CS 2009
Design and Analysis of Algorithms

Waheed Ahmed
Email : waheedahmed@nu.edu.pk

Week 7:
Linear Time Sorting (Counting Sort, Radix Sort, Bucket Sort)

COMPARISON-BASED SORTING

- **You want to sort an array of items**
- **You can't access the items' values directly: you can only *compare* two items and find out which is bigger or smaller.**
- Examples: Insertion Sort, MergeSort, QuickSort

"Comparison-based sorting algorithms" are general-purpose.

The worst case complexity of comparison-based sorting can not be reduced more than "n.logn" (Proof in textbook)

Linear-time Sorting

Beyond comparison-based sorting algorithms!

A New Model Of Computation

The elements we're working with have meaningful values.

Before:

arbitrary elements whose values we could never directly access, process, or take advantage of (i.e. we could only interact with them via comparisons)

Now (examples):

9	18	27	4	9	18	27

not-too-large integers

Dec	Feb	Oct	May

months in a year

- The worst-case complexity can be reduced further from "n.logn" without making comparisons, called linear sorting. Counting, Radix and Bucket sort are three examples.
- However, it is possible only under restrictive circumstances, for example sorting small integers (exam score), characters etc.

Counting Sort

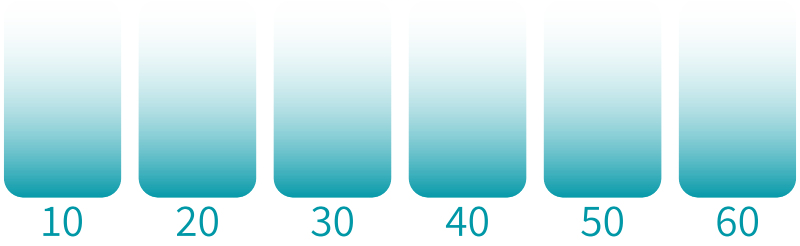
We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input:

30	50	20	30	10	60	50	20

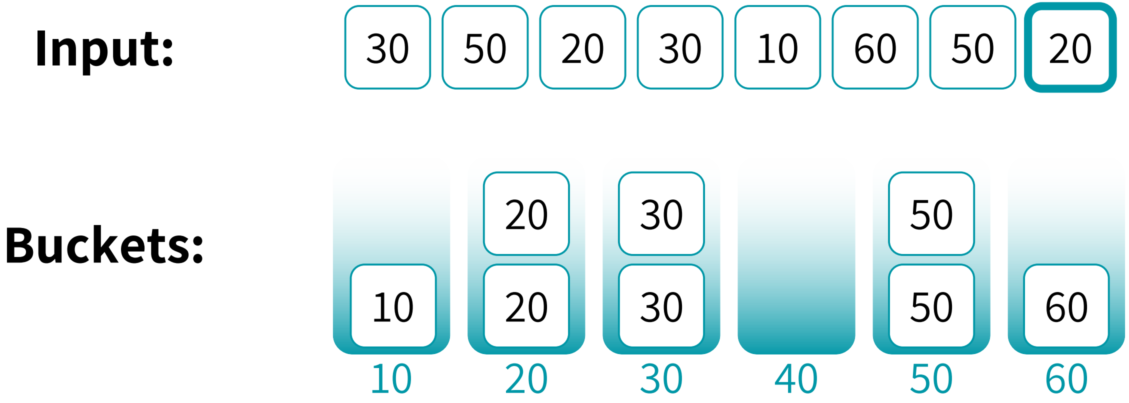
Buckets:



Counting Sort

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}



Counting Sort

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input: 30 | 50 | 20 | 30 | 10 | 60 | 50 | 20

Buckets:

10	20	30	40	50	60
	20	30		50	
10	20	30		50	60

Output: 10 | 20 | 20 | 30 | 30 | 50 | 50 | 60

Sorted in time:
O(n)

Counting Sort

- Input: array A[1, … , n]; k (elements in A have values from 1 to k)
- Output: sorted array A

Algorithm:

1. Create a counter array C[1, …, k]
2. Create an auxiliary array B[1, …, n]
3. Scan A once, record element frequency in C
4. Calculate prefix sum in C
5. Scan A in the reverse order, copy each element to B at the correct position according to C.
6. Copy B to A

Counting Sort: Pseudocode

```
COUNTING-SORT(A, B, k):
    let C[1..k] be a new array
    for i = 1 to k
        C[i ] = 0
    for j = 1 to A.length
        C[A[ j  ]] = C[A[ j  ]] + 1

    for i = 2 to k
        C[i ] = C[i ] + C[i  - 1]

      for j = A.length to 1
        B[C[A[ j  ]]] = A[ j  ]
        C[A[ j  ]] = C[A[ j  ]] - 1
```

