Sorting algorithms

William Hendrix

Lecture 14

Outline

- Brief review: InsertionSort, BubbleSort, SelectionSort, HeapSort, MergeSort
- Quicksort
- Comparison-based sorting
- Non-comparison-based sorting methods

Review: quadratic sorting algorithms

- Three main quadratic sorting algorithms
- BubbleSort
  - Swap adjacent values that are out of order $\Theta(n)$ time
  - Continue until no swaps are made $\Omega(1)$ to $O(n)$ iterations
  - Best/worst case complexity: $\Omega(n)/O(n^2)$
- InsertionSort
  - Insert each element into correct position $\Theta(n)$ elements
  - Shift sorted elements in order to make a space $\Omega(1)$ to $O(n)$ time
  - Best/worst case complexity: $\Omega(n)/O(n^2)$
- SelectionSort
  - Find min in unsorted portion of array $\Theta(n)$ time
  - Swap to beginning and repeat $\Theta(n)$ iterations
  - Best/worst case complexity: $\Theta(n^2)/\Theta(n^2)$

Linearithmic sorting algorithms

- HeapSort
  - Application of appropriate data structure to SelectionSort
  - Organize data in heap (usually max heap) $\Theta(n)$
  - Extract max, swap to end $\Theta(\lg n)$
  - Repeat until sorted $\Theta(n)$ iterations
  - Best/Worst case: $\Theta(n \lg n)/\Theta(n \lg n)$

- MergeSort
  - Application of divide-and-conquer strategy
  - Split array in two $\Theta(1)$
  - Recursively call MergeSort on two halves $2T(n/2)$
  - Merge arrays $\Theta(n)$
  - Best/Worst case: $\Theta(n \lg n)/\Theta(n \lg n)$, by MT

QuickSort

- Alternative divide-and-conquer 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

