Comparison Sorts (cont.)

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Announcements

- P2 Checkpoint due Tuesday!
- Q2 grades released soon
- Lecture questions: pollev.com/cse332

Lecture Outline

- **Comparison Sorting with Divide-and-Conquer**
  - Fancier Algorithms: MergeSort
  - Fancier Algorithms: QuickSort

Technique: Divide and Conquer

- Very important technique in algorithm design!
  1. Divide problem into smaller parts
  2. Solve the parts independently
     - Recursion
     - Or potentially parallelism!
  3. Combine solution of parts to produce overall solution

- Example:
  - Sort each half of the array, then combine together
  - To sort each half, split into halves …

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
    - Divide elements into those less-than pivot and those greater-than pivot
    - Sort the two divisions (recursively on each)
    - Answer is [*sorted-less-than*] then [*pivot*] then [*sorted-greater-than*]

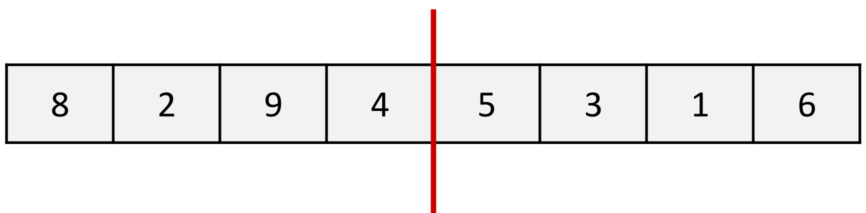
Lecture Outline

- Comparison Sorting with Divide-and-Conquer
  - **Fancier Algorithms: MergeSort**
  - Fancier Algorithms: QuickSort

MergeSort

- To sort array from position **`lo`** to position **`hi`**:
  - If range is 1 element long, it's sorted! (Base case)
  - Else, split into two halves:
    - Sort from **`lo`** to **`(hi+lo)/2`**
    - Sort from **`(hi+lo)/2`** to **`hi`**
    - Merge the two halves together

- Merging takes two sorted parts and sorts everything
  - $O(n)$ time but requires $O(n)$ auxiliary space…



MergeSort: Merging Example (1 of 10)

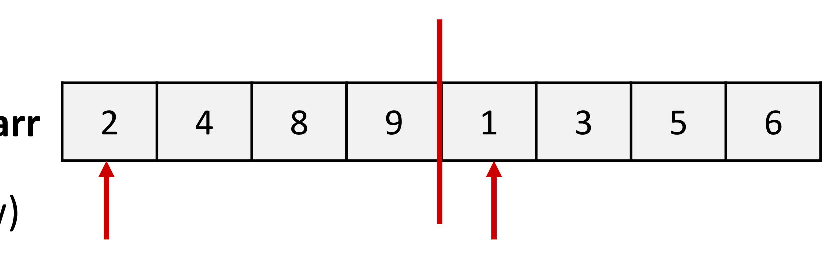
- Start with:

arr

8	2	9	4	5	3	1	6

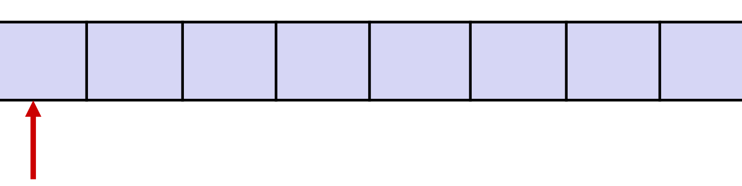
- Return from left and right recursion
  - (pretend it works for now)

arr



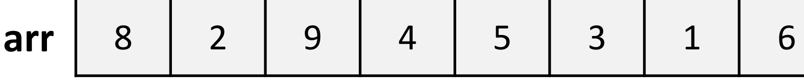
- Merge
  - Use 3 cursors and an extra array
  - When done, copy the extra array back to the original

aux

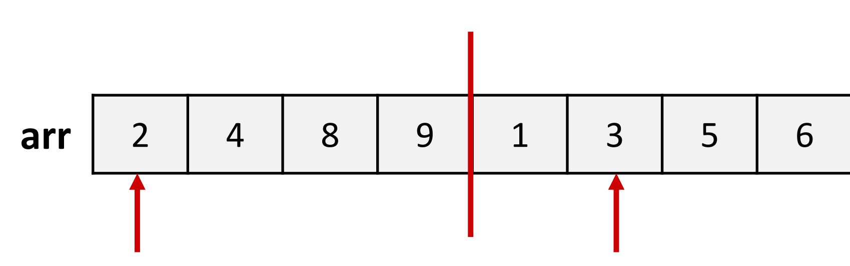


MergeSort: Merging Example (2 of 10)

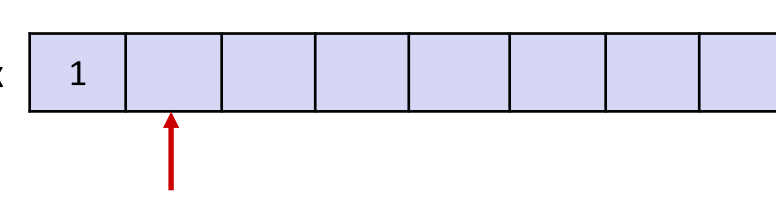
- Start with:



- Return from left and right recursion
  - (not magic ☺)



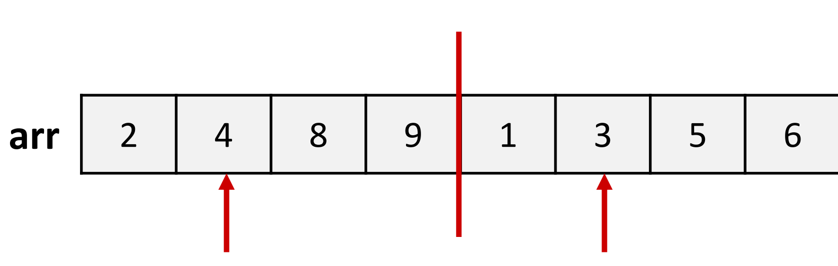
- Merge
  - Use 3 cursors and an extra array
  - When done, copy the extra array back to the original