1. Create a counter array C[1, …, k]
2. Create an auxiliary array B[1, …, n]
3. Scan A once, record element frequency in C
4. Calculate prefix sum in C
5. Scan A in the reverse order, copy each element to B at the correct position according to C.
6. Copy B to A

Analysis of Counting Sort

- Input: array A[1, … , n]; k (elements in A have values from 1 to k)
- Output: sorted array A

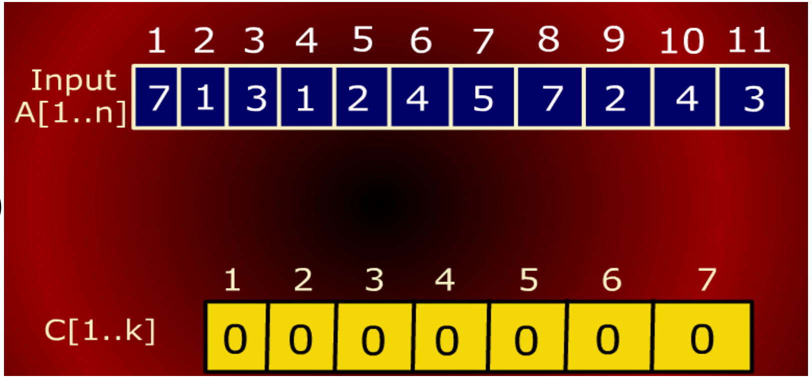
Algorithm:

1. Create a counter array C[1, …, k]
2. Create an auxiliary array B[1, …, n]
3. Scan A once, record element frequency in C
4. Calculate prefix sum in C
5. Scan A in the reverse order, copy each element to B at the correct position according to C.
6. Copy B to A

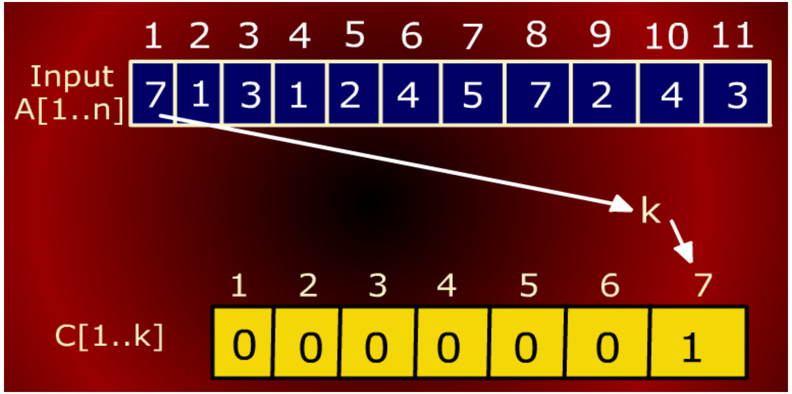
	Time	Space
1		O(k)
2		O(n)
3	O(n)	
4	O(k)	
5	O(n)	
6	O(n)	
	O(n+k)=O(n) (if k=O(n))	O(n+k)=O(n) (if k=O(n))

Counting sort

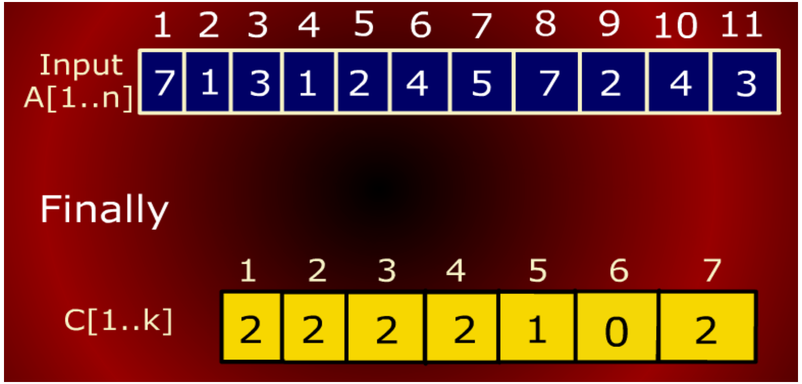
(1)



(2)



(3)