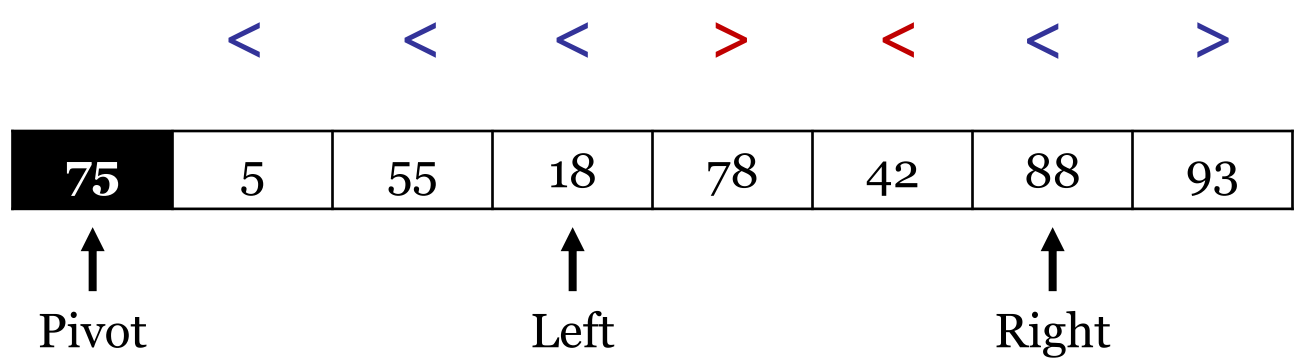
QuickSort example

- 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

	<	<	>			<	>
75	5	55	88	78	42	18	93
Pivot	Left						Right

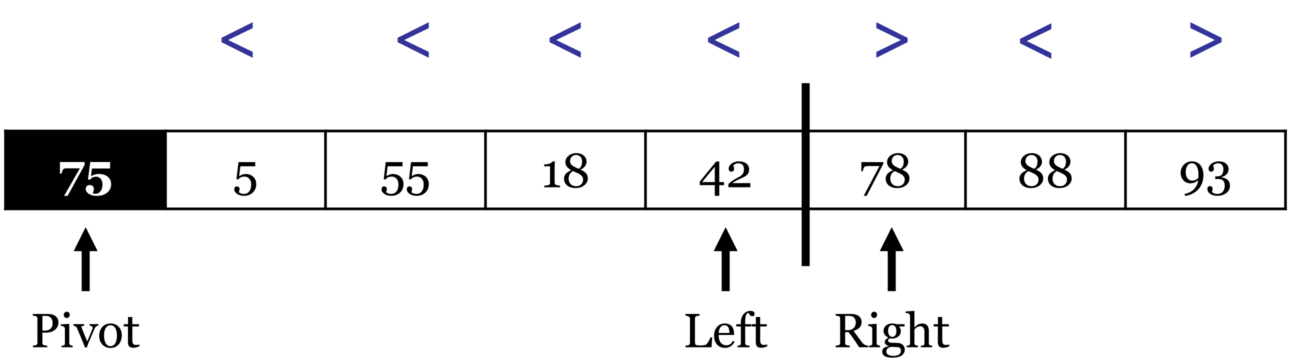
QuickSort example

- 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



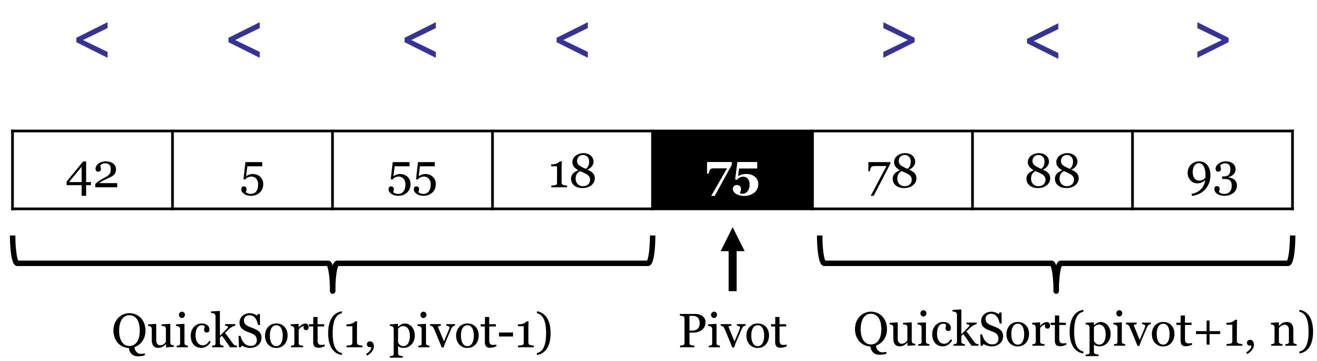
QuickSort example

- 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

<	<	<	<		>	<	>
5	18	42	55	**75**	78	88	93

QuickSort(1, pivot-1)   Pivot   QuickSort(pivot+1, n)

QuickSort complexity

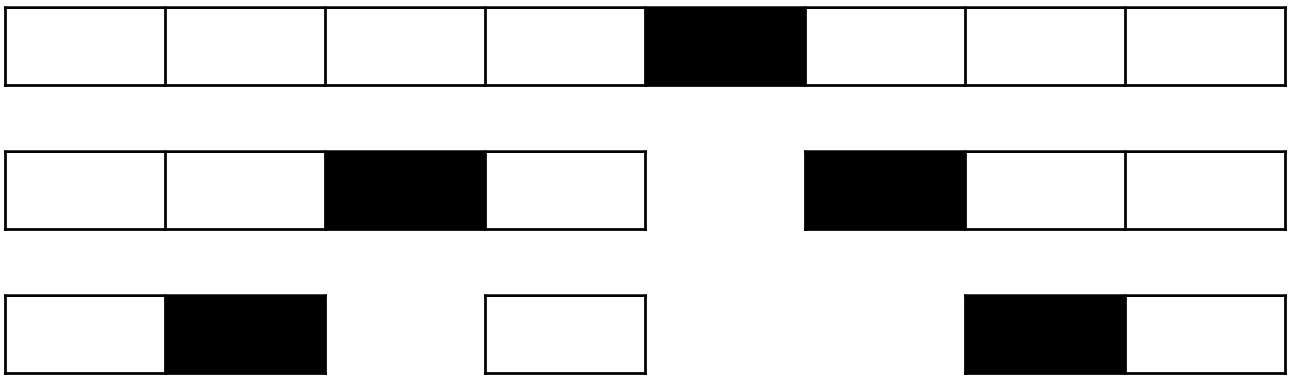
1. Choose pivot
   – Different strategies
2. Swap elements
   – Everything left of pivot is <, right of pivot >
3. Recurse on both sides

- Step 1: $\Theta(1)$
- Step 2: $\Theta(n)$
- Recursion
  – Depends on pivot!
  – Best case: $2T(n/2) \Rightarrow \Omega(n \lg n)$
  – Worst case: $T(n\text{-}1) \Rightarrow O(n^2)$

