

Comparison Sorting Algorithms

CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements

- ❖ P2 checkpoint 2 tomorrow – google form will release this afternoon
 - QuickSort should be the only thing we haven't covered in lecture yet
- ❖ Exercises 7 & 8 out!
 - Ex7 Canvas Groups: Join group and post in *group discussion*
 - Reflection questions subject to change before Wednesday
- ❖ Midterm out Wednesday!
- ❖ P1 Grades will release

Lecture Outline

- ❖ **Intro to Sorting**
- ❖ Simple Sorts
 - InsertionSort
 - SelectionSort
- ❖ Fancier Sorts
 - HeapSort
 - “Data Structure Sorts”
- ❖ Divide & Conquer Sorts
 - MergeSort
 - QuickSort



gradescope.com/courses/275833

- ❖ When you play cards, how do you order them in your hand?
- ❖ Why do you think we learning about sorting in this class?

Introduction to Sorting (1 of 2)

❖ Stacks, queues, priority queues, and dictionaries/sets all provide one element at a time

↳ But often we want “all the items” in some order

- Alphabetical list of people
- Population list of countries
- Search engine results by relevance

❖ Different sorting algorithms have different asymptotic and constant-factor trade-offs

- Knowing one way to sort just isn't enough; no single “best sort”

→ **Sorting is an excellent case-study in making trade-offs!**

Introduction to Sorting (2 of 2)

❖ Preprocessing (e.g. sorting) data to make subsequent operations faster is a general technique in computing!

▪ Example: Sort the items so that you can:

→ Find the k^{th} largest in constant time for any k

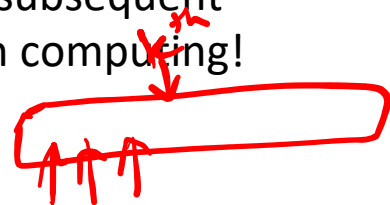
- ~~Perform binary search~~ to find an item in logarithmic time

▪ Whether preprocessing is beneficial depends on

- How often the items will change
- How many items there are

❖ Preprocessing's benefits depend on how often the items will change and how many items there are

→ **Sorting is an excellent case-study in making trade-offs!**



Comparison Sorting: Definition

❖ Problem: We have n comparable items in an array, and we want to rearrange them to be in increasing order

→ ❖ Input:

- An array A of (key, value) pairs
- A comparison function (consistent and total)
 - Given keys a & b , what is their relative ordering? $<$, $=$, $>$?
 - Ex: keys that implement Comparable or have a Comparator

❖ Output/Side-Effect:

- Reorganize the elements of A such that for any index i and j ,
if $i < j$ then $A[i] \leq A[j]$

→ ■ [Usually unspoken] **A** must have all the same items it started with

- Could also sort in reverse order, of course

Comparison Sort: Variations (1 of 2)

1. Maybe elements are in a linked list
 - Could convert to array and back in linear time, but some algorithms can still “work” on linked lists

unstable → 1, 2, 2, 4, 6
 1, 2, 2, 4, 6
 (1, 2) (2, 4) (4, 6)

→ Maybe if there are ties we should preserve the original ordering

- Sorts that do this naturally are called stable sorts.
3. Maybe we must not use more than $O(1)$ “auxiliary space”
 - These are called in-place sorts
 - Not allowed to allocate memory proportional to input (i.e., $O(n)$), but can allocate $O(1)$ # of variables
 - Work is done by swapping around in the array

Comparison Sort: Variations (2 of 2)

4. Maybe we can do more with elements than just compare
 - Comparison sorts assume a binary 'compare' operator
 - In special cases we can sometimes get faster algorithms

5. Maybe we have too many items to fit in memory
 - Use an **external sorting** algorithm

Big Picture of Comparison-Based Sorts

- ❖ Simple algorithms: $O(n^2)$
 - InsertionSort, SelectionSort
 - *BubbleSort, ShellSort*
- ❖ Fancier algorithms: $O(n \log n)$
 - HeapSort, MergeSort, QuickSort (randomized)
- ❖ Comparison-based sorting's lower bound: $\Omega(n \log n)$

Lecture Outline

- ❖ Intro to Sorting

- ❖ **Simple Sorts**
 - InsertionSort
 - SelectionSort

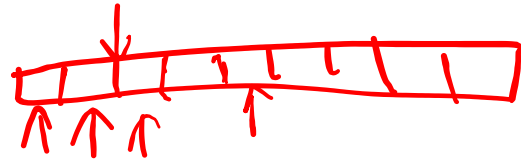
- ❖ Fancier Sorts
 - HeapSort
 - “Data Structure Sorts”