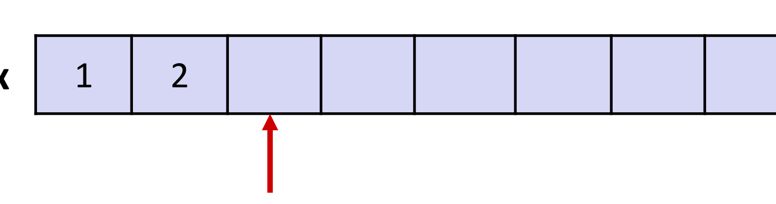
MergeSort: Merging Example (3 of 10)

- Start with:

- Return from left and right recursion
  - (not magic ☺)



- Merge
  - Use 3 cursors and an extra array
  - When done, copy the extra array back to the original

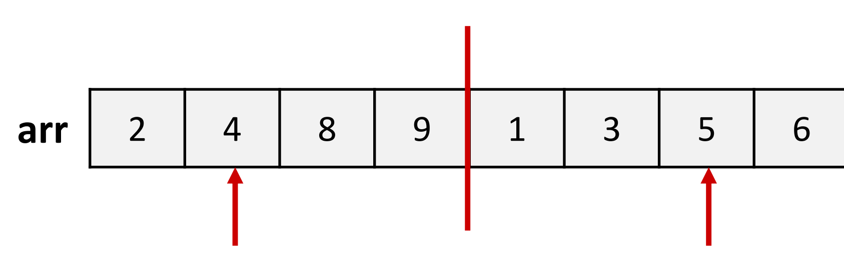


MergeSort: Merging Example (4 of 10)

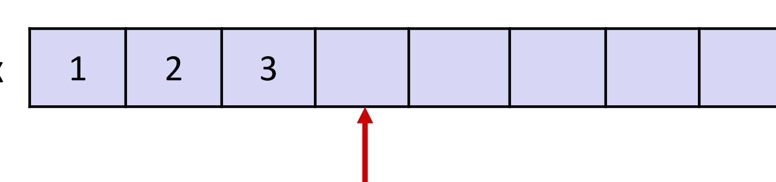
- Start with:

arr	8	2	9	4	5	3	1	6

- Return from left and right recursion
  - (not magic ☺)



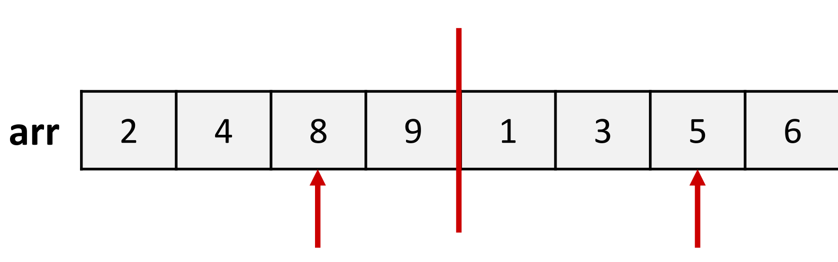
- Merge
  - Use 3 cursors and an extra array
  - When done, copy the extra array back to the original



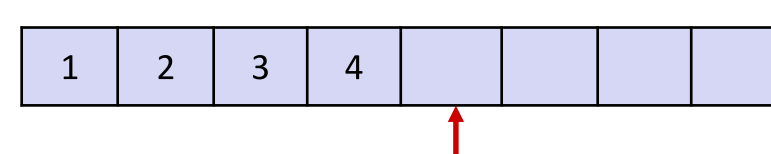
MergeSort: Merging Example (5 of 10)

- Start with:

- Return from left and right recursion
  - (not magic ☺)

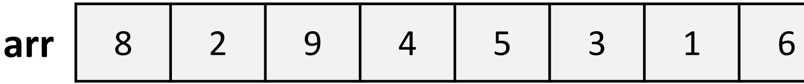


- Merge
  - Use 3 cursors and an extra array
  - When done, copy the extra array back to the original

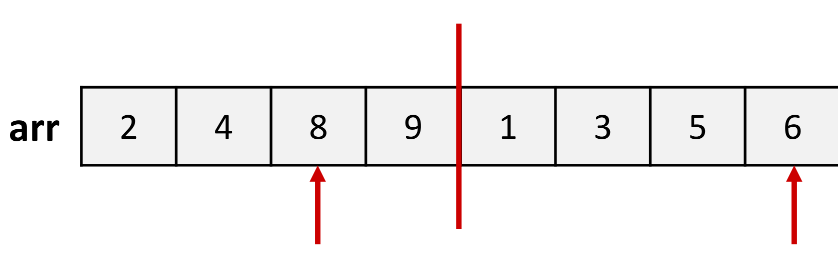


MergeSort: Merging Example (6 of 10)

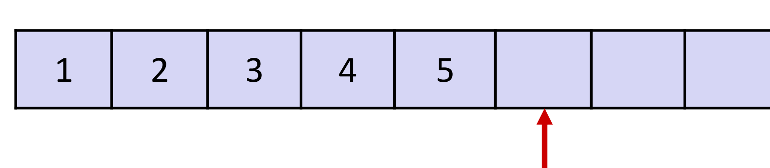
- Start with:



- Return from left and right recursion
  - (not magic ☺)



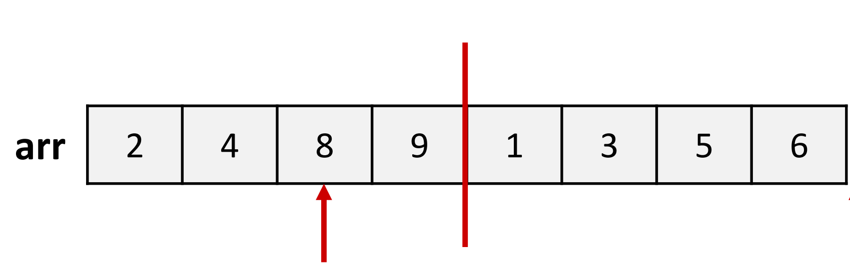
- Merge
  - Use 3 cursors and an extra array
  - When done, copy the extra array back to the original



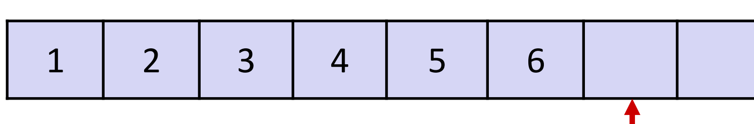
MergeSort: Merging Example (7 of 10)

- Start with:

- Return from left and right recursion
  - (not magic ☺)

