Lecture 18

Radix Sort
&
Counting Sort

October 25, 2021
Monday

Radix Sort

MOTIVATION

- Radix sort is a popular way of sorting used in everyday life.
    - To sort library cards
        - Create as many piles as many alphabets
        - Each card goes to the pile according to the starting alphabet
    - In early computers, it was used to sort 80-column cards of coding information
    - Sorting mail, as all zip codes have equal length.
- What about sorting list of integers. As they might have unequal length.
    - Consider [23 123 234 567 3]

RADIX SORT

- If we apply the same technique of piles of integers, we may end up like this
    - [123 23 234 3 567]
    - We can add zeros in front of each number.
    - [003 023 123 234 567]
    - Or we can look at each number as strings of bits, this way all numbers will have equal length.

RADIX SORT

- When sorting integers
  - 10 Piles numbered 0 through 9 are created.
  - In first pass, integers are put in appropriate pile according to their rightmost digit. E.g., 59 will go into pile: 9.
  - When all digits are put in piles, then piles are combined.
  - The process is repeated, now for the second rightmost digit. E.g., 59 now will go to the pile: 5.
  - The process ends after the leftmost digit of the longest number is processed.

PSEUDOCODE

```
RadixSort ( )
    for d = 1 to the position of the leftmost digit of longest number
        distribute all numbers among piles 0 through 9 according to the dth digit
        put all integers on one list.
```

PILES IMPLEMENTATION MATTERS

- The key to obtain the proper outcome
  - How 10 piles are implemented and then combined.
- If we follow First In Last Out Order (LIFO).
  - then 93 63 64 94
  - First pass:
    - pile 3: 63 93
    - pile 4: 94 64
  - Combined list: 63 93 94 64
  - Second pass
    - pile 6: 64 63
    - pile 9: 94 93
  - Combined List: 64 63 94 93

PILES IMPLEMENTATION MATTERS

- However, if we follow First In First Out (FIFO), the relative order of elements in the list is retained.
    - 93 63 64 94
    - First pass:
        - pile 3: 63 93
        - pile 4: 94 64
    - Combined list: 93 63 64 94
    - Second pass
        - pile 6: 64 63
        - pile 9: 94 93
    - Combined list: 63 64 93 94

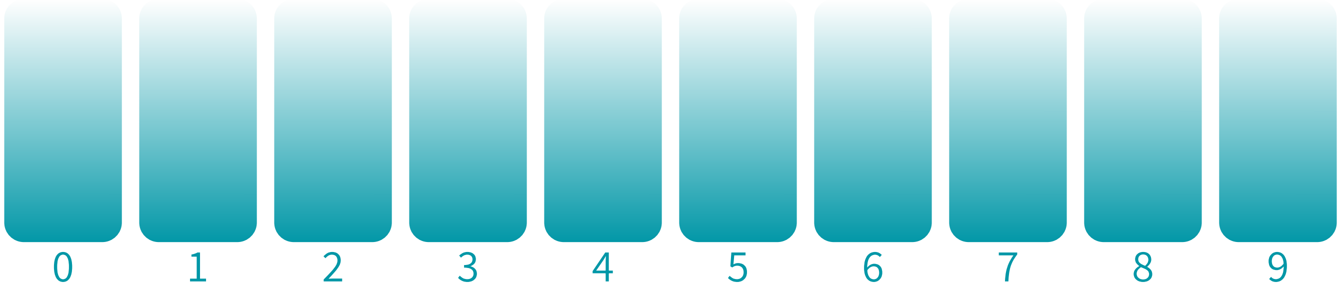
RADIX SORT

STEP 1: CountingSort on the least significant digit

Input:

21	345	13	101	50	234	1

Buckets:

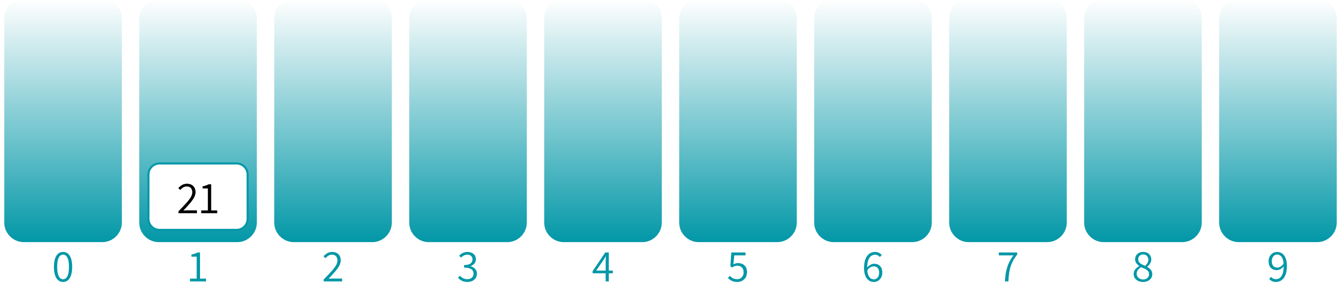


RADIX SORT

STEP 1: CountingSort on the least significant digit

Input: 21 | 345 | 13 | 101 | 50 | 234 | 1

Buckets:

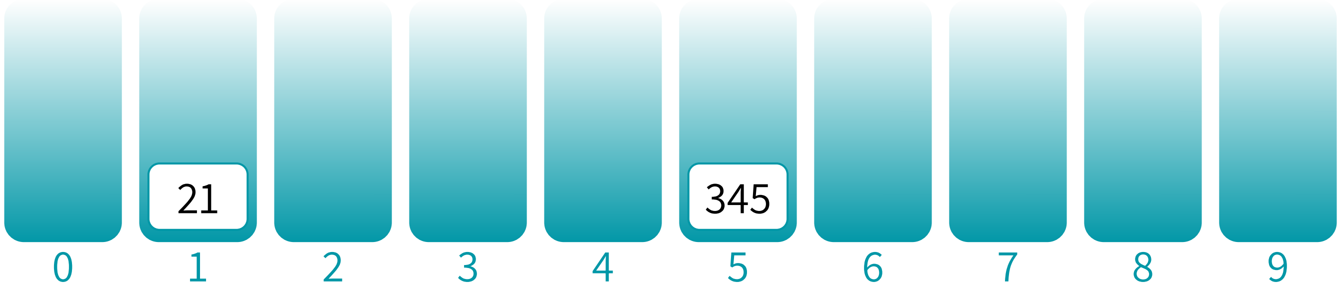


RADIX SORT

STEP 1: CountingSort on the least significant digit

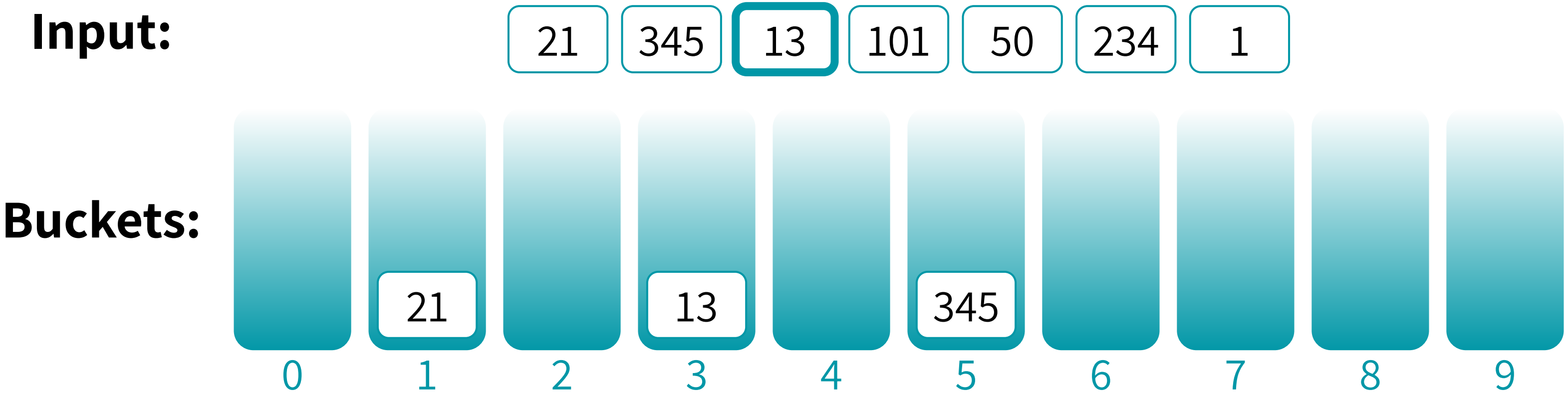
Input: 21 | 345 | 13 | 101 | 50 | 234 | 1

Buckets:



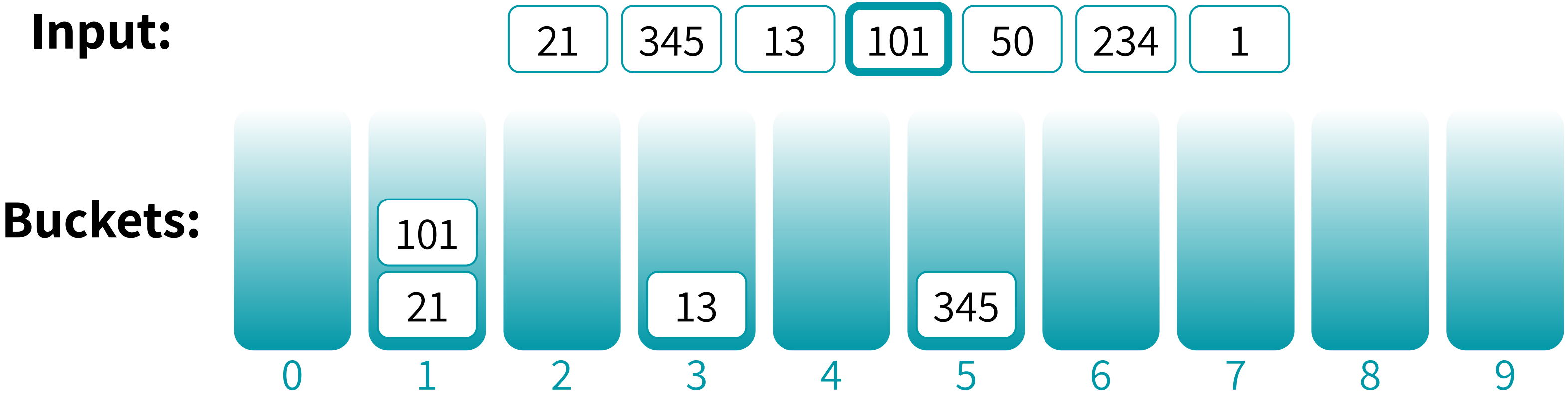
RADIX SORT

STEP 1: CountingSort on the least significant digit



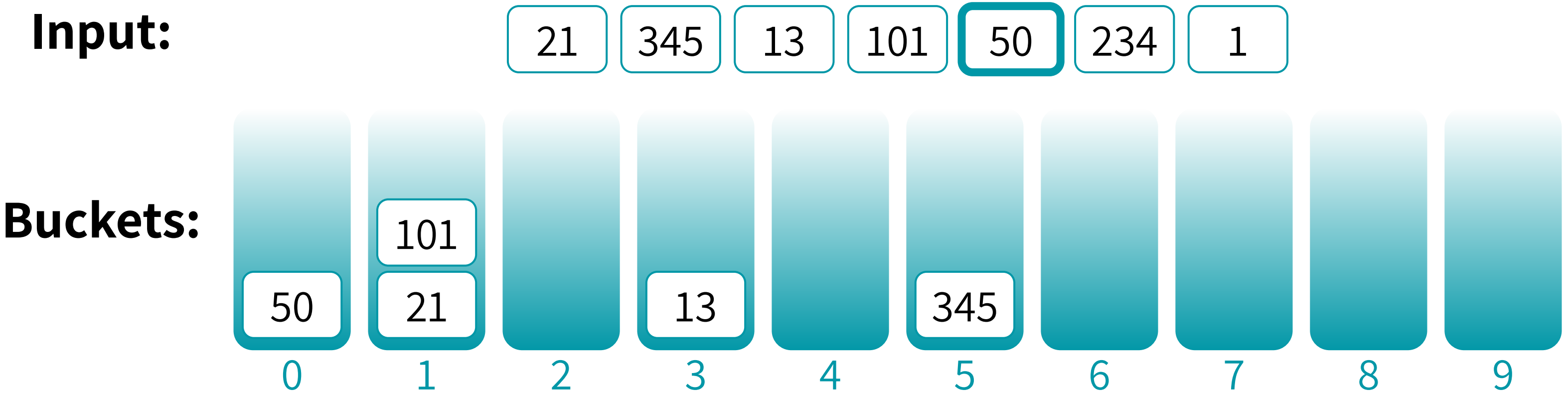
RADIX SORT

STEP 1: CountingSort on the least significant digit



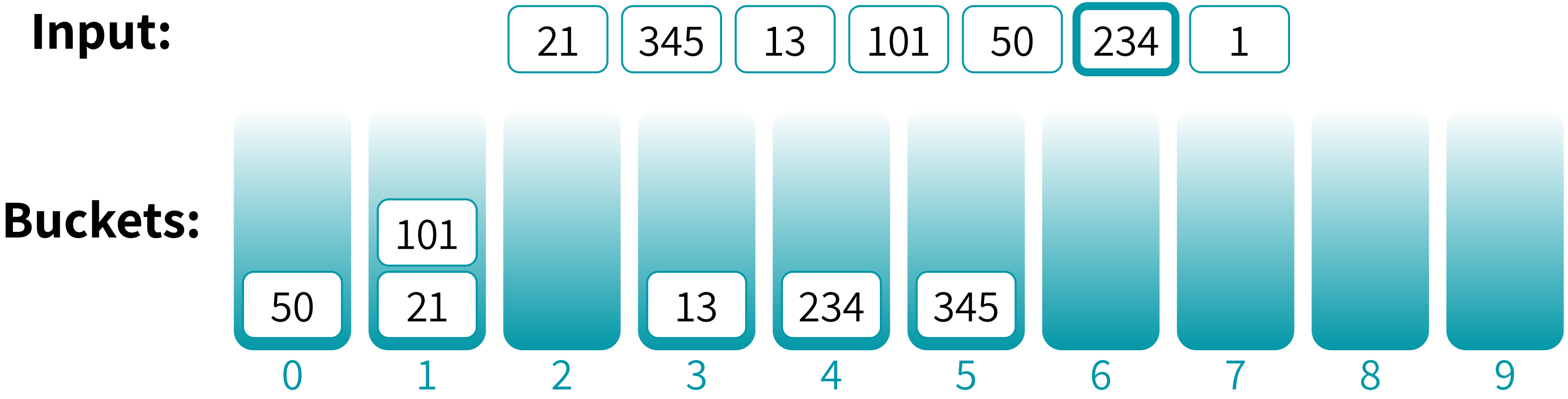
RADIX SORT

STEP 1: CountingSort on the least significant digit



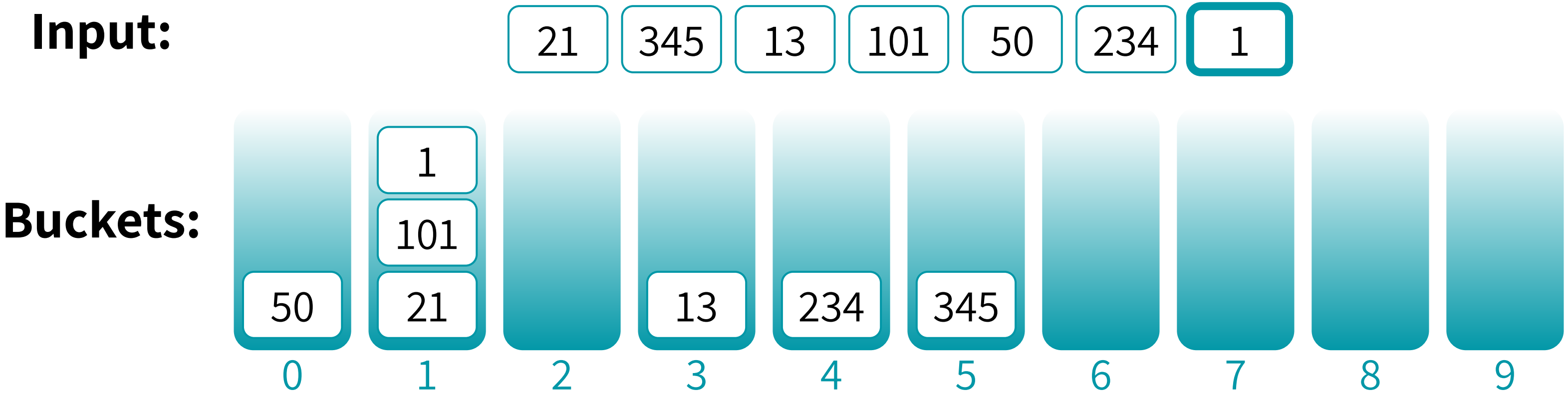
RADIX SORT

STEP 1: CountingSort on the least significant digit



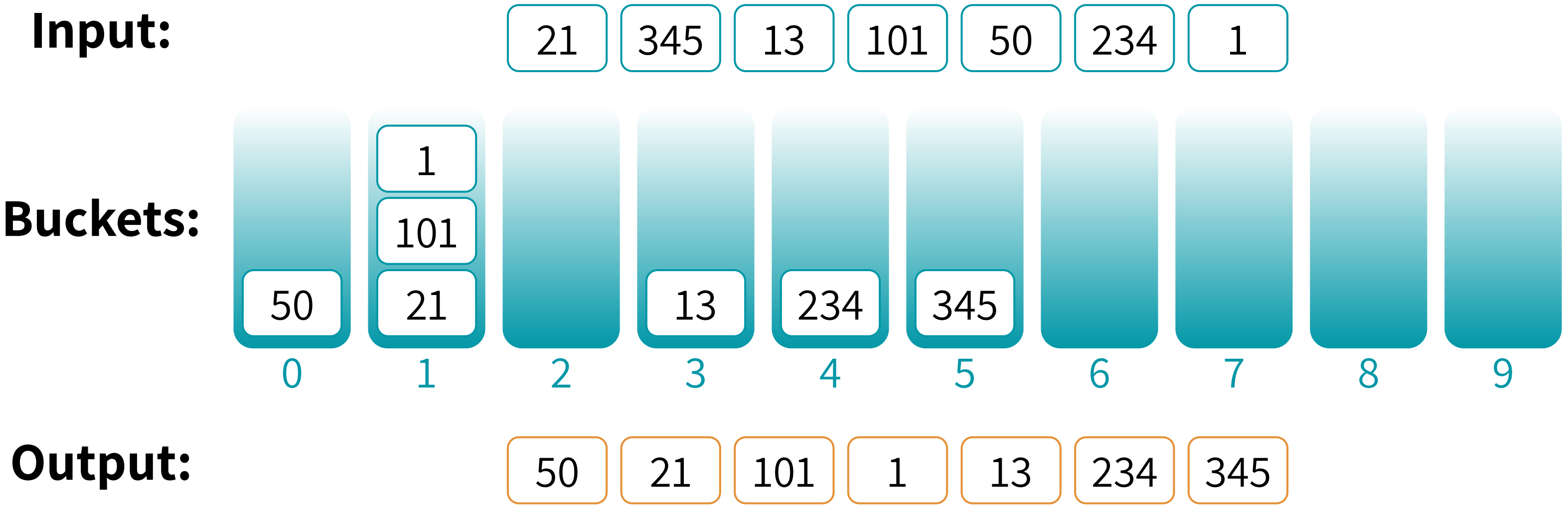
RADIX SORT

STEP 1: CountingSort on the least significant digit



RADIX SORT

STEP 1: CountingSort on the least significant digit



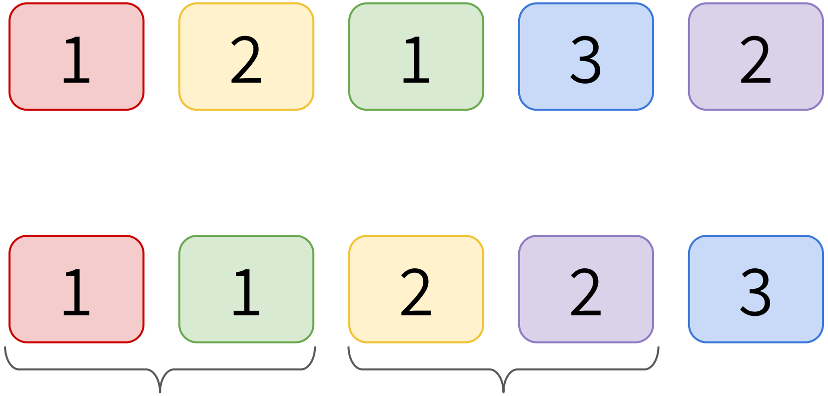