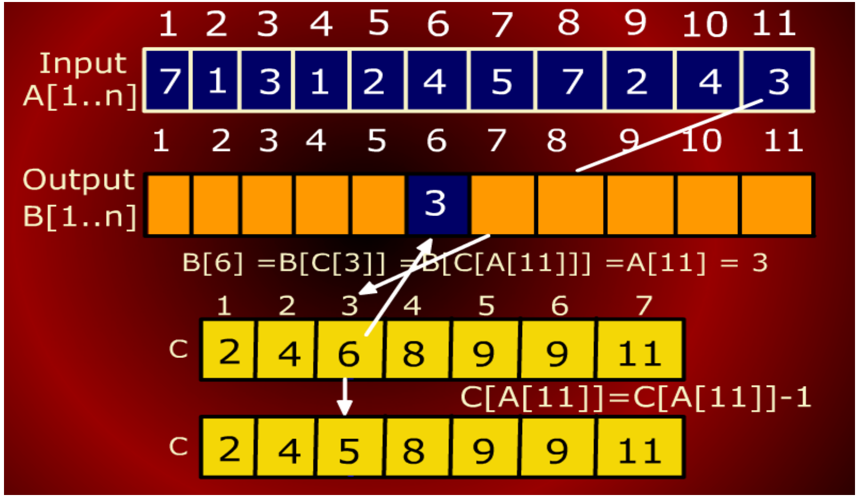


(4)

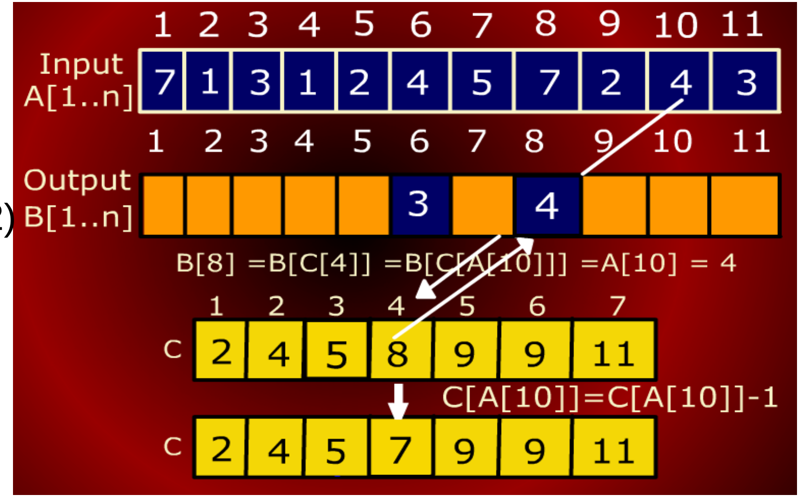
Counting sort

```
COUNTING-SORT(A, B, k):
   …
      for j = A.length to 1
            B[C[A[ j ]]] = A[ j ]
            C[A[ j ]] = C[A[ j ]] - 1
```

(1)



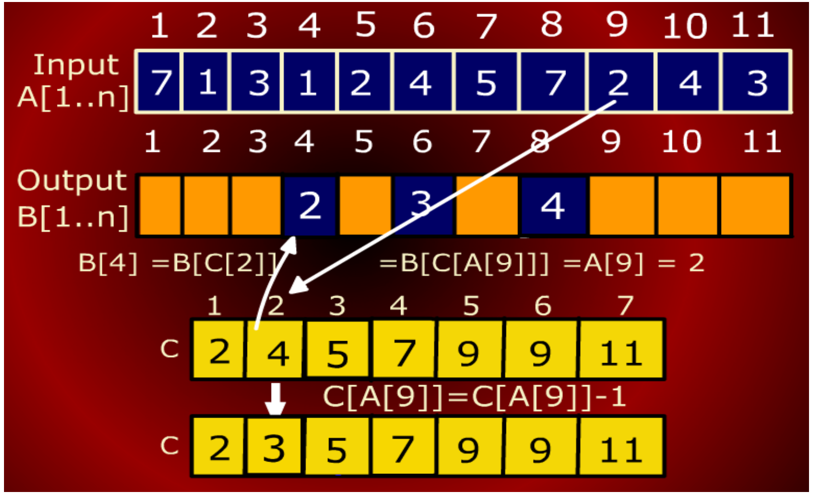
(2)



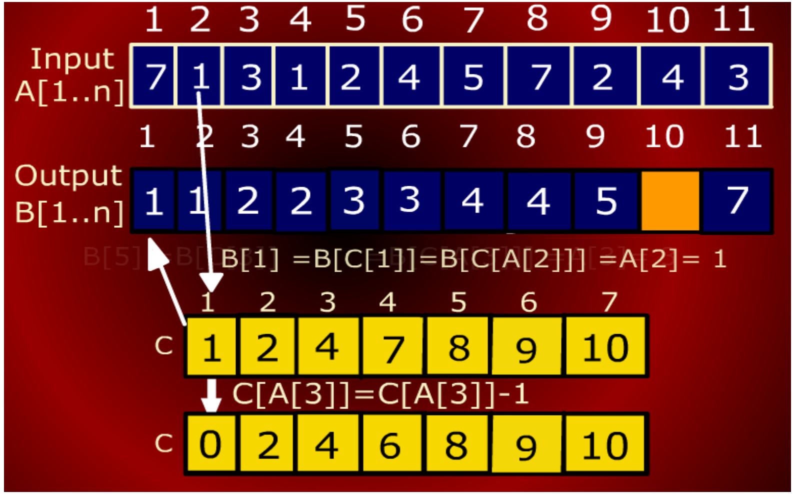
Counting sort

```
COUNTING-SORT(A, B, k):
    …
        for j = A.length to 1
            B[C[A[ j ]]] = A[ j ]
            C[A[ j ]] = C[A[ j ]] - 1
```

