



# **UNIT-1**

## **Sorting in Linear Time**

Lower bounds for sorting - Radix sort,  
Bucket sort

# How Fast Can We Sort?

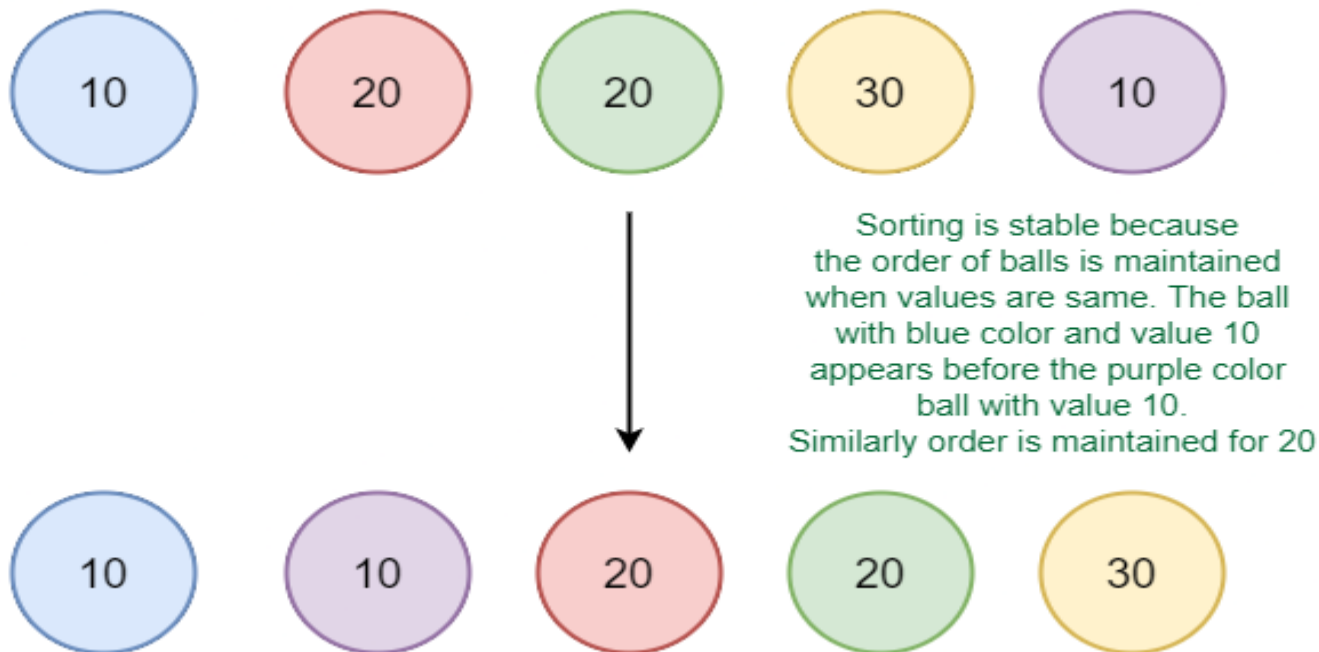
- An observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are  $\Omega(n \lg n)$

# Lower Bound For Comparison Sorts

- The time for comparison sort of  $n$  elements is  $\Omega(n \lg n)$
- Corollary: Heapsort and Mergesort are asymptotically optimal comparison sorts
- But the name of this lecture is “Sorting in linear time”!
  - *How can we do better than  $\Omega(n \lg n)$ ?*

# Stable Sorting Algorithm

- A sorting algorithm is said to be stable if **two objects with equal keys appear in the same order in sorted output as they appear in the input data set.**



# Radix Sort

- Key idea: Sort the *least* significant digit first

```
RadixSort(A, d)
```

```
    for i=1 to d
```

```
        StableSort(A) on digit i
```

# Radix Sort

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix  $2^{16}$  numbers
  - Can sort in just four passes with radix sort!
- Compares well with typical  $O(n \lg n)$  comparison sort
  - Requires approx  $\lg n = 20$  operations per number being sorted
- *So why would we ever use anything but radix sort?*

