Bucket, Bin, Radix and Lexicographic Sort

Sorting under assumptions

Linear Sorting

- If you know something about the values of the input set, you can potentially do better than the $\Omega(n \log n)$ sorting lower bound
- For example, if the values of the input set are in a certain range

Bucket or Bin Sort

- Bucket sort or bin sort partitions a set of elements into a finite number of buckets or bins
- Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm
- Bucket may run in linear time ($\underline{\Theta}(n)$)
- Each bucket must contain only a single element or it incurs a cost for additional sorts on the buckets themselves
- Since bucket sort is not a comparison sort, it is not subject to the $\Omega(n \log n)$ lower bound

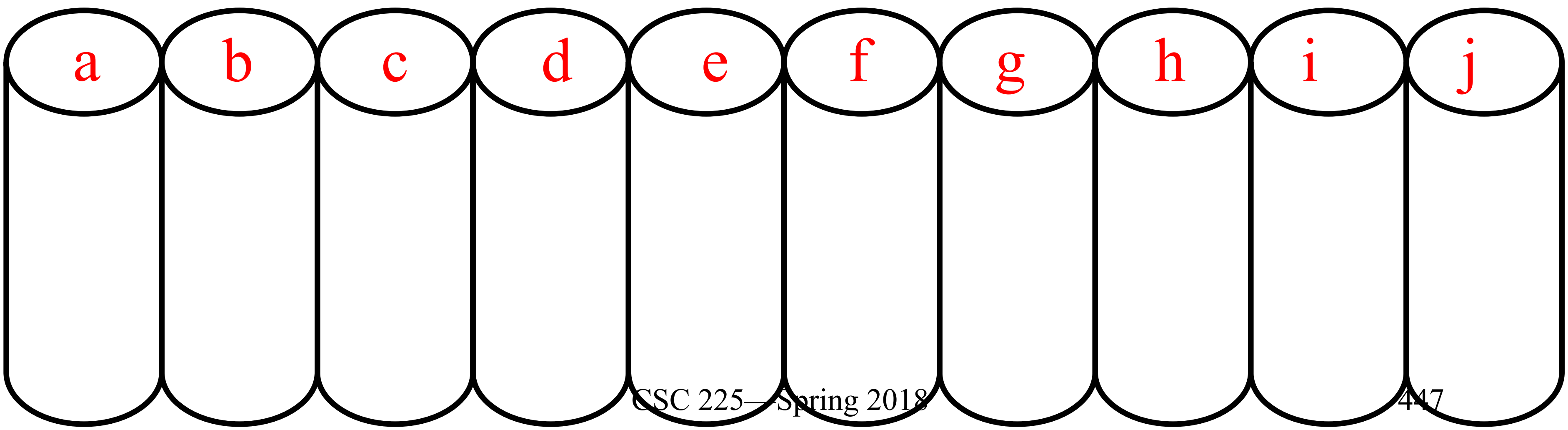
Bucket Sort

- Given n elements (e.g., words) to sort into N categories (e.g., letters of the alphabet)

- Input set
  - bucketsort insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort