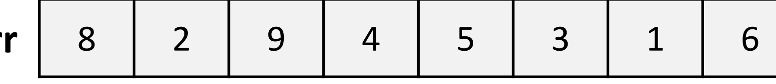


- Merge
  - Use 3 cursors and an extra array
  - When done, copy the extra array back to the original

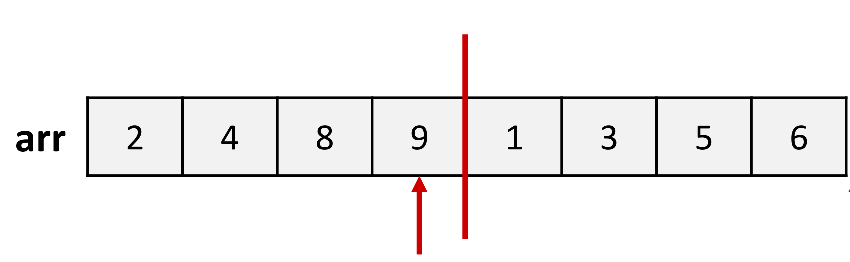


MergeSort: Merging Example (8 of 10)

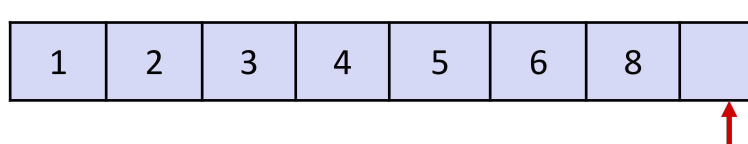
- Start with:



- Return from left and right recursion
  - (not magic ☺)

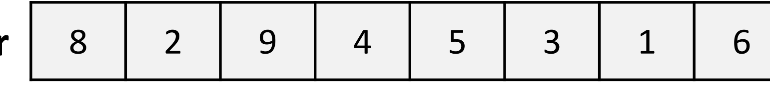


- Merge
  - Use 3 cursors and an extra array
  - When done, copy the extra array back to the original

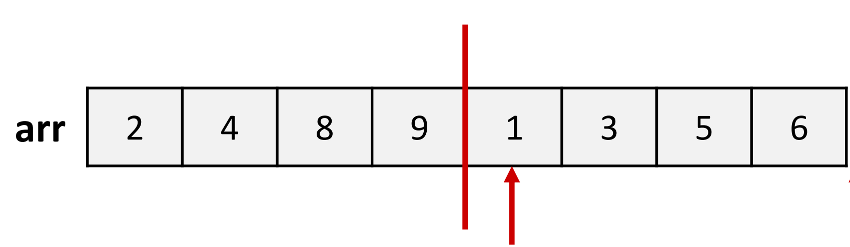


MergeSort: Merging Example (9 of 10)

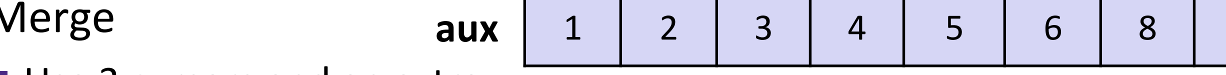
- Start with:



- Return from left and right recursion
  - (not magic ☺)

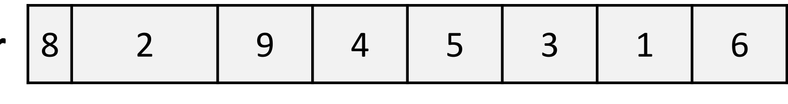


- Merge
  - Use 3 cursors and an extra array
  - When done, copy the extra array back to the original

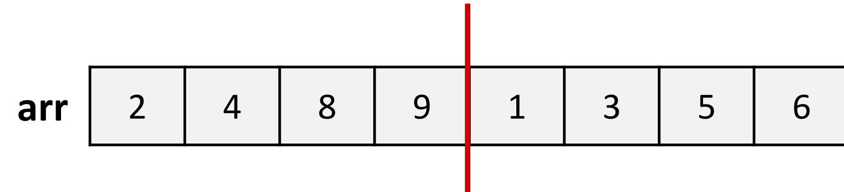


MergeSort: Merging Example (10 of 10)

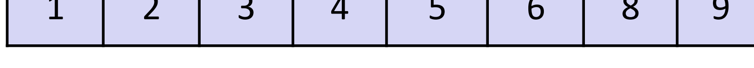
- Start with:



- Return from left and right recursion
  - (not magic ☺)



- Merge
  - Use 3 cursors and an extra array
  - When done, copy the extra array back to the original



MergeSort: Recursion Example (1 of 3)

8	2	9	4	5	3	1	6

Divide

8	2	9	4

5	3	1	6

Divide

8	2

9	4

5	3

1	6

One Element (done recurring!)

8 2 9 4 5 3 1 6

MergeSort: Recursion Example (2 of 3)

8	2	9	4	5	3	1	6

Divide

8	2	9	4

5	3	1	6

Divide

[8 | 2] [9 | 4] [5 | 3] [1 | 6]

One Element (done recurring!)

[8] [2] [9] [4] [5] [3] [1] [6]

Merge

[2 | 8] [4 | 9] [3 | 5] [1 | 6]

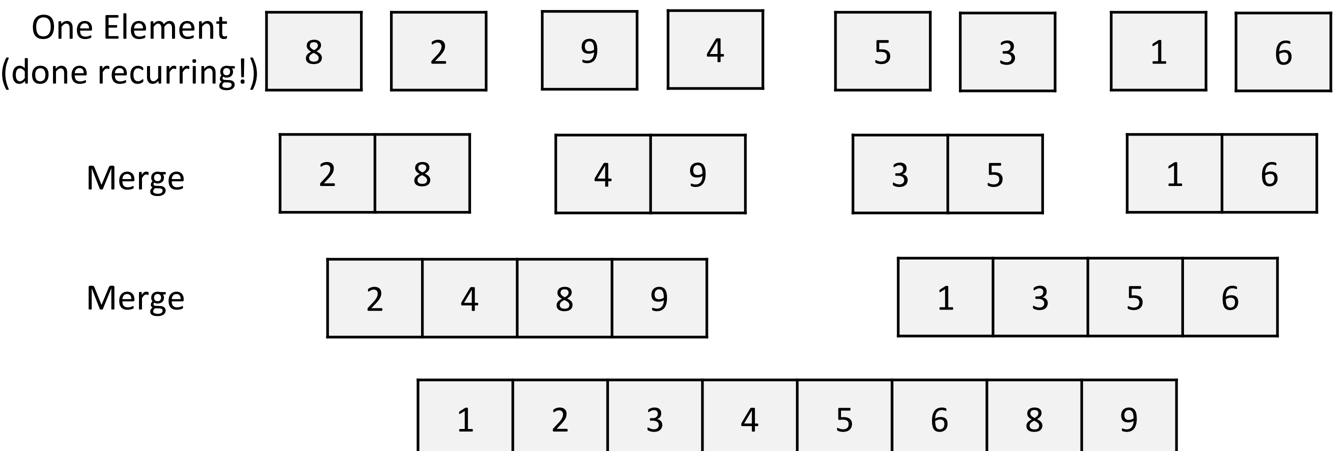
Merge

2	4	8	9

1	3	5	6

1	2	3	4	5	6	8	9

MergeSort: Recursion Example (3 of 3)