# Radix Sort

---

- In general, radix sort is
  - Fast
  - Asymptotically fast (i.e.,  $O(n)$ )
  - Simple to code
  - A good choice


# Radix Sort Example: LSD to MSD

- The following array needs sorting:

*Array[] = {455, 61, 63, 45, 67, 135, 74, 49, 15, 5}*

- We start off by finding the maximum element in the unsorted input array (maxim). The number of digits (d) in maxim gives us the number of passes we need to run to get the fully sorted output.
- Here, the maxim = 455 and d = 3. This tells us that 3 passes will be required to fully sort the array
- The loop will run once for units place, once for the tens place, and once for hundreds place before the array is finally sorted.
- *This array consists of integers in the decimal number system – so the digits will range from 0 to 9.*



- 
- *We start sorting from the least significant digit (LSD).*

*Note: “135 is before 15” and “15 is before 5” despite the same units place (5) because that’s the order in which they occur in the original array. This is a feature of stable sorting algorithms.*

- *Once the array is sorted based on units place, we sort the result of the previous step based on tens place. Finally, we sort the result based on hundreds place (MSD).*

*After the final pass, we get the sorted array.*

# Radix Sort - Example

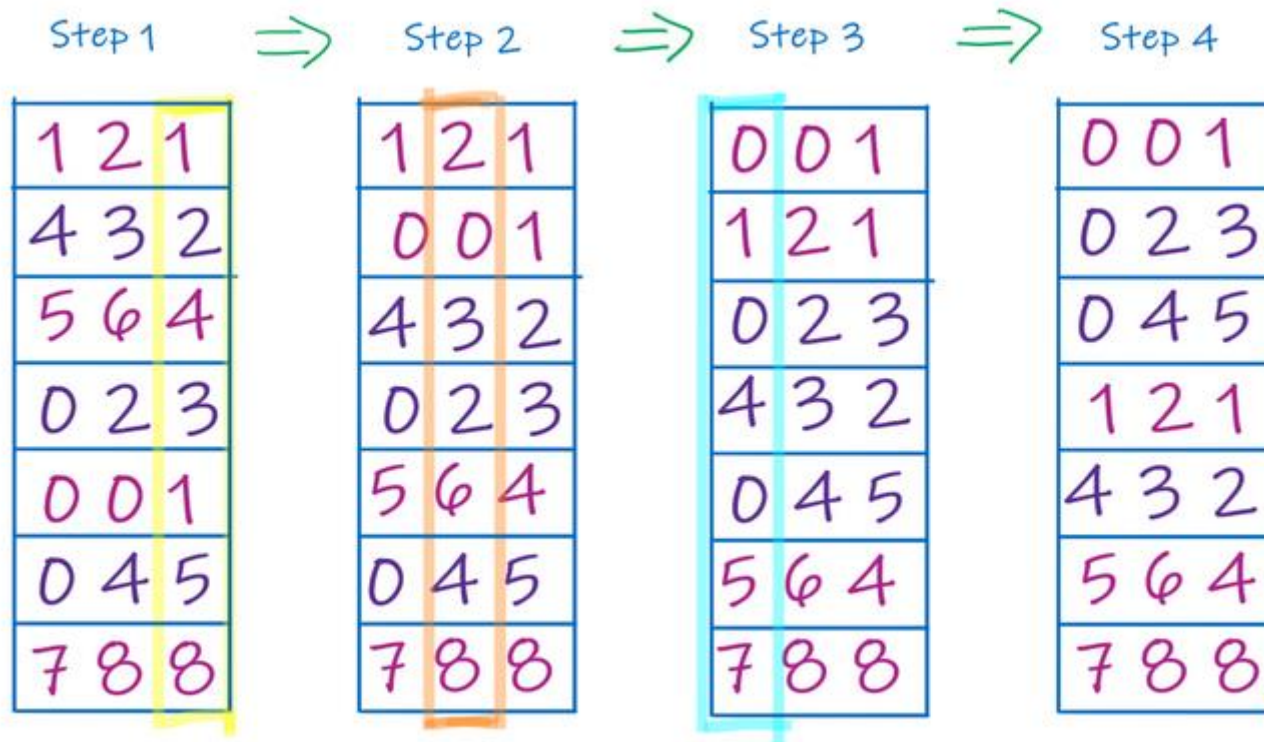


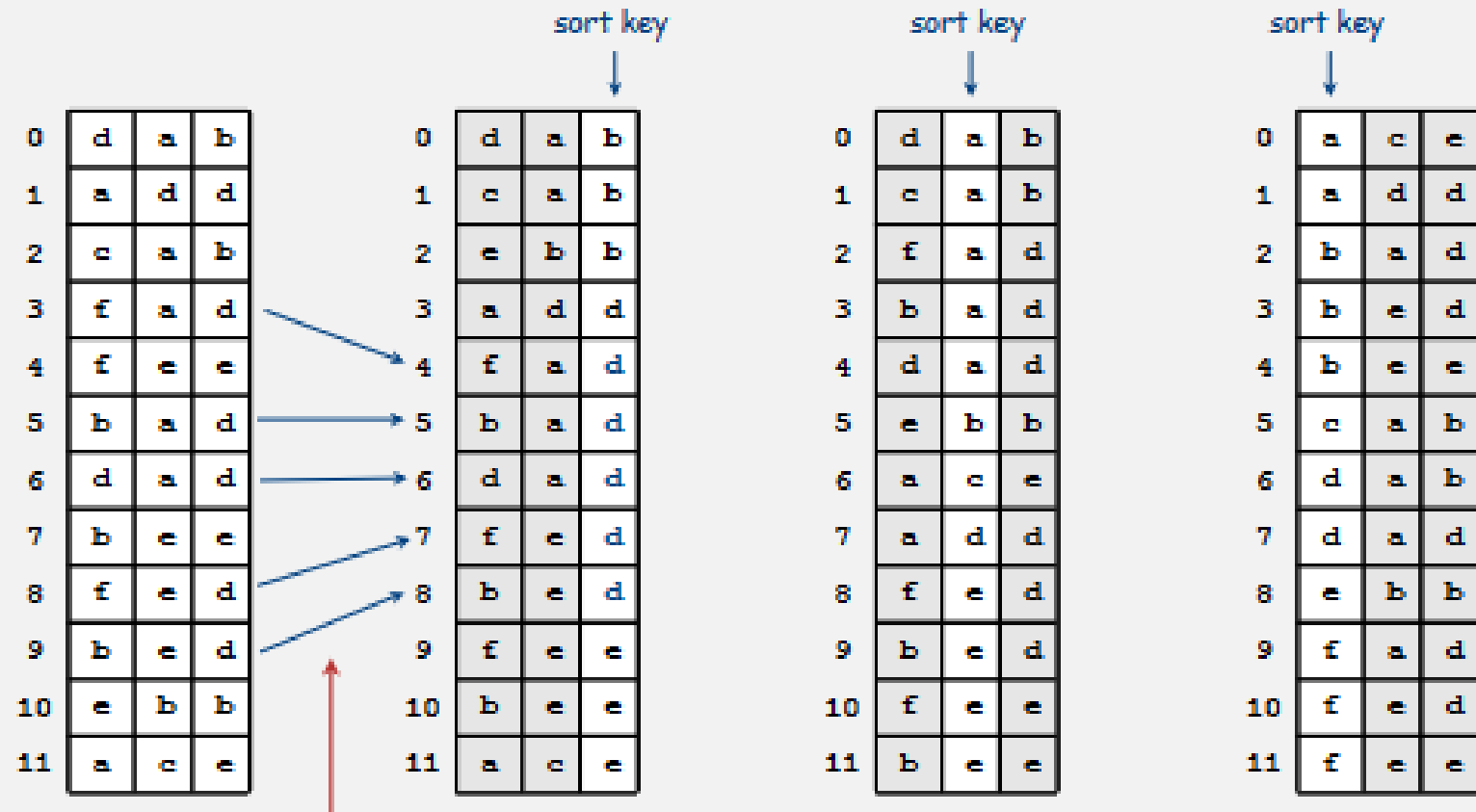
Fig: Working of Radix Sort (We get the final sorted array in Step 4)

# Radix Sort - Example

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

## LSD radix sort.

- Consider characters from right to left.
- Stably sort using  $d$ th character as the key via key-indexed counting.



sort must be stable  
(arrows do not cross)

# Applications of Radix Sort

---

- Radix sort can be applied to data that can be sorted lexicographically, such as words and integers.
- It is a good option when the algorithm runs on parallel machines, making the sorting faster. To use parallelization, we divide the input into several buckets, enabling us to sort the buckets in parallel, as they are independent of each other.

