Questions of the day

- We have seen a couple of sorting algorithms so far
  - Are there faster sorting algorithms?
  - What the fastest *possible* sorting algorithm?

Sorting and selection algorithms

William Hendrix

Today

- Review
  - Data structures
  - Sorting algorithms
- BubbleSort
- MergeSort
- QuickSort
- Average-case analysis
- Comparison-based sorting lower bound
- Non-comparison-based sorting algorithms
  - CountingSort
  - BucketSort
  - RadixSort
- QuickSelect
- RSelect

Review: data structures

- *For all:* know operations, implementation, and complexity
- **Stack:** access elements in LIFO order
- **Queue:** access elements in FIFO order
  - **Deque:** can simulate stack or queue
  - Array preferred, linked list okay
- **Priority queue:** access elements in "best first" order
  - Heap or Fibonacci heap
- **List:** store elements by index
  - Array or linked list
- **Set:** searchable collection of objects
- **Map:** searchable collection of associations (*key-value pairs*)
  - Balanced BST, hash table, or array
  - AVL tree operations
- **Union-Find:** represents partition/group memberships
  - Union by rank and path compression

Other worthwhile data structures

- **Prefix tree**
  - A.k.a., trie
  - Stores a collection of strings
  - Can efficiently search whether a string begins with query
    - Spellchecking or auto-complete
  - *Notable variants:* compressed prefix tree, suffix tree

- ***k*-d tree**
  - "k-dimensional tree"
  - Like BST for spatial data (2D, 3D, etc.)
  - Supports efficient nearest neighbor searching
  - Constructed to be perfectly balanced
    - Does not self-balance
  - *Alternatives:* quad-trees (2D data) or oct-trees (3D)

Review: sorting algorithms

- **SelectionSort**
  - Iteratively swap min into next spot
  - Best/worse case complexity: $\Theta(n^2)$

- **InsertionSort**
  - Iteratively insert next value into sorted array
  - Best/worse case complexity: $\Omega(n)/O(n^2)$

- **HeapSort**
  - Similar to SelectionSort but with heaps
  - Heapify array
  - DeleteMin() $n - 1$ times
  - Best/worse case complexity: $O(n \lg n)$

BubbleSort

- Another iterative sorting algorithm
- Scan array:
  - Swap adjacent elements if out of order
- Repeat until no swaps occur

```
Input: data: array of integers to sort
Input: n: the size of data
Output: permutation of data such that
        data[1] ≤ ... ≤ data[n]
Algorithm: BubbleSort
repeat
    for i = 1 to n − 1 do
        if data[i] > data[i + 1] then
            Swap data[i] and data[i + 1]
        end
    end
until the for loop makes no swaps
return data
```

BubbleSort analysis

Input: $data$: array of integers to sort
Input: n: the size of $data$
Output: permutation of $data$ such that $data[1] \leq \ldots \leq data[n]$

```
Algorithm: BubbleSort
repeat
    for i = 1 to n − 1 do
        if data[i] > data[i + 1] then
            Swap data[i] and data[i + 1]
        end
    end
until the for loop makes no swaps
return data
```

BubbleSort analysis

Input: *data*: array of integers to sort
Input: n: the size of *data*
Output: permutation of *data* such that
$data[1] \leq \ldots \leq data[n]$

Algorithm: BubbleSort

repeat
  for $i = 1$ to $n - 1$ **do** ← $\Theta(n)$ iters
    if $data[i] > data[i+1]$ **then**
      Swap $data[i]$ and $data[i+1]$ ← $\Theta(1)$
    end
  end
until the for loop makes no swaps
return *data*

BubbleSort analysis

Input: $data$: array of integers to sort
Input: n: the size of $data$
Output: permutation of $data$ such that
$data[1] \leq \ldots \leq data[n]$

Algorithm: BubbleSort

O(n)

repeat

Θ(n)

    for $i = 1$ to $n - 1$ **do**
        if $data[i] > data[i + 1]$ **then**
            Swap $data[i]$ and $data[i + 1]$
        end
    end
until the for loop makes no swaps
return $data$

After iteration k of the outer loop, the last k values will be the largest k values in the array, in sorted order

BubbleSort analysis

Input: *data*: array of integers to sort
Input: n: the size of *data*
Output: permutation of *data* such that
$data[1] \leq \ldots \leq data[n]$

Algorithm: BubbleSort

$\Theta(n^2)$

repeat
  for $i = 1$ to $n - 1$ **do**
    if $data[i] > data[i+1]$ **then**
      Swap $data[i]$ and $data[i+1]$
    end
  end
until the for loop makes no swaps

$\Theta(1)$

return *data*

Total: $O(n^2)$

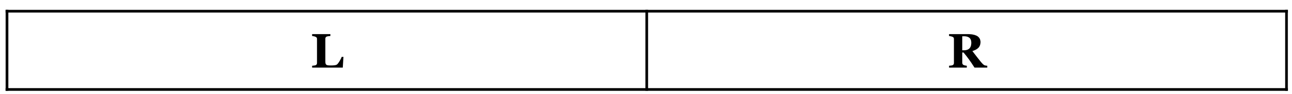
After iteration k of the outer loop, the last k values will be the largest k values in the array, in sorted order

MergeSort

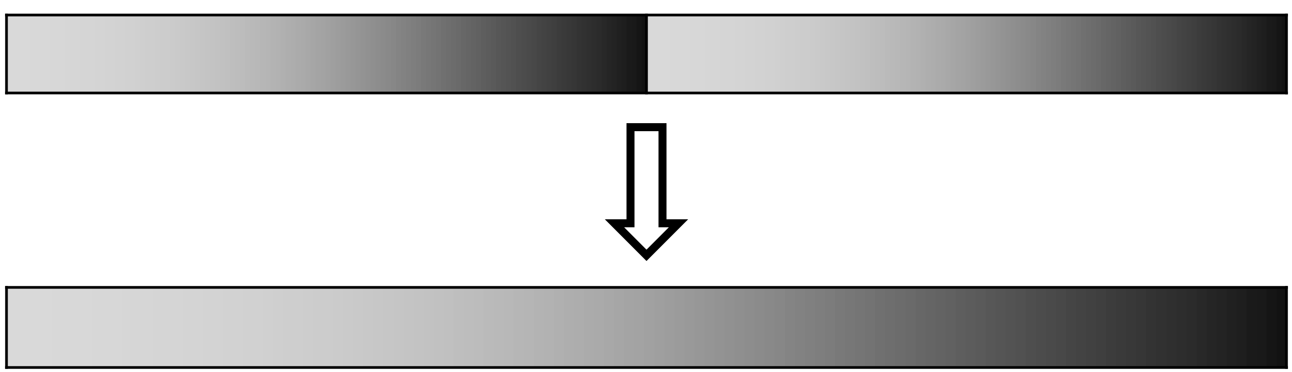
- Fast algorithm for sorting
  - InsertionSort and SelectionSort are $O(n^2)$

Description

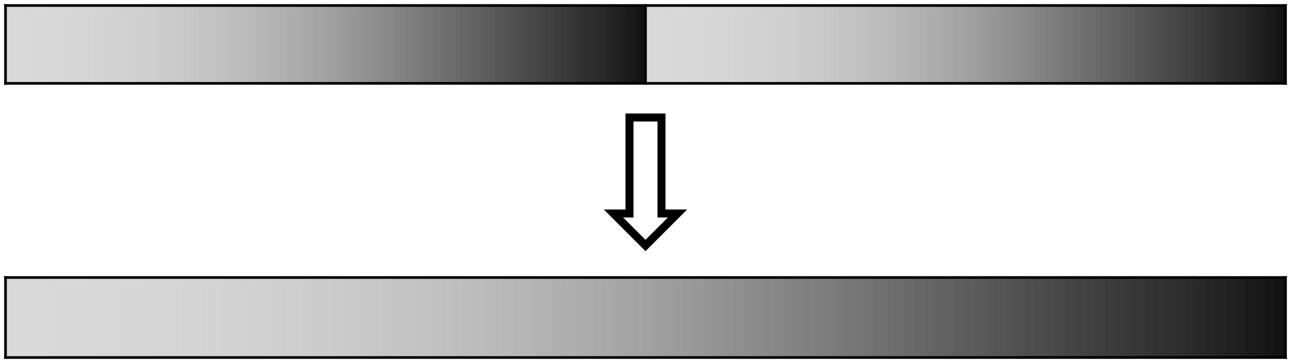
- Split array into two halves



  - Sort half-arrays recursively
    - Base case: one element
  - Combine two sorted half-arrays into one sorted array