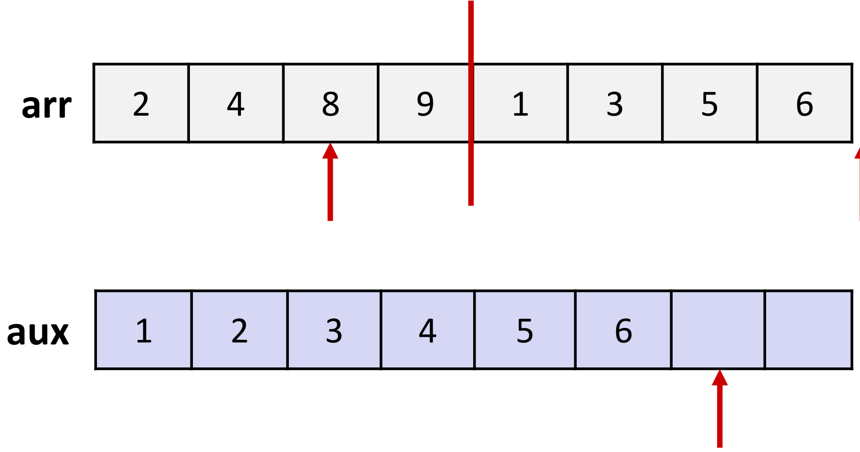


When a recursive call ends, its sub-arrays are *each in order*; we just need to merge them *in order together*

(and we already know how to merge!)

Optimizations: Reducing "Dregs Copies" (1 of 2)

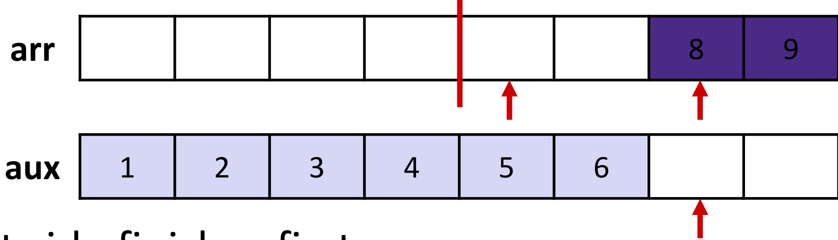
- Remember the final steps of our merge example?



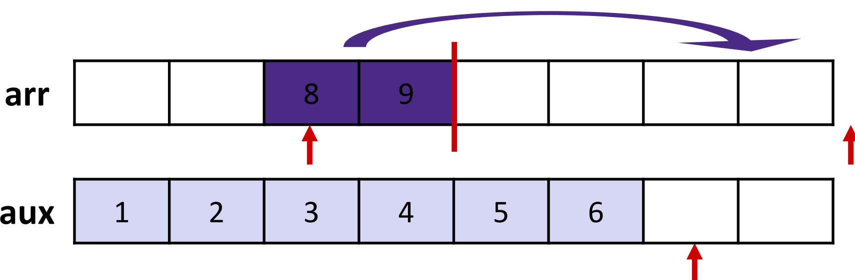
- It's wasteful to copy 8 & 9 to the auxiliary array, and then immediately copy them back into the original array!

Optimizations: Reducing "Dregs Copies" (2 of 2)

- If left side finishes first:
  - Stop the merge, and copy the auxiliary array back to the original



- If right side finishes first:
  - Stop the merge, and copy the dregs directly into right side
  - Then copy auxiliary array back to the original

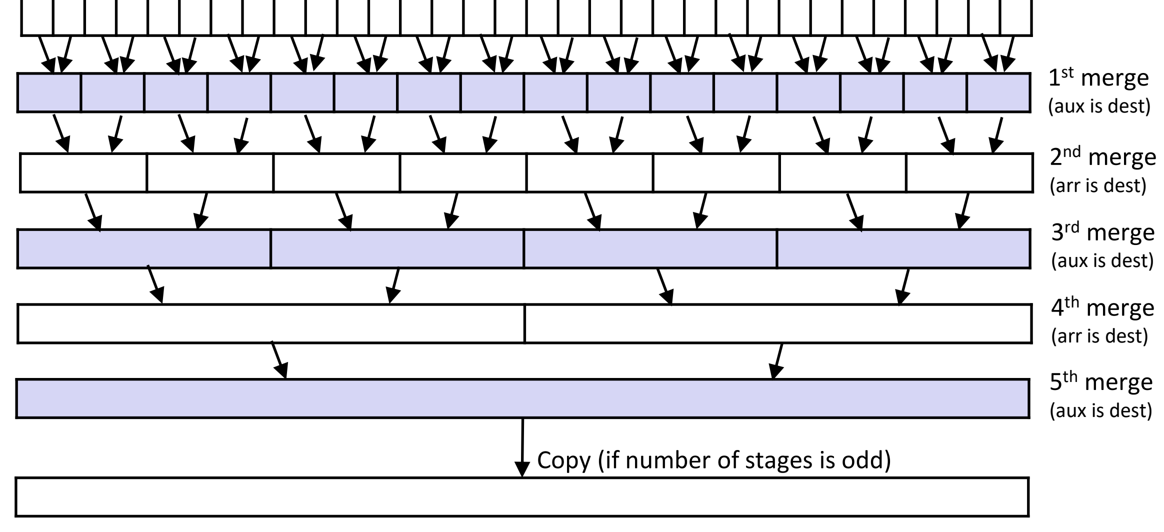


Optimizations: Reducing Temp Arrays (1 of 2)

