QuickSort

CSE 332 Spring 2021

Instructor: Hannah C. Tang

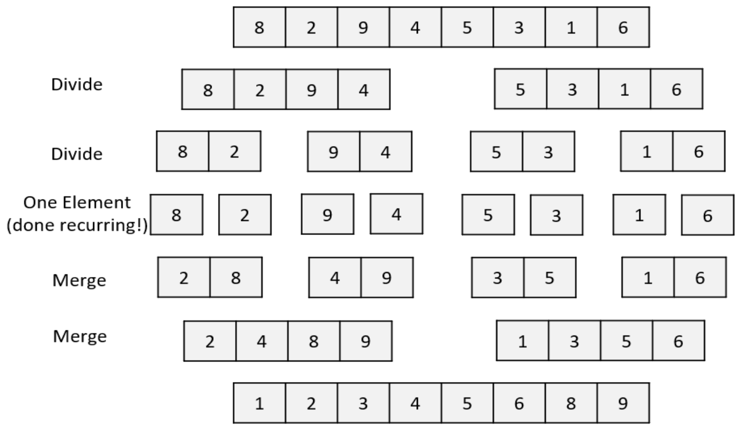
Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- Recall this image from last lecture, describing MergeSort:



- How many times is `mergeSort()` invoked for:
  - this 8-element array?
  - an *n*-element array? Assume that *n* is a power of 2 (ie, $n = 2^k$ for some *k*)

- *Bonus*: How many ways can you order {a, b, c}?

Announcements

- Quiz 1 grades released; we'll let regrade requests "cool off" before attending to them

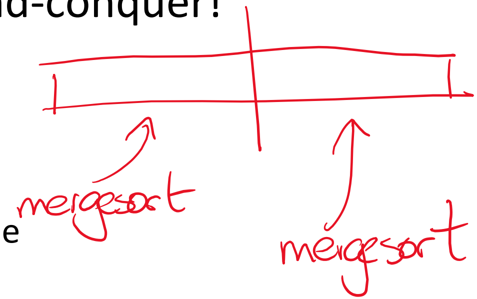
- P2 CP2 due *next week* (sorry for the confusion!)

Lecture Outline

- Comparison-based Sorting
  - **Review**
  - Fanciest algorithm using Divide-and-Conquer: QuickSort
  - External Sorting
  - Theoretical lower bound

Sorting with Divide and Conquer

- Two great sorting methods are divide-and-conquer!
  - MergeSort:
    - Sort the left half of the elements (recursively)
    - Sort the right half of the elements (recursively)
    - Merge the two sorted halves into a sorted whole



  - QuickSort:
    - Pick a "pivot" element
    - Partition elements into those *less-than* pivot and those *greater-than* pivot
    - Sort the *less-than* elements (recursively)
    - Sort the *greater-than* the elements (recursively)
    - All done! Answer is [*sorted-less-than*] [*pivot*] [*sorted-greater-than*]

QuickSort vs MergeSort (1 of 2)

Different algorithms, same problem

- MergeSort:
  - Execution
    - Does its work "on the way up"
      - i.e., in the merge, after the recursive call returns
    - Uses its auxiliary space very effectively:
      - Works well on linked lists
      - Linear merges minimize disk accesses
  - Time: always O(n log n)

- QuickSort:
  - Execution:
    - Does its work "on the way down"
      - i.e., in the partition, before the recursive call
    - Doesn't need auxiliary space
  - Runtime: O(n log n) in best and randomized cases ☺
    - But $O(n^2)$ worst-case ☹

Demo: https://docs.google.com/presentation/d/1h-gS13kKWSKd_5gt2FPXLYigFY4jf5rBkNFl3qZzRRw/present

QuickSort vs MergeSort (2 of 2)

- Asymptotic Runtime:
  - QuickSort is O(n log n) in best and randomized cases, but O(n^2) worst-case
  - MergeSort is always O(n log n)

- Constants Matter!
  - QuickSort does fewer copies and more comparisons, so it depends on the relative cost of these two operations
  - Typically, cost of copies is higher so QuickSort really *is* the “quickest”

Lecture Outline

- Comparison-based Sorting
  - Review
  - **Fanciest algorithm using Divide-and-Conquer: QuickSort**
  - External Sorting
  - Theoretical lower bound

QuickSort Steps

1. Pick the pivot value(s)
   - Any choice is correct; data will end up sorted
   - For efficiency, these value(s) ought to approximate the median

2. Partition all the values into:
   a. The values less than the pivot(s)
   b. The pivot(s)
   c. The values greater than the pivot(s)
   d. .. In linear time? In-place? Stably?