Merging



- Fill in result left-to-right
- Min is min(left) or min(right)
- 2nd value is next value of selected half or min(unselected half)
- Continue until both arrays have emptied into result
  - After one array is empty, just add the other
- Time to combine: $\Theta(n)$
  - Better than $O(n^2)$

MergeSort pseudocode

Input: $data$: the data to sort (must be comparable)
Input: n: the number of elements in data
Output: a permutation of $data$ such that $data[1] \leq \ldots \leq data[n]$
Algorithm: MergeSort

```
if n ≤ 1 then
    return data;
end
mid = floor((n + 1)/2);
left = MergeSort(data[1..mid], mid);          ←
right = MergeSort(data[mid + 1..n], n − mid);  ←
temp = Array(n);
ℓ = r = 1;
while ℓ ≤ mid and r ≤ n − mid do
    if left[ℓ] < right[r] then
        temp[ℓ + r − 1] = left[ℓ];
        ℓ = ℓ + 1;
    else
        temp[ℓ + r − 1] = right[r];
        r = r + 1;
    end
end
temp[ℓ + r − 1..mid + r − 1] = left[ℓ..mid];
temp[mid + r..n] = right[r..n − mid];
return temp;
```

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(n) = \Theta(n \lg n)$$

QuickSort

- Another *n* log *n* algorithm for sorting
- Splits dataset according to *value*, not *position*
  - "Small half": less than some value
  - "Large half": larger than some value
  - *Pivot*: the value used to split the dataset
- Pseudocode
  1. Select pivot
     - Naïve strategy: pick first element
  2. Start at left and right ends of array
     - Swap values larger than pivot to the RHS of array
     - Swap values smaller than pivot to LHS of array
     - Equal values can go on either side
  3. Insert pivot where two "halves" meet
  4. Recursively sort each "half"
     - Base case: arrays with 0 or 1 elements are sorted

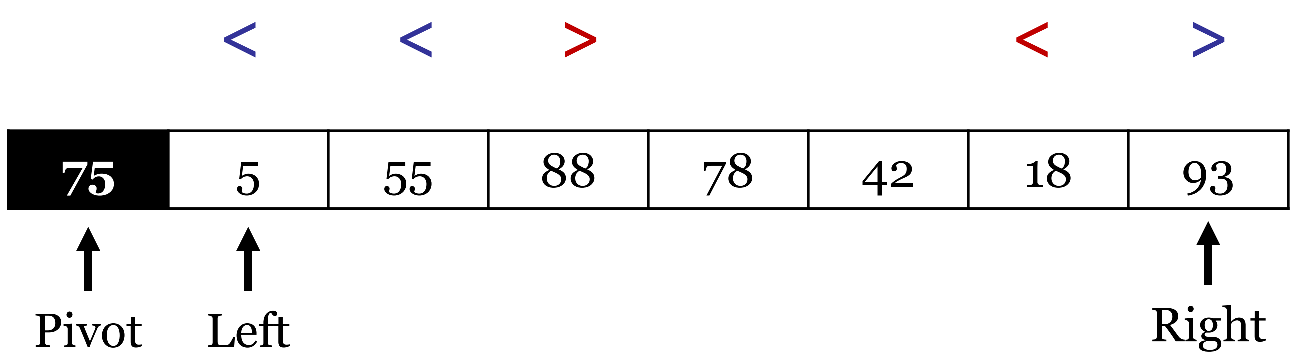
QuickSort example

- Apply QuickSort to the array below:
  1. Select pivot
     - Naïve strategy: pick first element

75	5	55	88	78	42	18	93

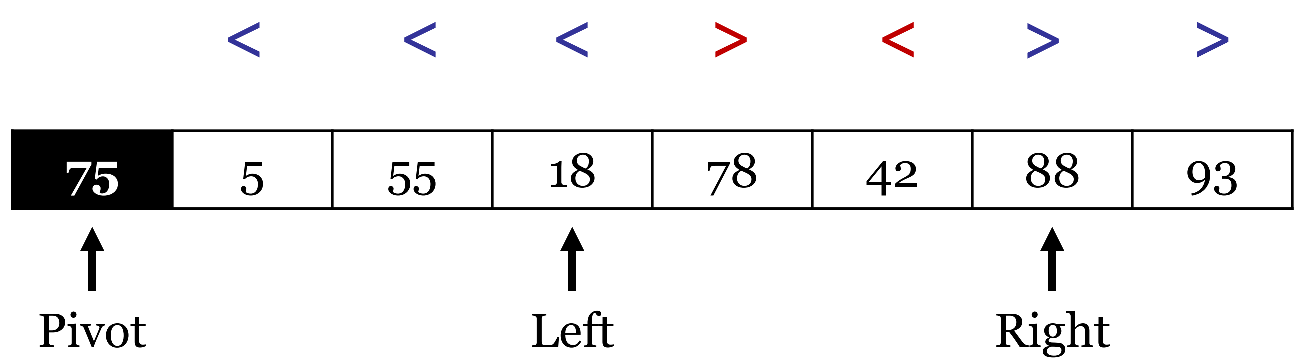
QuickSort example

- Apply QuickSort to the array below:
  1. Select pivot
     - Naïve strategy: pick first element
  2. Start at left and right ends of array
     - Swap values larger than pivot to the RHS of array
     - Swap values smaller than pivot to LHS of array
     - Equal values can go on either side



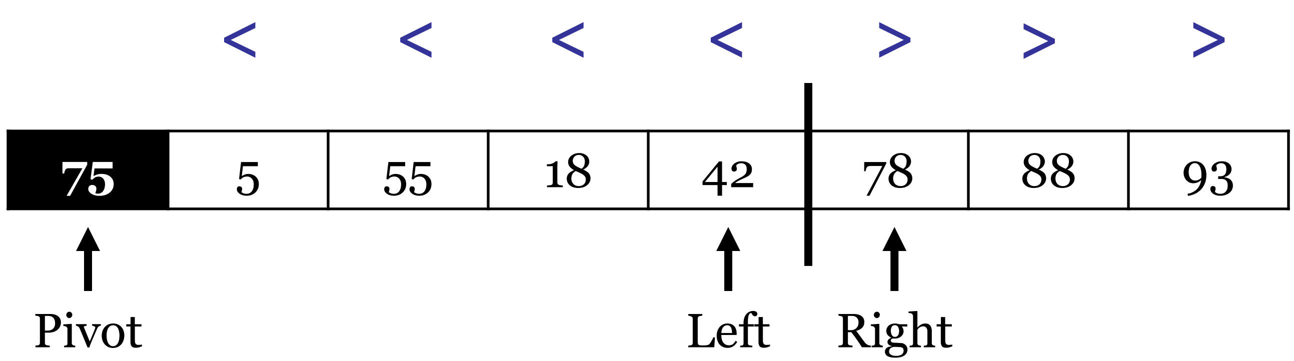
QuickSort example

- Apply QuickSort to the array below:
  1. Select pivot
     - Naïve strategy: pick first element
  2. Start at left and right ends of array
     - Swap values larger than pivot to the RHS of array
     - Swap values smaller than pivot to LHS of array
     - Equal values can go on either side