# Application of Queues

- Radix Sort
  - radix is a synonym for base. base 10, base 2
- Multi pass sorting algorithm that only looks at individual digits during each pass
- Use queues as buckets to store elements
- Create an array of 10 queues
- Starting with the least significant digit place value in queue that matches digit
- Empty queues back into array
- Repeat, moving to next least significant digit


# Radix Sort Example: MSD to LSD

- In the previous example, we sorted from LSD to MSD. If we'd gone the other way, we would have got the same result.
- However, we prefer sorting from LSD to MSD – the benefit is that after any  $n$  passes, positions of all numbers with digits  $\leq n$  are sorted relative to each other.
- There are certain cases where it is more advantageous to implement radix sort from MSD to LSD.
- One such case is when we have to sort strings in alphabetical order. In this case:
  - The radix will be 26, as there are 26 letters in the alphabet.
  - We need to go from the first letter of the string (MSD) to LSD as the string lengths may not be equal.
  - LSD to MSD will not work. Look at this example:

# LSD to MSD will not work. Look at this example:

Input	{abc, ac, cba}
Pass 1: Last letters	{cba, abc, ac}
Pass 2: Second-to-last letters	{ac, cba, abc}
Pass 3 [final pass]: Third-to-last letters	{ac, abc cba} ✗
The correct order is	{abc, ac, cba} ✓

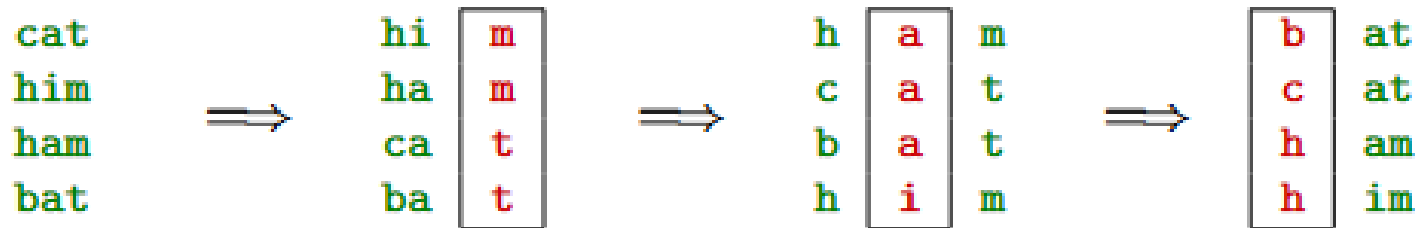


- 
- The right sorted order for the above example is {abc, ac, cba}, which we'll get if we move from MSD to LSD.
  - In the case of digits, the number of place values of numbers with fewer digits can effectively match that of the largest number since there can be as many zeroes to the left of any number as needed — we're effectively comparing, say, 0034 with 2139. There's no such equivalent for strings.
  - We can go LSD to MSD if the strings are of the same length.

We can go LSD to MSD if the strings are of the same length.  
For example:

Input	{bb, ad, ac, ba}
Pass 1: Last letters	{ba, bb, ac, ad}
Pass 2 (final pass): Second-to-last letters	{ac, ad, ba, bb} ✓

We can go LSD to MSD if the strings are of the same length. For example:



- While sorting from MSD to LSD, we need to use buckets to separate the result of previous passes to get the correct result. For example, without buckets:

Input	{bb, ad, ac, ba}
Pass 1: Last letters	{ <b>ad</b> , <b>ac</b> , <b>bb</b> , <b>ba</b> }
Pass 2 (final pass): Second letters	{ba, bb, ac, ad} ❌
The correct order is	{ac, ad, ba, bb} ✅