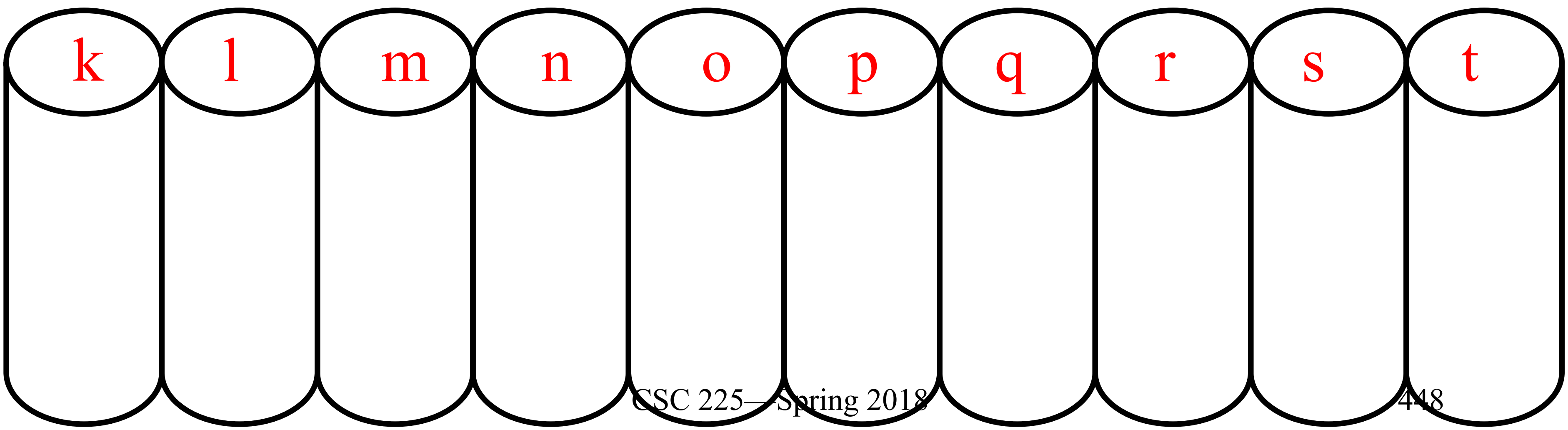
Bucket Sort

- bucketsort insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort



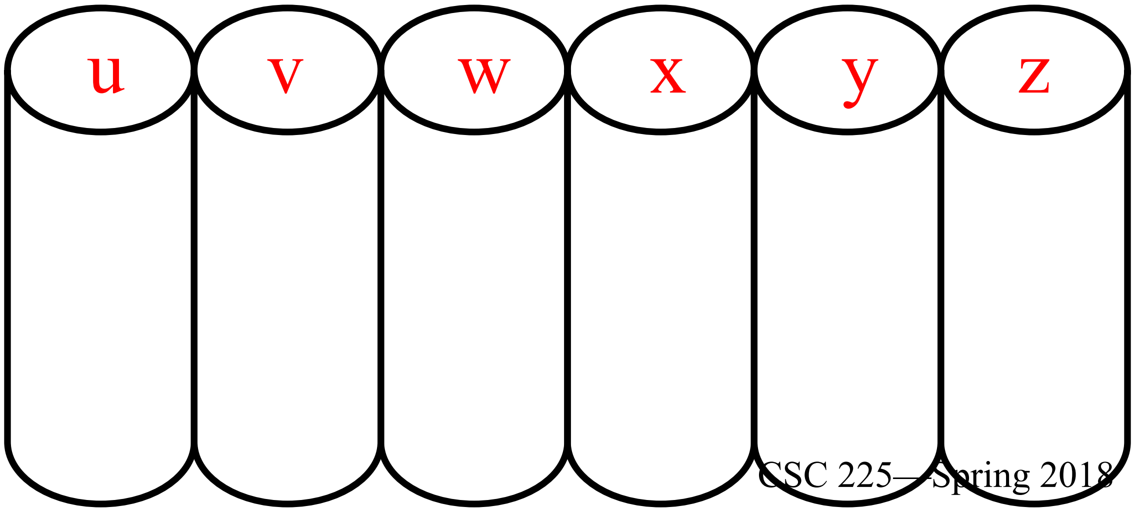
Bucket Sort

- bucketsort insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort



Bucket Sort

- bucketsort insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort

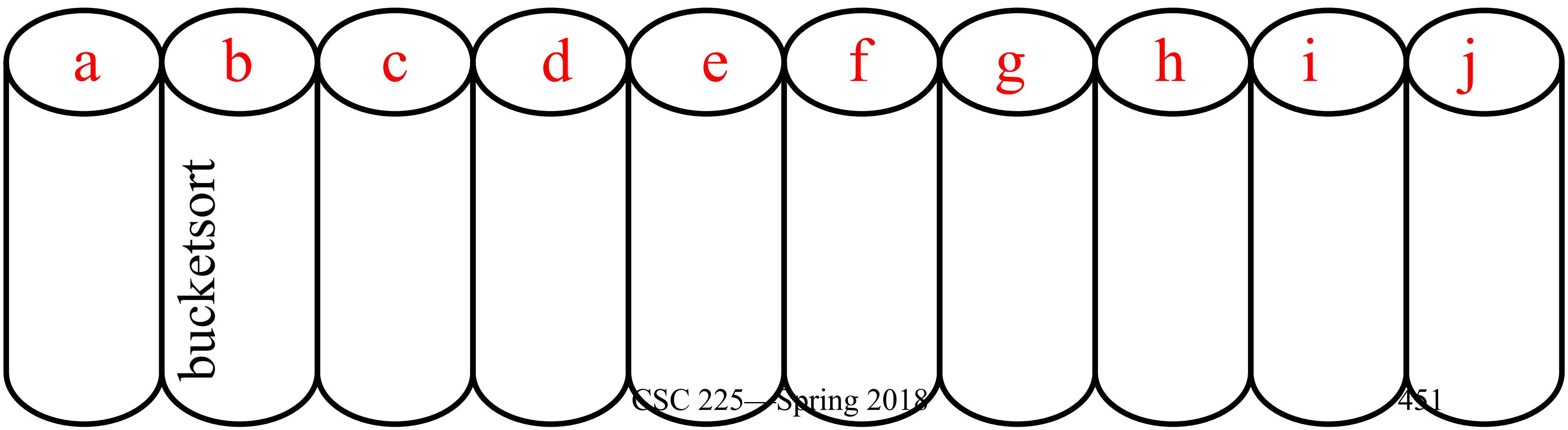


Bucket Sort

- bucketsort insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort

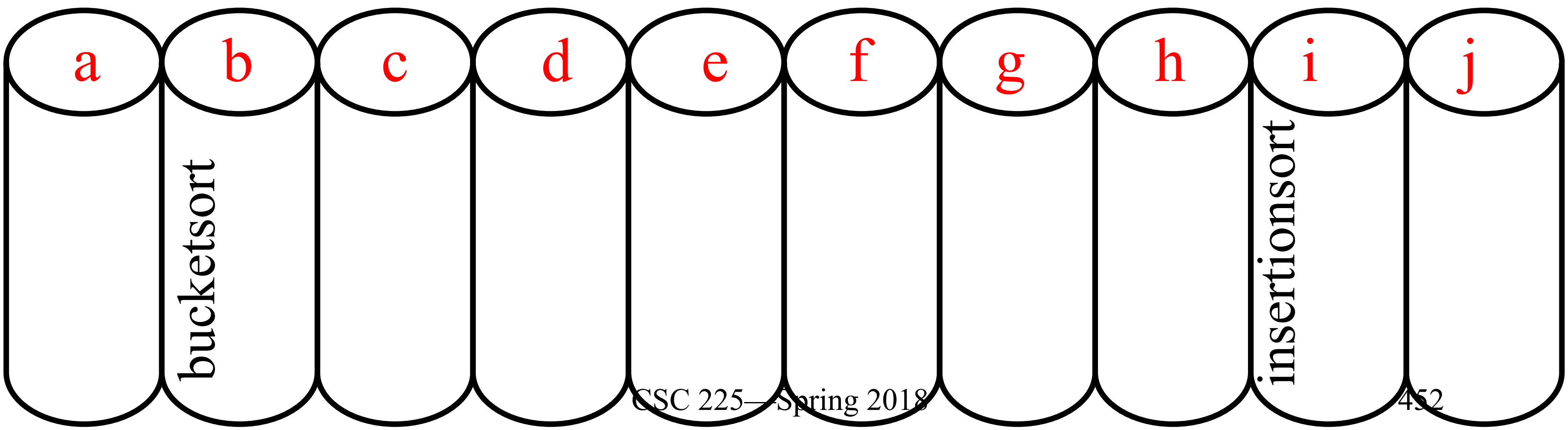
Bucket Sort

- **bucketsort** insertionsort selectionsort quicksort mergesort shellsort treeselection heapsort



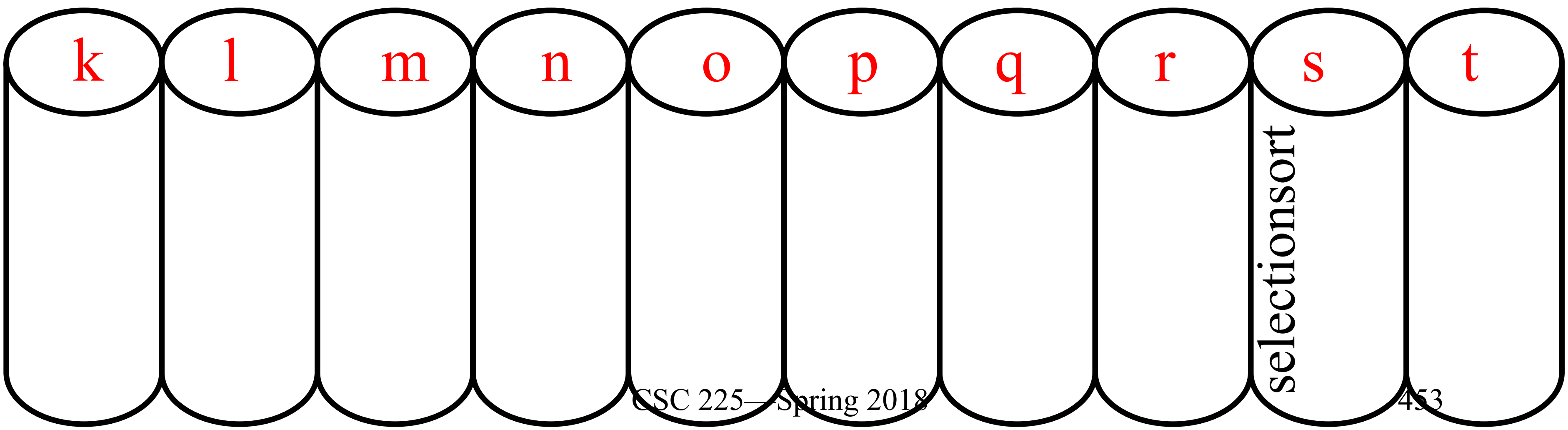
Bucket Sort

- **insertionsort** selectionsort quicksort mergesort shellsort treeselection heapsort