Average case complexity

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

Complexity



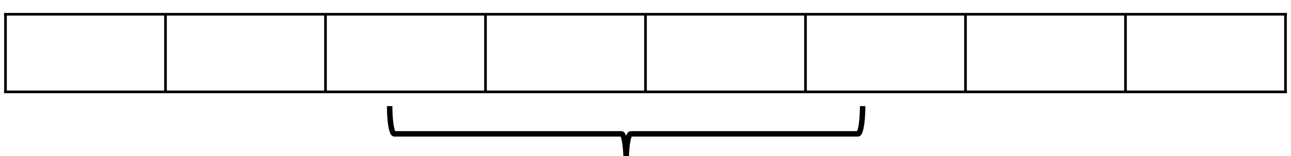
$\Theta(n)$

$\sim \Theta(n)$

$\sim \Theta(n)$

$\vdots$

- Total complexity: $\Theta(nh)$
- 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: $h = 2\log_{4/3}(n) \Rightarrow \Theta(n \lg n)$

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

Important sorting features

- Time complexity
- In-place
  - Needs no additional space
- Stable
  - Elements with equal value keep relative order
- **Example:**

a=3	b=2	c=2	d=3	$\Rightarrow$	b	c	a	d

Algorithm	Best case	Worst case	In-place?	Stable?
BubbleSort	$\Omega(n)$	$O(n^2)$	Y	Y
InsertionSort	$\Omega(n)$	$O(n^2)$	Y	Y
SelectionSort	$\Omega(n^2)$	$O(n^2)$	Y	Y
HeapSort	$\Omega(n\lg n)$	$O(n\lg n)$	Y	N
MergeSort	$\Omega(n\lg n)$	$O(n\lg n)$	N*	Y
QuickSort	$\Omega(n\lg n)$	$O(n^2)$	Y	N*

Fastest for small input

Generally considered the fastest

* Possible, but difficult

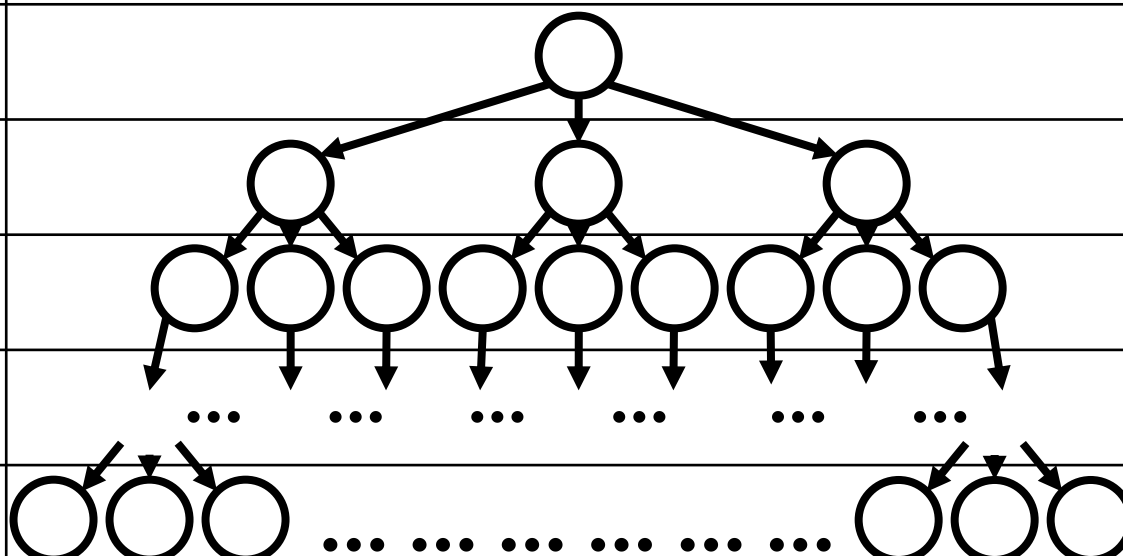
Comparison-based sorting algorithms

- Sorting algorithms that operate by comparing pairs of elements
  - BubbleSort, SelectionSort, InsertionSort, HeapSort, MergeSort, QuickSort
  - All $O(n^2)$ or $O(n \lg n)$

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

CBS algorithm lower-bound

- Array of size n has $n!$ total permutations
  - Correct algorithm must perform different swaps in every case
- Each comparison: up to 3 outcomes
  - <, >, =
- # of outcomes after k swaps:

Comparisons	Picture	Count
0		1
1		3
2		9
…		…
k		3^k

- # comparisons to distinguish $n!$ outcomes: $\log_3(n!) = \Theta(n \lg n)$

Coming up

- Final sorting algorithms
- Graphs

- **Project 1** will be due Oct. 18
- **Homework 7** (posted tonight) will be due Oct 22

- **Recommended readings:** Sections 5.1 and 5.2
- **Practice problems:** 4-2, 4-18, 4-29 (p. 139)