- This error happened because the LSD got more priority than the MSD. The fix for this error is to use buckets:

After the first pass, store the output in buckets:

$\{\{ad, ac\}, \{bb, ba\}\}$

Sort both buckets separately:

$\{\{ac, ad\}, \{ba, bb\}\}$

Result:

$\{ac, ad, ba, bb\}$  ✓

# MSD Radix Sort example

B	A	D	G	E	\0					
B	A	N	N	E	R	\0				
C	O	F	F	E	\0					
C	O	M	P	A	R	I	S	O	N	\0
C	O	M	P	U	T	E	R	\0		
M	I	D	N	I	G	H	T	\0		
W	A	N	D	E	R	\0				
W	A	R	D	R	O	B	E	\0		
W	O	R	K	E	R	\0				

Only Green highlighted  
elements are examined/  
compared

# MSD Radix Sort example

al	p	habet		al	g	orithm
al	i	gnment		al	i	gnment
al	l	ocate		al	i	as
al	g	orithm		al	l	ocate
al	t	ernative	⇒	al	l	
al	i	as		al	p	habet
al	t	ernate		al	t	ernative
al	l			al	t	ernate

# MSD string sort: example

Input

she	are	are	are	are	are	are	are	are
sells	by	by	by	by	by	by	by	by
seashells	she	sells	seashells	sea	sea	sea	seas	sea
by	sells	seashells	sea	seashells	seashells	seashells	seashells	seashells
the	seashells	sea	seashells	seashells	seashells	seashells	seashells	seashells
sea	sea	sells	sells	sells	sells	sells	sells	sells
shore	shore	seashells	sells	sells	sells	sells	sells	sells
the	shells	she	she	she	she	she	she	she
shells	she	shore	shore	shore	shore	shore	shore	shore
she	sells	shells	shells	shells	shells	shells	shells	shells
sells	surely	she	she	she	she	she	she	she
are	seashells	surely	surely	surely	surely	surely	surely	surely
surely	the	the	the	the	the	the	the	the
seashells	the	the	the	the	the	the	the	the

Annotations: 'd' points to the 'd' in 'seashells' in the second column. 'to' points to the 'o' in 'seashells' in the second column. 'hi' points to the 'h' in 'the' in the second column.

are	are	are	are	are	are	are	output
by	by	by	by	by	by	by	by
sea	sea	sea	sea	sea	sea	sea	sea
seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells
seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells
sells	sells	sells	sells	sells	sells	sells	sells
sells	sells	sells	sells	sells	sells	sells	sells
she	she	she	she	she	she	she	she
shore	shore	shore	shells	she	she	she	she
shells	hells	shells	she	shells	shells	shells	shells
she	she	she	shore	shore	shore	shore	shore
surely	surely	surely	surely	surely	surely	surely	surely
the	the	the	the	the	the	the	the
the	the	the	the	the	the	the	the

Annotations: 'need to examine every character in equal keys' points to the 's' in 'seashells' in the second column. 'end of string goes before any char value' points to the 'e' in 'she' in the fifth column.

Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted)



# Radix Sort in Action: 1s place

- original values in array

9, 113, 70, 86, 12, 93, 37, 40, 252, 7, 79, 12

- Look at ones place

9, 113, 70, 86, 12, 93, 37, 40, 252, 7, 79, 12

- Array of Queues (all empty initially):

0	5
1	6
2	7
3	8
4	9

# Radix Sort in Action: 1s

- original values in array

9, 113, 70, 86, 12, 93, 37, 40, 252, 7, 79, 12

- Look at ones place

9, 113, 70, 86, 12, 93, 37, 40, 252, 7, 79, 12

- Queues:

0	7 <u>0</u> , 4 <u>0</u>	5	
1		6	6 8 <u>6</u>
2	1 <u>2</u> , 25 <u>2</u> , 1 <u>2</u>	7	7 3 <u>7</u> , <u>7</u>
3	11 <u>3</u> , 9 <u>3</u>	8	
4		9	9 <u>9</u> , 7 <u>9</u>