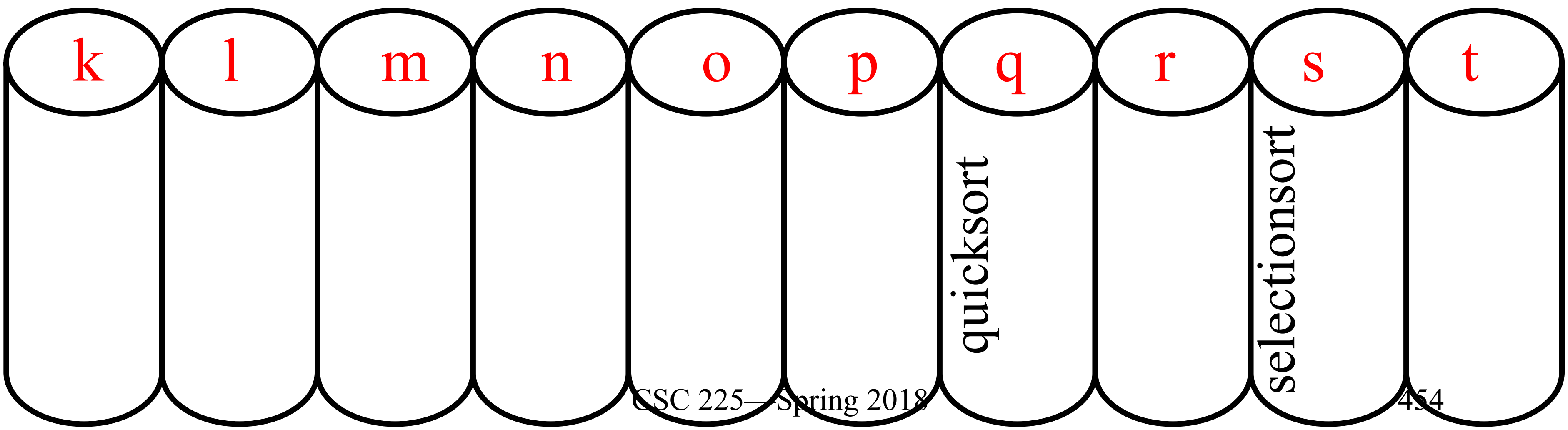
Bucket Sort

- **selectionsort** quicksort mergesort shellsort treeselection heapsort



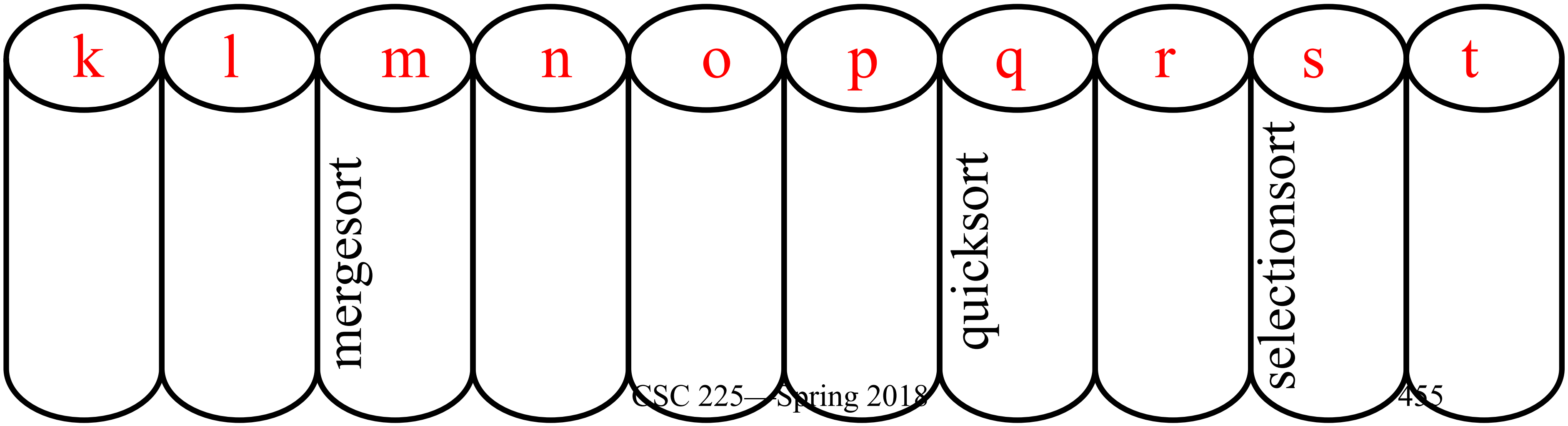
Bucket-Sort

- **quicksort** mergesort shellsort treeselection heapsort



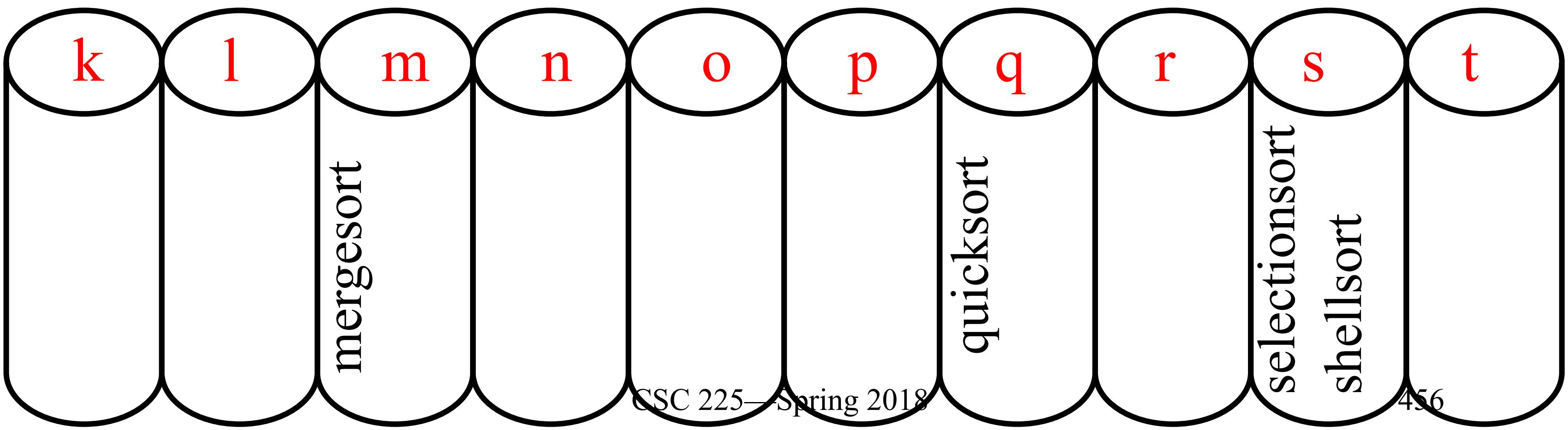
Bucket Sort

- **mergesort** shellsort treeselection heapsort



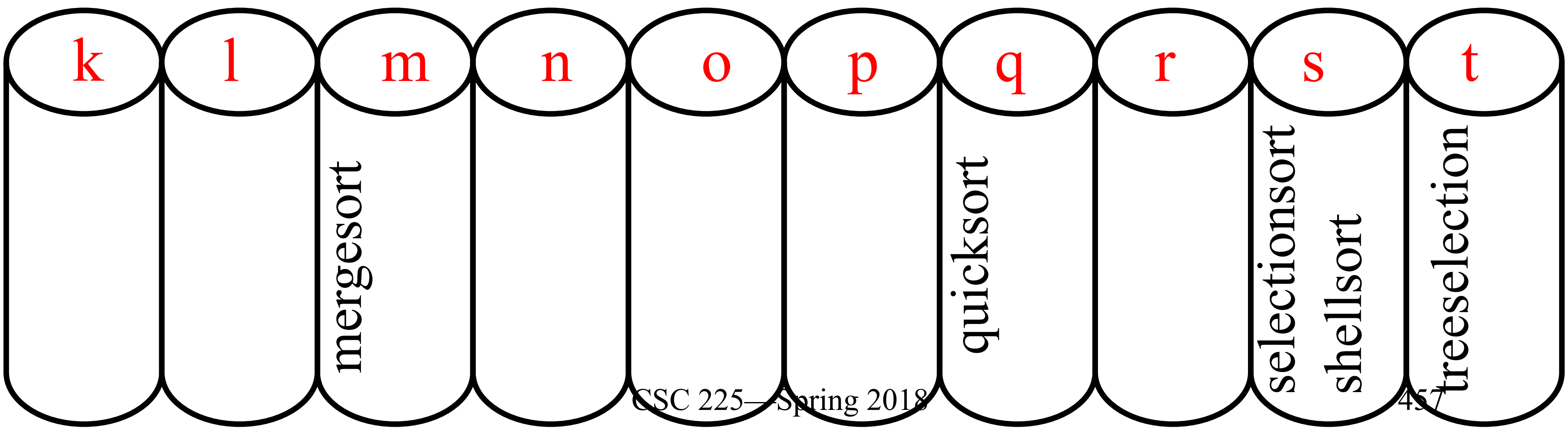
Bucket Sort

- **shellsort** treeselection heapsort



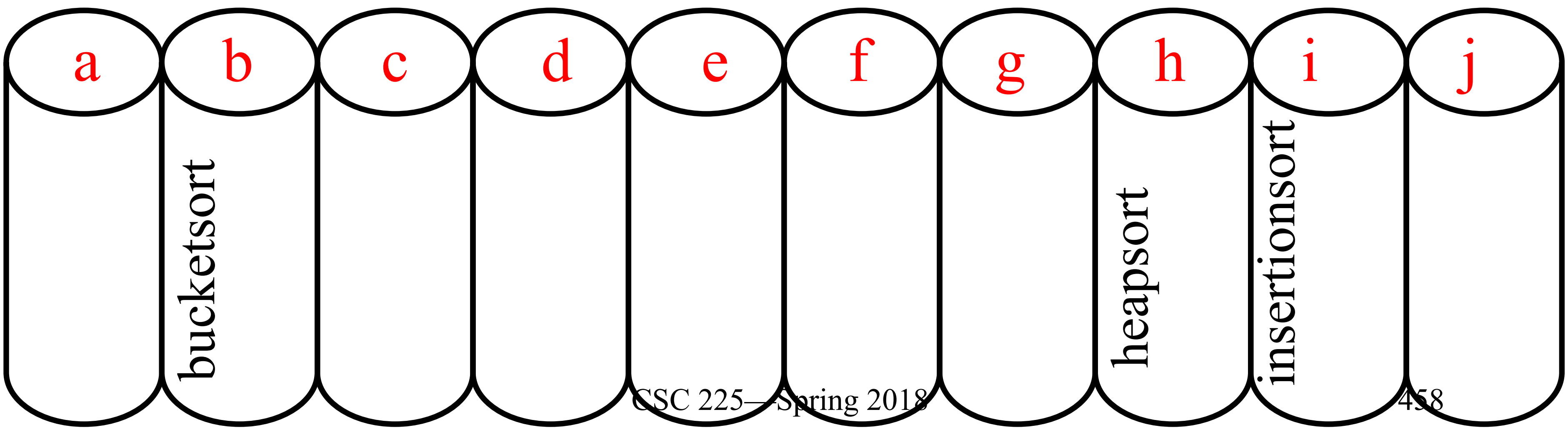
Bucket Sort

- **treeselection** heapsort



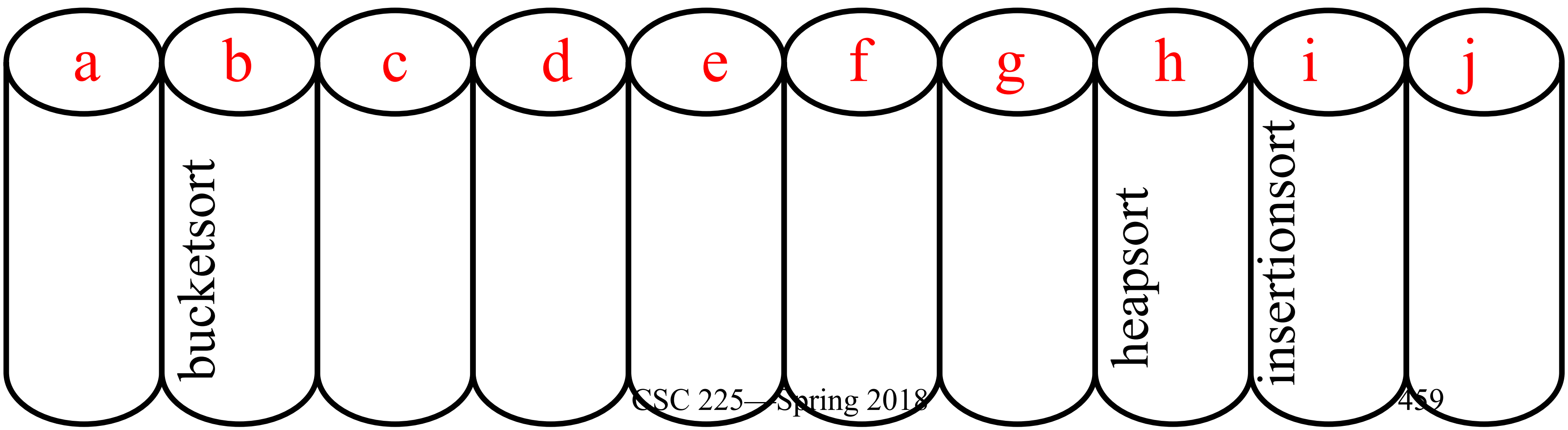
Bucket Sort

- heapsort



Bucket Sort

- Concatenate buckets
  - bucketsort heapsort insertionsort mergesort quicksort shellsort selectionsort treeselection



