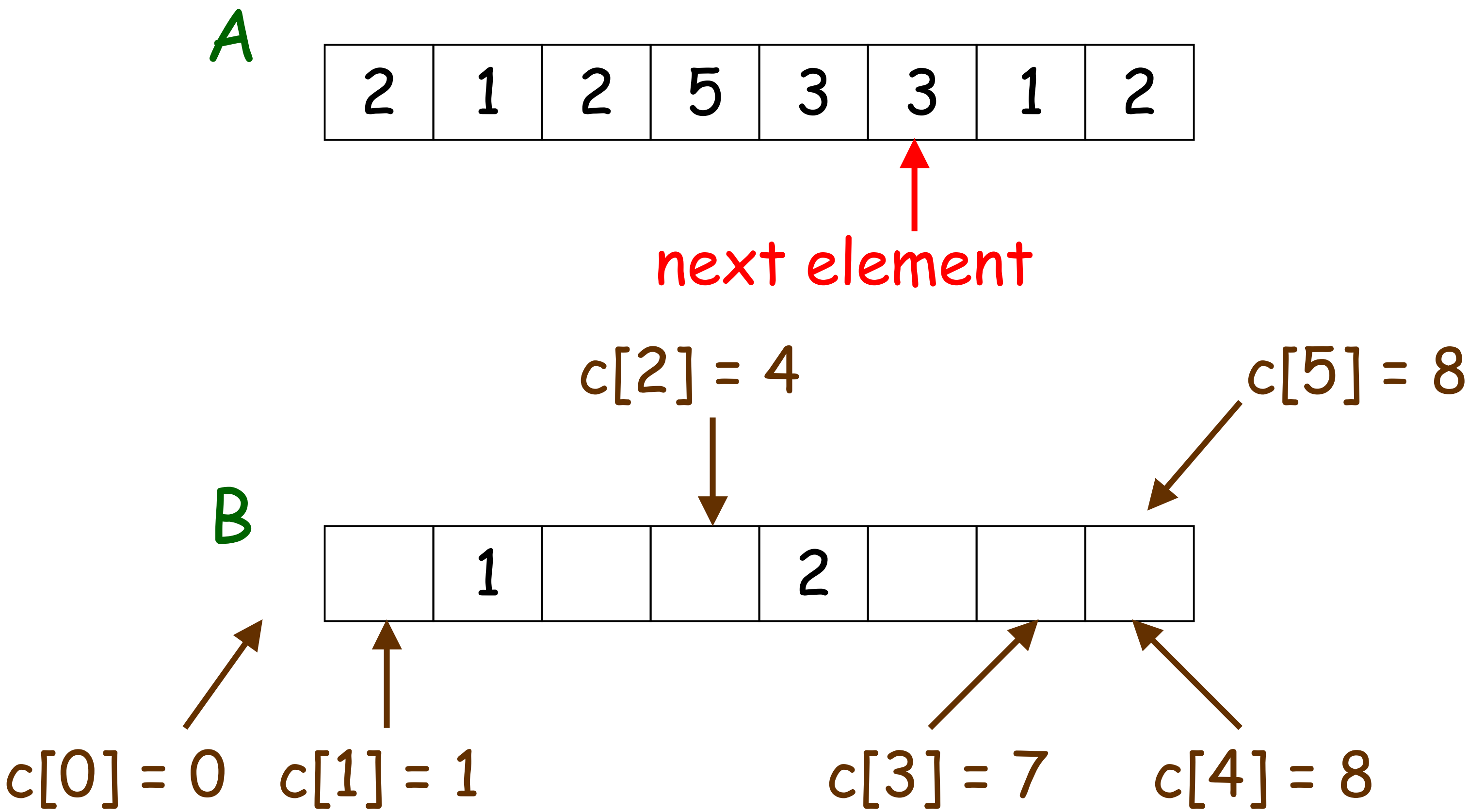
Counting Sort (Details)