- Simplest / worst approach:
  - Every divide: allocate two new auxiliary arrays of size `(hi-lo)/2`
  - Every merge: allocate another auxiliary array
- Better:
  - Allocate a single auxiliary array of size `n` at beginning to use throughout
  - Reuse "slices" of size `(hi-lo)/2` within that array at every merge
- Best (but a little tricky):
  - Don't copy back! At 2nd, 4th, 6th, … merges, use the original array as the auxiliary array; at odd-numbered merges, vice-versa
  - If the number of stages is odd, need one final copy at end

Optimizations: Reducing Temp Arrays (2 of 2)

1. Recur down to sub-arrays of size 1 (no copies)
2. As we return from the recursion, switch off arrays



3. Arguably easier to code up without recursion at all

MergeSort: Characteristics

- We've discussed arrays, but you may need to sort linked lists
  - One approach:
    - Convert to array: O(n)
    - Sort: O(n log n)
    - Convert back to list: O(n)
  - Alternatively: MergeSort works well on linked lists
    - HeapSort and QuickSort do not ☹
    - InsertionSort and SelectionSort can work, but they're slower

- MergeSort is the best choice for external sorting
  - Linear merges minimize disk accesses

MergeSort: Runtime Analysis (1 of 3)

- MergeSort sorts *n* elements by:
  - Return if immediately if *n*=1
  - Doing 2 subproblems of size *n*/2 + then an *O*(*n*) merge otherwise

- Runtime expression?
  - $T(1) = c_1$
  - $T(n) = 2T(n/2) + c_2 n$

MergeSort: Runtime Analysis (2 of 3)

$T(1) = c_1$

$T(n) = 2T(n/2) + c_2 n$ First expansion

$= 2(2T(n/4) + c_2 n/2) + c_2 n$ Second expansion

$= 4T(n/4) + 2c_2 n$

$= 4(2T(n/8) + c_2 n/4) + 2c_2 n$ Third expansion

$= 8T(n/8) + 3c_2 n$

$= 2^k T(n/2^k) + kc_2 n$ k^{th} expansion

Let $k = \log n$

Then $T(n) = 2^k T(n/2^k) + kc_2 n$

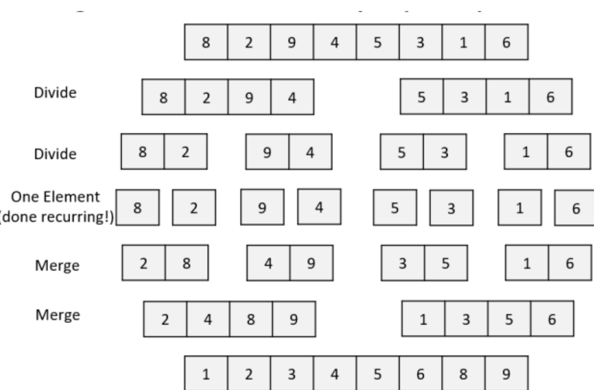
$= 2^{\log n} T(1) + \log n \, c_2 n$

$= c_1 n + c_2 n \log n$

$= O(n \log n)$

MergeSort: Runtime Analysis (3 of 3)

- More intuitively, this recurrence comes up often enough you should "just know" it's $O(n \log n)$
- MergeSort is relatively easy to intuit
  - Best, worst, and "average" all have the same runtime
  - The recursion "tree" will have $\log n$ height and at each level we do a *total* amount of merging equal to n



Lecture Outline

- Comparison Sorting with Divide-and-Conquer
  - **Fancier Algorithms: MergeSort**
  - Fancier Algorithms: QuickSort

QuickSort vs MergeSort (1 of 2)

- Execution:
  - QuickSort does its work "on the way down"
    - i.e., in the partition, before the recursive call
  - MergeSort does its work "on the way up"
    - i.e., in the merge, after the recursive call returns
  - QuickSort does not need auxiliary space
  - MergeSort uses its auxiliary space very effectively:
    - Works well on linked lists
    - Linear merges minimize disk accesses
- Runtime:
  - QuickSort is O(n log n) in best and "average" cases ☺
    - But $O(n^2)$ worst-case ☹
  - MergeSort is always O(n log n)

QuickSort vs MergeSort (2 of 2)

- Runtime:
  - QuickSort is O(n log n) in best and "average" cases ☺
    - But $O(n^2)$ worst-case ☹
  - MergeSort is always O(n log n)

- Bottom line:
  - QuickSort does fewer copies and more comparisons, so it depends on the relative cost of these two operations!

Quicksort Outline

1. Pick the pivot value(s)
   - Hopefully these value(s) 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)

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

4. ✨TADA✨! The answer is “as simple as A, B, C”

Quicksort Outline

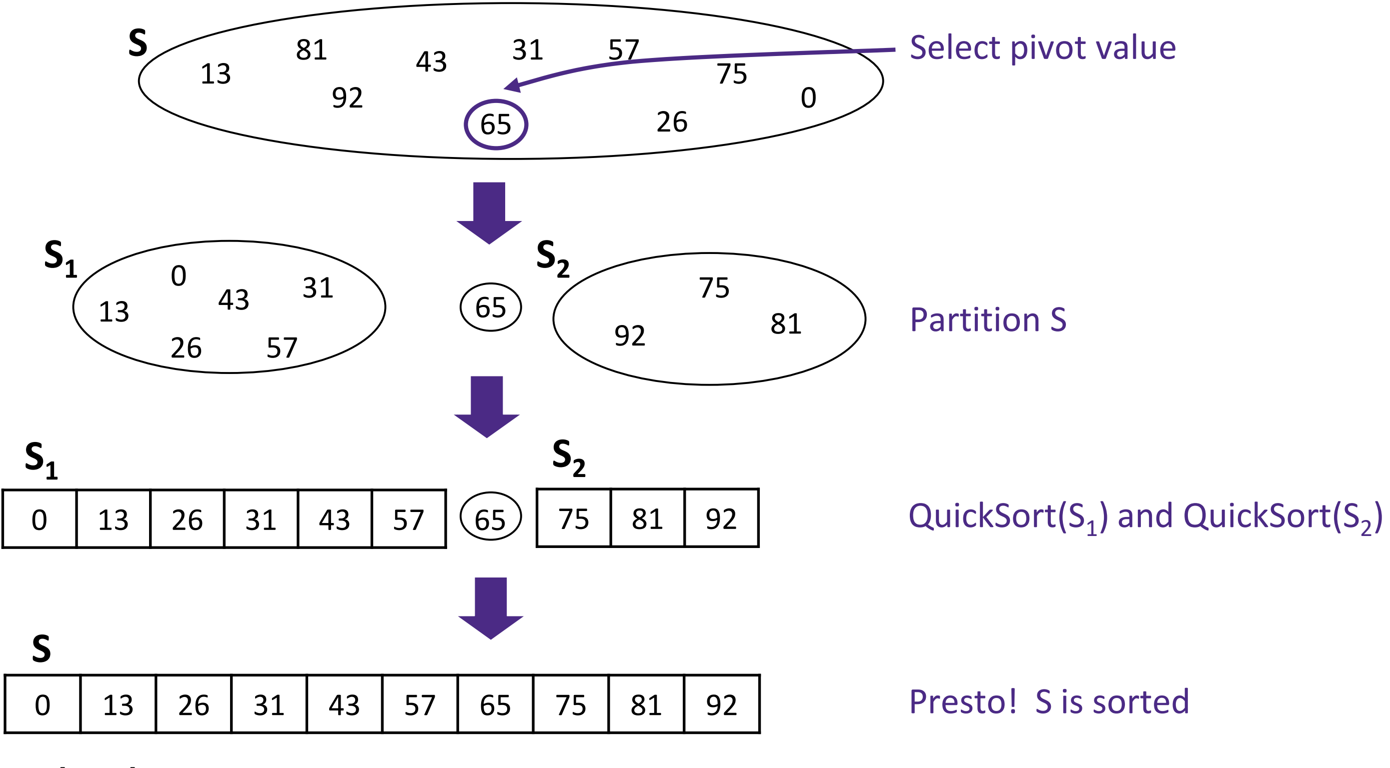
1. Pick the pivot value(s)
   - Hopefully these value(s) approximate the median