(3)



(4)

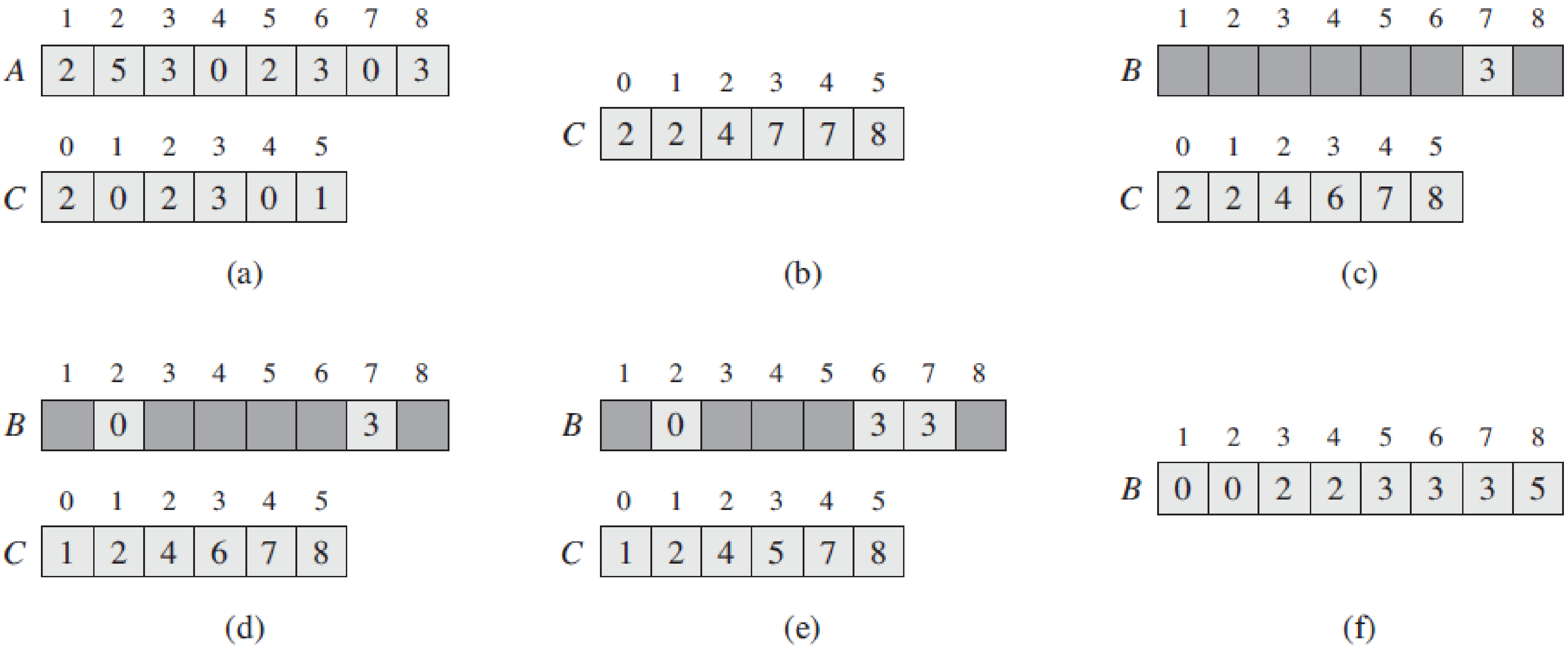


COUNTING SORT

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

```
COUNTING-SORT(A, B, k):
…
    for j = A.length to 1
        B[C[A[ j ]]] = A[ j ]
        C[A[ j ]] = C[A[ j ]] - 1
```

COUNTING SORT



RADIX SORT

A sorting algorithm for integers up to size M
(or more generally, for sorting strings)

RADIX SORT

For sorting integers where the maximum value of any integer is M.
(This can be generalized to lexicographically sorting strings as well)

IDEA:

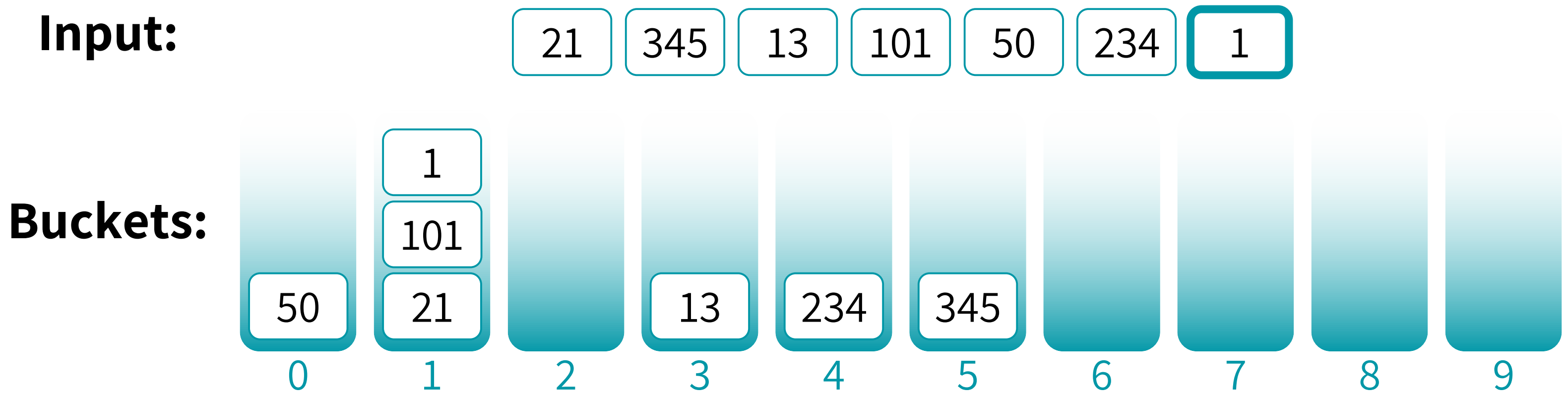
Perform CountingSort on the least-significant digit first,
then perform CountingSort on the next least-significant, and so on...

Instead of a bucket per possible value, **we just need to maintain a bucket per possible value that a single digit (or character) can take on!**

e.g. 10 buckets labeled 0, 1, …, 9

RADIX SORT

STEP 1: CountingSort on the least significant digit



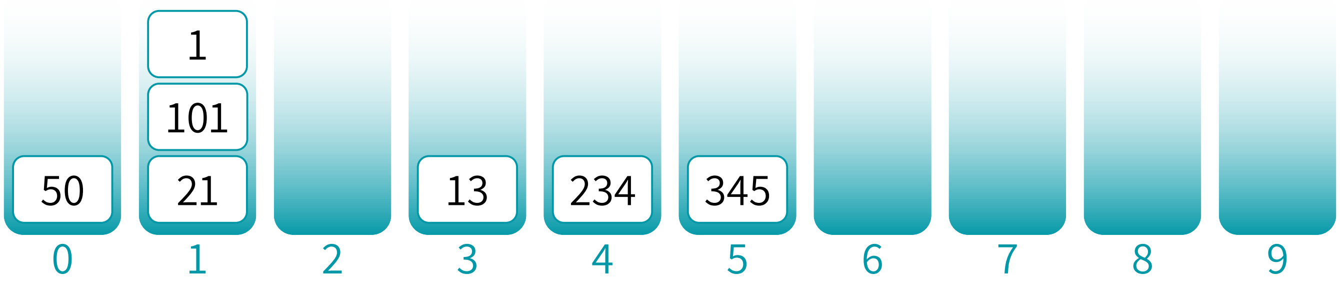
RADIX SORT

STEP 1: CountingSort on the least significant digit

Input:

21	345	13	101	50	234	1

Buckets:



Output:

50	21	101	1	13	234	345

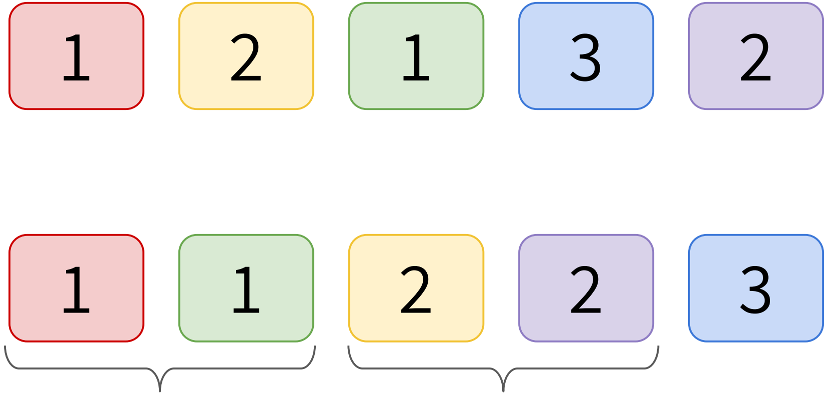
When creating the output list, make sure bucket items exit in FIFO order
(i.e. use a *stable* implementation of CountingSort, where buckets are FIFO queues)

QUICK ASIDE: STABLE SORTING

We say a sorting algorithm is STABLE if two objects with equal values appear in the same order in the sorted output as they appear in the input.