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



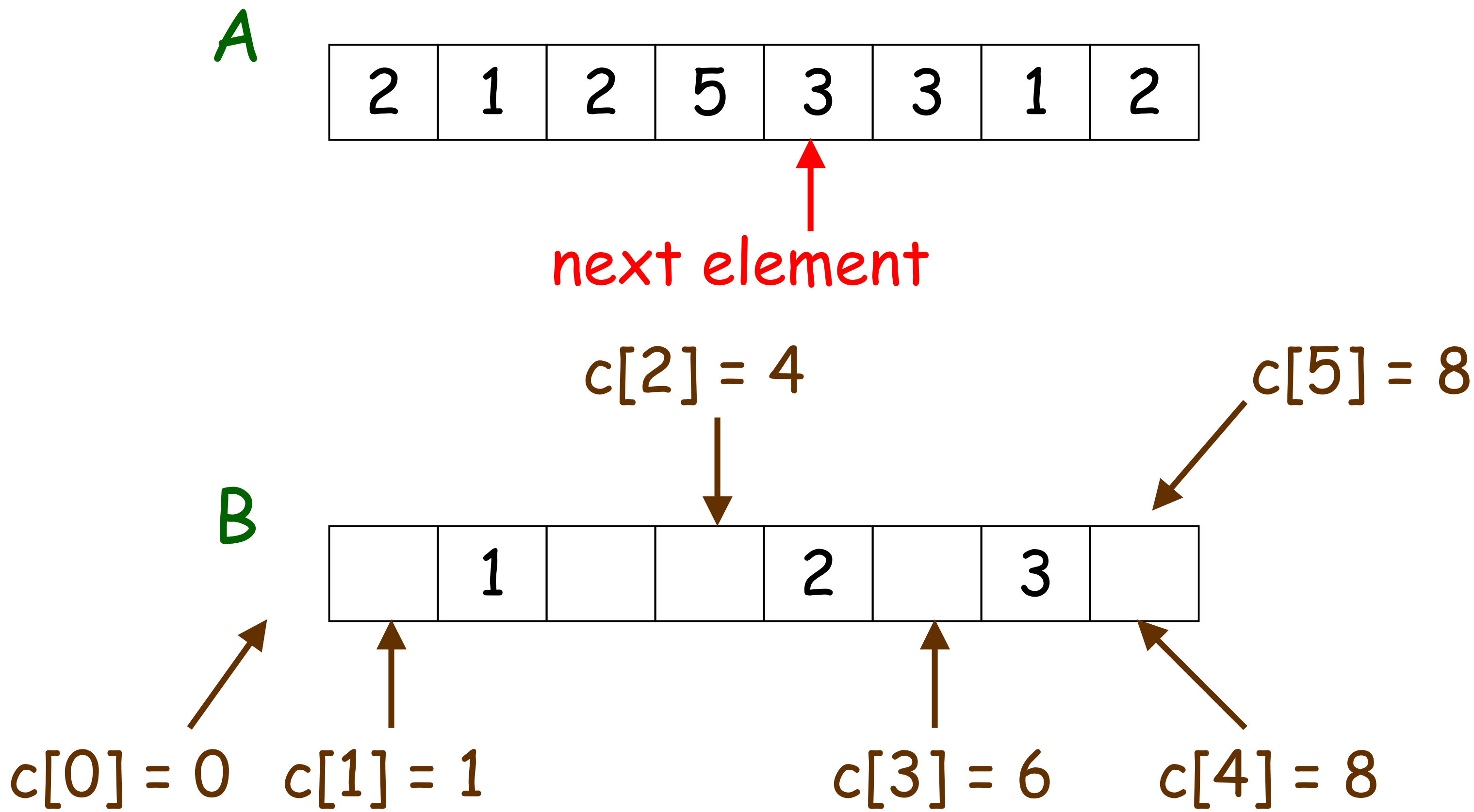
Counting Sort (Details)

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



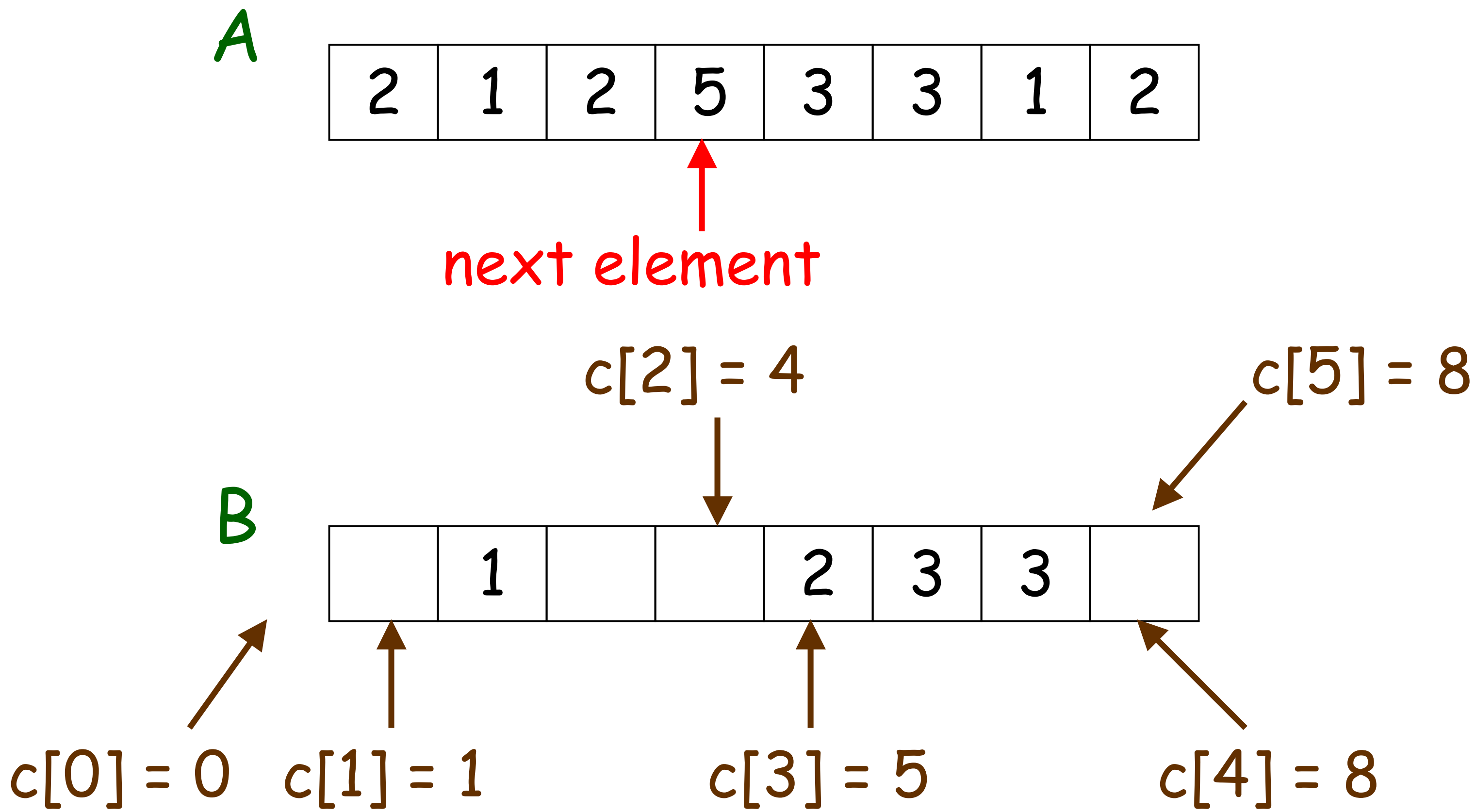
Counting Sort (Details)