When creating the output list, make sure bucket items exit in FIFO order
(i.e. use a *stable* implementation of CountingSort, where buckets are FIFO queues)

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!

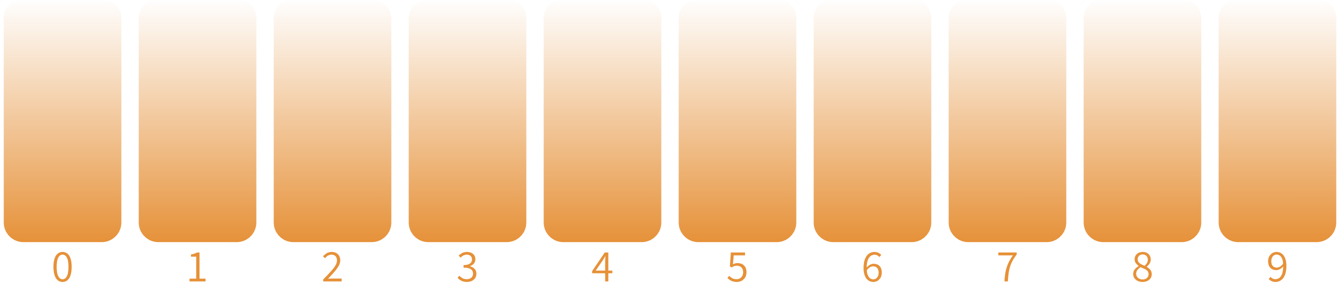
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50	21	101	1	13	234	345

Buckets:



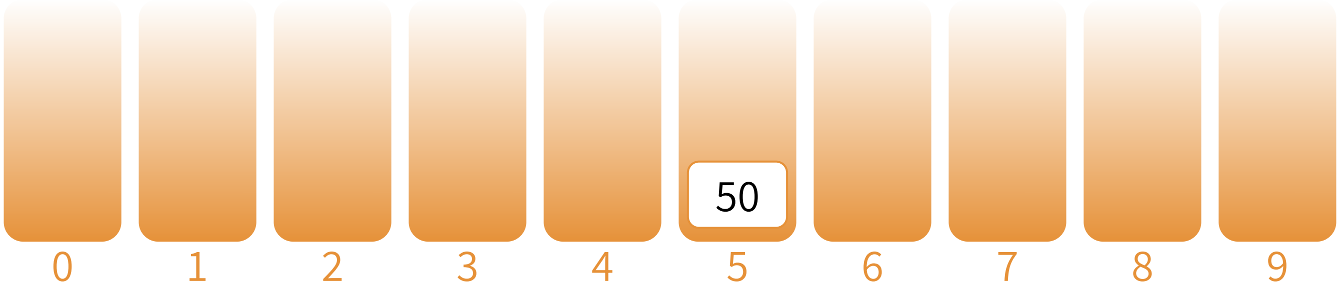
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

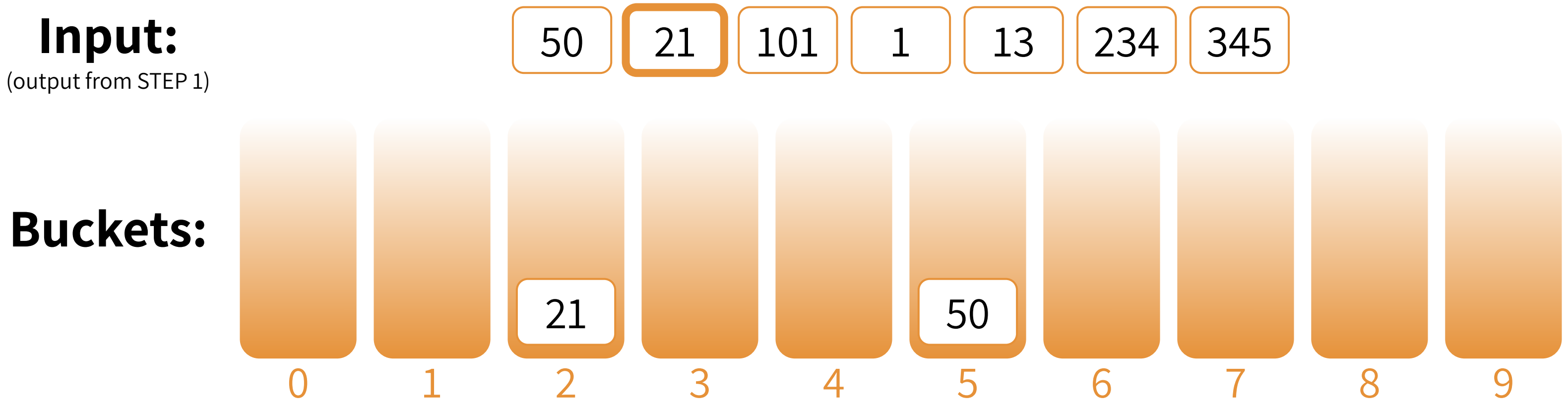
50	21	101	1	13	234	345

Buckets:



RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

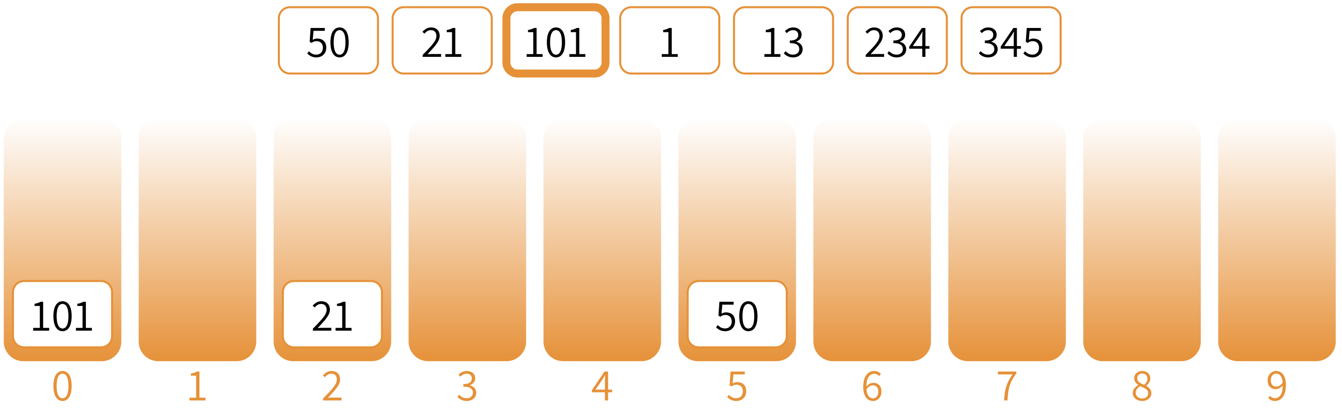


RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

Buckets:

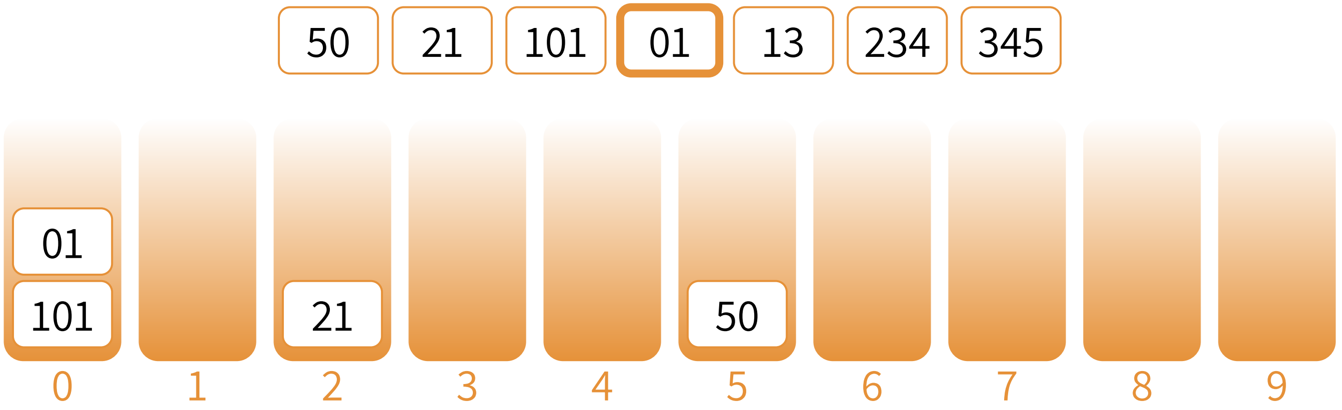


RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

Buckets:

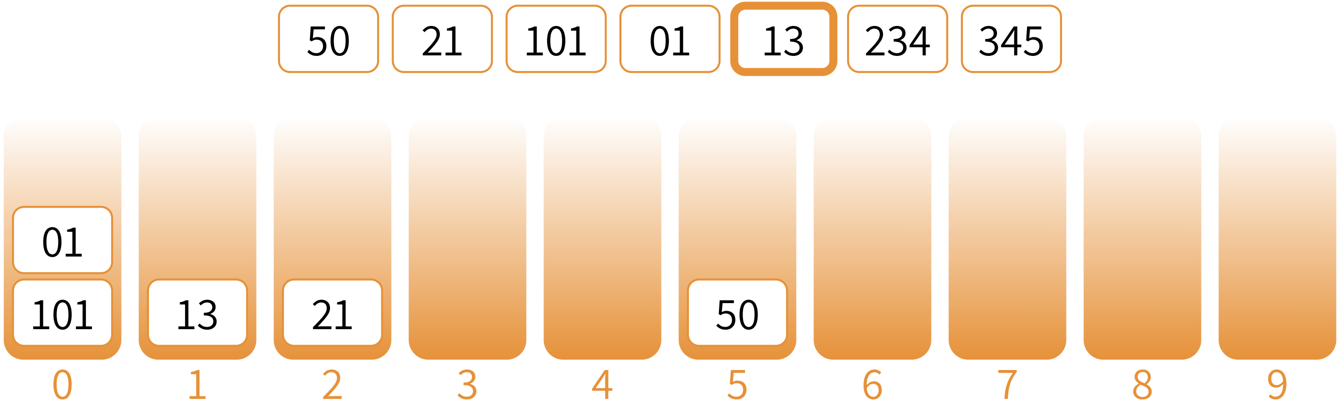


RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

Buckets:

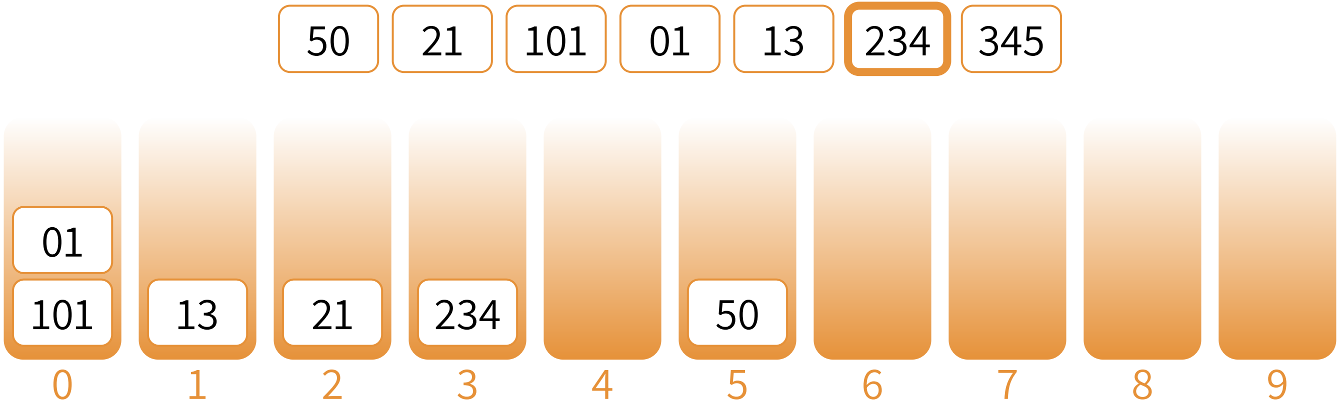


RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

Buckets:



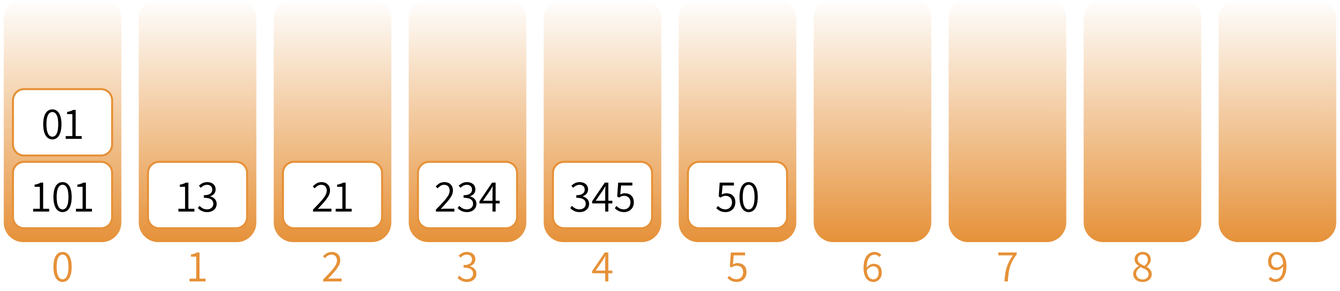
RADIX SORT

STEP 2: CountingSort on the 2nd 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)

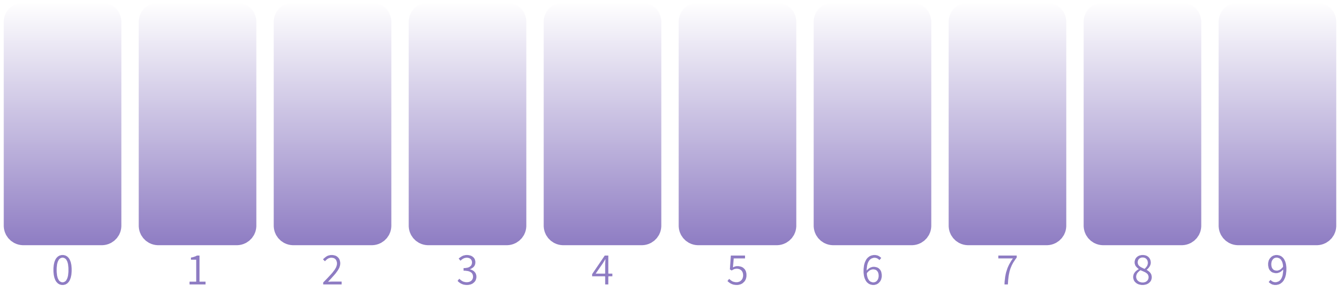
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	01	13	21	234	345	50

Buckets:



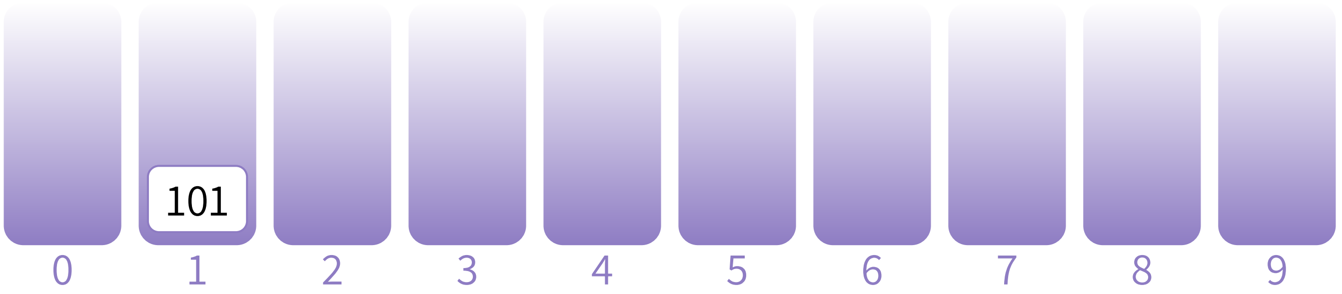
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	01	13	21	234	345	50

Buckets:



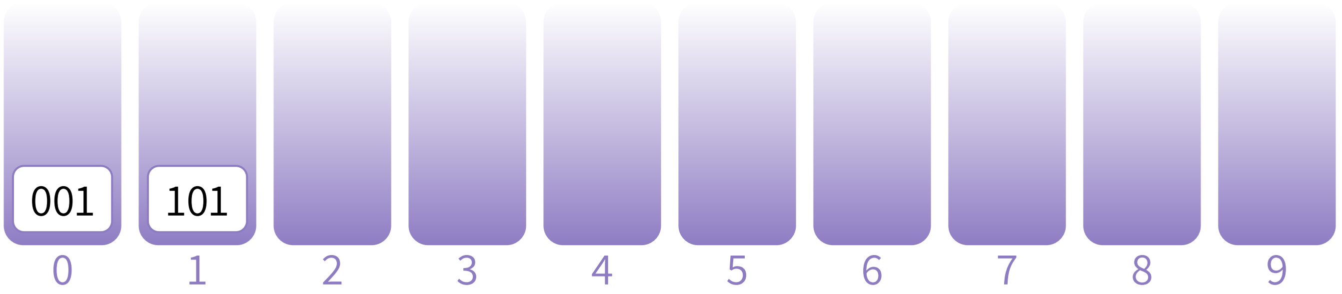
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	**001**	13	21	234	345	50

Buckets:

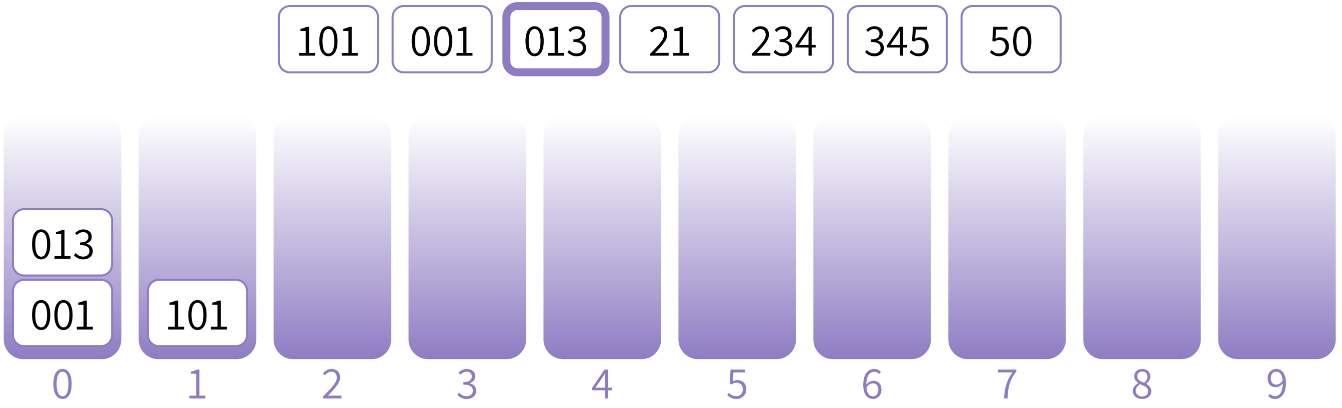


RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

Buckets:

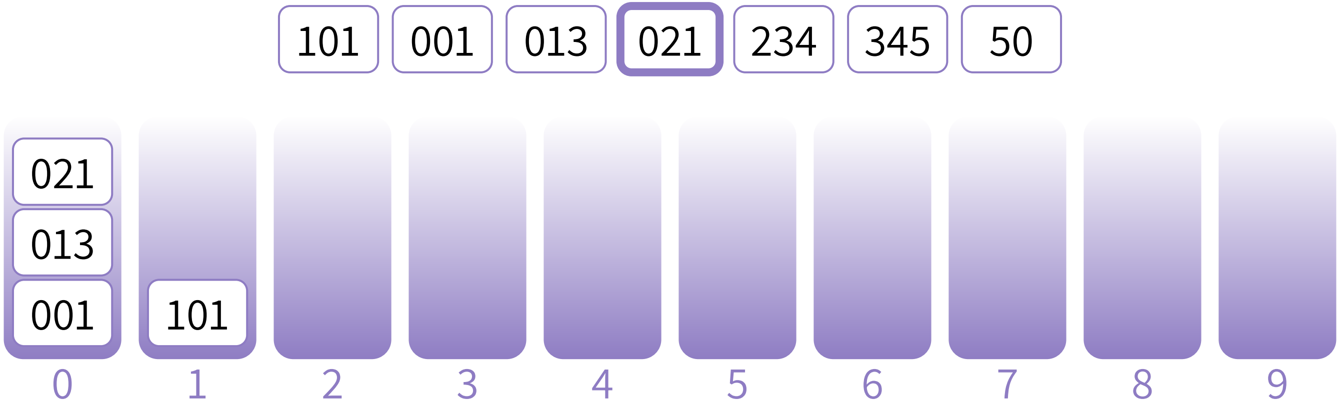


RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

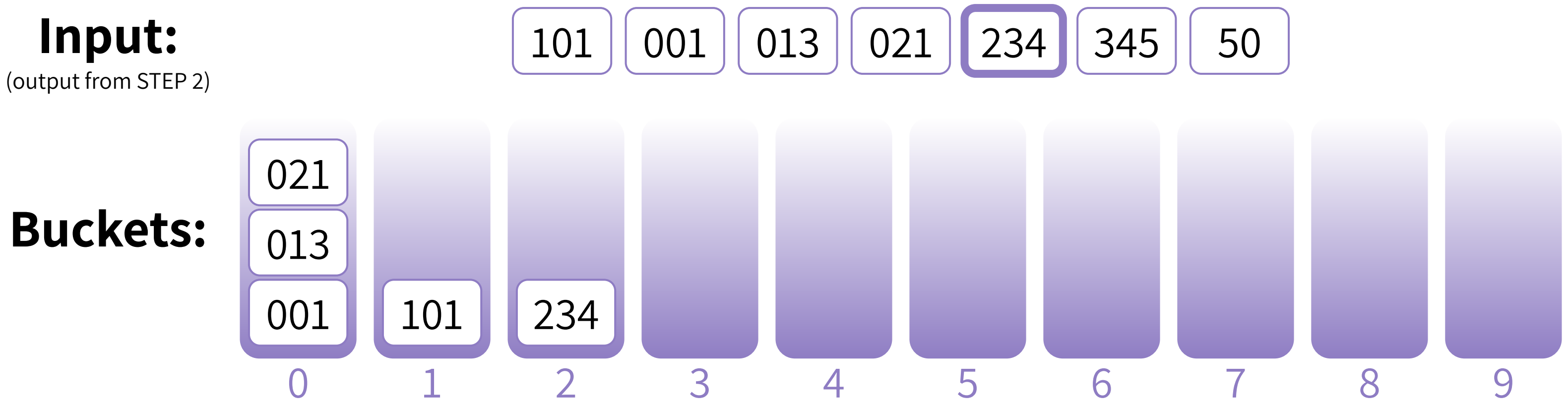
Input:
(output from STEP 2)

Buckets:



RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit



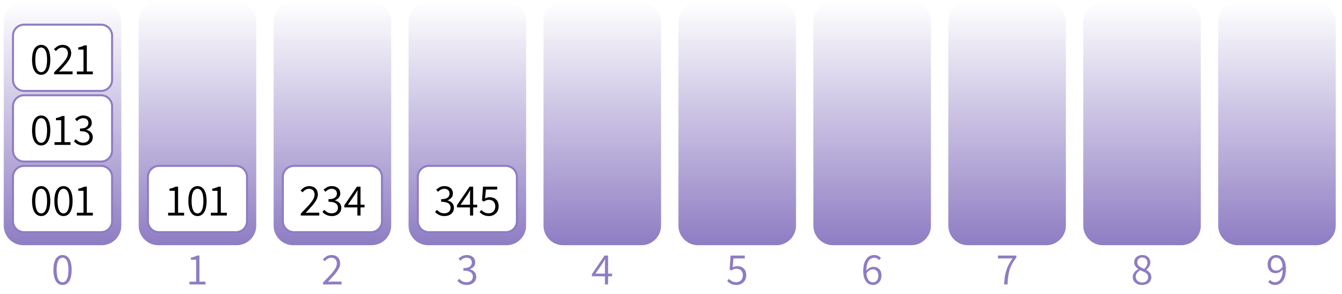
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	001	013	021	234	**345**	50

Buckets:



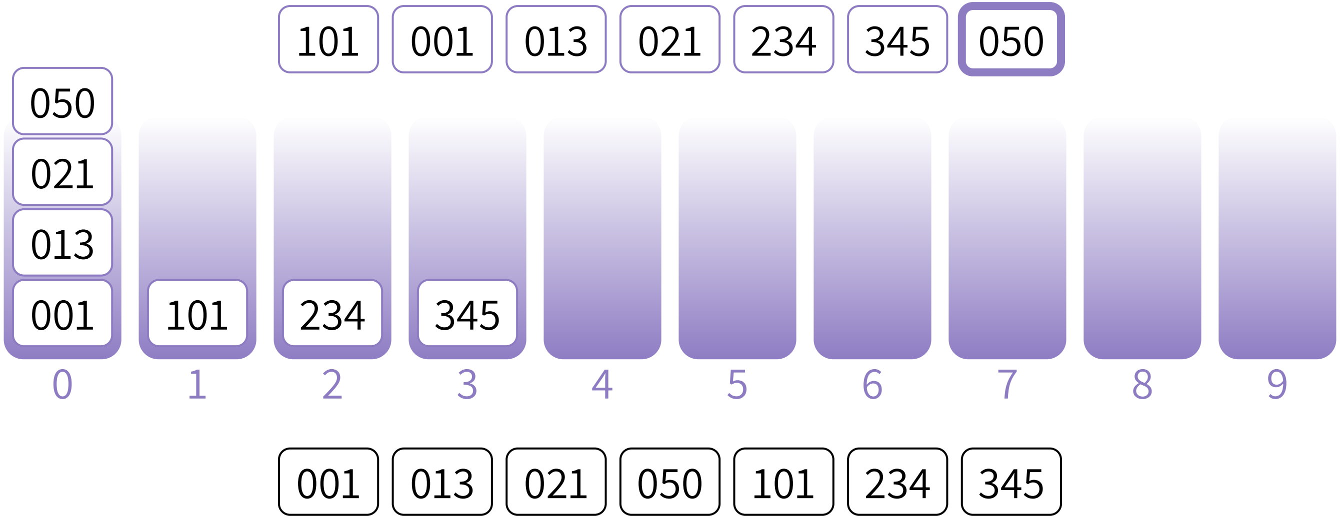
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

Buckets:

Output:



It worked! But why does it work???