Bucket Sort

```
Algorithm bucketSort(S)
    Input: Sequence S of items with integer keys in the range [0, k−1]
    Output: Sequence S sorted in nondecreasing order of the keys
    Let B be an array of k sequences, each of which is
    initially empty
    for each item x in S do
        Let k be the key of x
        Remove x from S
        insert x at the end of bucket B[k]
    end
    for i ← 0 to N−1 do
        for each item x in sequence B[i] do
            remove x from B[i]
            insert x at the end of S
        end
    end
```

Running Time of Bucket Sort

- First loop
  - Iterates n times
  - n removes from list S
  - n inserts into buckets B
- Second loop
  - Iterates N times
  - n removes from buckets B
  - n inserts into list S

Note that Bucket sort only sorts by one component of the key (e.g., first letter)

➔ **The time complexity of bucket sort is $O(n+N)$ and uses $O(n+N)$ space**
  - Usually the range of N is small compared to n
  - The second loop deals with the same elements as the first loop

➔ **The time complexity of bucket sort is $O(n)$ and uses $O(n)$ space**

Postman's Sort

- This sorting algorithm is a variant of bucket sort
- The keys are described so the algorithm can allocate buckets efficiently
- This is the algorithm used by letter-sorting machines in the post office
  - first province, then post offices, then routes, etc
- Since keys are not compared against each other, sorting time is O(cn), where c depends on the size of the key and number of buckets
- This is similar to a radix sort that works "top down" or "most significant digit first"

Radix Sort

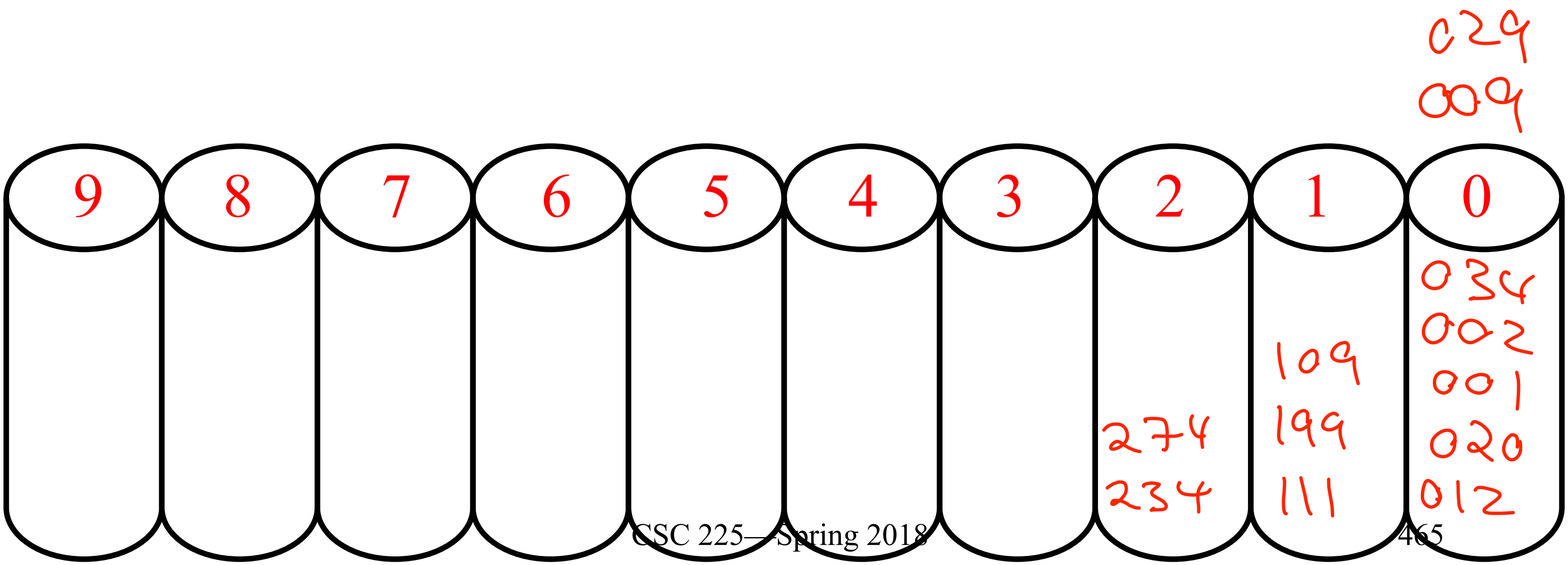
- Apply bucket sort multiple times to the components of a key
- Integer representations can be used to represent things such as strings of characters (e.g., names of people, places)
- Two classifications of radix sorts are
  - Least significant digit (LSD) radix sorts (i.e., usually right most digit)
  - Most significant digit (MSD) radix sorts (i.e., usually left most digit)
- LSD radix sorts process the integer representations starting from the <u>least significant digit</u> and move the processing towards the <u>most significant digit</u>
- MSD radix sorts process the integer representations starting from the <u>most significant digit</u> and move the processing towards the <u>least significant digit</u>

MSD Radix Sort

012	234	274	020	001	111	002	034
009	029	199	109	005	203	123	401
568	073	193	122	033	120	040	081
006	221	032					

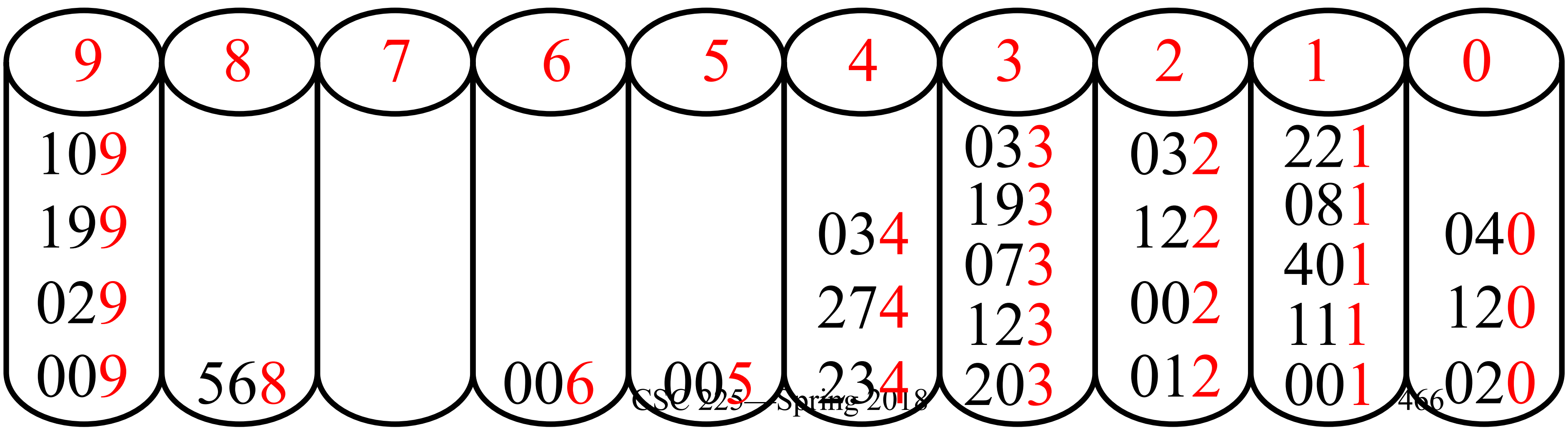
MSD Radix Sort (descending order)

012 234 274 020 001 111 002 034
009 029 199 109 005 203 123 401
568 073 193 122 033 120 040 081
006 221 032



LSD Radix Sort—insert into buckets by LSD

012	234	274	020	001	111	002	034
009	029	199	109	005	203	123	401
568	073	193	122	033	120	040	081
006	221	032					