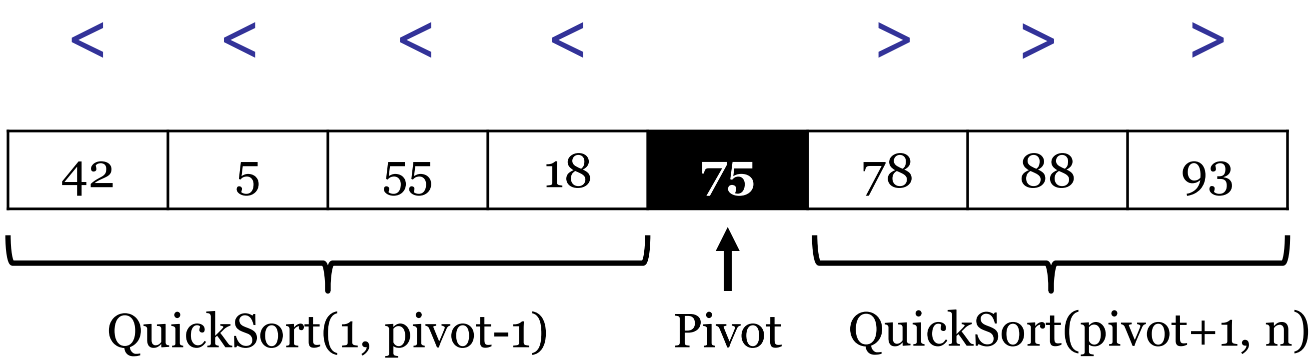
QuickSort example

- Apply QuickSort to the array below:
  1. Select pivot
     - Naïve strategy: pick first element
  2. Start at left and right ends of array
     - Swap values larger than pivot to the RHS of array
     - Swap values smaller than pivot to LHS of array
     - Equal values can go on either side
  3. Swap pivot with element where left and right meet



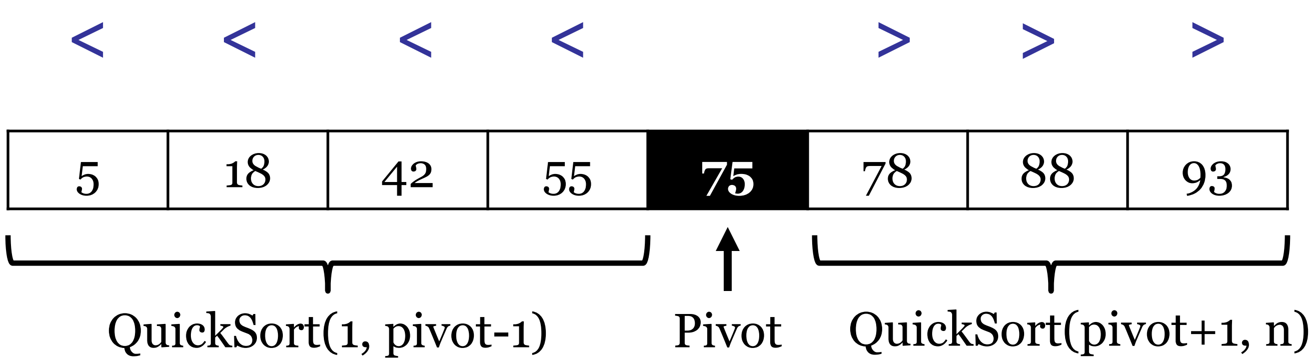
QuickSort example

- Apply QuickSort to the array below:
  1. Select pivot
     - Naïve strategy: pick first element
  2. Start at left and right ends of array
     - Swap values larger than pivot to the RHS of array
     - Swap values smaller than pivot to LHS of array
     - Equal values can go on either side
  3. Swap pivot with element where left and right meet
  4. Recurse on both sides



QuickSort example

- Apply QuickSort to the array below:
  1. Select pivot
     - Naïve strategy: pick first element
  2. Start at left and right ends of array
     - Swap values larger than pivot to the RHS of array
     - Swap values smaller than pivot to LHS of array
     - Equal values can go on either side
  3. Swap pivot with element where left and right meet
  4. Recurse on both sides



QuickSort complexity

1. Choose pivot
   – Different strategies
2. Swap elements
   – Everything left of pivot is <, right of pivot >
3. Recurse on both sides

- Step 1: $\Theta(1)$
- Step 2: $\Theta(n)$
- Recursion
  – Depends on pivot!
  – $T(n) = T(L) + T(R) + \Theta(n)$
- Worst case: $T(n) = T(n-1) + \Theta(n)$
  – $O(n^2)$
- Best case: $T(n) = 2T(n/2) + \Theta(n)$
  – $\Omega(n \lg n)$

Analysis of QuickSort

- Once the two halves are sorted, entire array is sorted
  - Can use tail recursion
- Pivot might not divide dataset exactly in half
- Pivot value impacts runtime
- Pivot selection strategies
  - First/last
    - Simple
    - Bad on sorted or constant data
  - Median-of-three
    - Choose median of first, last, and middle
    - Partitions sorted data well
  - Random
    - Always average case, unless constant values
  - Median
    - Overhead of calculating is too high: $\Theta(n)$
- Constant data can be fixed by splitting array into <, =, and >
  - Code is more complex

Average/expected case complexity

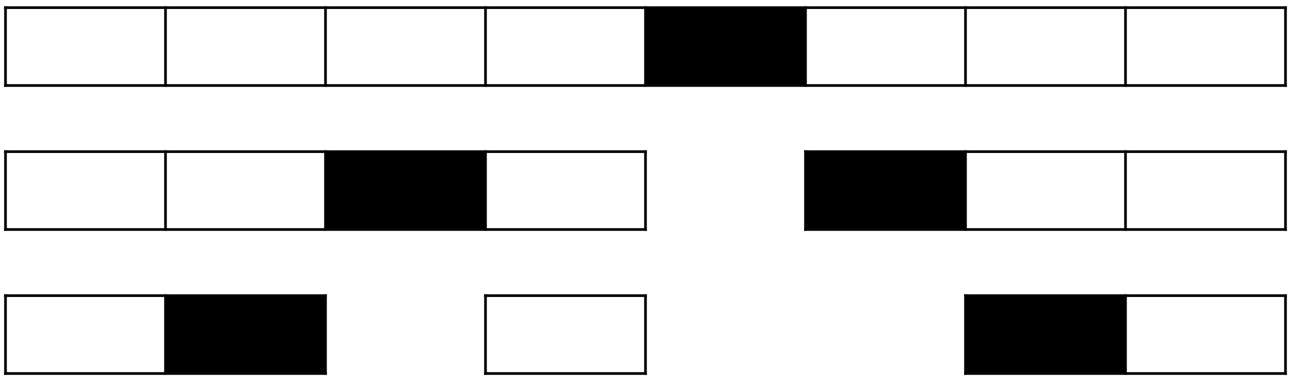
- Best-case complexity
  - Least amount of time to compute
  - E.g., InsertionSort is $\Theta(n)$ when input is sorted
- Worst-case complexity
  - Greatest amount of time to compute
  - E.g., InsertionSort is $\Theta(n^2)$ when input is reverse-sorted

- Average-case complexity
  - Average complexity across all possible inputs
  - Always falls between best and worst case
  - E.g., inner loop of InsertionSort needs to shift $n/2$ elements on average: $\Theta(n^2/2) = \Theta(n^2)$
- Expected-case complexity
  - Similar to average-case, but with assumptions about inputs

Average-case complexity of QuickSort

- Steps 1 & 2: $\Theta(n)$
- Recursion tree

Complexity



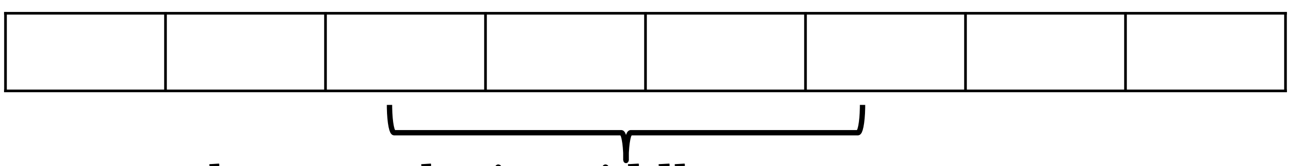
$\Theta(n)$

$O(n)$

$O(n)$

$\vdots$

- Total complexity: $O(nh)$
- Where is the pivot?