- ❖ Divide & Conquer Sorts
 - MergeSort
 - QuickSort

InsertionSort

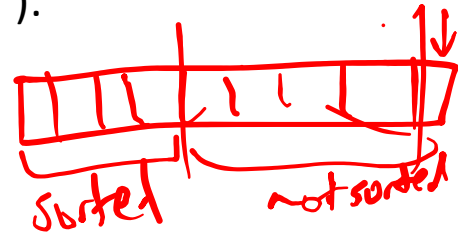
❖ Idea: At step k , insert the k^{th} element in the correct position

- Sort first two elements
- Now insert 3rd element in order
- ...



❖ Loop invariant (“when loop index is i ”):

- First i elements are in sorted order



❖ Time:

Best-case: $O(n)$ Worst-case: $O(n^2)$

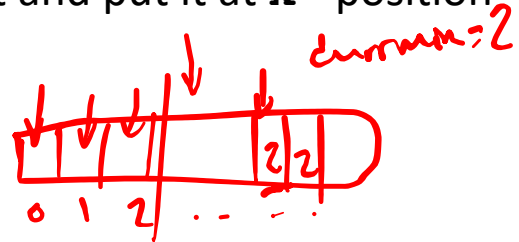
❖ Characteristics:

Stable: ✓ In-place: ✗

SelectionSort

❖ Idea: At step k , select the smallest elt and put it at k^{th} position

- Find smallest element, put it 1st
- Find next smallest element, put it 2nd
- ...



❖ Loop invariant (“when loop index is i ”):

- First i elements are the i smallest elements in sorted order

❖ Time:

Best-case: $O(n^2)$ Worst-case: $O(n^2)$

❖ Characteristics:

Stable: ✓ In-place: ✓

InsertionSort vs. SelectionSort (1 of 2)

Different algorithms, same problem

❖ InsertionSort

■ Loop invariant:

- First i elements are in sorted order

■ Characteristics:

- Stable: yes

■ Time:

- Worst-case: $O(n^2)$

→ “Average” case: $O(n^2)$

❖ SelectionSort

■ Loop invariant:

- First i elements are *the i smallest elements* in sorted order

■ Characteristics:

- Stable: no

■ Time:

- Worst-case: $O(n^2)$

→ “Average” case: $O(n^2)$

InsertionSort vs. SelectionSort (2 of 2)

- ❖ InsertionSort has better best-case complexity
 - Best case is when input is “mostly sorted”
- ❖ Different constants
 - InsertionSort may do well on small arrays (empirically: $N < \sim 15$)
 - Java’s built-in sort prefers InsertionSort for arrays < 47 items

$O(n^2)$

Lecture Outline

- ❖ Intro to Sorting
- ❖ Simple Sorts
 - InsertionSort
 - SelectionSort
- ❖ **Fancier Sorts**
 - HeapSort
 - “Data Structure Sorts”
- ❖ Divide & Conquer Sorts
 - MergeSort
 - QuickSort

Naïve HeapSort

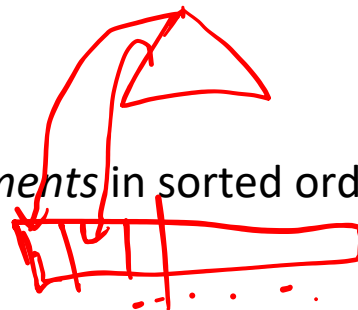
- ❖ Idea: Put everything in a **MIN** heap; successively deleteMin
 - add() all elements into heap – OR – better yet, use buildHeap
 - for($i=0$; $i < arr.length$; $i++$)
 $arr[i] =$ ~~deleteMin()~~; $n \log n$

- ❖ Loop invariant (“when loop index is i ”):
 - First i elements are *the i smallest elements* in sorted order

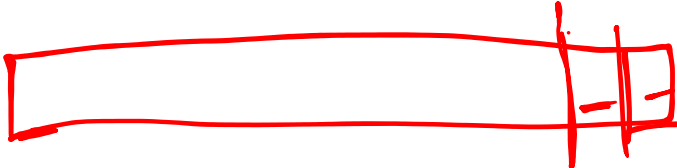
- ❖ Time: $O(n \log n)$

- ❖ Characteristics:

Stable: NO In-place: NO



In-place HeapSort

- ❖ Idea: Put everything in a **MAX** heap ; successively delete Max
 - insert each `arr[i]` –OR– better yet, use `buildHeap`
 - `for(i=0; i < arr.length; i++)`
`arr[arr.length - i] = deleteMax();`
- ❖ Loop invariant (“when loop index is **i**”): same as naïve version
- ❖ Time: $n \log n$

- ❖ Characteristics:
 Stable: ~~x~~ In-place: ✓

Aside: “AVLSort” and “DataStructureSort”

- ❖ We can also use a balanced tree to:
 - **add** each element: total time $O(n \log n)$
 - Do an in-order traversal $O(n)$
- ❖ But a balanced tree cannot be made in-place, and constants worse than HeapSort
 - Both are $O(n \log n)$ in worst, best, and average case
 - Neither sorts parallelizes well
- ❖ Don't even think about trying to sort with a hash table ...