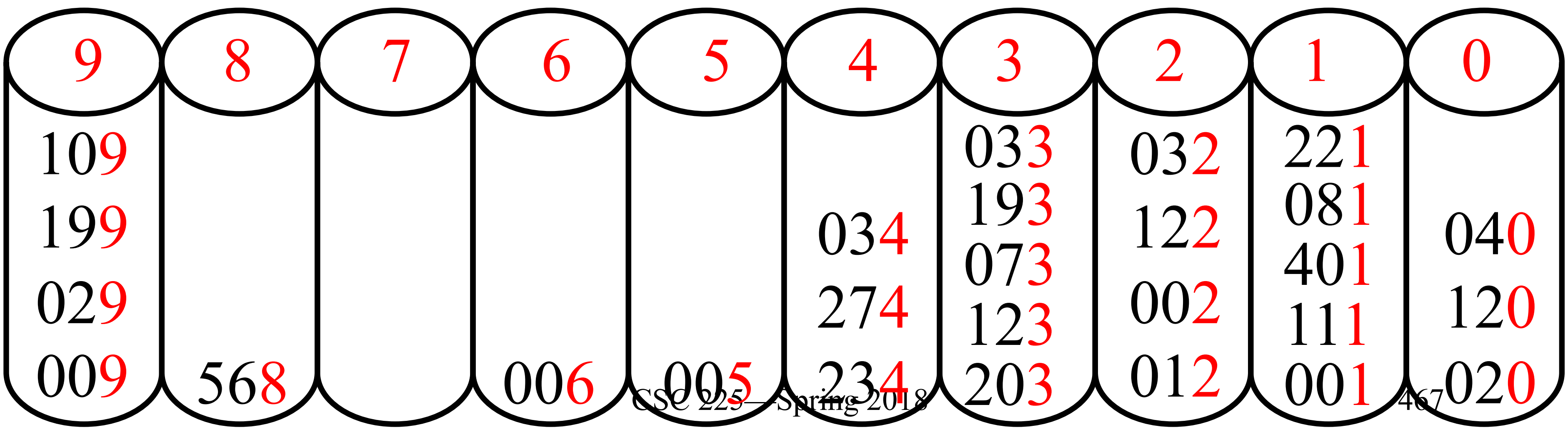
LSD Radix Sort—Concatenate

009 029 199 109 568 006 005 234

274 034 203 123 073 193 033 012

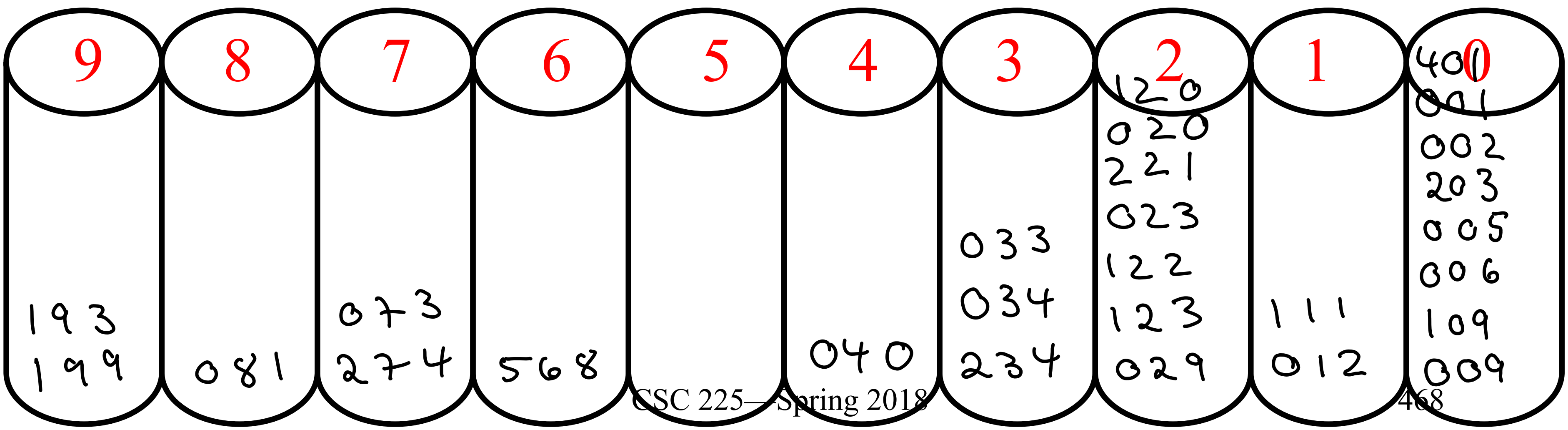
002 122 023 001 111 401 081 221

020 120 040



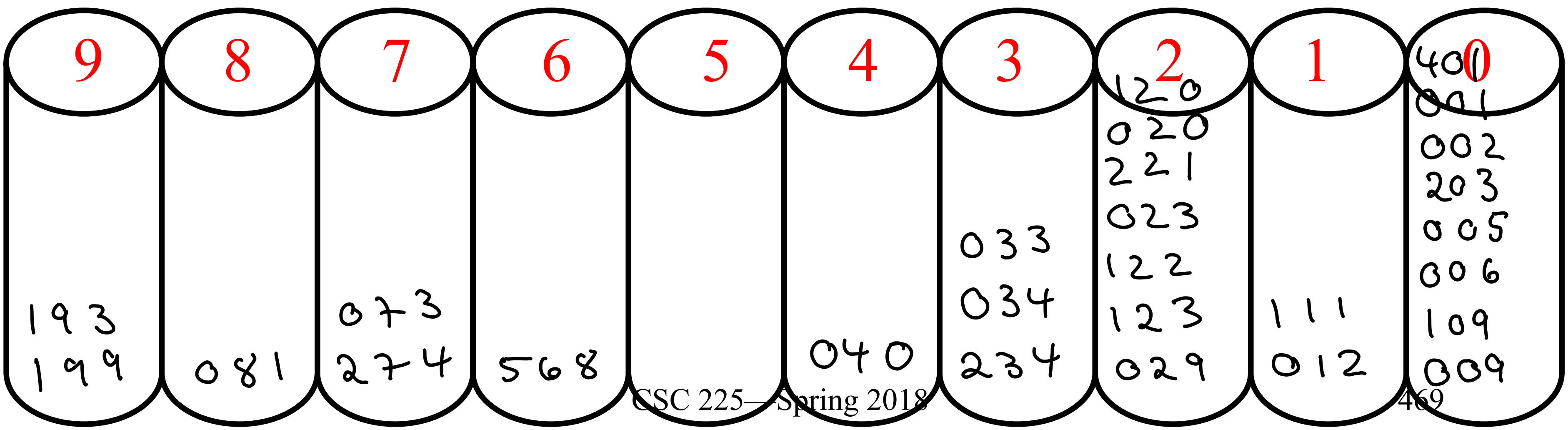
LSD Radix Sort—insert into buckets by 2nd digit

009	029	199	109	568	006	005	234
274	034	203	123	073	193	033	012
002	122	023	001	111	401	081	221
020	120	040					



LSD Radix Sort—concatenate

199	193	081	274	073	568	040	234
034	033	029	123	122	023	221	020
120	012	111	009	109	006	005	203
002	001	401					

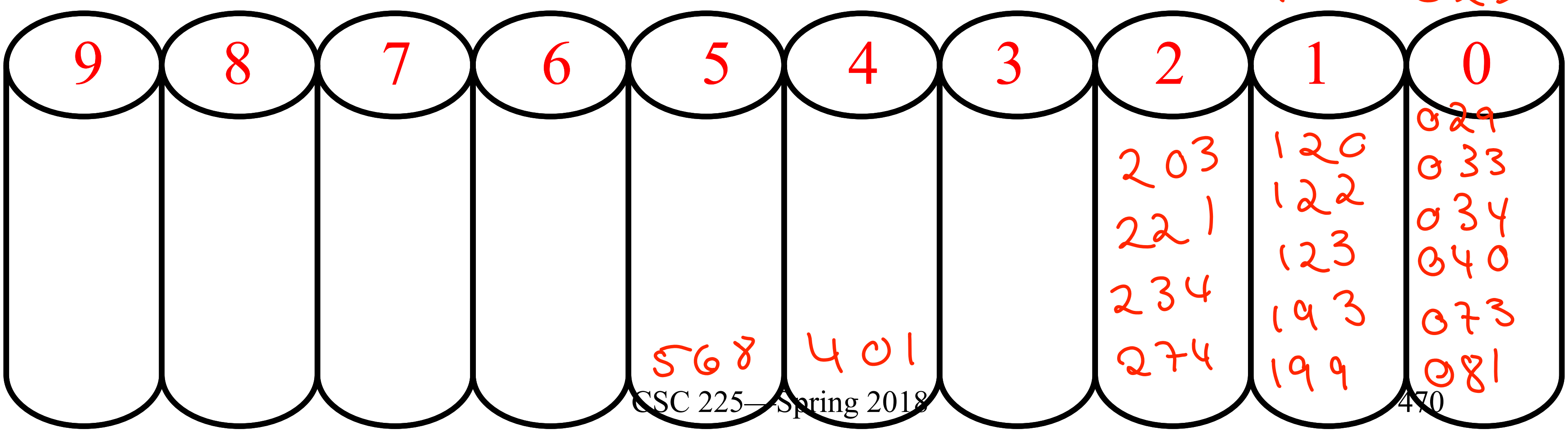


LSD Radix Sort

199	193	081	274	073	568	040	234
034	033	029	123	122	023	221	020
120	012	111	009	109	006	005	203
002	001	401					