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



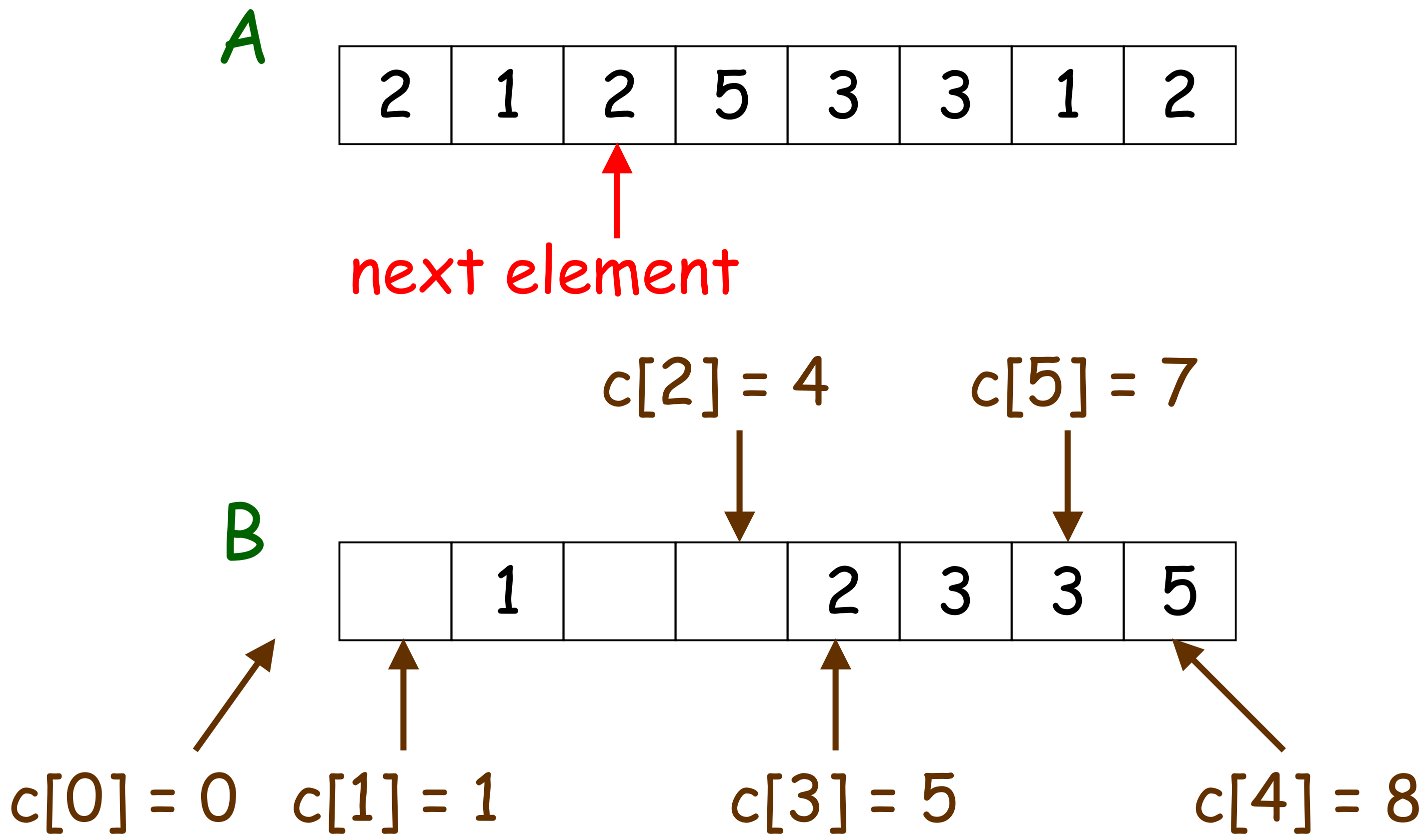
Counting Sort (Details)