3. Recursively QuickSort(A) and QuickSort(C)

TA-DA!

QuickSort Steps

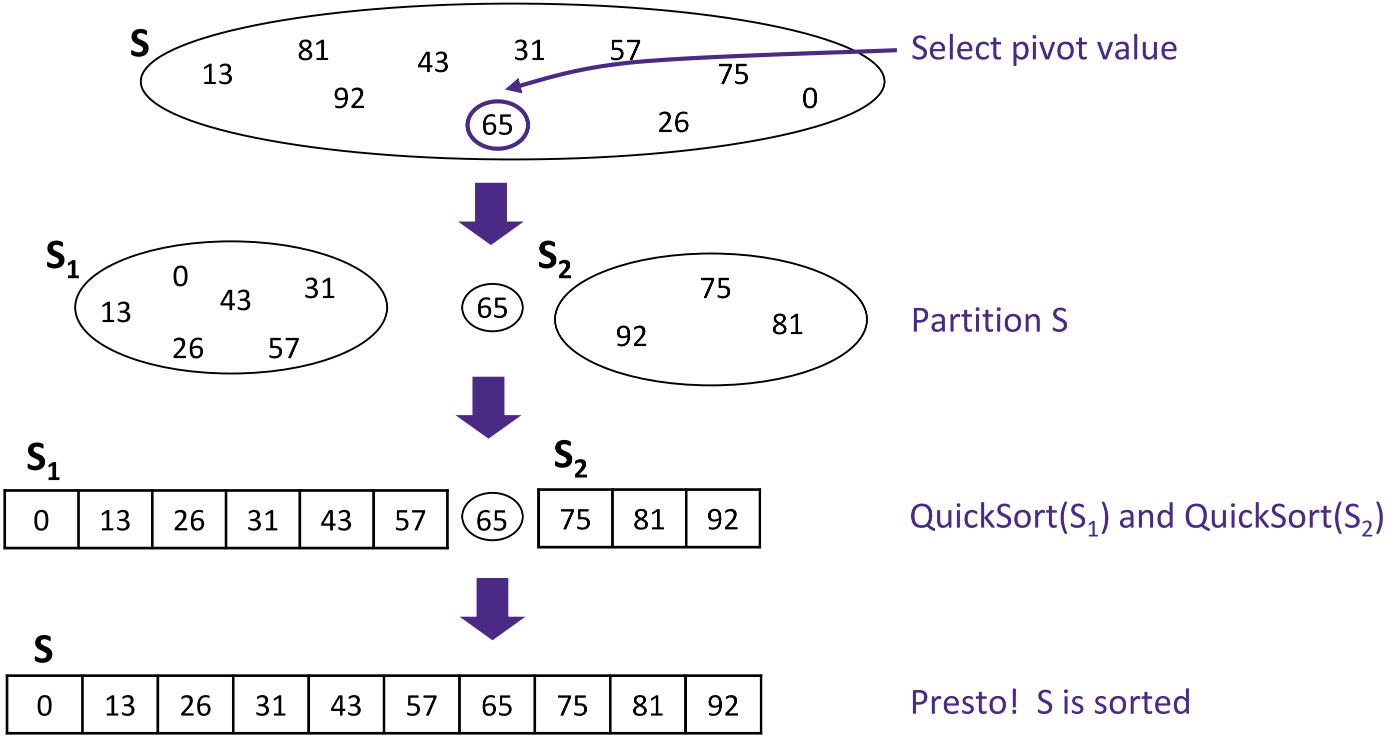
1. Pick the pivot value(s)
   - Any choice is correct; data will end up sorted
   - For efficiency, these value(s) ought to approximate the median

2. Partition all the values into:
   a. The values less than the pivot(s)
   b. The pivot(s)
   c. The values greater than the pivot(s)
   d. … In linear time? In-place? Stably?

3. **Recursively QuickSort(A) and QuickSort(C)**

TA-DA!