- ❖ Why might I care about a sort being stable or in place? Would having these two qualities ever be worth the tradeoff of having a slower algorithm?

.

Lecture Outline

- ❖ Intro to Sorting

- ❖ Simple Sorts
 - InsertionSort
 - SelectionSort

- ❖ Fancier Sorts
 - HeapSort
 - “Data Structure Sorts”

- ❖ **Divide & Conquer Sorts**
 - MergeSort
 - 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

- ❖ Examples:
 - Sort each half of the array, then combine together
 - Split the array into “small part” and “big part”, then sort the parts

Sorting with Divide and Conquer

❖ Two great sorting methods are divide-and-conquer!

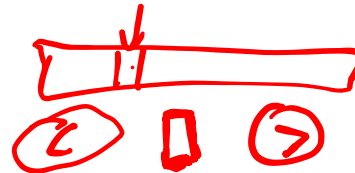
■ MergeSort:

- 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



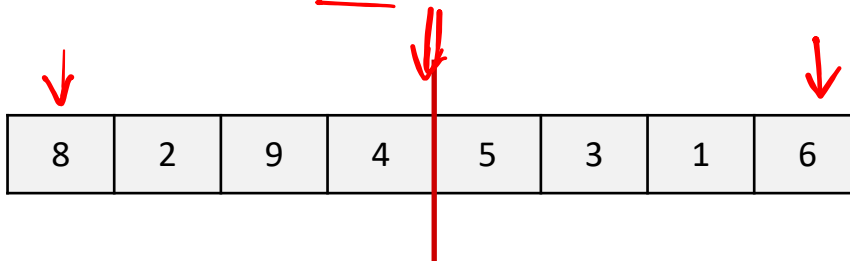
■ QuickSort:

- 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*]



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:
 - "Somehow" sort from lo to $(hi+lo)/2$
 - "Somehow" 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...



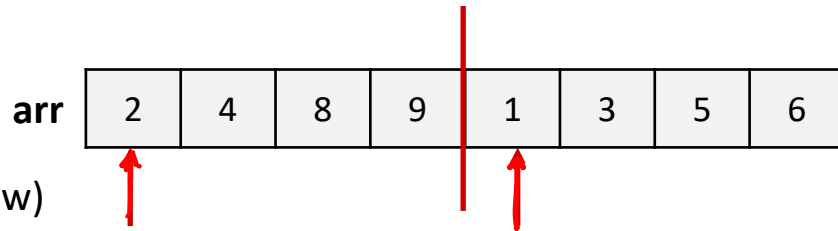
MergeSort: Merging Example (1 of 10)

❖ Start with:



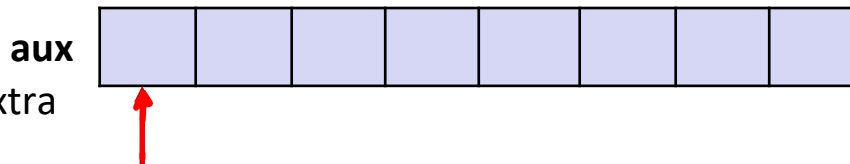
❖ Return from left and right recursion

- (pretend it works for now)



❖ Merge

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



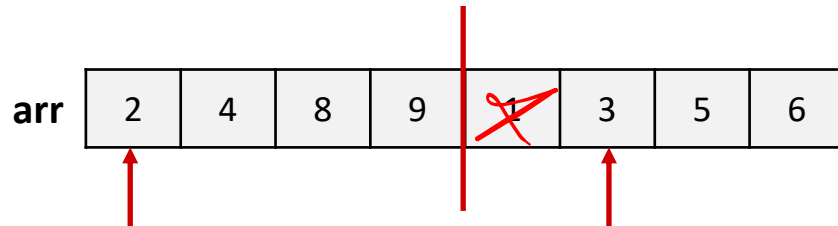
MergeSort: Merging Example (2 of 10)