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



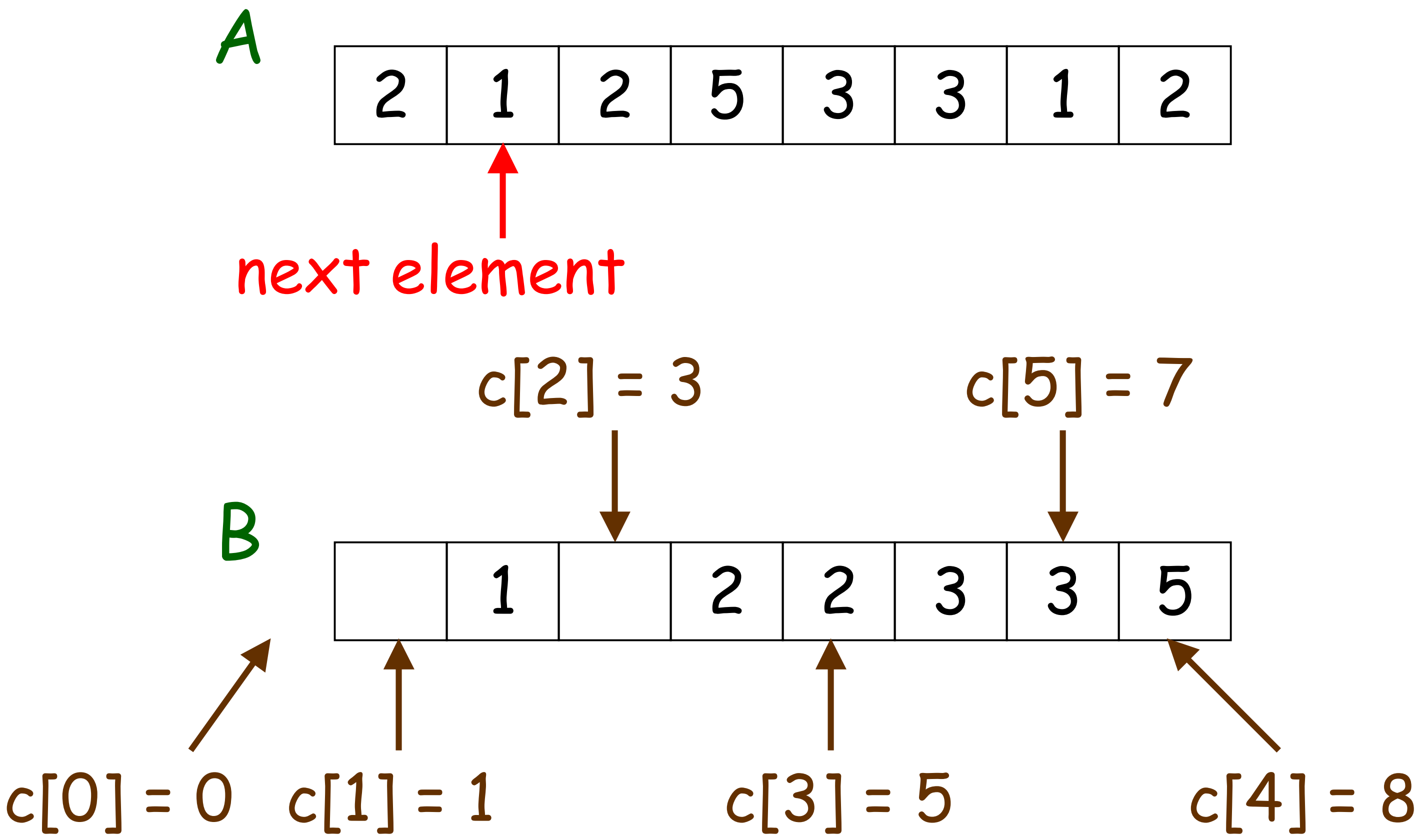
Counting Sort (Details)

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



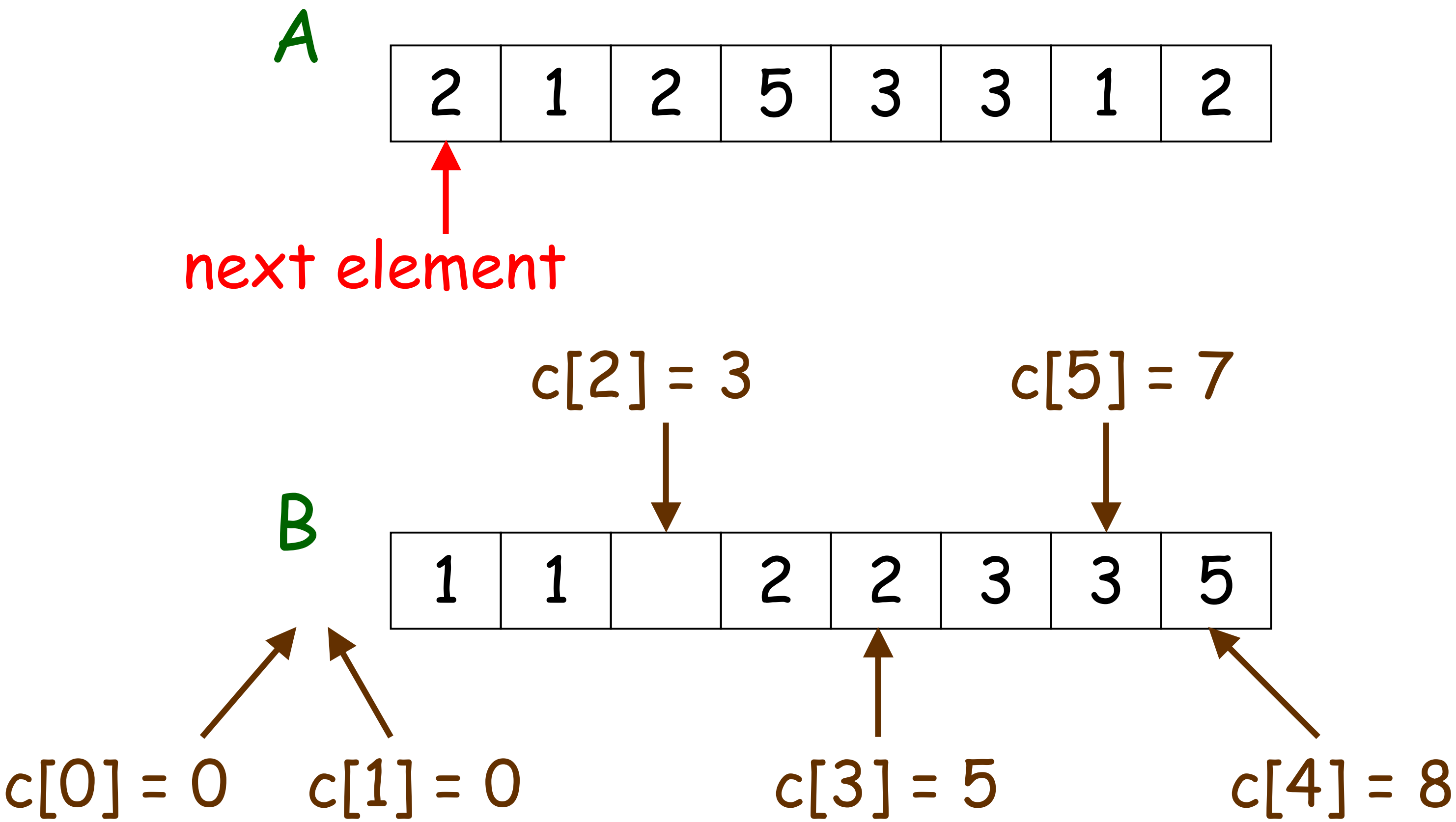
Counting Sort (Details)