QuickSort Intuition: Set Partitioning

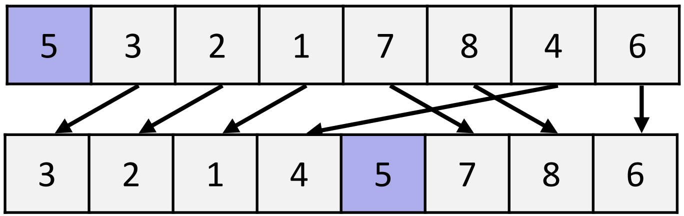


[Weiss]

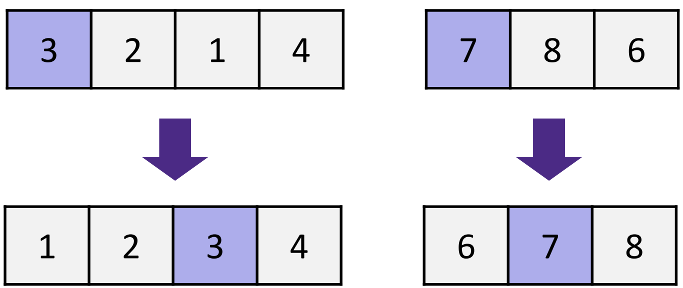
Recursive Call (1 of 3)

Note: for the remainder of this section, our pivot-selection algorithm is "first item in the subarray"

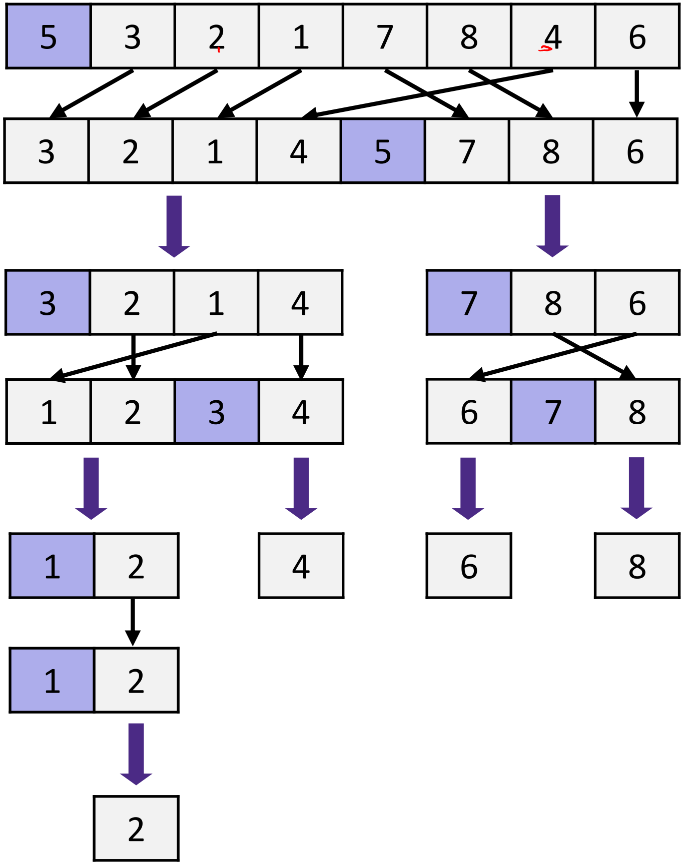
- After partitioning on 5:
  - 5 is in its "correct place" (ie, where it'd be if the array were sorted)



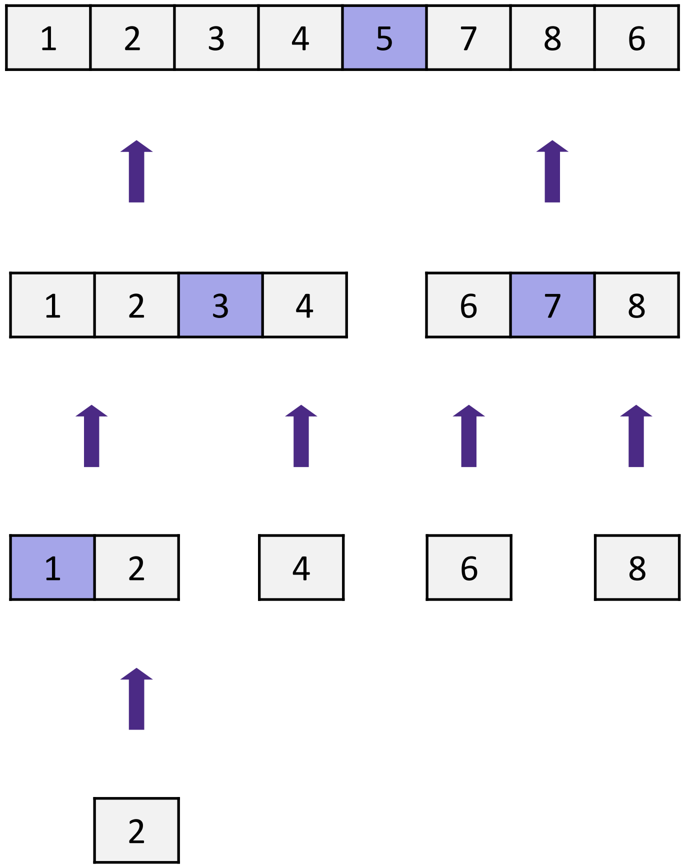
  - Can now sort two halves separately (eg, through recursive use of partitioning)