Input:

Sorted Output:
(if algorithm is stable)



The red 1 appeared before the green 1 in the input, so they have to also appear in this order in the output!

The yellow 2 appeared before the purple 2 in the input, so they have to also appear in this order in the output!

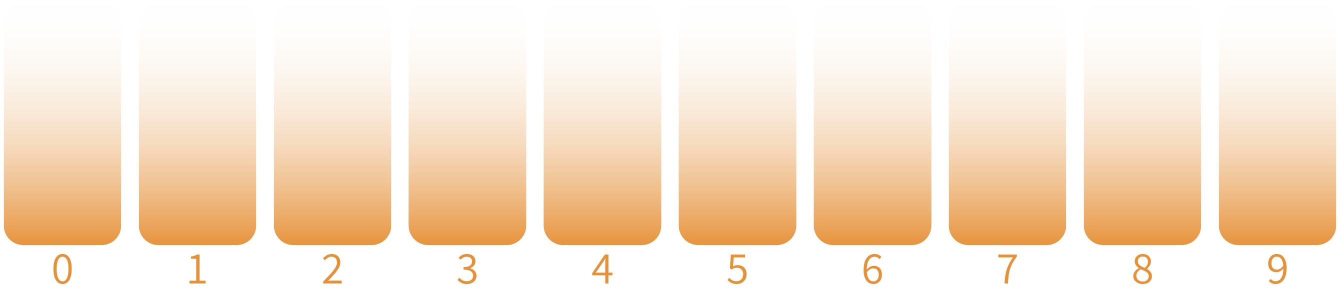
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50	21	101	1	13	234	345

Buckets:



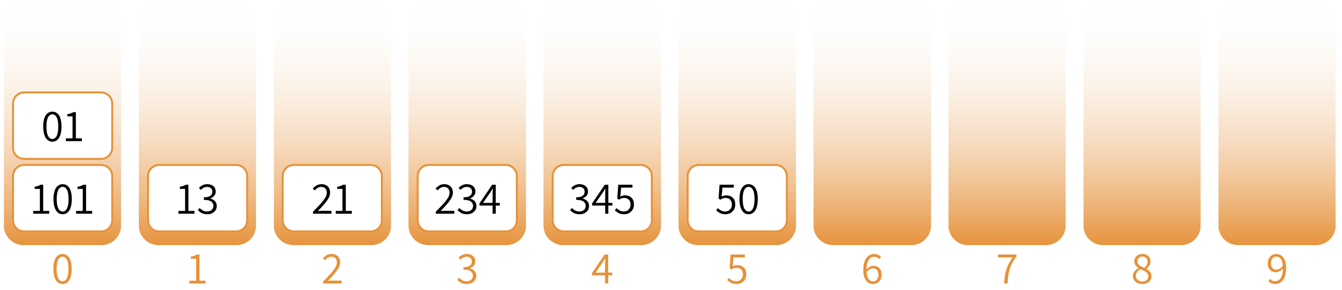
RADIX SORT

STEP 2: CountingSort on the 2^{nd} least significant digit

Input:
(output from STEP 1)

50	21	101	01	13	234	**345**

Buckets:



Output:

101	01	13	21	234	345	50

When creating the output list, make sure bucket items exit in FIFO order
(i.e. use a *stable* implementation of CountingSort, where buckets are FIFO queues)

RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	001	013	021	234	345	050

Buckets:

0	1	2	3	4	5	6	7	8	9
001, 013, 021, 050	101	234	345						

Output:

001	013	021	050	101	234	345

It worked! But why does it work???

RADIX SORT RUNTIME

Suppose we are sorting **n** (up-to-)**d**-digit numbers in base 10 (e.g. n = 7, d = 3):

21	345	13	101	50	234	1

How many iterations are there?

d iterations

How long does each iteration take?

Initialize 10 buckets + put n numbers in 10 buckets ⇒ **O(n)**

What is the total running time?