# Radix Sort in Action: 10s

- Empty queues in order from 0 to 9 back into array

70, 40, 12, 252, 12, 113, 93, 86, 37, 7, 9, 79

- Now look at 10's place

70, 40, 12, 252, 12, 113, 93, 86, 37, 7, 9, 79

- Queues:

0    7, 9

1    12, 12, 113

2

3    37

4    40

5    252

6

7    70, 79

8    86

9    93

# Radix Sort in Action: 100s

- Empty queues in order from 0 to 9 back into array

7, 9, 12, 12, 113, 37, 40, 252, 70, 79, 86, 93

- Now look at 100's place

  7,   9,   12,   12,   113,   37,   40,   252,   70,   79,   86,   93

- Queues:

0     7,   9,   12,   12,   37,   40,   70,   79,   86,   93

5

1     113

6

2     252

7

3

8

4

9

# Radix Sort in Action: Final Step

---

- Empty queues in order from 0 to 9 back into array

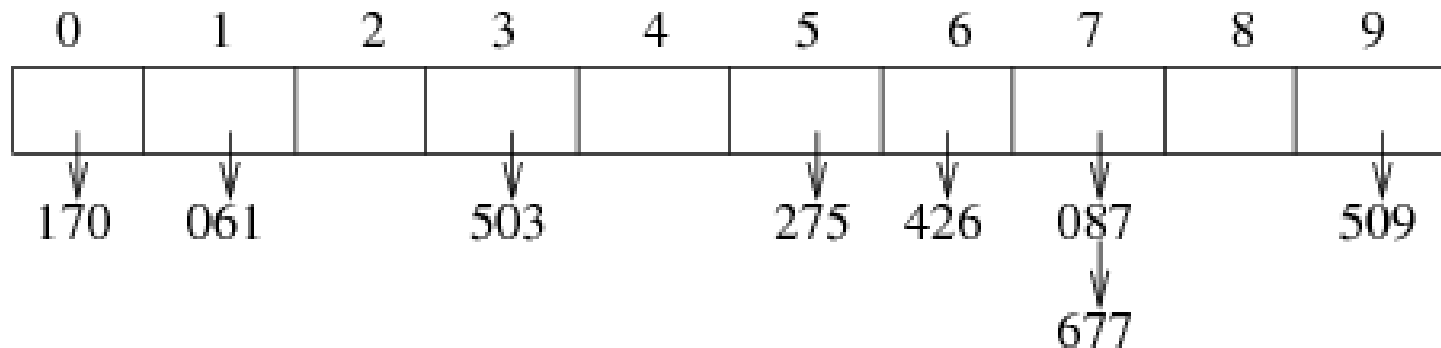
7, 9, 12, 12, 40, 70, 79, 86, 93, 113, 252