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)

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

4. ✨TADA✨! The answer is “as simple as A, B, C”

Quicksort Intuition: Set Partitioning

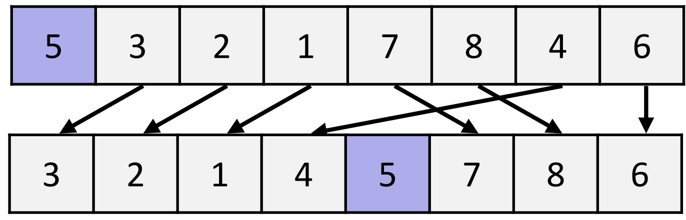


[Weiss]

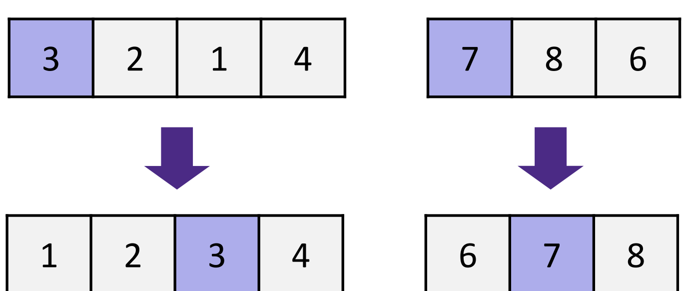
Step #3: Recursion (1 of 3)

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

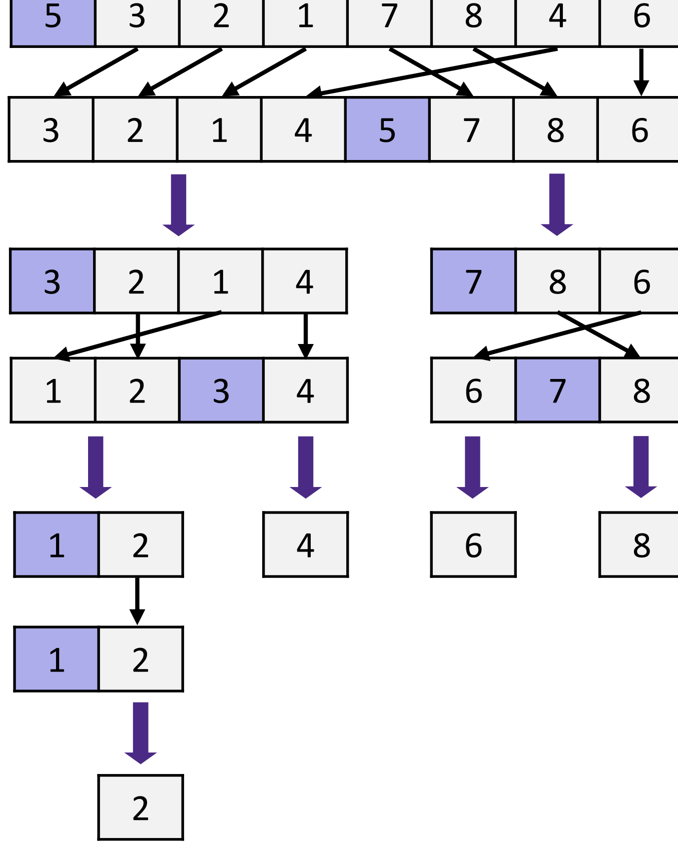
- After partitioning on 5:
  - 5 is in its "correct place" (ie, where it'd be if the array were sorted)



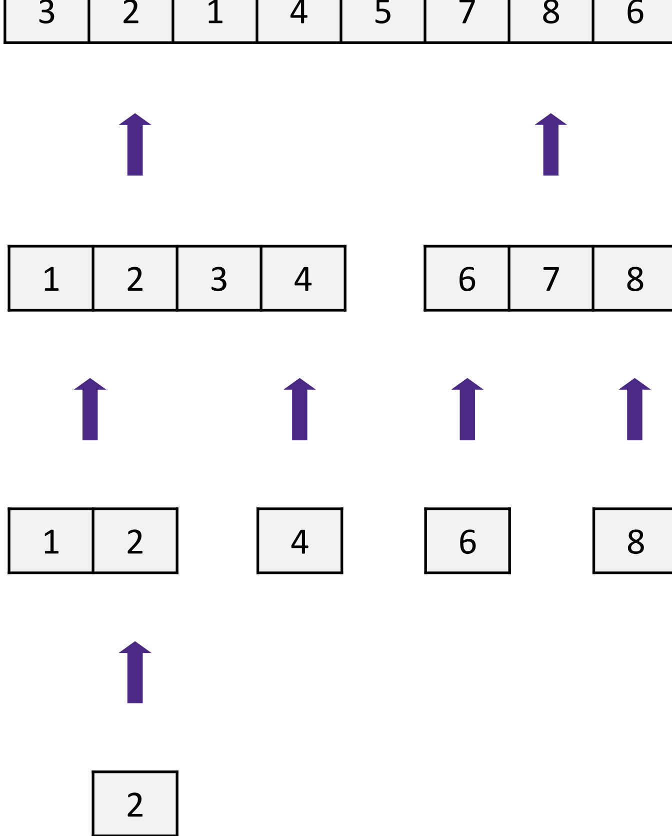
  - Can now sort two halves separately (eg, through recursive use of partitioning)