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



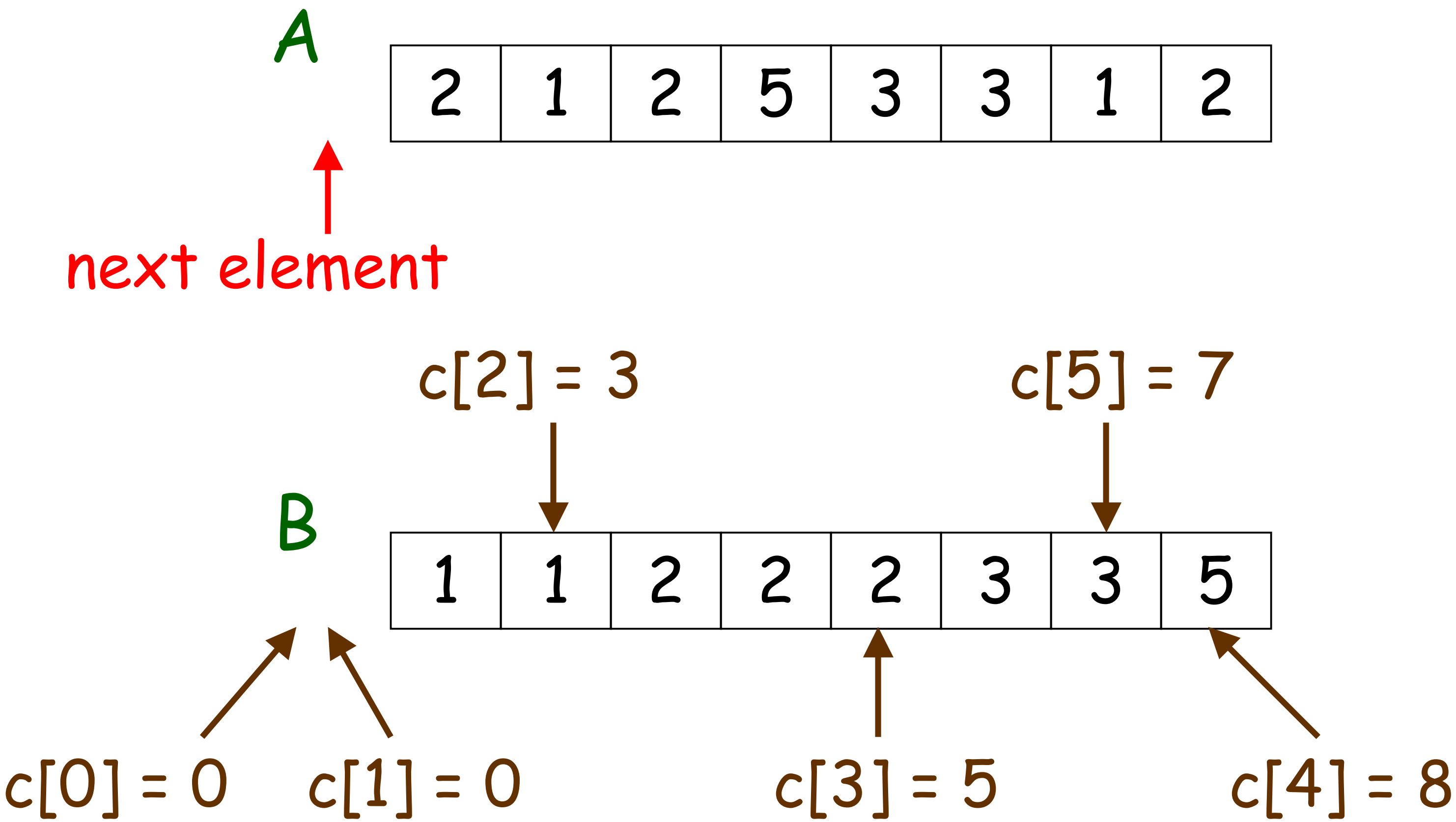
Counting Sort (Details)

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 (Step 1)

How can we perform Step 1 smartly?