Recursive Call (2 of 3)



Recursive Call (3 of 3)



QuickSort Steps

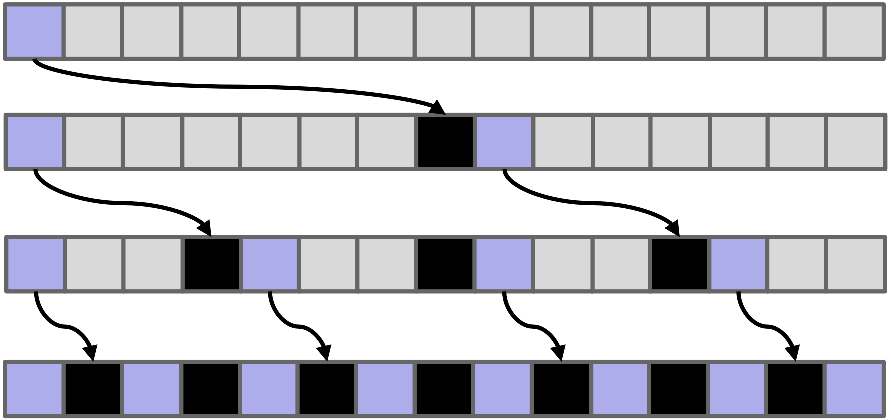
1. **Pick the pivot value(s)**
   - **Any choice is correct; data will end up sorted**
   - **For efficiency, these value(s) ought to approximate the median**

2. Partition all the values into:
   a. The values less than the pivot(s)
   b. The pivot(s)
   c. The values greater than the pivot(s)
   d. … In linear time? In-place? Stably?

3. Recursively QuickSort(A) and QuickSort(C)

✨ *TA-DA!* ✨

Pivot Selection: Pivot is the Median



Only size 1 problems remain, so we're done.

partition

$T(0) = T(1) = c_1$

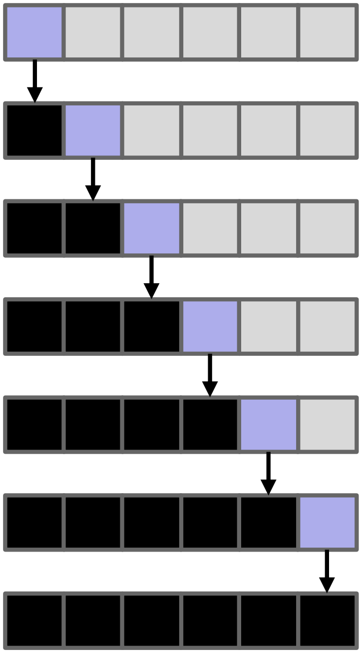
$T(n) = 2T(n/2) + c_2\, n$

(partition is linear-time)

Same recurrence as MergeSort:

$O(n \log n)$

Pivot Selection: Pivot is the Min/Max



$T(0) = T(1) = c_1$

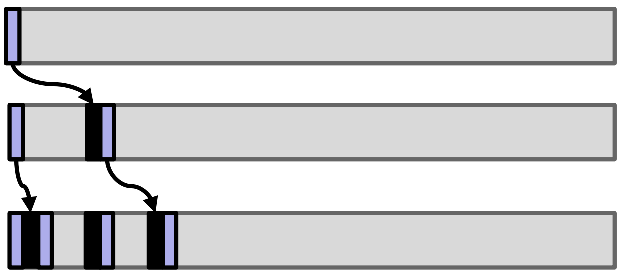
$T(n) = T(n\text{-}1) + c_2 n$

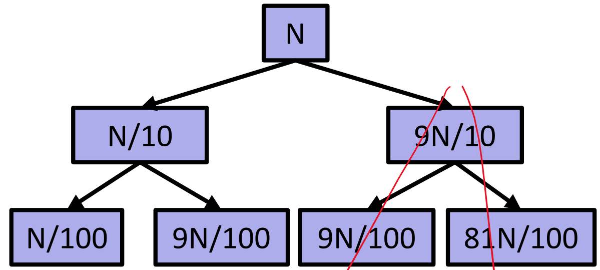
still partitioning

Basically same recurrence as SelectionSort: $O(n^2)$

Pivot Selection: Pivot is Random

- Suppose pivot always ends up *at least 10% from either edge*





- Work at each level: O(N) and Runtime is O(NH)
  - Height is approximately $\log_{10/9} N = O(\log N)$
- Runtime: O(N log N)
  - See proof in text