  - 50% chance to be in middle 50%
  - Reduces "big" side to ¾ $\quad h = \log_{4/3}(n)$
  - If other pivots do nothing: $h = 2\log_{4/3}(n) \Rightarrow O(n \lg n)$

Comparison-based sorting algorithms

- Sorting algorithms that operate by comparing pairs of elements
  - Everything we've discussed so far
  - All $O(n^2)$ or $O(n \lg n)$

- Is there a faster sorting algorithm?
  - No!
  - At least not comparison-based

- **Proof idea**
  - Based on *execution paths* of algorithm
    - Instructions executed for a given input
    - If statements, loop conditions, etc.
  - Execution path defines all comparisons and swaps
  - Execution path must be different for different inputs

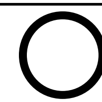
CBS algorithm lower-bound

- Array of size n has $n!$ total permutations
  - Min # of execution paths to be correct
- Each comparison: 2 outcomes
  - true or false
- # of execution paths after k comparisons:

Comparisons	Picture	Count

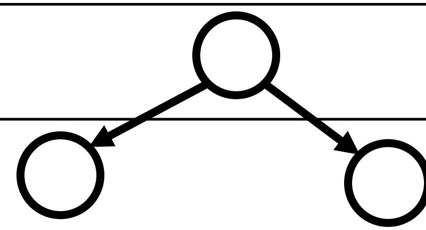
CBS algorithm lower-bound

- Array of size n has $n!$ total permutations
  - Min # of execution paths to be correct
- Each comparison: 2 outcomes
  - true or false
- # of execution paths after k comparisons:

Comparisons	**Picture**	**Count**
0		1

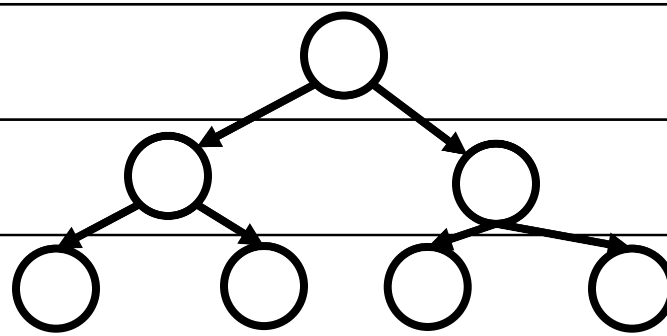
CBS algorithm lower-bound

- Array of size n has $n!$ total permutations
  - Min # of execution paths to be correct
- Each comparison: 2 outcomes
  - true or false
- # of execution paths after k comparisons:

Comparisons	**Picture**	**Count**
0		1
1		2

CBS algorithm lower-bound

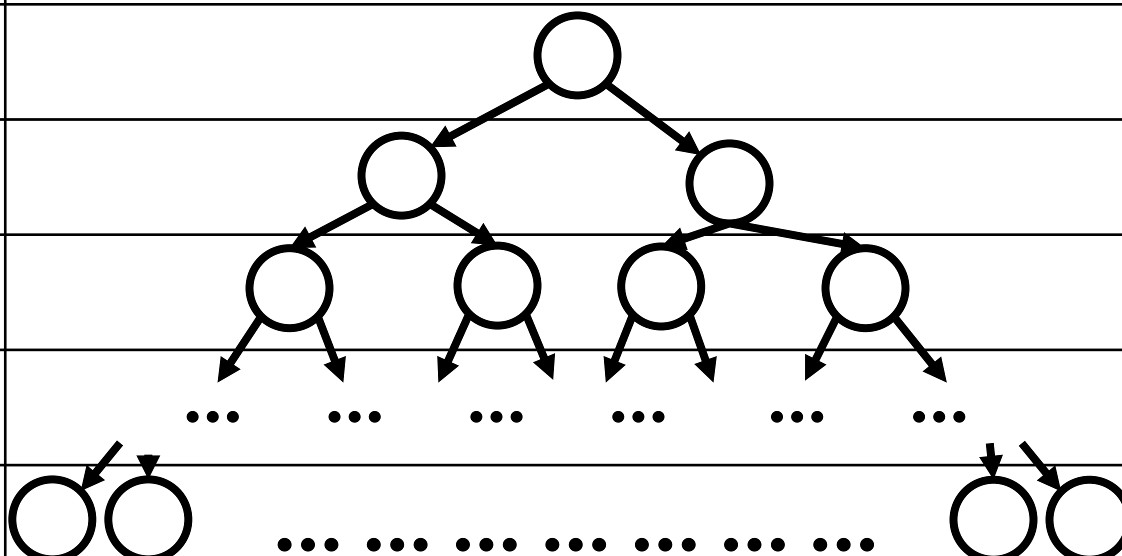
- Array of size n has $n!$ total permutations
  - Min # of execution paths to be correct
- Each comparison: 2 outcomes
  - true or false
- # of execution paths after k comparisons:

Comparisons	Picture	Count
0		1
1		2
2		4

CBS algorithm lower-bound

- Array of size n has $n!$ total permutations
  - Min # of execution paths to be correct
- Each comparison: 2 outcomes
  - true or false
- # of execution paths after k comparisons:

Comparisons	Picture	Count
0		1
1		2
2		4
…		…
k		2^k



- # comparisons to distinguish $n!$ outcomes: $\lg(n!) = \Theta(n \lg n)$

Lower-bound exercise

- The *counterfeit coin* problem:
  - **Input:** a stack of n coins, all identical except for one counterfeit, which weighs less
  - **Output:** index of counterfeit coin
  - *Primary operation:* weigh any two sets of coins on a scale
    - Returns <, >, or =

1. What is the number of possible outputs for an input of size n?
2. How many execution paths do k weighings yield?
3. Find a lower bound on the weighings required for this problem.

Lower-bound exercise

- The *counterfeit coin* problem:
  - **Input:** a stack of n coins, all identical except for one counterfeit, which weighs less
  - **Output:** index of counterfeit coin
  - *Primary operation:* weigh any two sets of coins on a scale
    - Returns <, >, or =

1. What is the number of possible outputs for an input of size n?
   - n (any of the n coins could be the counterfeit)
2. How many execution paths do k weighings yield?
   - 3^k (each has 3 possibilities)
3. Find a lower bound on the weighings required for this problem.
   - $3^k \geq n$
   - $k \geq \log_3(n)$

- **Food for thought:** can you find the optimum algorithm that makes $\log_3(n)$ comparisons?