1. Initialize c[0], c[1], …, c[k] to 0

2. /* First, set c[j] = # elements with value j */

   For x = 1,2,…,n, increase c[A[x]] by 1

3. /* Set c[j] = location in B to place next element whose value is j (iteratively) */

   For y = 1,2,…,k, c[y] = c[y-1] + c[y]

Time for Step 1 = O(n + k)

Counting Sort (Step 2)

How can we perform Step 2 ?

```
/*  Process A from right to left */
For x = n, n-1,…,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 (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

2	1	2	5	3	3	1	2

After Sorting

1	1	2	2	2	3	3	5

Radix Sort

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

1 9 0 4	4 4 3 2
2 5 7 9	1 9 0 4
1 8 7 4	1 8 7 4
6 3 5 5	1 3 0 4
4 4 3 2	6 3 5 5
8 3 1 8	8 3 1 8
1 3 0 4	2 5 7 9

Radix Sort (Example Run)

Round 2: Stable sort digit 2

4 4 3 2	1 9 0 4
1 9 0 4	1 3 0 4
1 8 7 4	8 3 1 8
1 3 0 4	4 4 3 2
6 3 5 5	6 3 5 5
8 3 1 8	1 8 7 4
2 5 7 9	2 5 7 9

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

Radix Sort (Example Run)

Round 3: Stable sort digit 3

1 9 0 4	1 3 0 4
1 3 0 4	8 3 1 8
8 3 1 8	6 3 5 5
4 4 3 2	4 4 3 2
6 3 5 5	2 5 7 9
1 8 7 4	1 8 7 4
2 5 7 9	1 9 0 4

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

Radix Sort (Example Run)

Round 4: Stable sort digit 4

1 3 0 4	1 3 0 4