001
002
005
006
009
012
020
023

109
111

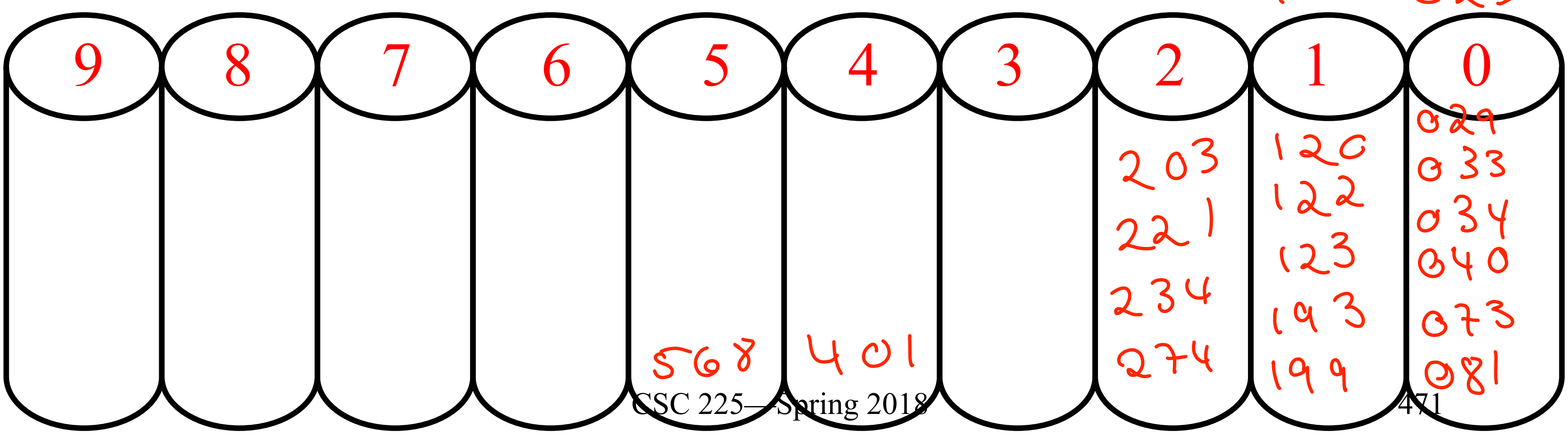


LSD Radix Sort—concatenate

568	401	274	234	221	203	199	193
123	122	120	111	109	081	073	040
034	033	029	023	020	012	009	006
005	002	001					

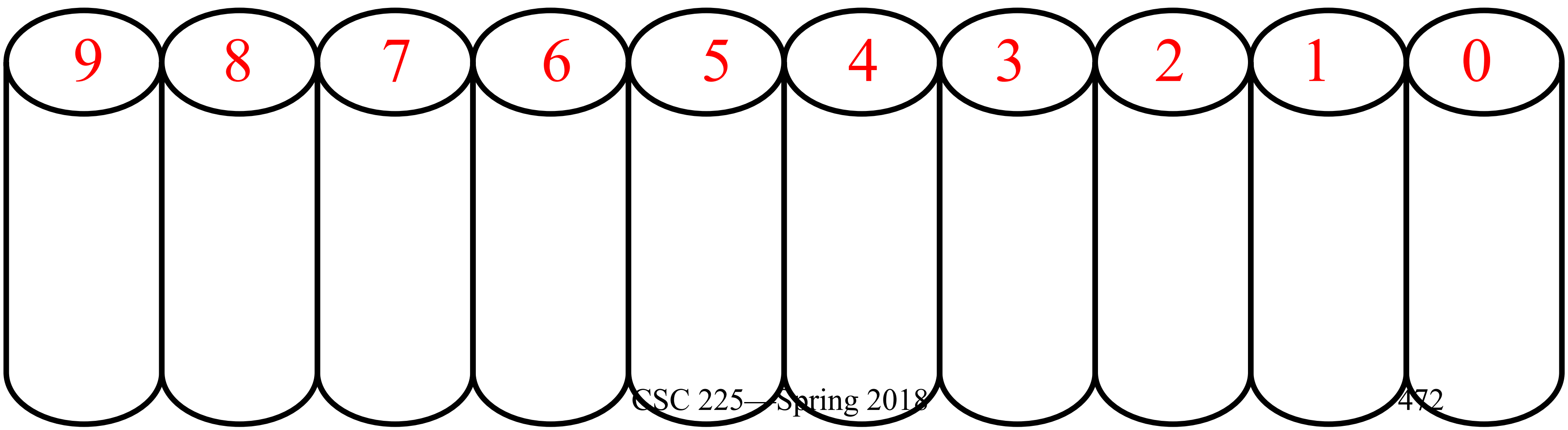
001
002
005
006
009
012
020
023

109
111



LSD Radix Sort—sorted

568 401 274 234 221 203 199 193
123 122 120 111 109 081 073 040
034 033 029 023 020 012 009 006
005 002 001



Radix Sort

- Repeated sorting by means of Bucket Sort
  - For each component of the key perform one Bucket Sort
- Start with the least significant component of the key and end with most significant component
- Implement buckets as queues
- Let the number of components per key be d

- **Theorem. The time complexity of Radix Sort is** $\boxed{O(d(n+N))}$ **or** $\boxed{O(dn)}$ **for large** ***n*****.**

Radix Sort

Theorem. Radix sort is stable

A sorting algorithm for sequence $S = ((k_0, e_0), \ldots, (k_{n-1}, e_{n-1}))$ is *stable* if, for any two items (k_i, e_i) and (k_j, e_j) in S, such that $k_i = k_j$ and (k_i, e_i) precedes (k_j, e_j) before sorting (that is $i < j$), item (k_i, e_i) precedes (k_j, e_j) also after sorting.

Sorting Models and Lower Bound

Reading Assignment

Sections 4.4 and 4.6

What are the sorting models used so far?

What have all of these sorting strategies in common?

- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Shell Sort
- Bubble Sort
- Shaker Sort
- Tree Selection

What have these sorting strategies in common?

- Bucket Sort
- Radix Sort

Comparison based sorting

- Let $S = (x_0, x_1, x_2, \ldots, x_{n\text{-}1})$ be a sequence.
- Assume all elements in S are distinct.
- To sort elements, an algorithm compares two elements x_i and x_j (is $x_i < x_j$?).
- Depending on the outcome (yes or no) the computer will perform either no comparisons or more comparisons.

How many comparisons are required for an optimal sorting algorithm to sort *n* elements?

- The number of comparisons must hold for any sequence that is inputted in the optimal algorithm.

How many comparisons are required for an optimal sorting algorithm?

2 9 10 6 4 8 1 5 7 3

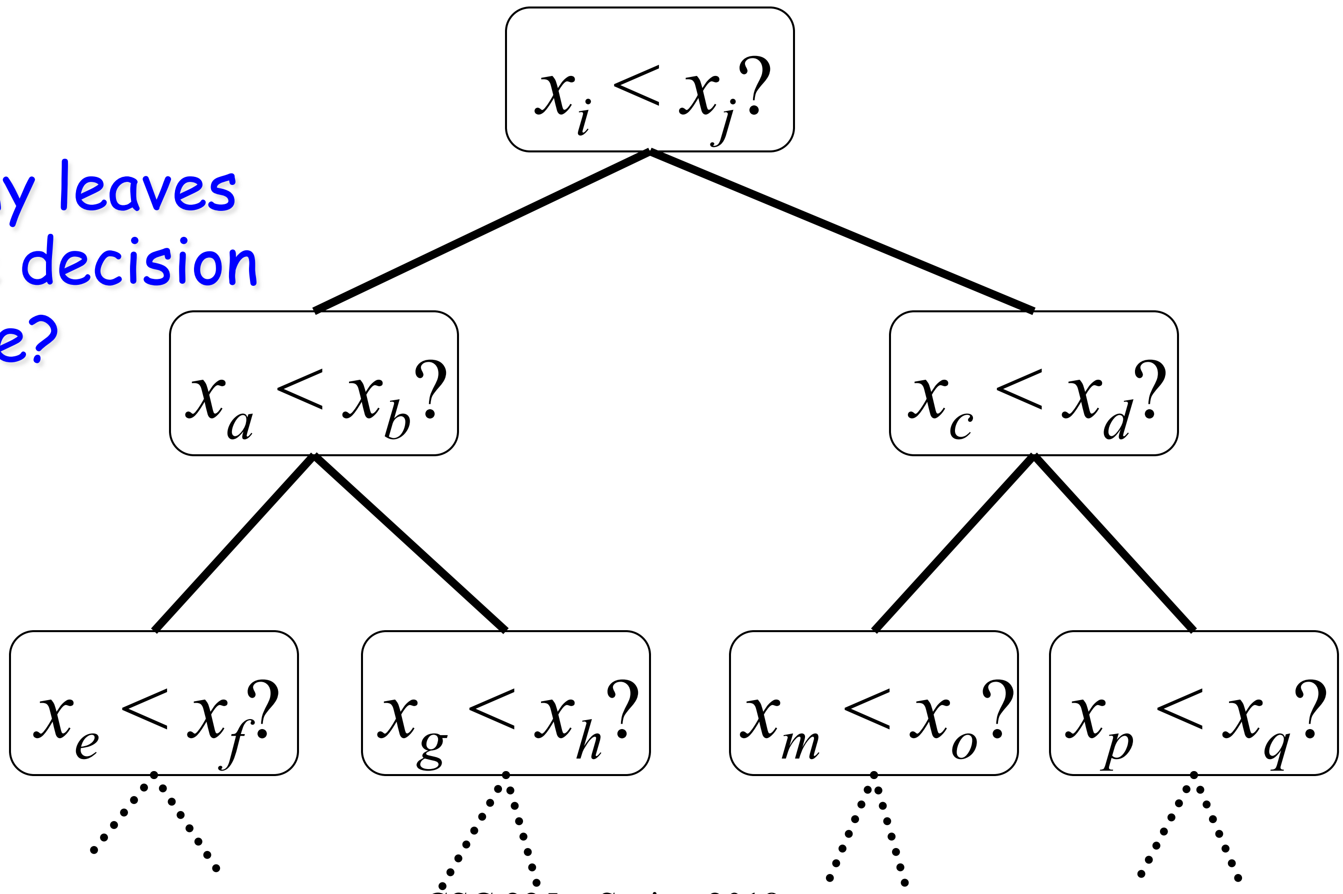
Comparison based sorting

- Let $S = (x_0, x_1, x_2, \ldots, x_{n\text{-}1})$ be a sequence.
- Assume all elements in S are distinct.
- To sort elements, an algorithm compares two elements x_i and x_j (is $x_i < x_j$?).
- Depending on the outcome (yes or no) the computer will perform either no comparisons or more comparisons.