POINT TO FOCUS

- When integers are sorted according to the digit in position *d*.
  - Then within each pile, integers are sorted to the part of the integer extending from digit 1 to d-1.
  - Consider the pile 5 after third pass containing items
    - 12534, 554, 3590.
    - This pile is ordered with respect to two rightmost digits of each number

IMPLEMENTATION

```
Void RadixSort ( long data [ ], int n)
    register int d, j, k, factor;
    const int radix = 10;  const int digits = 10;
    Queue<long>    queues [ radix ] ;

    for ( d = 0, factor = 1; d < digits; factor *= radix, d++) {
        for ( j = 0; j < n; j++ )
            queues [ ( data [ j ] / factor ) % radix ].enqueue (data [ j ] );

        for ( j = k = 0; j < radix; j++ )
            while ( ! queues [ j ].empty ( ) )
                data [ k++ ] = queues [ j ] .dequeue ( ) ;
    }
}
```

EXAMPLE PAGE 523 TEXTBOOK

Figure 9.15

TIME COMPLEXITY

- The algorithm does not rely on data comparison as did the previous sorting methods did.
- For each integer from *data [ ]*, two operations are performed.
  - Division by a factor, to disregard digits following the digit *d*
  - Modulus by a radix, to disregard digits preceding the digit *d*
  - For a total of 2n digits = ***O (n)*** operations.
  - All integers are moved to piles and then back to *data [ ]*,
  - For a total of 2n digits = ***O (n)*** moves.

TIME COMPLEXITY

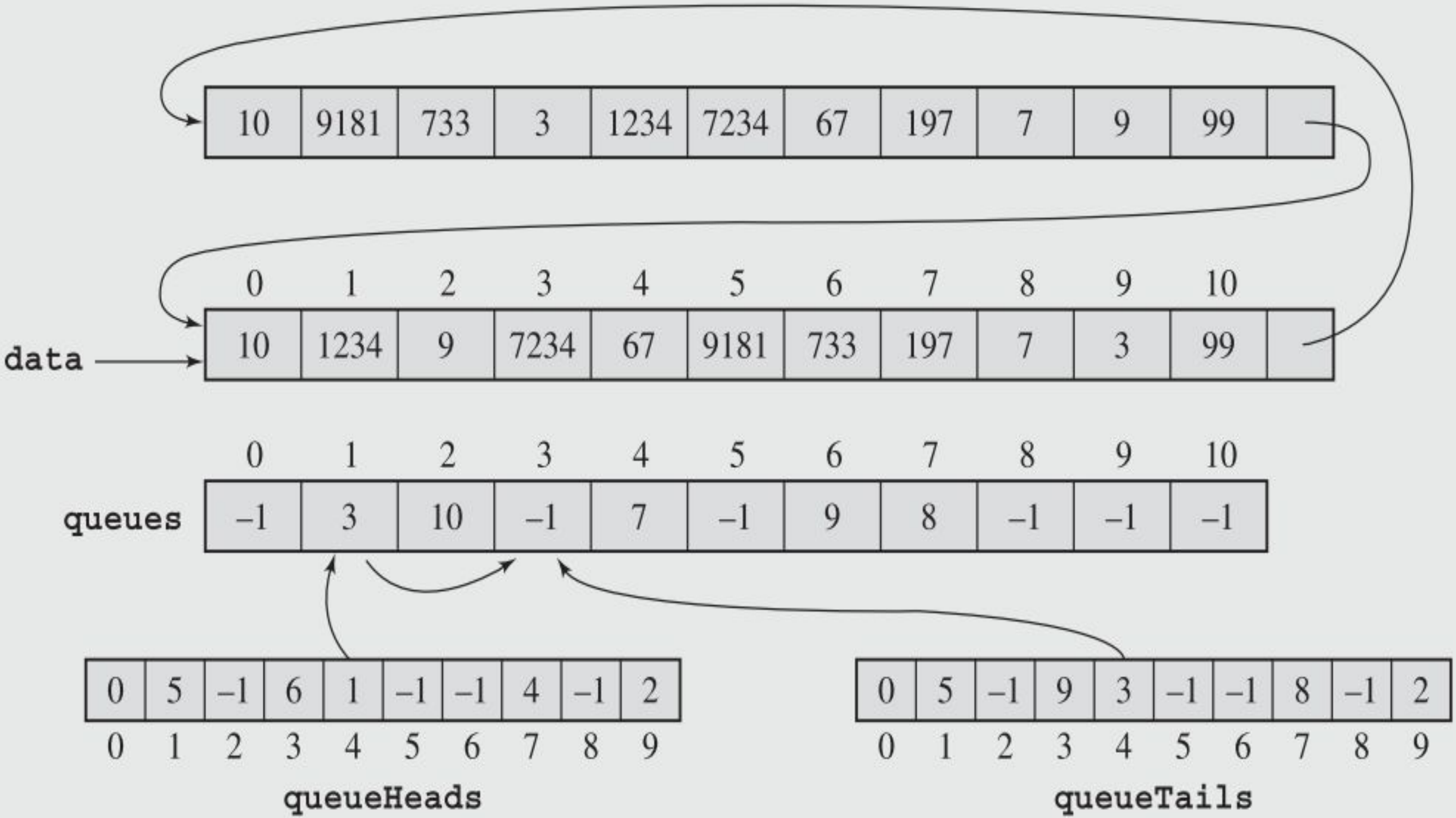
- Since the implementation uses only *for* loops with counters.
  - Therefore, it requires the same amount of passes for each case.
  - Best, Average and Worst Case are equal.
- The body of the only while loop is executed *n* times to dequeue integers from all queues.

LIMITATIONS OF LINKED LISTS APPROACH

- The algorithm requires additional space for piles.
  - Which is implemented as linked lists
  - Occupying *kn* bytes depending on the size *k* of the pointers.

A BETTER APPROACH

- A better approach is an array of size *n* for each queue.
  - This requires creating these queues only once.
- The efficiency of the algorithm depends only on the number of exchanges.
  - Copying data to the queues.
  - Copying data from the queues.
- What if radix r is a large number and a large amount of data has to be stored, then the solution requires r queues of size n.
  - The number (r + 1) . n may become unrealistically large.


10
9181
733
3
1234
7234
67
197
7
9
99
0
1
2
3
4
5
6
7
8
9
10
data
10
1234
9
7234
67
9181
733
197
7
3
99
0
1
2
3
4
5
6
7
8
9
10
queues
–1
3
10
–1
7
–1
9
8
–1
–1
–1
0
5
–1
6
1
–1
–1
4
–1
2
0
1
2
3
4
5
6
7
8
9
queueHeads
0
5
–1
9
3
–1
–1
8
–1
2
0
1
2
3
4
5
6
7
8
9
queueTails

A BETTER APPROACH

- The next stage orders data according to the information gathered in *queues*
  - Copies all the data from the original array to some temporary storage and then back to this array.
- The improvement is significant because the new implementation runs several times faster than the implementation that uses queues.


3. 백의 자릿수를 기준으로
각 버킷에 넣고 순서대로 뺀다
7
9
5
1
5
3
7
9
3
4
5
6
7

Counting Sort

COUNTING SORT

- The Counting Sort counts the number of times each number occurs in the array data [ ].
    - Using an array count [ ].
    - count [ ] is indexed with numbers from data [ ].
    - Counters indicating the number of integers ≤ *i* are added and stored in count [i]
    - This way count [i] - 1 indicates the home position of i in data [ ].

IMPLEMENTATION

CPP FILE IS PROVIDED

COUNTING SORT

- Counting sort is linear in max (n, largest number in data [ ])
- This means that, even for small arrays it can be very expensive
  - If at least one number in data is very large.
  - Consider: [1, 2, 1, 10000].
  - Count would have 10001 cells all of them would have to be processed.
- If it can be guaranteed that all numbers in data [ ] are small
  - Then counting sort is very efficient even for very large arrays.