8 3 1 8	1 8 7 4
6 3 5 5	1 9 0 4
4 4 3 2	2 5 7 9
2 5 7 9	4 4 3 2
1 8 7 4	6 3 5 5
1 9 0 4	8 3 1 8

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

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:

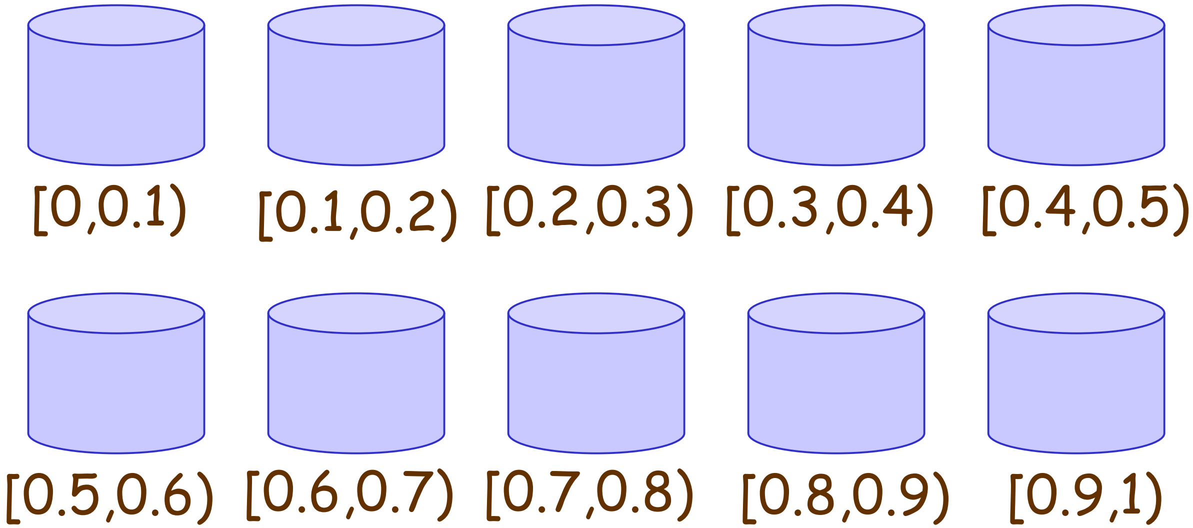
  Distribute elements into n buckets, so that each bucket is likely to have fewer elements → easier to sort

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

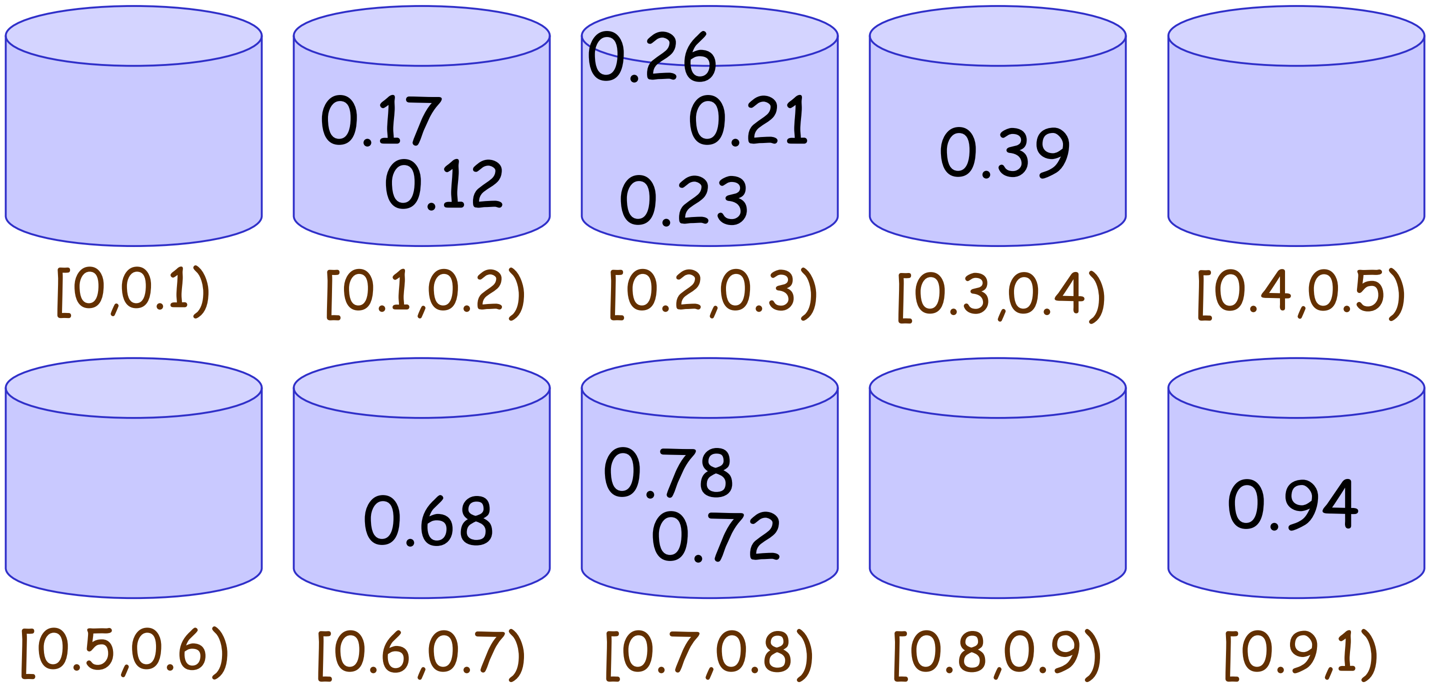


n = #buckets
= #elements

each bucket represents a
subinterval of size 1/n

Bucket Sort (Details)

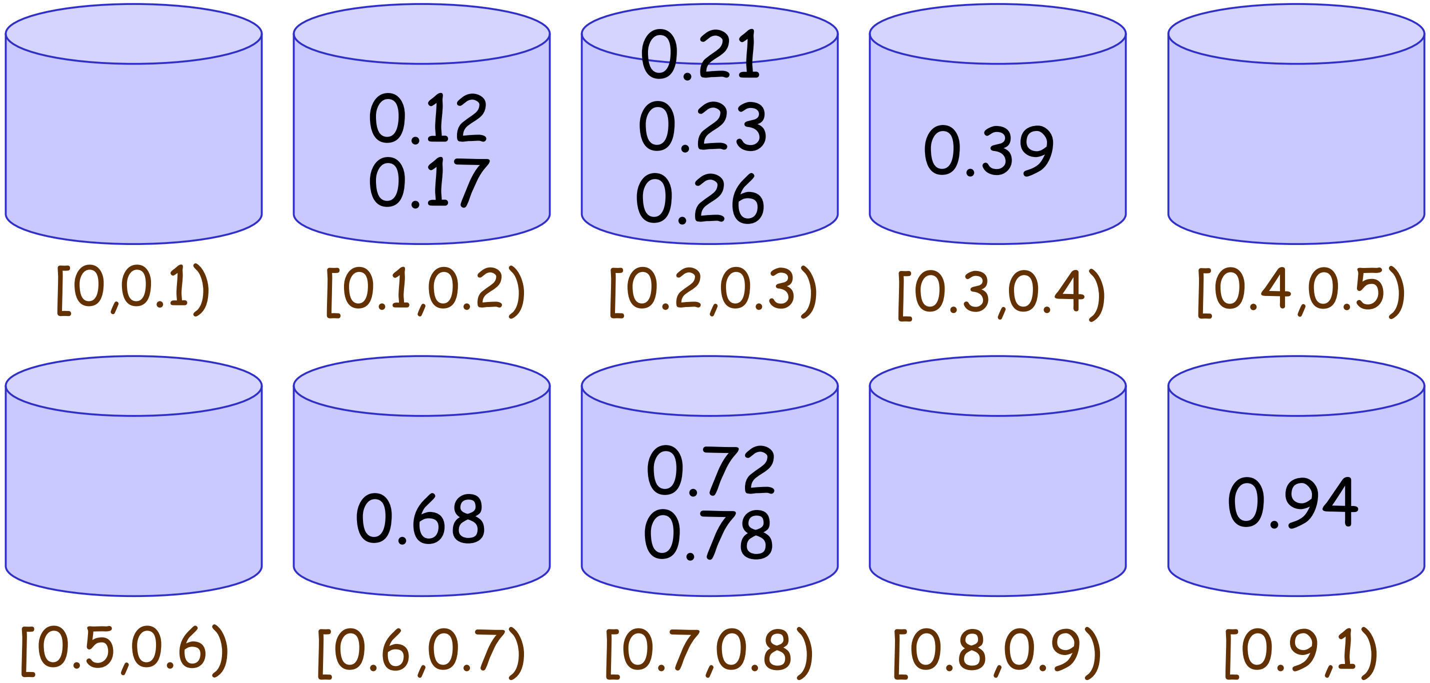
Step 2: Distribute each element to correct bucket



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

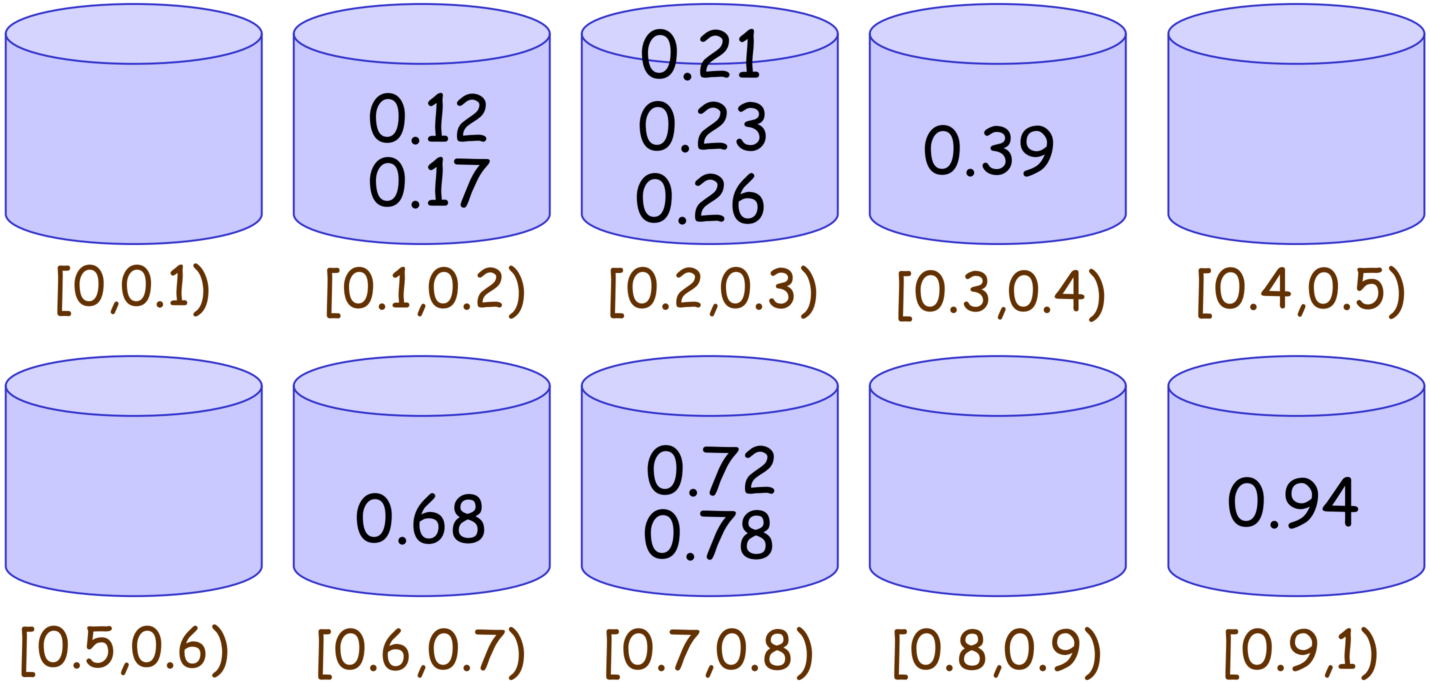
Bucket Sort (Details)

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

  ➔ wosrt-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

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

varies on input

So, $E[X] \leq E[c(n_0^2 + n_1^2 + \ldots + n_{n-1}^2)]$

$$= c\, E[n_0^2 + n_1^2 + \ldots + n_{n-1}^2]$$

$$= c\, (E[n_0^2] + E[n_1^2] + \ldots + E[n_{n-1}^2])$$

$$= cn\, E[n_0^2] \qquad \text{(by symmetry)}$$

Average Running Time

Textbook (pages 175-176) shows that