O(nd)

Bucket Sort

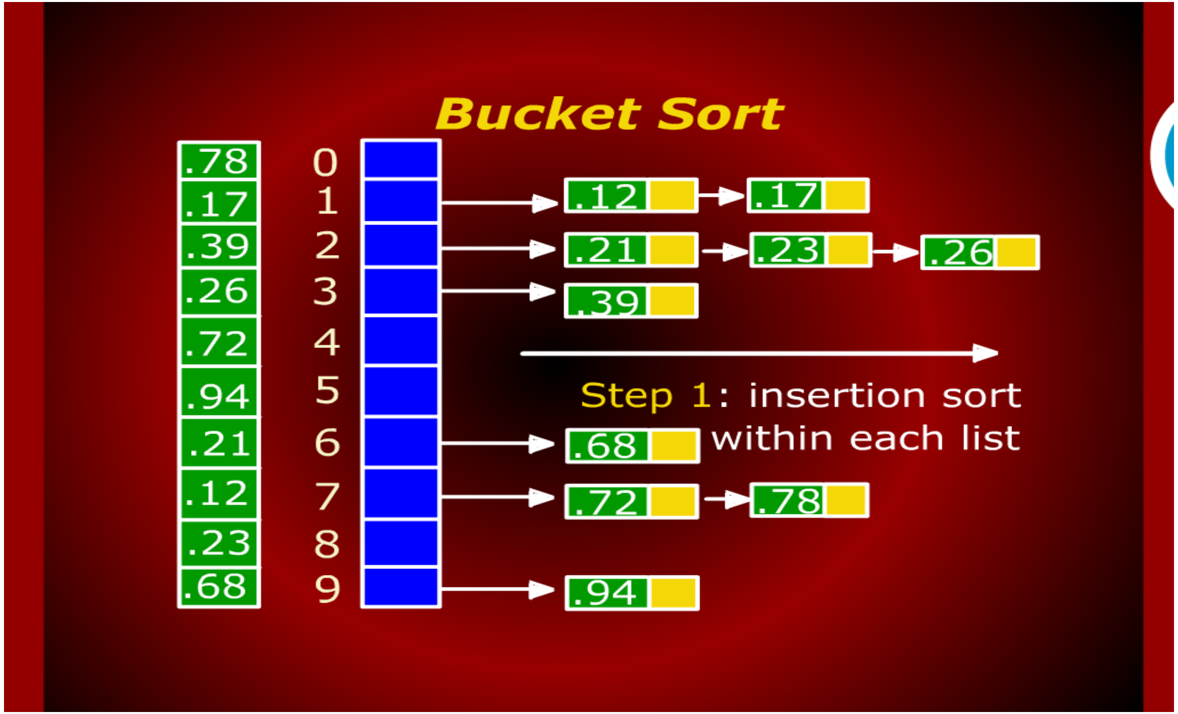
- Assumptions :

❖ Input elements are uniformly distributed over [0,1]

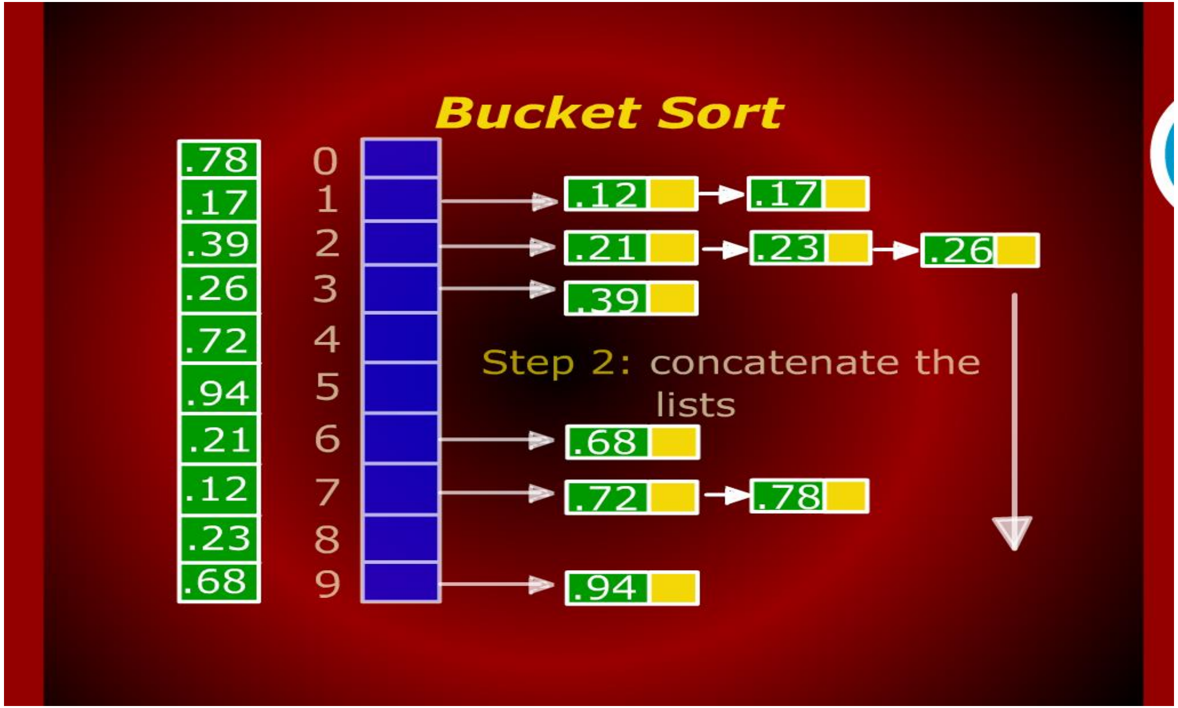
Bucket Sort

```
BUCKET-SORT(A)
let B[0..n-1] be a new array
n = A.length
for i = 0 to n - 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊n A[i]⌋]
for i = 0 to n - 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n - 1] together in order
```