# Radix Sort as Bucket Sort

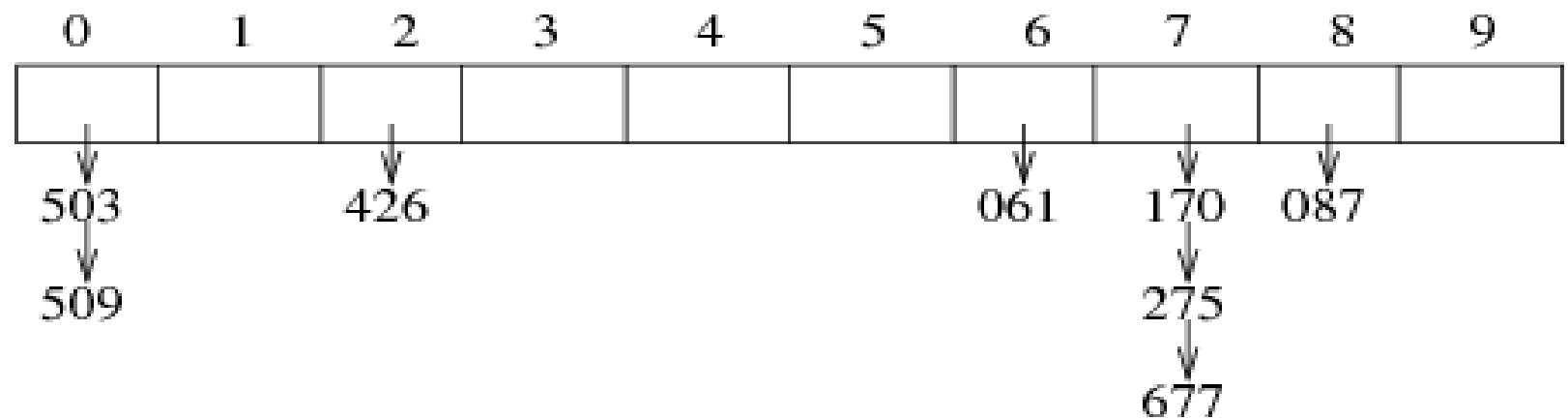
## □ Least-significant-digit-first

Example: 275, 087, 426, 061, 509, 170, 677, 503

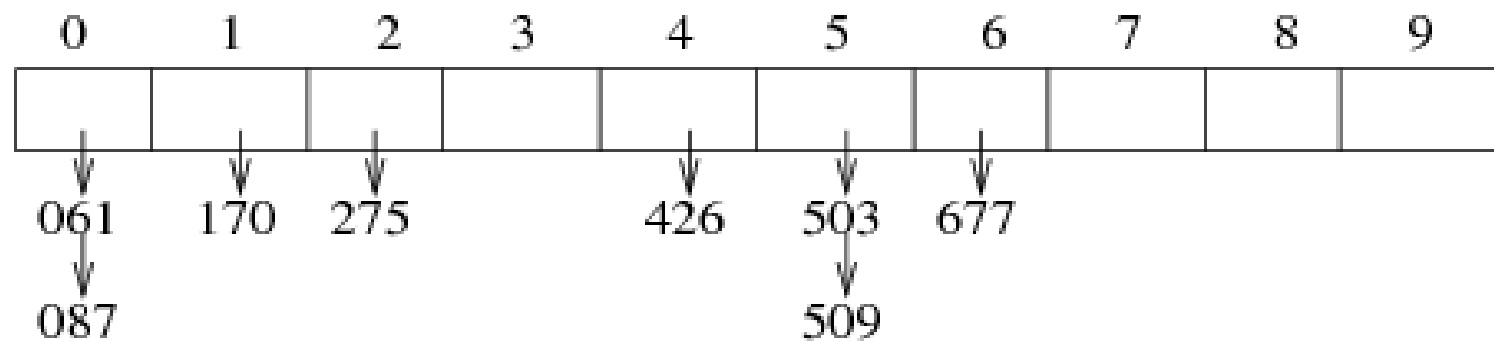
1st pass



2nd pass



3rd pass



in sorted order

# Summary: Radix Sort

- Radix sort:
  - Assumption: input has  $d$  digits ranging from 0 to  $k$
  - Basic idea:
    - Sort elements by digit starting with *least* significant
    - Use a stable sort (like counting sort) for each stage
  - Each pass over  $n$  numbers with  $d$  digits takes time  $O(n+k)$ , so total time  $O(dn+dk)$ 
    - When  $d$  is constant and  $k=O(n)$ , takes  $O(n)$  time
  - Fast! Stable! Simple!
  - Doesn't sort in place