Pivot Selection Dictates Runtime!

- If pivot lands "somewhere good", Quicksort is Θ(N log N)
- However, the very rare $\Theta(N^2)$ cases do happen in practice
  - **Bad ordering**: Array already in (almost-)sorted order and pivot is first or last index
  - **Bad elements**: Array with all duplicates
- Three philosophies for avoiding worst-case behavior:
  1. **Randomness**: pick a random pivot; shuffle before sorting
     - Elegant, but (pseudo)random number generation can be slow
  2. **Smarter Pivot Selection**: calculate or approximate the median
     - Median-of-3: median of `arr[lo]`, `arr[hi-1]`, `arr[(hi+lo)/2]`
  3. **Introspection**: switch to safer sort if recursion goes too deep

Avoiding Worst-Case Pivots

- Example worst-cases:
  - **Bad ordering**: Array already in (almost-)sorted order and pivot is first or last index
  - **Bad elements**: Array with all duplicates

- Three philosophies for avoiding worst-case behavior:
  1. **Randomness**: pick a random pivot; shuffle before sorting
     - Elegant, but (pseudo)random number generation can be slow
  2. **Smarter Pivot Selection**: calculate or approximate the median
     - Median-of-3: median of `arr[lo]`, `arr[hi-1]`, `arr[(hi+lo)/2]`
  3. **Introspection**: switch to safer sort if recursion goes too deep
     - … what algorithm might be safer in the presence of badly-ordered elements?

QuickSort Steps

1. Pick the pivot value(s)
   - Any choice is correct; data will end up sorted
   - For efficiency, these value(s) ought to approximate the median

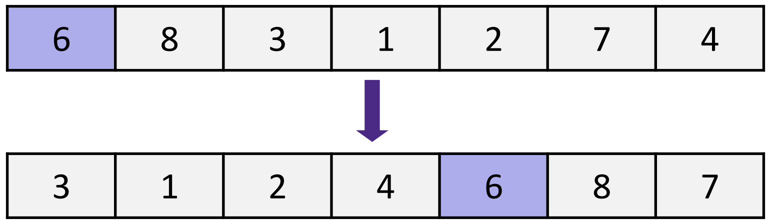
2. **Partition all the values into:**
   a. **The values less than the pivot(s)**
   b. **The pivot(s)**
   c. **The values greater than the pivot(s)**
   d. **… In linear time? In-place? Stably?**

3. Recursively QuickSort(A) and QuickSort(C)

TA-DA!

Partitioning: Problem Statement

- Given an array of elements and the 0th value as the pivot, write pseudocode that partitions the array



- Constraints:
  - Must complete in O(N log N) time, but ideally Θ(N)
  - Must use O(N) space, but ideally Θ(1)
  - May use any data structure (eg, BSTs, stacks/queues, etc)
  - Ideally, preserves the elements’ relative ordering (“stable”)

- Conceptually simple, but hardest part to code up correctly!

Partitioning: Option 1: Three-Pass

- Overview:
  - Copy “less than”s, then copy pivot(s), finally copy “greater-than”s
  - Demo: https://docs.google.com/presentation/d/16pOLboxhtJlaDxF7iRT5XcltDKmwab_wbvjZ4wPmJYk/edit
- Stable! ☺
- Constants aren’t great; very slow ☹

Partitioning: Option 2: Hoare Partitioning (1 of 2)

- As published in Hoare's original QuickSort paper!