Bucket Sort

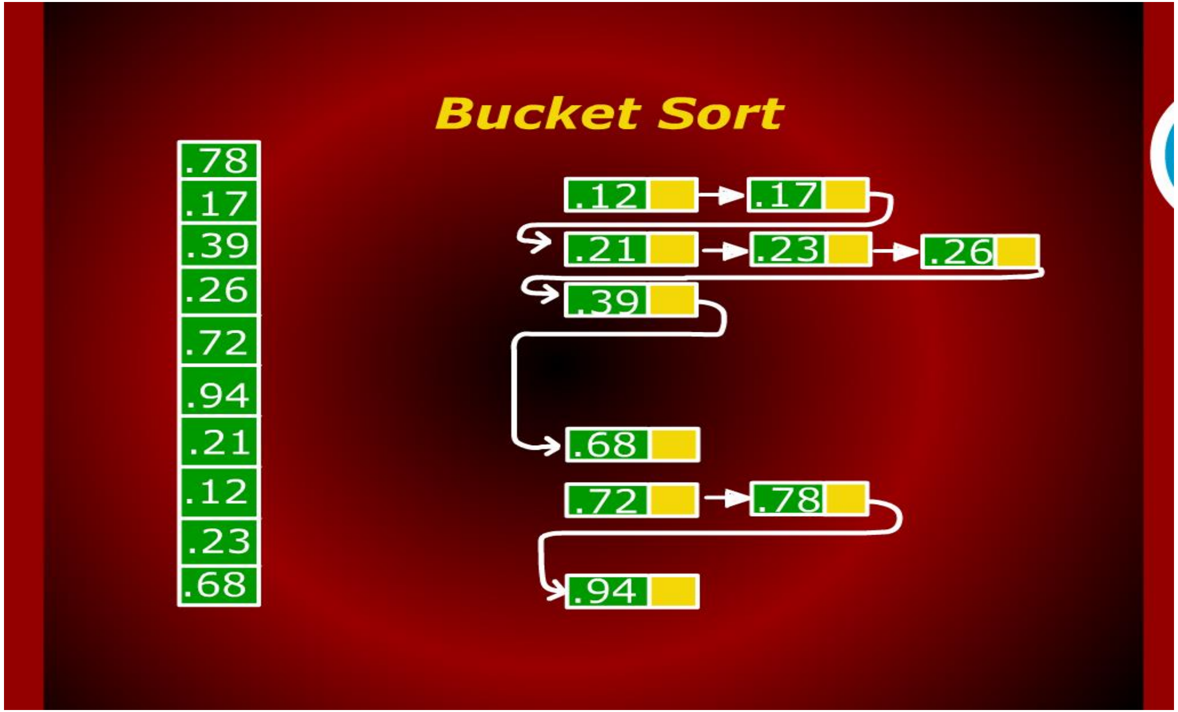

Bucket Sort
.78
.17
.39
.26
.72
.94
.21
.12
.23
.68
0
1
2
3
4
5
6
7
8
9
.12
.17
.21
.23
.26
.39
Step 1: insertion sort within each list
.68
.72
.78
.94

Bucket Sort



Bucket Sort
.78
.17
.39
.26
.72
.94
.21
.12
.23
.68
0
1
2
3
4
5
6
7
8
9
.12
.17
.21
.23
.26
.39
Step 2: concatenate the lists
.68
.72
.78
.94

Bucket Sort



Bucket Sort
.78
.17
.39
.26
.72
.94
.21
.12
.23
.68
.12
.17
.21
.23
.26
.39
.68
.72
.78
.94

Comparison of Sorting Algorithms

Algorithm	Worst Time	Extra Memory	Stable
Insertion sort	$O(n^2)$	O(1) (in place)	Yes
Merge sort	$O(n\ lgn)$	O(n)	Yes
Quick sort	$O(n^2)$	O(1) (in place)	Yes
Heap sort	$O(n\ lgn)$	O(1) (in place)	No
Counting sort	$O(n+k)$	$O(n+k)$	Yes