How many comparisons are required for an optimal sorting algorithm?

- Consider the (binary) decision tree.

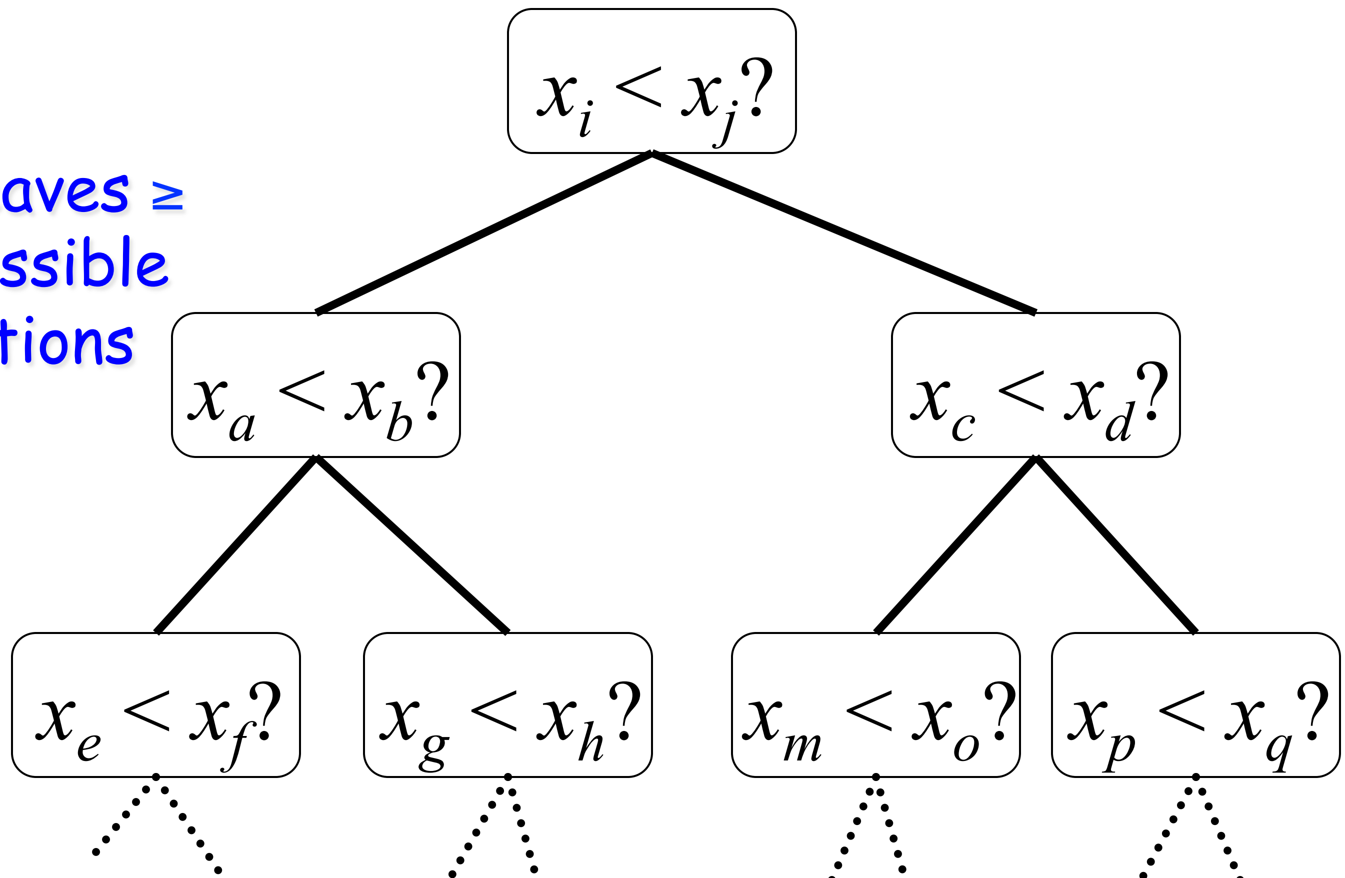
How many leaves does the decision tree have?



How many comparisons are required for an optimal sorting algorithm?

- Consider the (binary) decision tree.

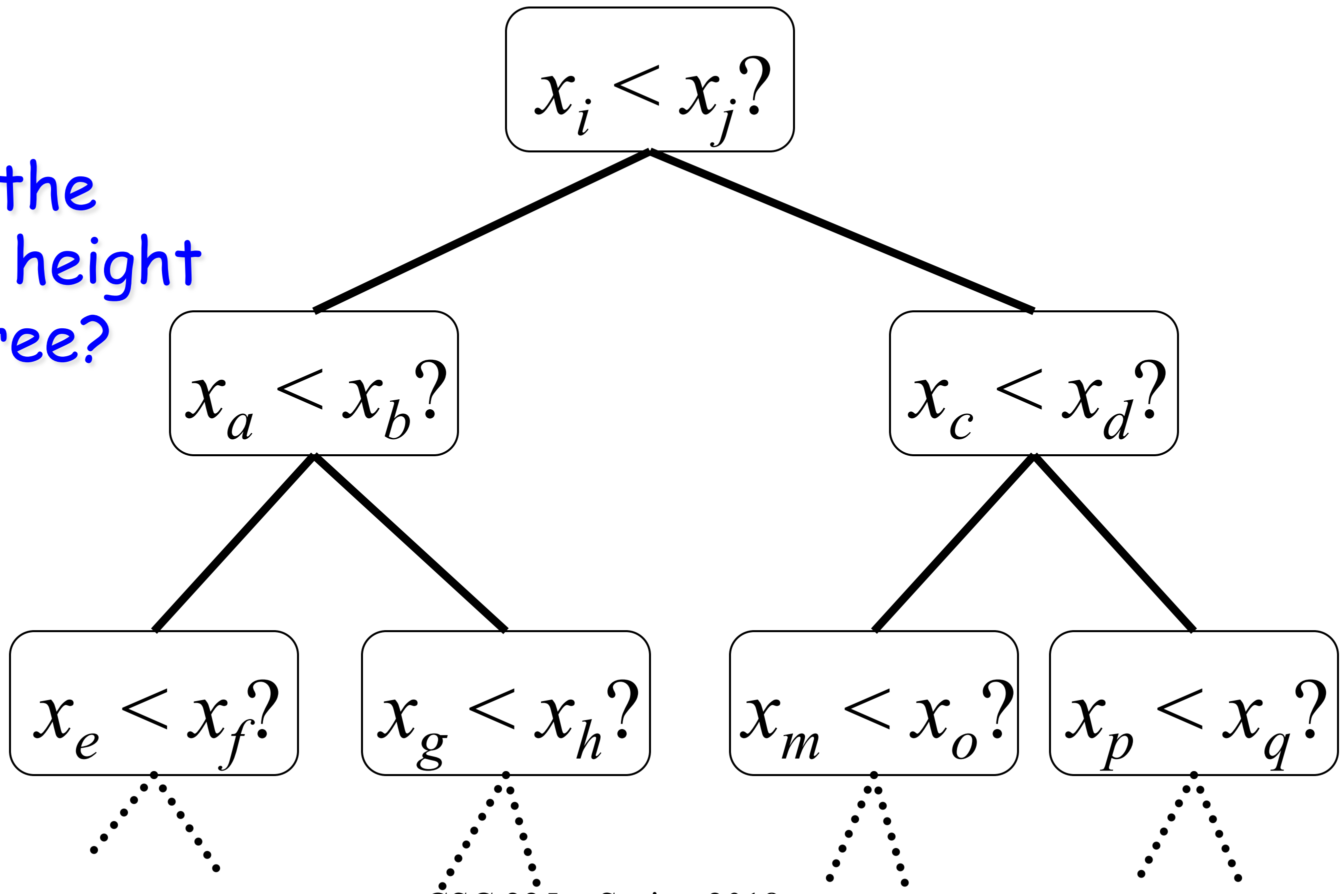
No. of leaves ≥ no. of possible permutations of S.