- Intuition:
  - L loves small items (i.e., <pivot) and R loves large items (i.e., >pivot)
  - Walk towards eachother, swapping anything they don't like

- Algorithm:
  1. Swap pivot with **`arr[lo]`** ("move it out of the way")
  2. Start **`i`** at **`lo`**, and **`j`** at **`hi-1`**
  3. Move **`j`** leftward until we hit value **`<pivot`** ("belongs on left")
  4. Move **`i`** rightward until we hit value **`>pivot`** ("belongs on right")
  5. Swap **`arr[i]`** and **`arr[j]`**
  6. When they meet, swap **`arr[lo]`** and **`arr[i]`** ("put pivot in correct place")

```
while (i < j)
  if (arr[j] > pivot) j--;
  else if (arr[i] <= pivot) i++;
  else swap(arr[i], arr[j])
```

Partitioning: Option 2: Hoare Partitioning (2 of 2)

- Unstable ☹
- Good constants: single-pass and in-place ☺
- Demo: https://docs.google.com/presentation/d/1zmoLw5stDFxRLYSrYJP4BzExZZJWkLLHQhYIOBUy70o/edit

gradescope

gradescope.com/courses/256241

- Partition the following array using Hoare's partitioning algorithm
  - The pivot, 5, has already been moved to the front
  - Sort only by the numbers (eg, 1); the extra letter (eg, 1a) is to help you determine stability

5	1a	6	9	3	7	2	4a	4b	1b

Swap pivot with arr[lo] ("move it out of the way")
Start i at lo+1, and j at hi-1
  Move j rightward until we hit value <pivot ("belongs on left")
  Move i leftward until we hit value >pivot ("belongs on right")
  Swap arr[i] and arr[j]
When they meet, swap arr[0] and arr[i] ("put pivot in correct place")

```
while (i < j)
  if (arr[j] > pivot) j--;
  else if (arr[i] <= pivot) i++;
  else swap(arr[i], arr[j])
```

Partitioning: Option 3: Three-Way

- Pick *two* pivots
  - Same intuition as median-of-three: it's hard to pick multiple bad pivots simultaneously

- Like Hoare Partitioning, use two pointers walking to the middle
  - But split array into three pieces, not two
  - Good constants: single-pass and in-place; $\log_3 N$ vs $\log_2 N$ ☺
  - Still an unstable sort ☹

- Used in Java's Arrays.sort(), Python's unstable sort, etc
  - Basically the de-facto partition algorithm circa 2020

QuickSort Steps

1. Pick the pivot value(s)
   - Any choice is correct; data will end up sorted
   - For efficiency, these value(s) ought to approximate the median

2. Partition all the values into:
   a. The values less than the pivot(s)
   b. The pivot(s)
   c. The values greater than the pivot(s)
   d. … In linear time? In-place? Stably?

3. Recursively QuickSort(A) and QuickSort(C)

✨TA-DA!✨

QuickSort: End-to-end Example (1 of 3)

1. Pick pivot (we'll use median-of-3)

8	1	4	9	0	3	5	2	7	6

2. Partition (we'll use Hoare Partitioning)
   - Move pivot to the beginning position

6	1	4	9	0	3	5	2	7	8

   - Let **`lo`** = 1 and **`hi`** = 9; loop until we find "swappable" values

6	1	4	9	0	3	5	2	7	8

6	1	4	9	0	3	5	2	7	8

6	1	4	9	0	3	5	2	7	8

QuickSort: End-to-end Example (2 of 3)

- Swap `lo` = 3 and `hi` = 7

6	1	4	2	0	3	5	9	7	8

- Keep looping

6	1	4	2	0	3	5	9	7	8

6	1	4	2	0	3	5	9	7	8

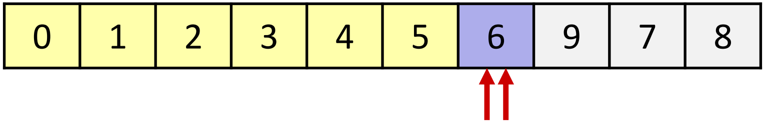
6	1	4	2	0	3	5	9	7	8

- Done! Swap pivot into position

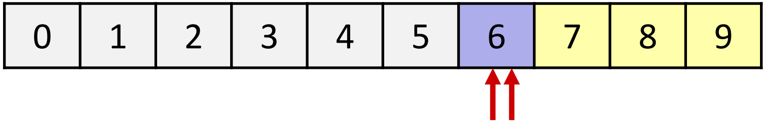
5	1	4	2	0	3	6	9	7	8

QuickSort: End-to-end Example (3 of 3)

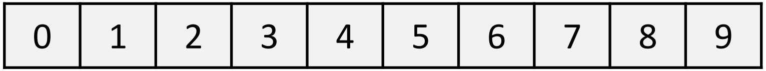
3. Recursively sort left (0 to **`lo-1`** = 5)