COUNTING SORT

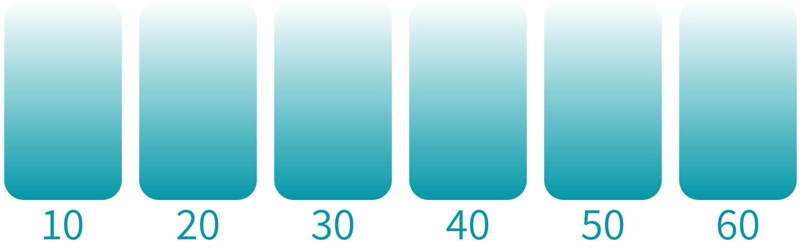
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:

30	50	20	30	10	60	50	20

Buckets:



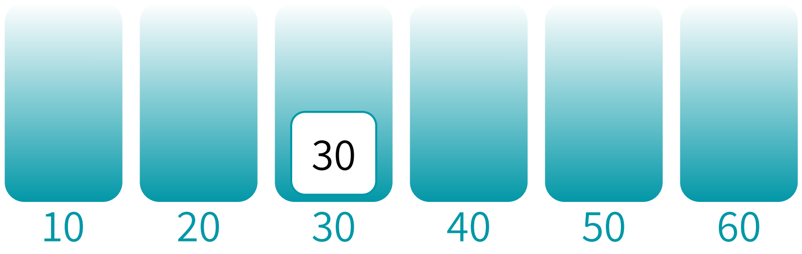
COUNTING SORT

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}

Input: 30 50 20 30 10 60 50 20

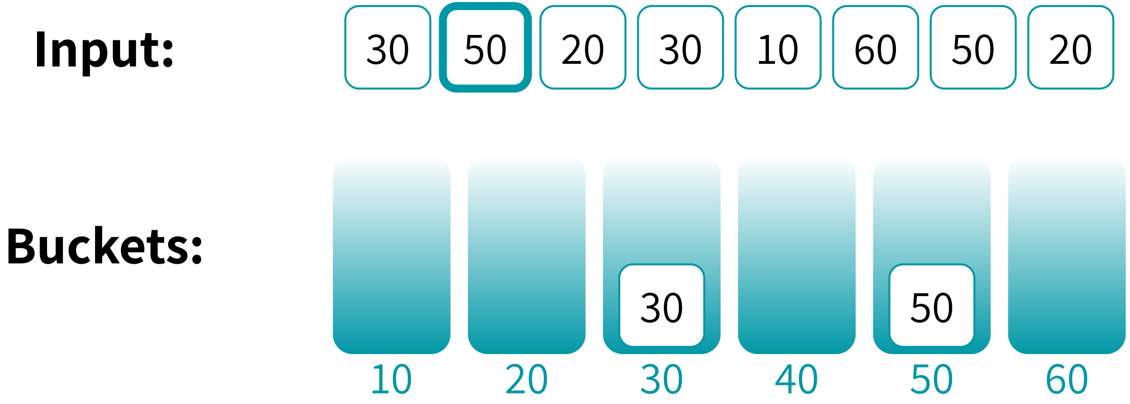
Buckets:



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	

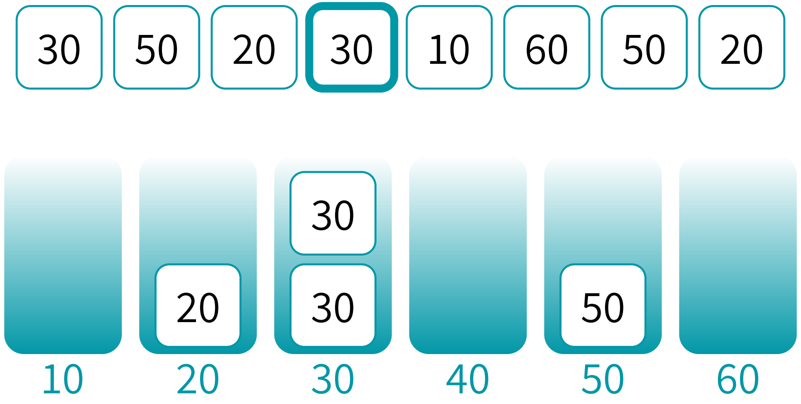
COUNTING SORT

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:

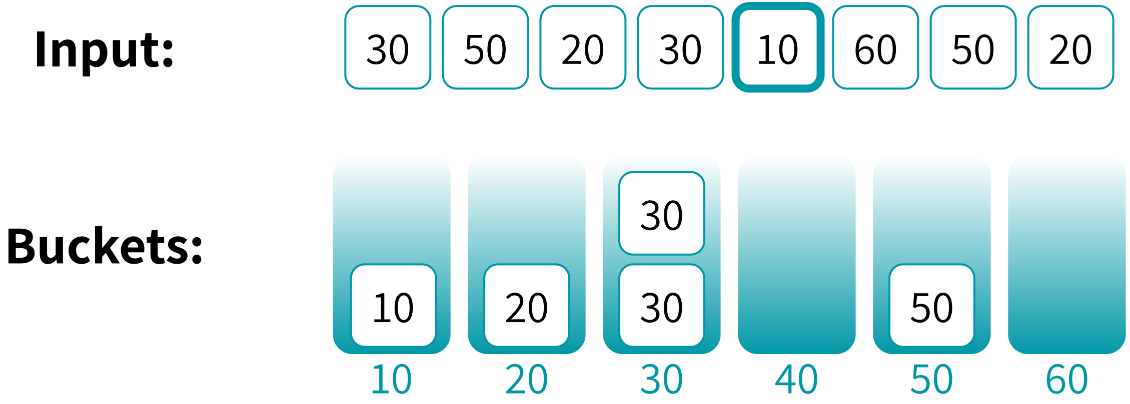
Buckets:



COUNTING SORT

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

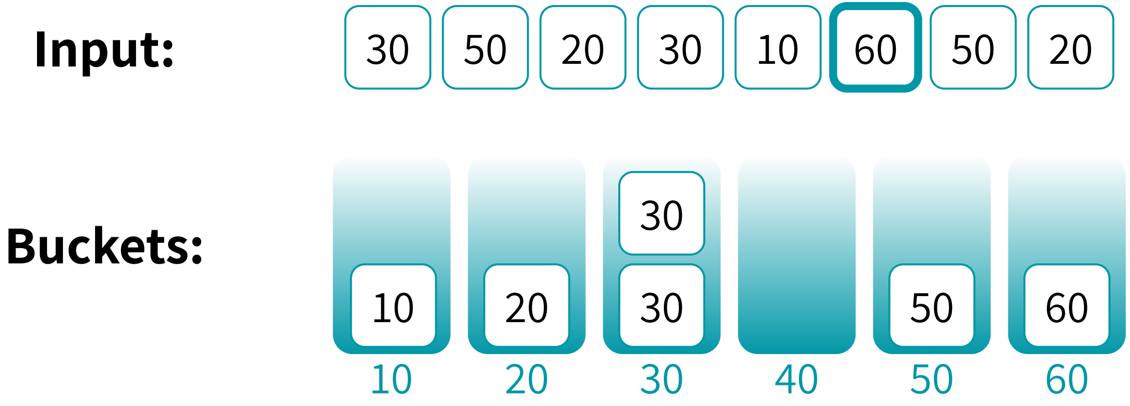
For example: elements are integers in {10, 20, 30, 40, 50, 60}



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}



COUNTING SORT

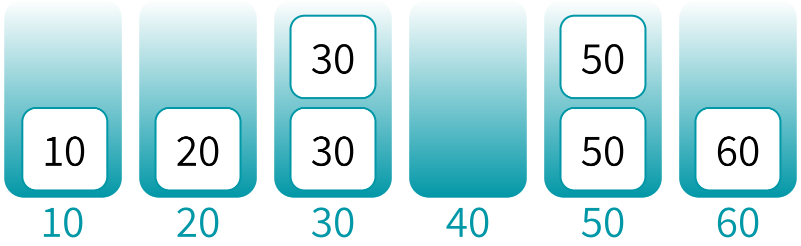
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:

30	50	20	30	10	60	50	20

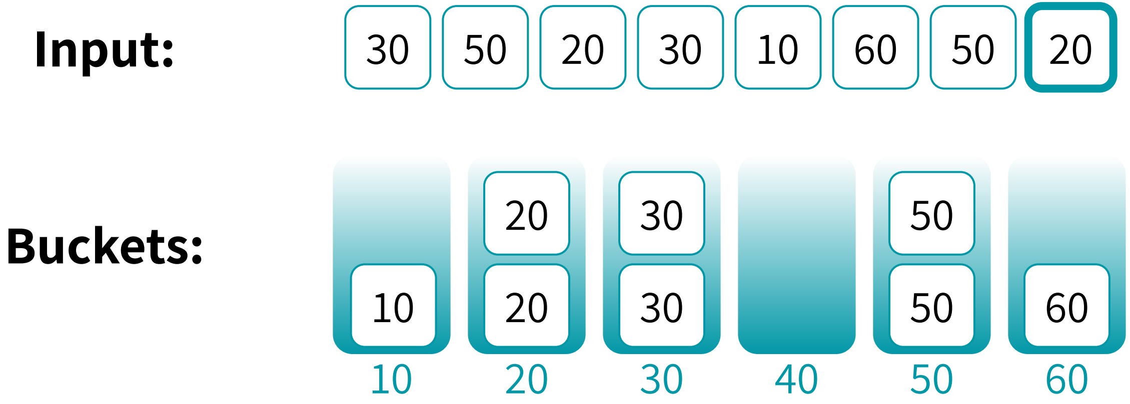
Buckets:



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

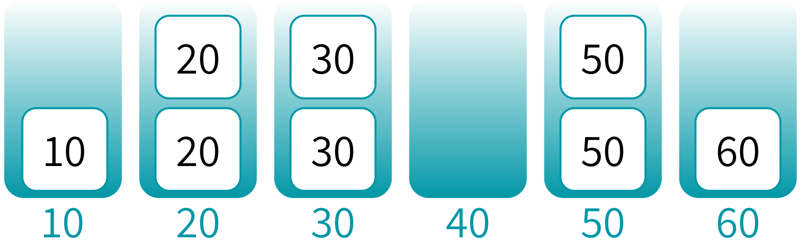
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:

30	50	20	30	10	60	50	20

Buckets:

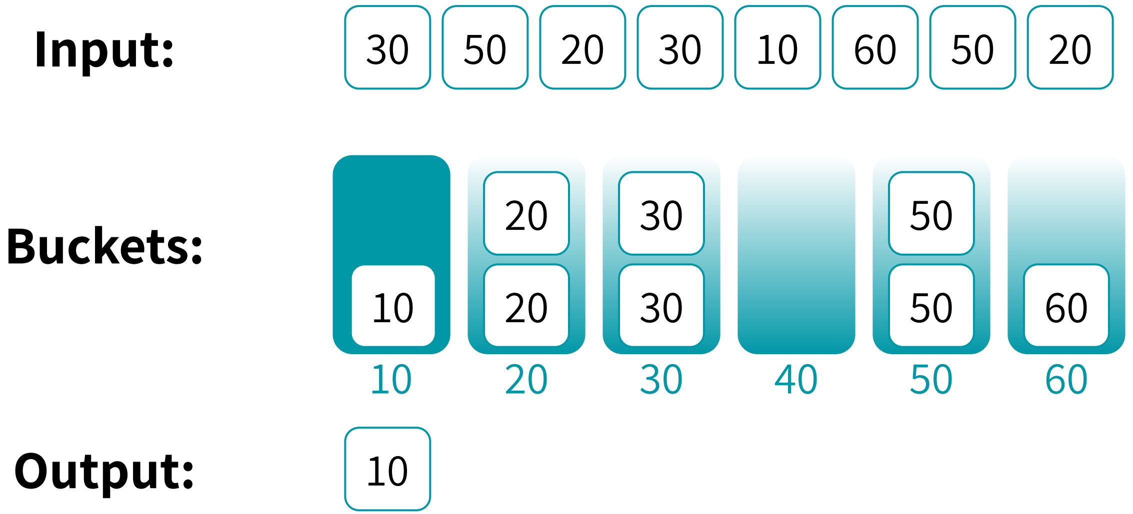


Output:

COUNTING SORT

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

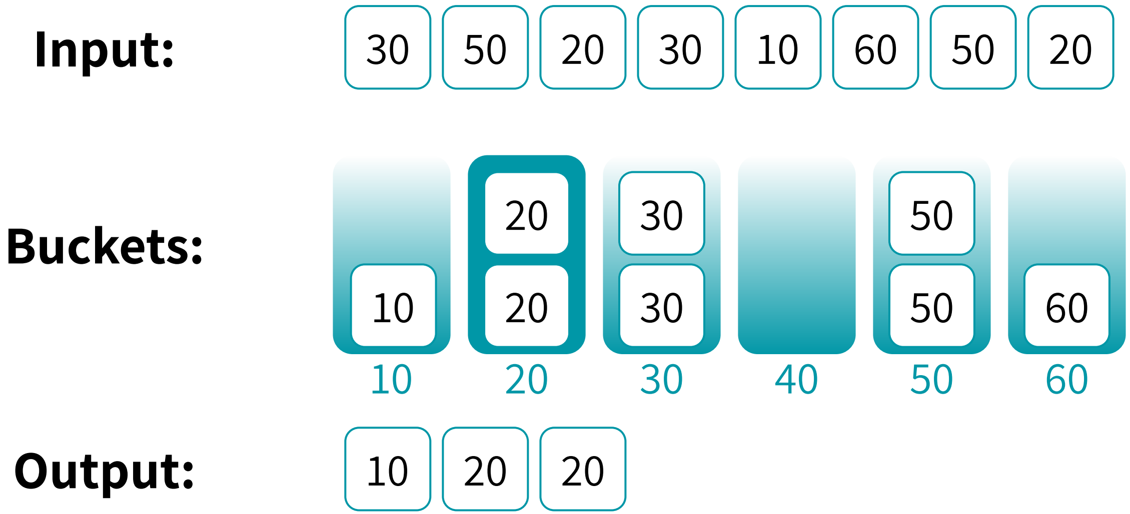
For example: elements are integers in {10, 20, 30, 40, 50, 60}



COUNTING SORT

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

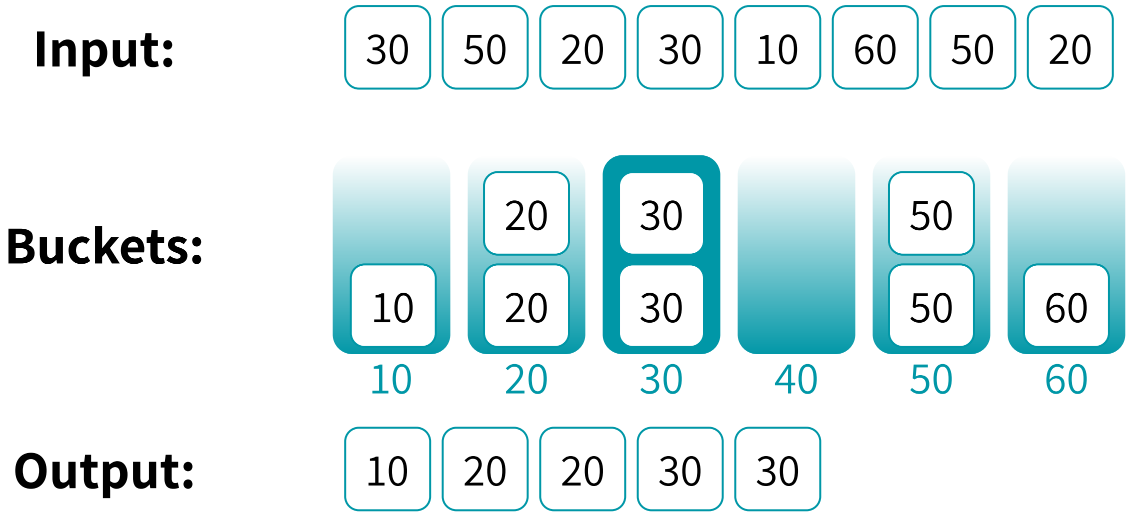
For example: elements are integers in {10, 20, 30, 40, 50, 60}



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}



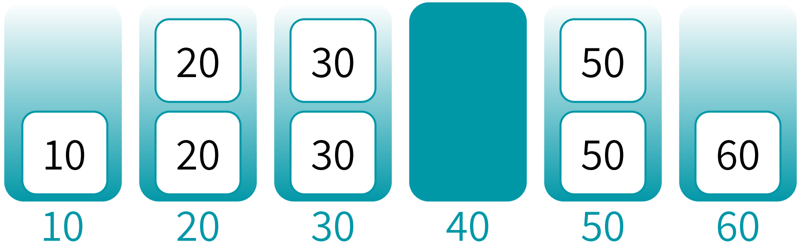
COUNTING SORT

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:



Output: 10 | 20 | 20 | 30 | 30

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

COUNTING SORT

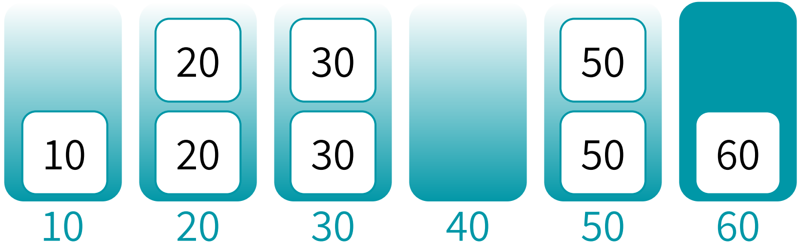
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:

30	50	20	30	10	60	50	20

Buckets:



Output:

10	20	20	30	30	50	50	60

COUNTING SORT

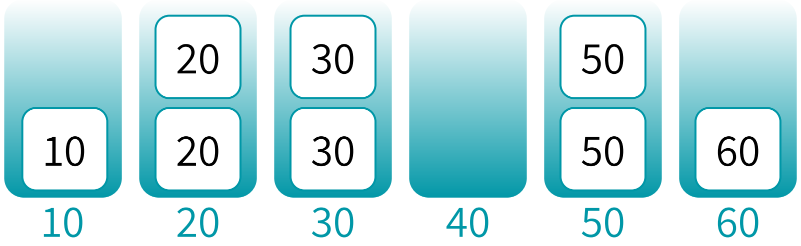
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:

30	50	20	30	10	60	50	20

Buckets:



Output:

10	20	20	30	30	50	50	60

Sorted in time: **O(n)**

COUNTING SORT

- Counting sort can be embedded with Radix Sort
  - Because it is stable.
  - Partial order obtained in one pass is not disturbed by a next pass.

EXAMPLE PAGE 526 TEXTBOOK
Figure 9.17

TIME COMPLEXITIES SORTING ALGORITHMS					
Sort	**Best**	**Average**	**Worst**	**Space**	**Stable**
Insertion	*O (n)*	$O (n^2)$	$O (n^2)$	*O (1)*	*Yes*
Selection	$O (n^2)$	$O (n^2)$	$O (n^2)$	*O (1)*	*No*
Bubble	$O (n^2)$	$O (n^2)$	$O (n^2)$	*O (1)*	*Yes*
Shell	*O (n log n)*	*O (n log n)*	$O (n^2)$	*O (1)*	*No*
Quick	*O (n log n)*	*O (n log n)*	$O (n^2)$	*O (1)*	*No*
Merge	*O (n log n)*	*O (n log n)*	*O (n log n)*	*O (n)*	*Yes*
Radix	*O (n)*	*O (d (n + r))*	*O (d (n + r))*	*O (n + b)*	*Yes*
Counting	*O (n)*	*O (n)*	*O (n)*	*O (n + r)*	*Yes*