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

➔ $E[X] \leq cn\, E[n_0^2] = 2cn - c$

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

➔ 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

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

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

Then, $n_0 = Y_1 + Y_2 + \ldots + Y_n$

For Interested Classmates

Then,

$$
\begin{aligned}
E[n_0^2] &= E[(Y_1 + Y_2 + \ldots + Y_n)^2] \\
&= E[\, Y_1^2 + Y_2^2 + \ldots + Y_n^2 \\
&\quad + Y_1Y_2 + Y_1Y_3 + \ldots + Y_1Y_n \\
&\quad + Y_2Y_1 + Y_2Y_3 + \ldots + Y_2Y_n \\
&\quad + \ldots \\
&\quad + Y_nY_1 + Y_nY_2 + \ldots + Y_nY_{n-1} \,]
\end{aligned}
$$

$$= E[Y_1^2] + E[Y_2^2] + \dots + E[Y_n^2]$$
$$+ E[Y_1Y_2] + \dots + E[Y_nY_{n-1}]$$
$$= n\,E[Y_1^2] + n(n-1)\,E[Y_1Y_2]$$
(by symmetry)

The value of Y_1^2 is either 1 (when $Y_1 = 1$),
or 0 (when $Y_1 = 0$)

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

$$E[Y_1^2] = 1/n * 1 + (1- 1/n) * 0 = 1/n$$

For Y_1Y_2, it is either 1 (when Y_1=1 and Y_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

$$E[Y_1Y_2] = 1/n^2 * 1 + (1 - 1/n^2) * 0 = 1/n^2$$

Thus,
$$\begin{aligned} E[n_0^2] &= n\, E[Y_1^2] + n(n-1)\, E[Y_1Y_2] \\ &= n\,(1/n) + n(n-1)\,(1/n^2) \\ &= 2 - 1/n \end{aligned}$$