4. Recursively sort right (**`hi`**+1 = 7 to **`arr.length`**)



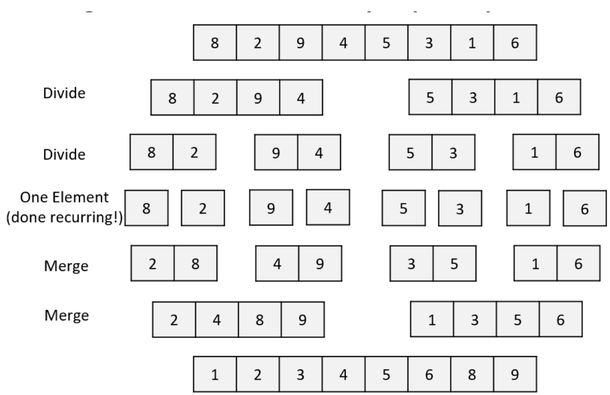
5. Sorted!



QuickSort Optimization: Cutoffs (1 of 2)

- For small *n*, recursion tends to cost more than a quadratic sort
  - Remember: asymptotic complexity applies to large *n*
  - Recursive calls add overhead (which “isn’t worth it” for small *n*)

- Recursive calls for small *n* are the most common (“leaf calls”)
  - Calls for small *n* are the vast majority of the recursive calls!



QuickSort Optimization: Cutoffs (2 of 2)

- So, switch algorithms for subproblems below a **cutoff** size
  - Eg, Java 12 uses InsertionSort for primitive types when $n < 47$

```
void quickSort(int[] arr, int lo, int hi) {
  if (hi – lo < CUTOFF)
    insertionSort(arr, lo, hi);
  else
    …
}
```

- Switching algorithms after a cutoff is a common technique!
  - E.g. *parallel* algorithms switch to *sequential* after a certain cutoff
  - E.g. MergeSort also uses cutoffs to switch to InsertionSort

- Does not affect asymptotic complexity, just the constants

Lecture Outline

- Comparison-based Sorting
  - Review
  - Fanciest algorithm using Divide-and-Conquer: QuickSort
  - **External Sorting**
  - Theoretical lower bound

Sorting Linked Lists

- We defined the sorting problem as over an array, but sometimes you want to sort linked lists
- One approach:
  - Convert to array: $O(n)$
  - Sort: $O(n \log n)$
  - Convert back to list: $O(n)$
- Or: MergeSort works very nicely on linked lists directly
  - HeapSort and QuickSort does not
  - InsertionSort and SelectionSort do, but they're slower

Sorting Massive Data: External Sorting (1 of 2)

- Need sorting algorithms that minimize disk access?
  - QuickSort and HeapSort jump all over the array; their random disk accesses don't utilize special locality effectively
  - MergeSort scans linearly through arrays, leading to (relatively) efficient sequential disk access

- MergeSort is the algorithm of choice for external sorting!

Sorting Massive Data: External Sorting (2 of 2)

- Can we make MergeSort even more efficient? Yes!
  - Load one page of elements into memory, sort, store this “run” on disk/tape
  - Use the `merge()` routine to merge successively larger runs
  - Repeat until you have only one run

- MergeSort can leverage multiple disks; see Weiss

Comparison-based Sorts: Summary

	Best-Case Time	Worst-Case Time	Randomized Case	In-Place?	Stable?	Notes
InsertionSort	Θ(N)	$\Theta(N^2)$	$\Theta(N^2)$	Yes	Yes	Fastest for small or partially-sorted input
SelectionSort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N^2)$	Yes	No	
In-Place HeapSort	Θ(N)	Θ(N log N)	Θ(N log N)	Yes	No	Slow in practice
MergeSort	Θ(N log N)	Θ(N log N)	Θ(N log N)	No	Yes	Fastest stable sort
QuickSort *(1st-element pivot + 3-pass partition)*	Θ(N log N)	$\Theta(N^2)$	Θ(N log N)	No	Yes	>=2x slower than MergeSort
QuickSort *(Median-of-three pivot + Hoare partition + cutoffs)*	Ω(N)	$O(N^2)$	Θ(N log N)	Yes	No	Fastest comparison sort

Lecture Outline

- Comparison-based Sorting
  - Review
  - Fanciest algorithm using Divide-and-Conquer: QuickSort
  - External Sorting
  - **Theoretical lower bound**

A Different View of Sorting

- Assume we have *n* elements, none are equal (no duplicates)

- **Sorting** is like finding one specific ordering out of all possible ordering of elements!