Non-comparison-based sorting

- **Main idea**
  - Locate the correct position without comparing to other values in array
  - Compare values to constants rather than each other
  - Use additional memory to avoid direct comparisons

- Three main algorithms
  - CountingSort
  - BucketSort
  - RadixSort

- Each has different advantages/disadvantages
  - None are unconditionally better than comparison-based

CountingSort

- **Main idea**
  - For each element:
    - Count number of smaller elements in array
    - Drop into correct position in sorted array
  - Data must be nonnegative integers
    - Negative values more difficult

- **Pseudocode**
  - Find max(*data*)
  - Create array *count* of length max(*data*)
  - Increment count so that *count*[*i*] is # of *i*'s in *data*
  - Prepend 0 to count array
  - Perform a cumulative sum
    - *count[i]* is number of values < *i* (where *i* goes in sorted order)
  - Place each *data[i]* into correct position according to *count*

CountingSort example

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that $data[0] \leq \ldots \leq data[n-1]$

Algorithm: CountingSort

$max = \max(data)$
$count = \text{Array}(max)$
Initialize *count* to 0
for $i = 0$ to $n - 1$ **do**
    Increment $count[data[i]]$
end
Perform a cumulative sum on *count*
$sorted = \text{Array}(n)$
for $i = n - 1$ down to 0 **do**
    Decrement $count[data[i]]$
    $sorted[count[data[i]]] = data[i]$
end
return *sorted*

2	1	3	2	2	1

* Pseudocode uses 0 base and assumes *data* > 0

CountingSort example

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that $data[0] \leq \ldots \leq data[n-1]$

Algorithm: CountingSort
$max = \max(data)$
$count = \text{Array}(max)$
Initialize *count* to 0
for $i = 0$ to $n - 1$ **do**
    Increment $count[data[i]]$
end
Perform a cumulative sum on *count*
$sorted = \text{Array}(n)$
for $i = n - 1$ down to 0 **do**
    Decrement $count[data[i]]$
    $sorted[count[data[i]]] = data[i]$
end
return *sorted*

* Pseudocode uses 0 base and assumes *data* > 0

2	1	3	2	2	1

count:

0	1	2	3
0	0	0	0

CountingSort example

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that $data[0] \leq \ldots \leq data[n-1]$

Algorithm: CountingSort

$max = \max(data)$
$count = \text{Array}(max)$
Initialize *count* to 0
for $i = 0$ to $n - 1$ **do**
    Increment $count[data[i]]$
end
Perform a cumulative sum on *count*
$sorted = \text{Array}(n)$
for $i = n - 1$ down to 0 **do**
    Decrement $count[data[i]]$
    $sorted[count[data[i]]] = data[i]$
end
return *sorted*

2	1	3	2	2	1

count:

0	1	2	3
0	2	3	1

CountingSort example

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that $data[0] \leq \ldots \leq data[n-1]$

```
Algorithm: CountingSort
max = max(data)
count = Array(max)
Initialize count to 0
for i = 0 to n − 1 do
    Increment count[data[i]]
end
Perform a cumulative sum on count
sorted = Array(n)
for i = n − 1 down to 0 do
    Decrement count[data[i]]
    sorted[count[data[i]]] = data[i]
end
return sorted
```

2	1	3	2	2	1

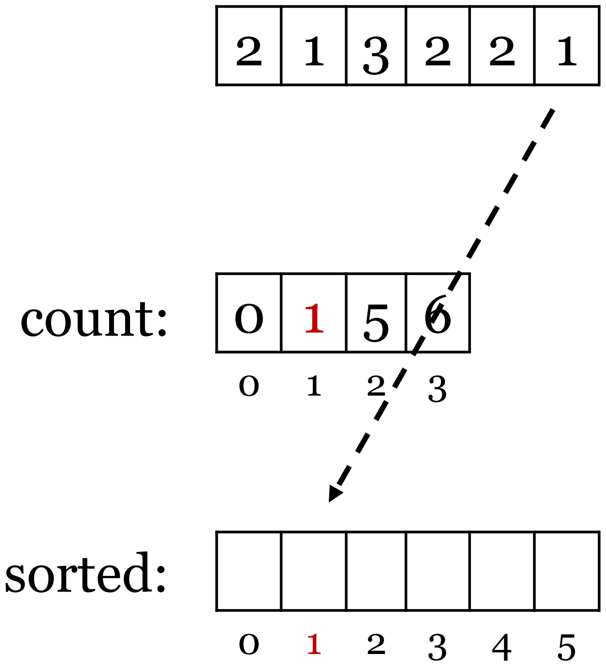
count:

0	2	5	6
0	1	2	3

CountingSort example

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that
$data[0] \leq \ldots \leq data[n-1]$

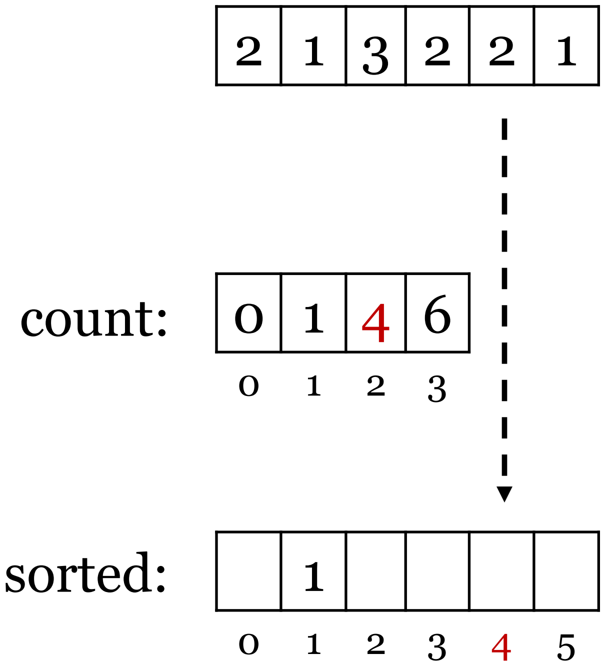
```
Algorithm: CountingSort
max = max(data)
count = Array(max)
Initialize count to 0
for i = 0 to n − 1 do
    Increment count[data[i]]
end
Perform a cumulative sum on count
sorted = Array(n)
for i = n − 1 down to 0 do
    Decrement count[data[i]]
    sorted[count[data[i]]] = data[i]
end
return sorted
```



CountingSort example

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that $data[0] \leq \ldots \leq data[n-1]$

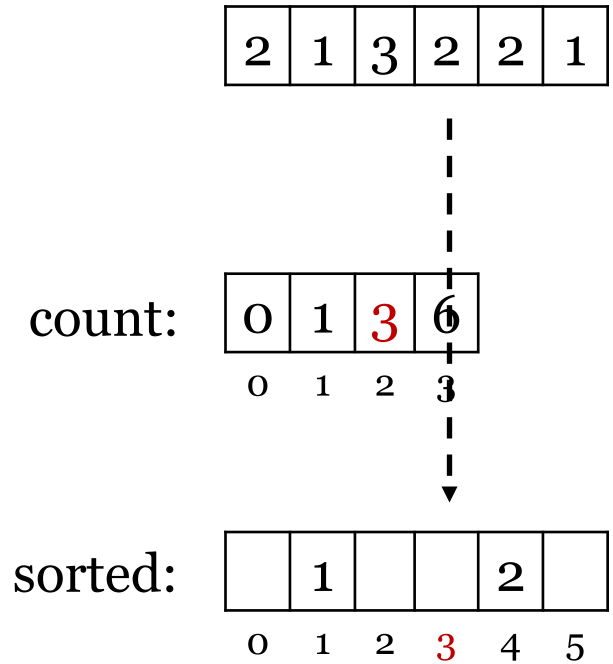
Algorithm: CountingSort
$max = \max(data)$
$count = \text{Array}(max)$
Initialize *count* to 0
for $i = 0$ to $n-1$ **do**
  Increment $count[data[i]]$
end
Perform a cumulative sum on *count*
$sorted = \text{Array}(n)$
for $i = n-1$ down to 0 **do**
  Decrement $count[data[i]]$
  $sorted[count[data[i]]] = data[i]$
end
return *sorted*