How many comparisons are required for an optimal sorting algorithm?

- Consider the (binary) decision tree.

What is the minimum height of the tree?

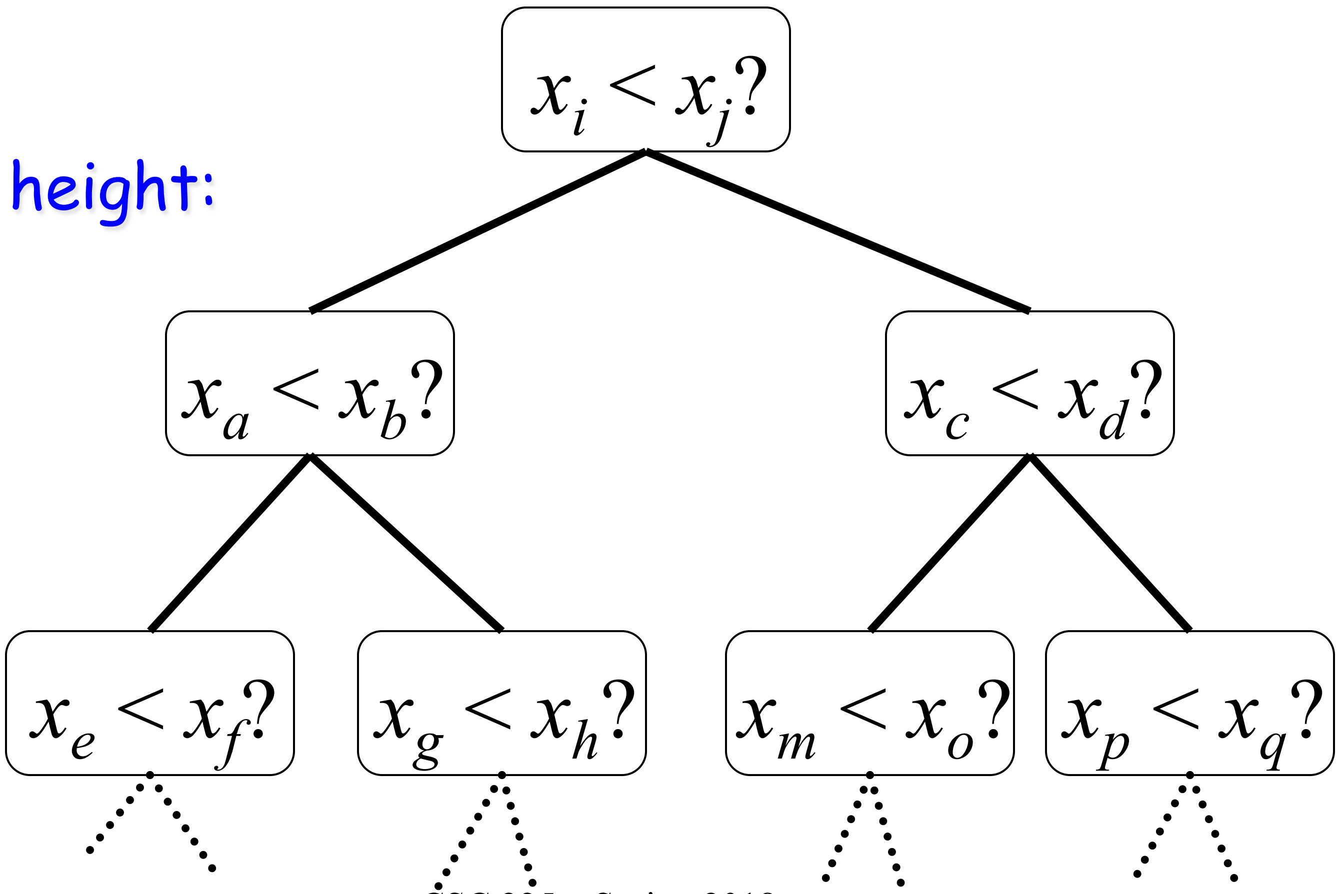


No. of leaves $\geq n!$

How many comparisons are required for an optimal sorting algorithm?

- Consider the (binary) decision tree.

minimum height:
$\log(n!)$

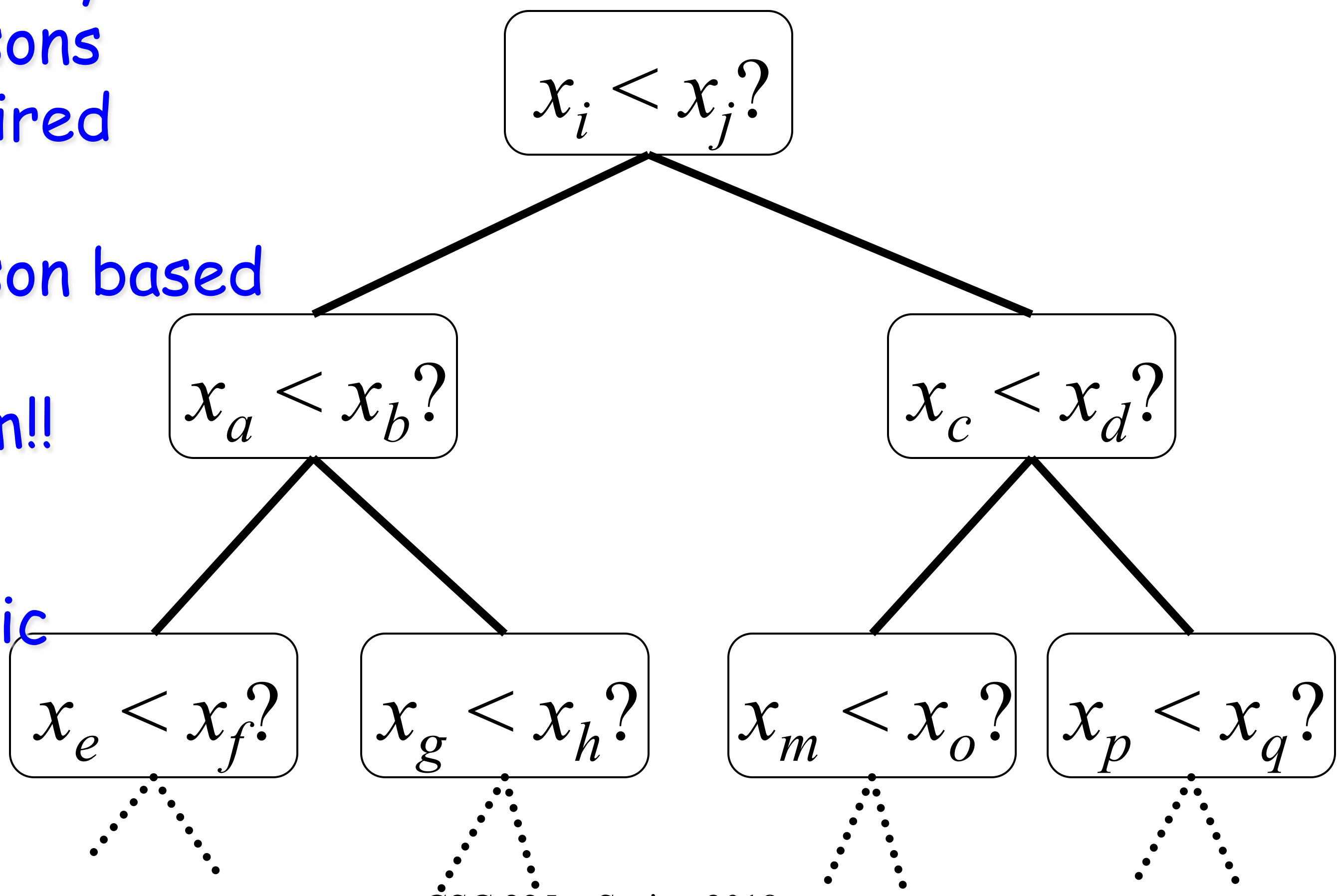


No. of leaves $\geq n!$

How many comparisons are required for an optimal sorting algorithm?

$\log(n!)$ many comparisons are required for any comparison based sorting algorithm!!

Asymptotic notation?



Big-Omega Notation

Let $f: \mathbb{N} \to \mathbb{R}$ and $g: \mathbb{N} \to \mathbb{R}$.

$f(n)$ is $\Omega(g(n))$

if and only if

$g(n)$ is $O(f(n))$.

- Lower bound

lower bound on the number of comparisons

- $\log(n!) \geq$

lower bound on the number of comparisons

- Lemma: $\log(n!) \geq \log((n/2)^{(n/2)})$

- We show: $n! \geq (n/2)^{(n/2)}$

lower bound on the number of comparisons

- $\log (n!) \geq \log((n/2)^{(n/2)})$

Proof: $n! \geq n\,(n\text{-}1)\,(n\text{-}2) \ldots 1$

$$= n\,(n\text{-}1)\,(n\text{-}2) \ldots n/2 \ldots 1$$

$$\geq (n/2)^{(n/2)}$$

Lower bound on the number of comparisons

- $\log(n!) \geq \log((n/2)^{(n/2)}) = (n/2)\log(n/2)$

$\Omega(n \log n)$