Step #3: Recursion (2 of 3)



Step #3: Recursion (3 of 3)



Quicksort Outline

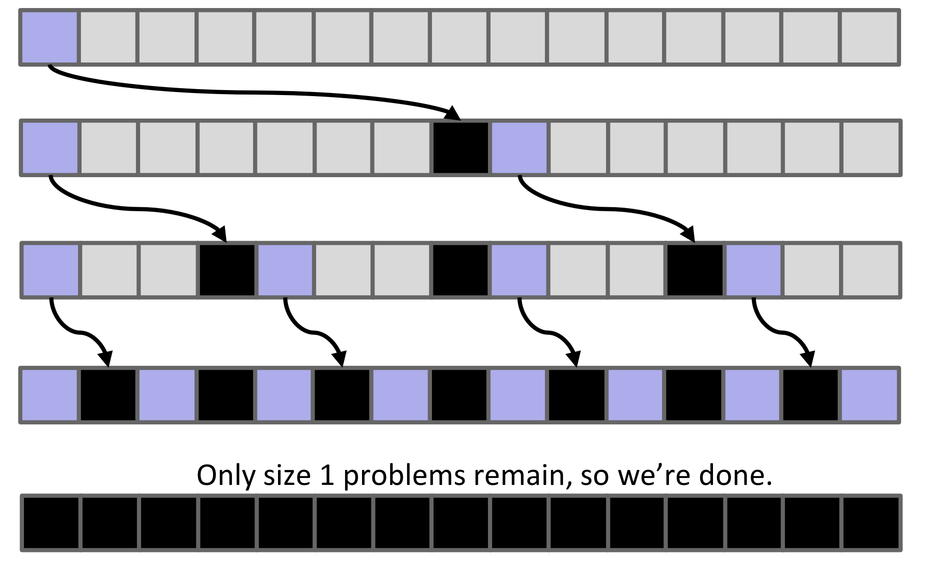
1. **Pick the pivot value(s)**
   - **Any choice is correct; data will end up sorted**
   - **But hopefully these value(s) approximate the median**

2. Partition all the values into:
   A. The value less than the pivot(s)
   B. The pivot(s)
   C. The value greater than the pivot(s)

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

4. ✨TADA✨! The answer is “as simple as A, B, C”

Step #1 (Best Case): Pivot is the Median



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

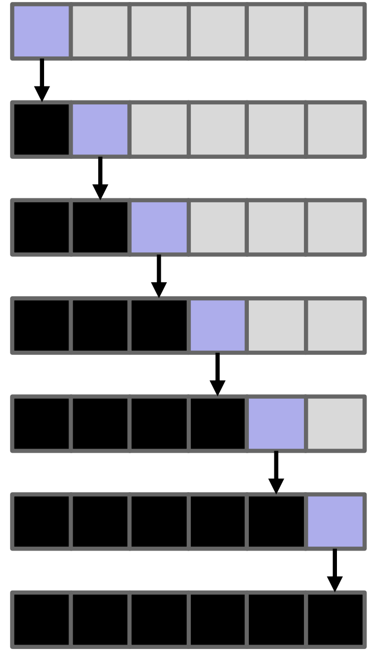
$T(n) = 2T(n/2) + c_2\, n$

(partition is linear-time)

Same recurrence as MergeSort:

$O(n \textbf{ log } n)$

Step #1 (Worst Case): Pivot is the Min/Max



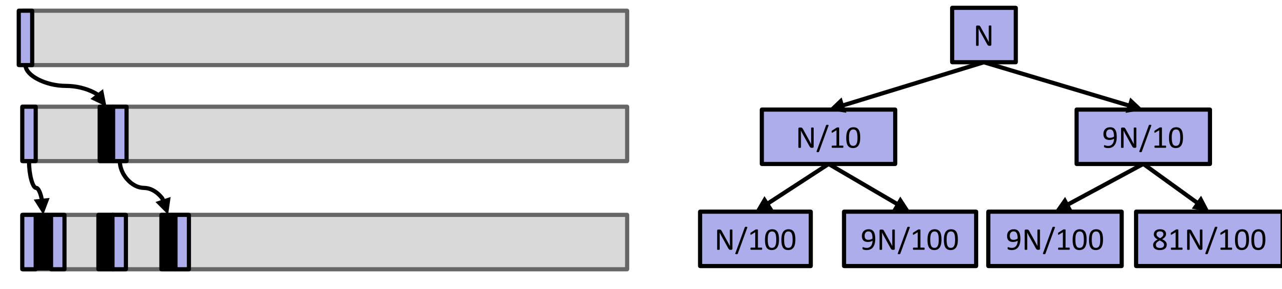
$T(0) = T(1) = c_1$

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

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

Step #1 (Randomized Case)

- 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

tl;dr: 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

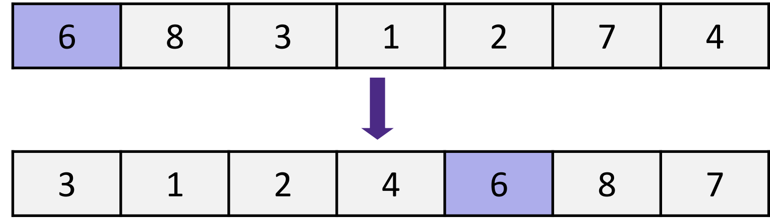
- Three philosophies for avoiding worst-case behavior:
  1. **Randomness**: pick a random pivot; shuffle before sorting
     - Elegant, but (pseudo)random number generation can be slow
  2. **Smarter Pivot Selection**: calculate or approximate the median
     - Median-of-3: median of `arr[lo]`, `arr[hi-1]`,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

Quicksort Outline

1. Pick the pivot value(s)
   - Hopefully these value(s) approximate the median
2. **Partition all the values**
   - **In linear time**
   - **In-place**
   - **Stably?**
3. Recursively QuickSort(A) and QuickSort(C)
4. ✨TADA✨! The answer is “as simple as A, B, C”