CountingSort example

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that
$data[0] \leq \ldots \leq data[n-1]$

Algorithm: CountingSort
$max = \max(data)$
$count = \text{Array}(max)$
Initialize *count* to 0
for $i = 0$ to $n-1$ **do**
  Increment $count[data[i]]$
end
Perform a cumulative sum on *count*
$sorted = \text{Array}(n)$
for $i = n-1$ down to 0 **do**
  Decrement $count[data[i]]$
  $sorted[count[data[i]]] = data[i]$
end
return *sorted*

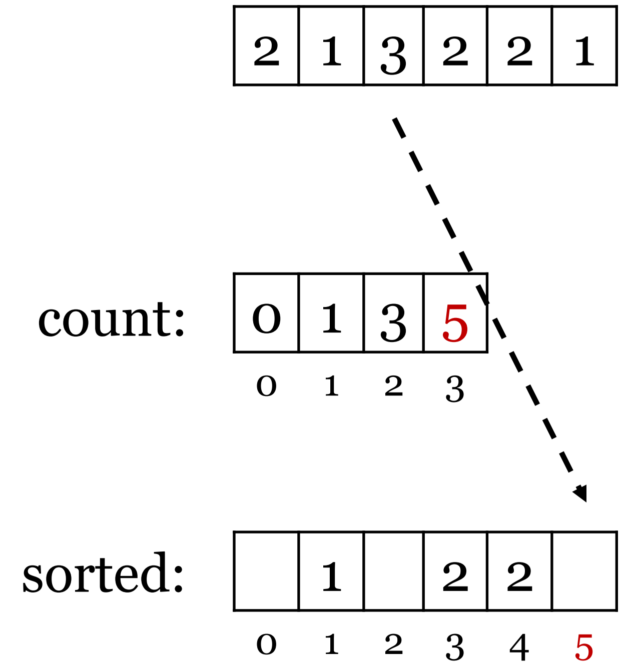


CountingSort example

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that $data[0] \leq \ldots \leq data[n-1]$

Algorithm: CountingSort

$max = \max(data)$
$count = \text{Array}(max)$
Initialize $count$ to 0
for $i = 0$ to $n - 1$ **do**
  Increment $count[data[i]]$
end
Perform a cumulative sum on $count$
$sorted = \text{Array}(n)$
for $i = n - 1$ down to 0 **do**
  Decrement $count[data[i]]$
  $sorted[count[data[i]]] = data[i]$
end
return *sorted*

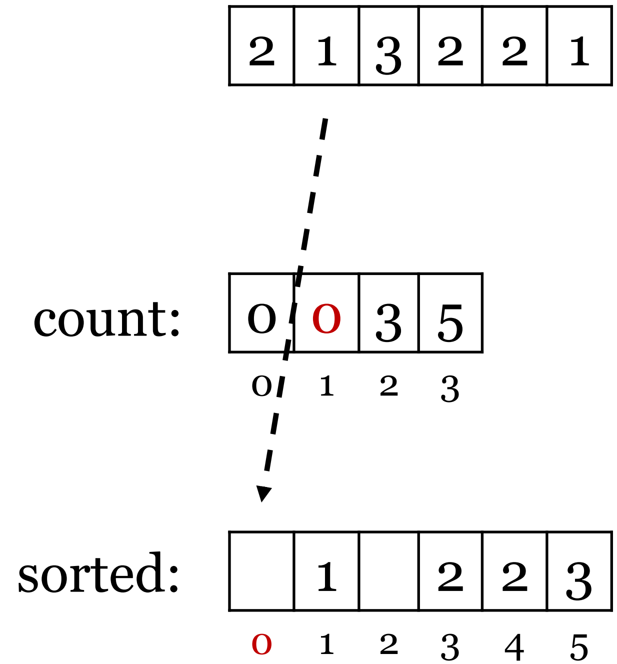


CountingSort example

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that $data[0] \leq \ldots \leq data[n-1]$

Algorithm: CountingSort

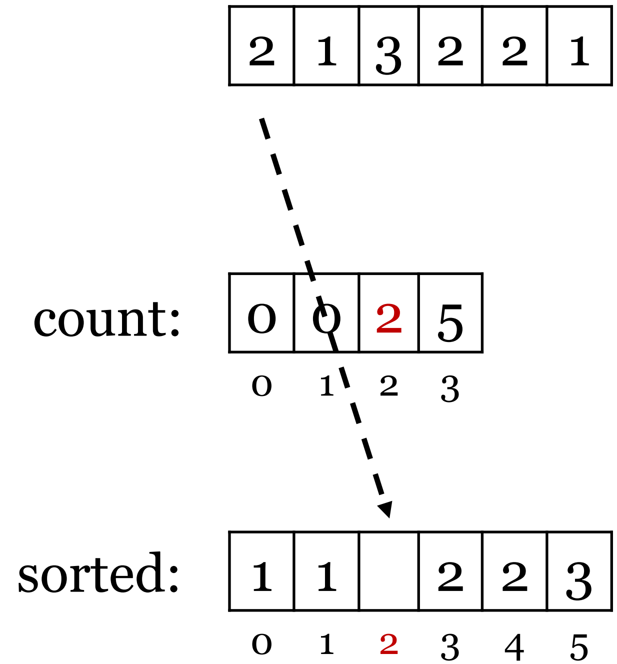
$max = \max(data)$
$count = \text{Array}(max)$
Initialize $count$ to 0
for $i = 0$ to $n-1$ **do**
  Increment $count[data[i]]$
end
Perform a cumulative sum on $count$
$sorted = \text{Array}(n)$
for $i = n-1$ down to 0 **do**
  Decrement $count[data[i]]$
  $sorted[count[data[i]]] = data[i]$
end
return $sorted$



CountingSort example

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that $data[0] \leq \ldots \leq data[n-1]$

```
Algorithm: CountingSort
max = max(data)
count = Array(max)
Initialize count to 0
for i = 0 to n − 1 do
    Increment count[data[i]]
end
Perform a cumulative sum on count
sorted = Array(n)
for i = n − 1 down to 0 do
    Decrement count[data[i]]
    sorted[count[data[i]]] = data[i]
end
return sorted
```



CountingSort example

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that $data[0] \leq \ldots \leq data[n-1]$

```
Algorithm: CountingSort
max = max(data)
count = Array(max)
Initialize count to 0
for i = 0 to n − 1 do
    Increment count[data[i]]
end
Perform a cumulative sum on count
sorted = Array(n)
for i = n − 1 down to 0 do
    Decrement count[data[i]]
    sorted[count[data[i]]] = data[i]
end
return sorted
```

2	1	3	2	2	1

count:

0	1	2	3
0	0	2	5

sorted:

0	1	2	3	4	5
1	1	2	2	2	3

CountingSort analysis

Input: *data*: array of n values
Input: n: size of *data*
Output: a permutation of *data* such that $data[0] \leq \ldots \leq data[n-1]$

Algorithm: CountingSort

$max = \max(data)$
$count = \text{Array}(max)$
Initialize *count* to 0
for $i = 0$ to $n-1$ **do**
  Increment $count[data[i]]$
end
Perform a cumulative sum on *count*
$sorted = \text{Array}(n)$
for $i = n-1$ down to 0 **do**
  Decrement $count[data[i]]$
  $sorted[count[data[i]]] = data[i]$
end
return *sorted*

CountingSort analysis

<u>Worst case complexity</u>	
	Input: *data*: array of n values
	Input: n: size of *data*
	Output: a permutation of *data* such that $data[0] \leq \ldots \leq data[n-1]$
	Algorithm: CountingSort
O(n)	$max = \max(data)$
O(max)	$count = \text{Array}(max)$
	Initialize *count* to 0
O(n)	**for** $i = 0$ to $n - 1$ **do**
	Increment $count[data[i]]$
	end
O(max)	Perform a cumulative sum on *count*
O(1)	$sorted = \text{Array}(n)$
O(n)	**for** $i = n - 1$ down to 0 **do**
	Decrement $count[data[i]]$
	$sorted[count[data[i]]] = data[i]$
	end
O(1)	**return** *sorted*