- How many ***permutations*** (possible orderings) of the elements?
  - Example, *n*=3

a < b < c	a < c < b	b < a < c	b < c < a	c < a < b	c < b < a

  - *n* choices for least element, then *n*-1 for next, then *n*-2 for next, …
  - *n*(*n*-1)(*n*-2)…(2)(1) = ***n! possible orderings***

Describing Every Comparison Sort

- A different way of thinking about **sorting** is that it "finds" the right answer among the n! possible answers
  - Starts "knowing nothing"; "anything is possible"
  - Each comparison gains information, eliminating some possibilities
    - Comparisons are binary: `a < b` or `b < a`
    - Intuition: each comparison eliminates (at most) half of remaining possibilities
  - In the end, narrows down to a single possibility

- Where are the comparisons in:
  - InsertionSort?
  - QuickSort?

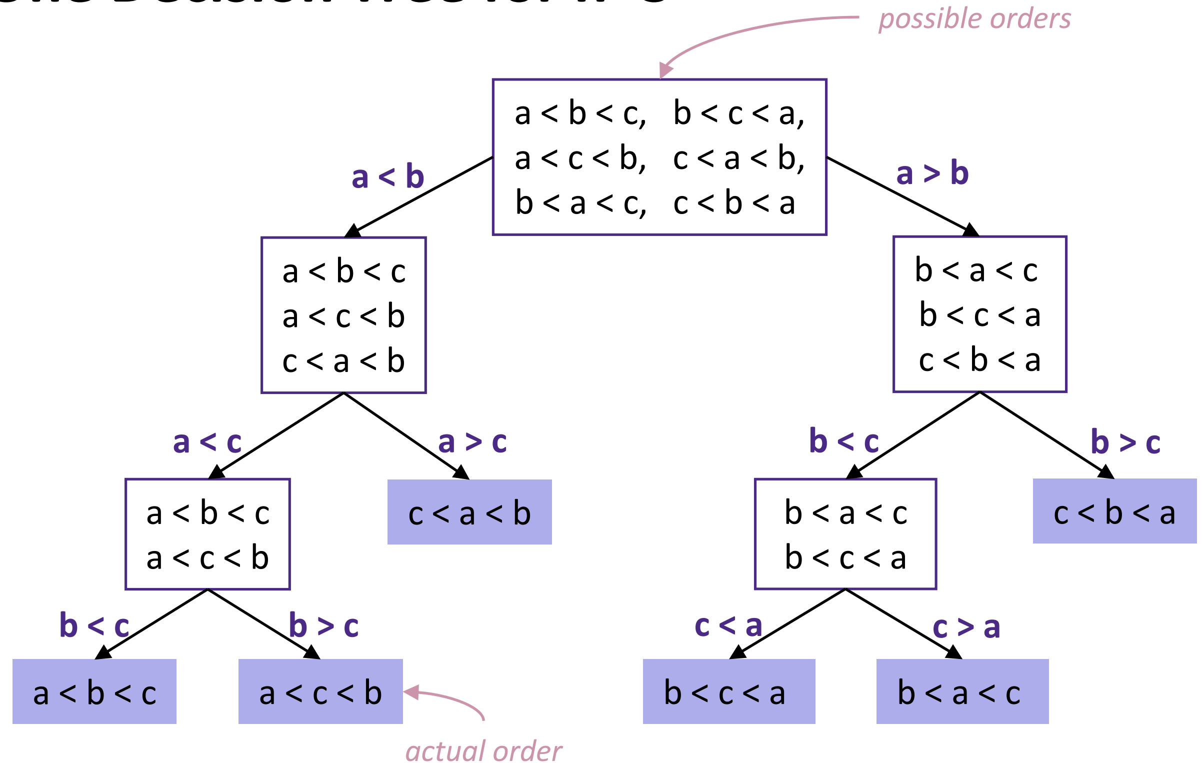
Representing Comparison Sorts

- Let’s represent these binary comparisons as a binary tree!

- Called a *Decision Tree*
  - Nodes contain “set of remaining possible orderings”
  - The root contains all possible orderings; anything is possible
  - The leaves contain exactly one specific ordering
  - Edges are “answers from a comparison”

We are not actually building the tree; it’s what our *proof* uses to represent “the most the algorithm could know so far”

One Decision Tree for n=3



- The leaves contain all the possible orderings of a, b, c
- A different algorithm would lead to a different tree