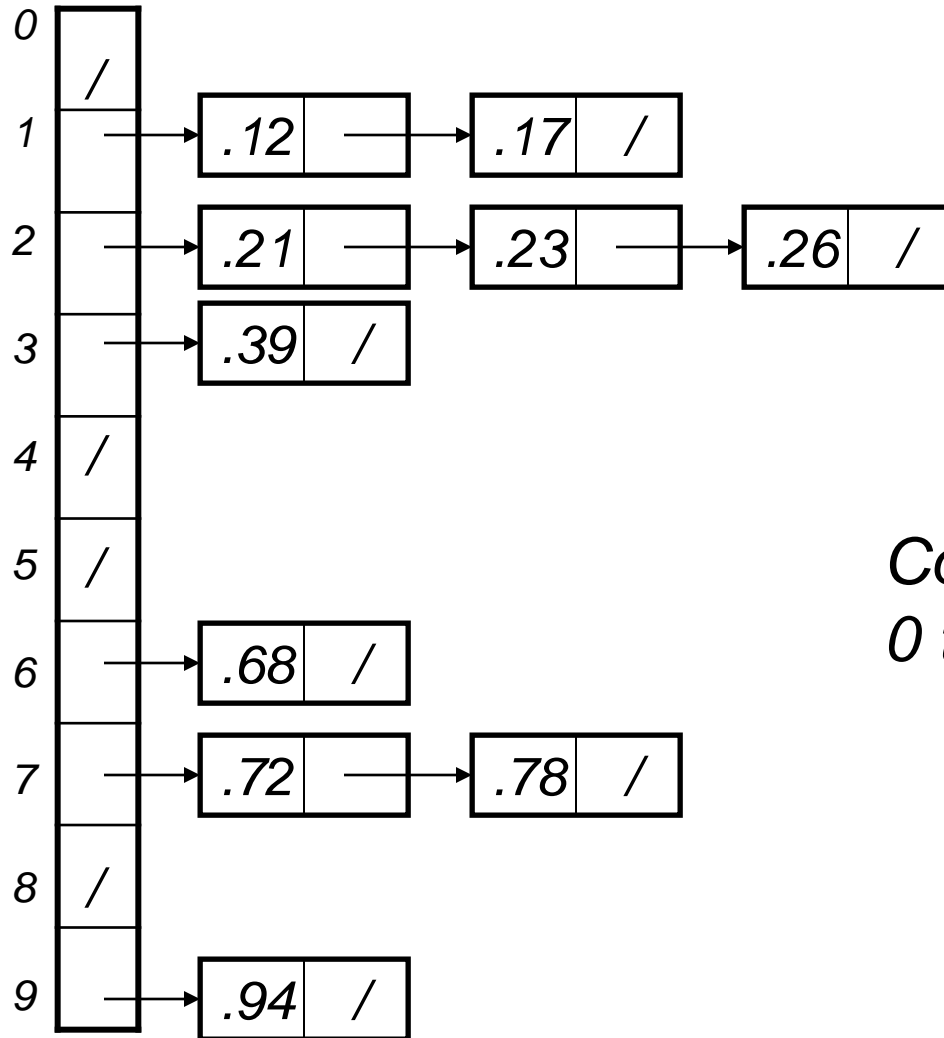
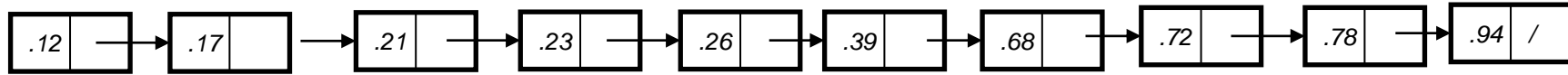
# Open interval and closed interval

- Open interval and closed interval are used to represent a range of numeric values. The open interval includes only the values between the endpoints and is represented as  $( )$ . The closed interval includes even the endpoints of the range of values and is represented as  $[ ]$ .
- The open interval can be presented as an expression  $a < x < b$ , and the closed interval is represented as an expression  $a \leq x \leq b$ .
- A half-open interval includes only one of its endpoints, and is denoted by mixing the notations for open and closed intervals. For example,  $(0, 1]$  means **greater than 0 and less than or equal to 1**, while  $[0, 1)$  means greater than or equal to 0 and less than 1.

# Bucket Sort

- Bucket sort
- Assumption: input is  $n$  reals from  $[0, 1)$  // means greater than or equal to 0 and less than 1.
  - Basic idea:
    - Create  $n$  linked lists (*buckets*) to divide interval  $[0,1)$  into subintervals of size  $1/n$
    - Add each input element to appropriate bucket and sort buckets with insertion sort
  - Uniform input distribution  $\rightarrow O(1)$  bucket size
    - Therefore the expected total time is  $O(n)$
  - These ideas will return when we study *hash tables*

# Example - Bucket Sort



*Concatenate the lists from 0 to  $n - 1$  together, in order*



The End