<u>Total:</u>
O(n + max)

- Complexity depends on data range
  - Potentially linear
- Good when range is small
- Terrible if range is large
  - Increases memory requirements, too
- Only works with integers or integer-like data

BucketSort

- **Main idea**
  - Divide data range into *n* “buckets”
  - Assign data to buckets and sort independently
  - Buckets “should” take *O(1)* to sort

- **Pseudocode**
  - Calculate max and min of data
  - Create *n* lists
  - For each element *data[i]*,
    - Compute $b = \text{floor}(n(\mathit{data}[i] - \text{min}) / (\text{max} - \text{min} + 1))$
    - Insert *data[i]* into *list[b]* in sorted order
  - Concatenate all lists

BucketSort example

- **Data:**

27	63	30	91	0	13	76	61	99	55

  - 10 elements

BucketSort example

- **Data:**

27	63	30	91	0	13	76	61	99	55

  – 10 elements
- **Data range:** [0, 99]
- **Bucket size:** (max – min + 1) / n = 10
- **Bucket ranges:**

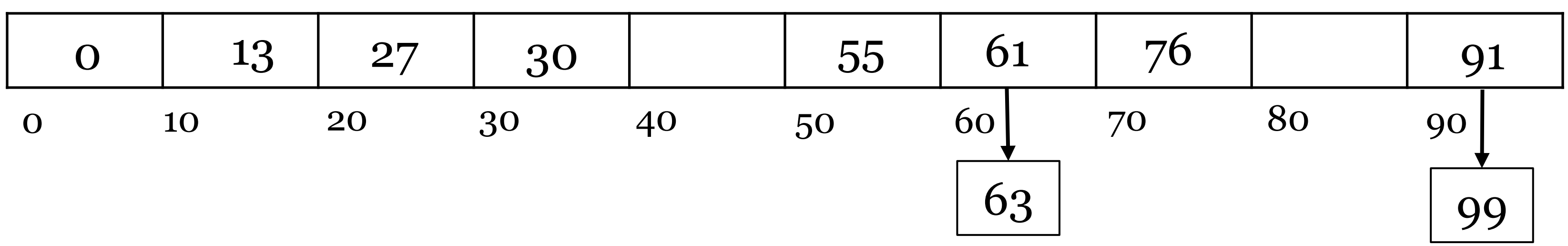
0	13	27	30			63	76		91
0	10	20	30	40	50	60	70	80	90

BucketSort example

- **Data:**

27	63	30	91	0	13	76	61	99	55

  – 10 elements
- **Data range:** [0, 99]
- **Bucket size:** (max – min + 1) / n = 10
- **Bucket ranges:**



- **Solution:**

0	13	27	30	55	61	63	76	91	99

BucketSort analysis

Analysis

- Calculate max and min of data
- Create *n* lists
- For each element *data[i]*,
  - Compute $b = \left\lfloor \dfrac{n(data[i] - \min)}{\max - \min + 1} \right\rfloor$
  - Insert *data[i]* into *list[b]* in sorted order
- Concatenate all lists

BucketSort analysis

Analysis

<u>Complexity</u>

- Calculate max and min of data $\Theta(n)$
- Create n lists $\Theta(n)$
- For each element *data[i]*, $\Theta(n)$
  - Compute $b = \left\lfloor \frac{n(data[i] - \min)}{\max - \min + 1} \right\rfloor$ $\Theta(1)$
  - Insert *data[i]* into *list[b]* in sorted order $\Omega(1) \text{ to } O(n)$
- Concatenate all lists $\Theta(n)$
- Total complexity: $\Omega(n) \text{ to } O(n^2)$
  - If each list is *O(1)*: $\Theta(n)$
- Linear time if data is evenly distributed
- Stable
- Quadratic time if not
- Uses $O(n)$ space

RadixSort

- **Main idea**
  - Treat numbers as a sequence of digits
    - Also works for words (sequence of characters)
  - Sort digits from least to most significant
    - By the time you reach the last digit, you're done!

- **Pseudocode**
  - For each digit d_i from least to most significant
    - Use CountingSort to sort data by d_i

RadixSort example

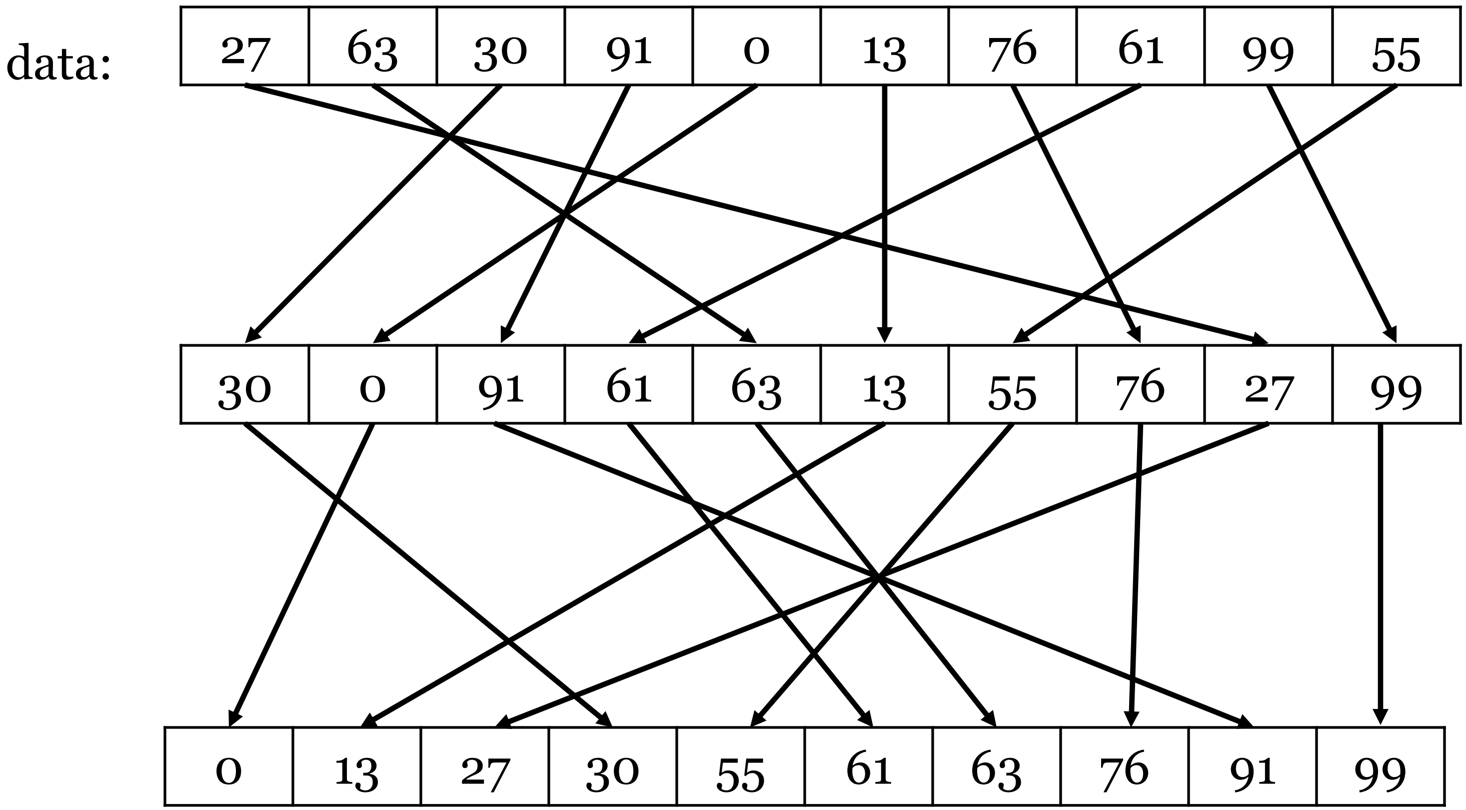
- For each digit d_i from least to most significant
  - Use CountingSort to sort data by d_i
- **Example**

data:

27	63	30	91	0	13	76	61	99	55

RadixSort example

- For each digit d_i from least to most significant
  - Use CountingSort to sort data by d_i
- **Example**



RadixSort analysis

Analysis

- For each digit d_i from least to most significant
  - Use CountingSort to sort data by d_i

RadixSort analysis

Analysis	Complexity
• For each digit d_i from least to most significant	$\Theta(\lg r)$
– Use CountingSort to sort data by d_i	$\Theta(n)$

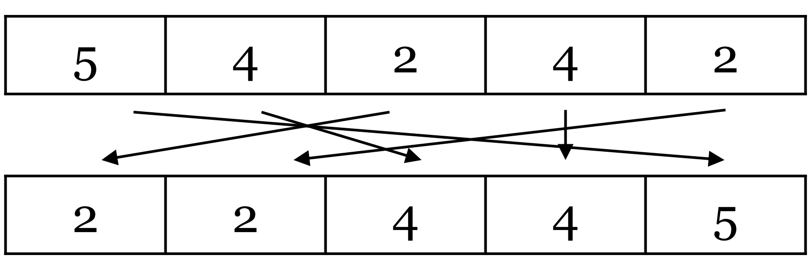
- Total complexity: $\Theta(n \lg r)$
  - If $r = O(n)$, complexity is $O(n \lg n)$
  - If $r = O(1)$, complexity is $O(n)$

- Linear complexity if range is bounded
- Uses $O(n)$ space
- Performance depends on range/length of longest entry

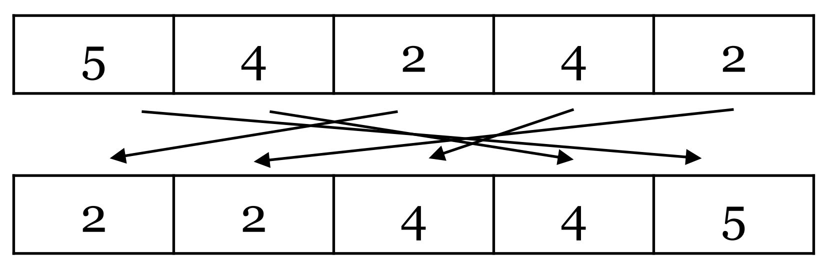
Sorting algorithm evaluation

- There are 3 main features to consider when sorting data
  - **Speed/time complexity**
    - Usually most important factor
  - ***In-place* sorting**
    - Algorithm uses no data structures other than array
      - No temp arrays!
    - Very memory efficient
  - ***Stable* sorting**
    - Equal values retain their relative order
    - *Unstable*: equal values appear in random order

Stable



Unstable



- Important for some applications
  - E.g., sorting rows in Excel file

Sorting algorithm comparison

	Best-case complexity	Worst case	Avg/exp case	Stable?	In-place?
SelectionSort	$\Omega(n^2)$	$O(n^2)$	$\Theta(n^2)$	✓	✓
InsertionSort	$\Omega(n)$	$O(n^2)$	$\Theta(n^2)$	✓	✓
BubbleSort	$\Omega(n)$	$O(n^2)$	$\Theta(n^2)$	✓	✓
HeapSort	$\Omega(n \lg n)$	$O(n \lg n)$	$\Theta(n \lg n)$	✗	✓
MergeSort	$\Omega(n \lg n)$	$O(n \lg n)$	$\Theta(n \lg n)$	✓	✗
QuickSort	$\Omega(n \lg n)$	$O(n^2)$	$\Theta(n \lg n)$	*	✓
CountingSort	$\Omega(n + r)$	$O(n + r)$	$\Theta(n)^\dagger$	✓	✗
BucketSort	$\Omega(n)$	$O(n^2)$	$\Theta(n)^\dagger$	✓	✗
RadixSort	$\Omega(n)$	$O(n \lg r)$	$\Theta(n \lg n)^\dagger$	✓	*

† with "good" data
* possible but slower

Selection

- Consider related problem of finding the *k*th smallest element in an array of size *n*

- **Example:**

27	**82**	**3**	**100**	**96**	**15**	**41**

  - Find 3rd smallest value

- If $k = O(1)$ or $n-O(1)$, we can find the value using a modified min or max algorithm
  - Keep track of k largest/smallest values seen
  - Insert values until entire array is processed
  - $O(n + k^2)$
  - Modified HeapSort: $O(n + k \lg n)$
- If $k = O(n)$, this is no better than sorting!

QuickSelect

- **Main idea:** partition as in QuickSort and recursively search one half
  - Partition as in QuickSort
  - Search one half recursively (like Binary Search)

- **Example (first element as pivot):**

27	**82**	**3**	**100**	**96**	**15**	**41**

QuickSelect

- **Main idea:** partition as in QuickSort and recursively search one half
  - Partition as in QuickSort
  - Search one half recursively (like Binary Search)

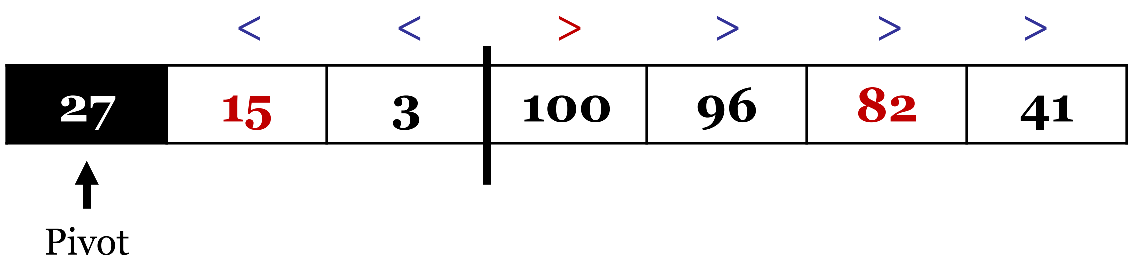
- **Example (first element as pivot):**

	>				<	>
27	**82**	**3**	**100**	**96**	**15**	**41**
Pivot						

QuickSelect

- **Main idea:** partition as in QuickSort and recursively search one half
  - Partition as in QuickSort
  - Search one half recursively (like Binary Search)

- **Example (first element as pivot):**



QuickSelect

- **Main idea:** partition as in QuickSort and recursively search one half
  - Partition as in QuickSort
  - Search one half recursively (like Binary Search)

- **Example (first element as pivot):**

$<$	$<$		$>$	$>$	$>$	$>$
3	**15**	**27**	**100**	**96**	**82**	**41**

- Complexity depends on pivot
  - Best case: $T(n) = T(n/2) + \Theta(n)$
    - $\Omega(n)$
  - Worst case: $T(n) = T(n-1) + \Theta(n)$
    - $O(n^2)$
  - Random pivot: $\Theta(n)$ (average case)
- Is there a selection algorithm with *Θ(n)* worst case behavior?

Coming up

- Brute force
- Greedy algorithms
- Midterm Exam

- **Recommended readings:** Sections 8.1-9.2
- *Practice problems:* R-8.2, R-8.4, R-8.7, C-8.1, C-8.3, C-8.5, C-8.9, A-8.2, R-9.4, R-9.5, C-9.1, C-9.3, C-9.12, A-9.2