Step #2: 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+1`**, and **`j`** at **`hi-1`**
  3. Move **`j`** rightward until we hit value <pivot ("belongs on left")
  4. Move **`i`** leftward until we hit value >pivot ("belongs on right")
  5. Swap **`i`** and **`j`**
  6. 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 2: Hoare Partitioning (2 of 2)

- Unstable ☹

- Good constants: single-pass and in-place ☺

- Demo: https://docs.google.com/presentation/d/1DOnWS59PJOa-LaBfttPRseIpwLGefZkn450TMSSUiQY/pub?start=false&loop=false&delayms=3000&slide=id.g463de7561_042

Partitioning, Option 3: Three-Way

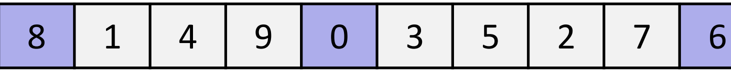
- Pick *two* pivots
  - Same intuition as median-of-three: it's hard to pick multiple bad pivots simultaneously

- 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 ☹

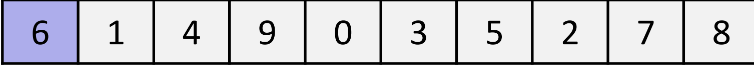
- Used in Java's Arrays.sort(), Python's unstable sort, etc
  - Basically the de-facto partition algorithm circa 2020

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

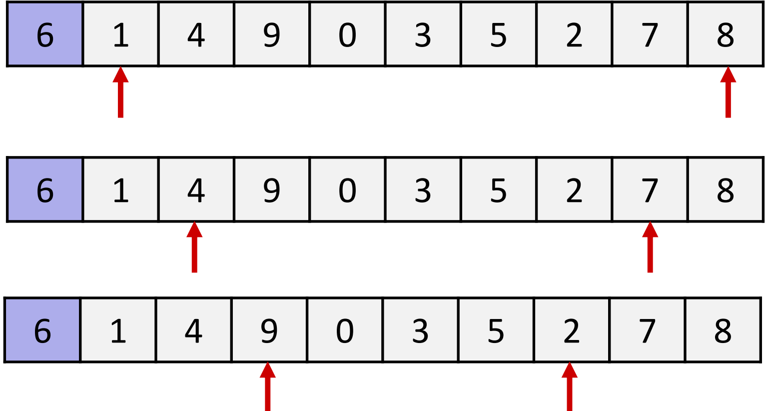
1. Pick pivot (we'll use median-of-3)



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

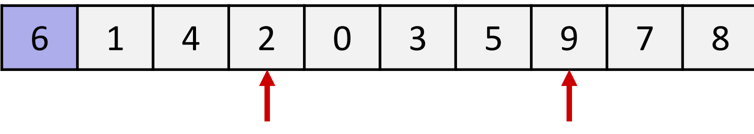


   - Let **lo** = 1 and **hi** = 9; loop until we find "swappable" values

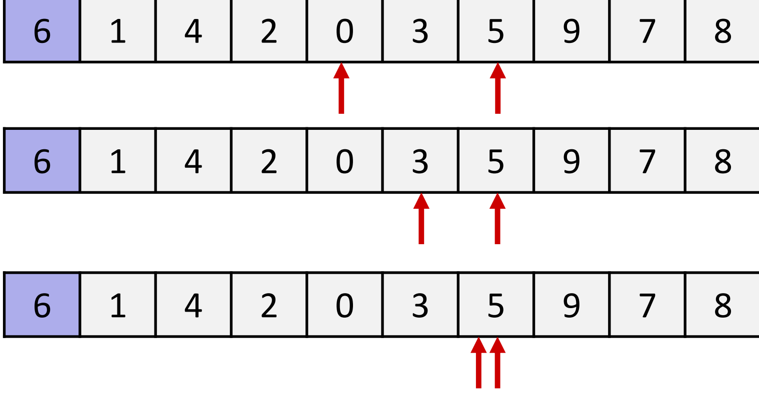


QuickSort: End-to-end Example (2 of 3)

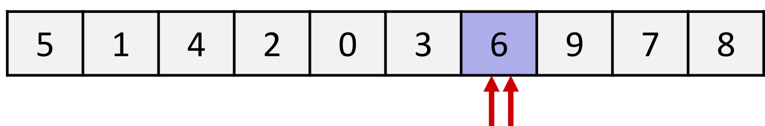
- Swap `lo` = 3 and `hi` = 7



- Keep looping

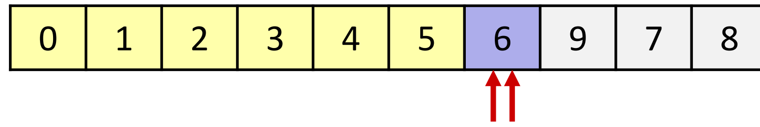


- Done! Swap pivot into position

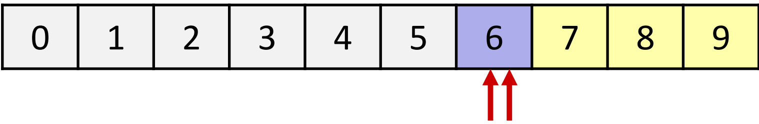


QuickSort: End-to-end Example (3 of 3)

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")
- So, switch algorithms for subproblems below a **cutoff** size
  - Reasonable rule of thumb: use InsertionSort for *n* < 10
  - 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
    …
}
```

QuickSort Optimization: Cutoffs (2 of 2)

- Notes
  - Switching algorithms after a cutoff is a common technique!
    - E.g. *parallel* algorithms switch to *sequential* algorithms after a certain cutoff
    - E.g. MergeSort also uses cutoffs to switch to InsertionSort
  - Does not affect asymptotic complexity, just the constants
    - Notice how this cuts out the vast majority of the recursive calls
    - If the QuickSort's recursive calls are a tree, we've trimmed out the bottom layers

Summary

	Best-Case Time	Worst-Case Time	Randomized Case	In-Place?	Stable?	Notes
In-Place InsertionSort	Θ(N)	$Θ(N^2)$	$Θ(N^2)$	Yes	Yes	Fastest for small or partially-sorted input
SelectionSort	$Θ(N^2)$	$Θ(N^2)$	$Θ(N^2)$	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)	$Θ(N^2)$	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