❖ Start with:



❖ Return from left and right recursion

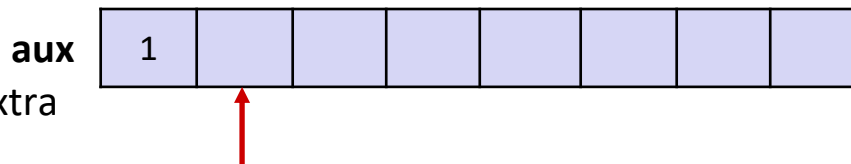
▪ (not magic 😊)



❖ Merge

▪ Use 3 cursors and an extra auxiliary array

▪ When done, copy the extra array back to the original



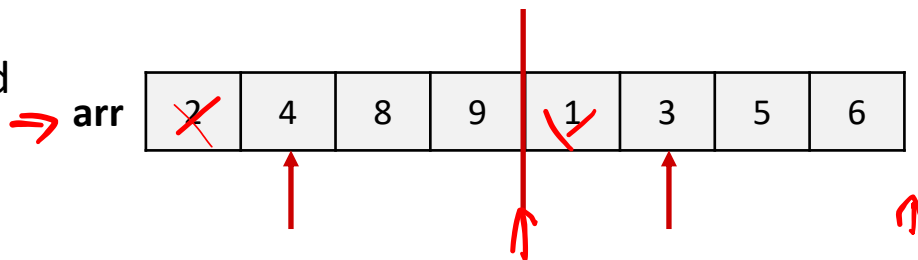
MergeSort: Merging Example (3 of 10)

❖ Start with:



❖ Return from left and right recursion

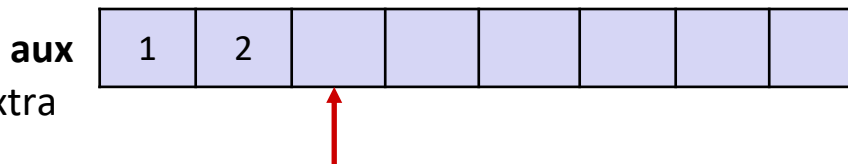
▪ (not magic 😊)



❖ Merge

▪ Use 3 cursors and an extra auxiliary array

▪ When done, copy the extra array back to the original



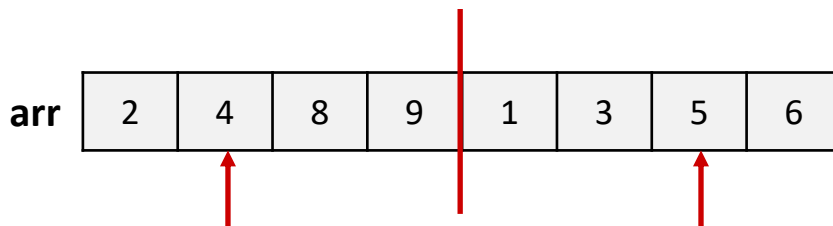
MergeSort: Merging Example (4 of 10)

❖ Start with:



❖ Return from left and right recursion

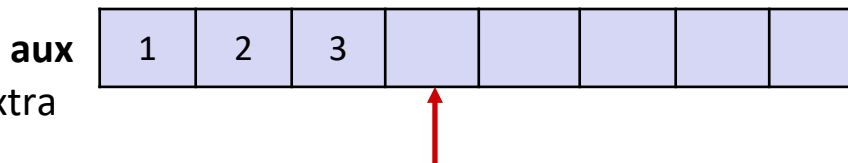
▪ (not magic 😊)



❖ Merge

▪ Use 3 cursors and an extra auxiliary array

▪ When done, copy the extra array back to the original



MergeSort: Merging Example (5 of 10)

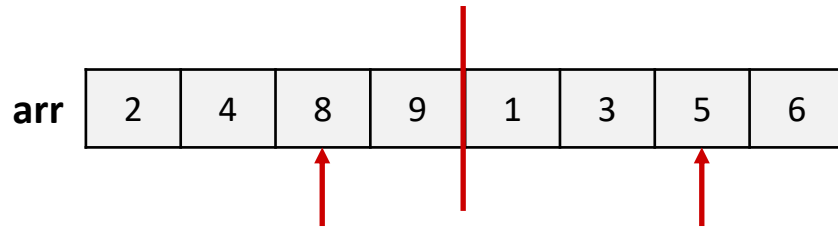
❖ Start with:

arr	8	2	9	4	5	3	1	6
-----	---	---	---	---	---	---	---	---

❖ Return from left and right recursion

▪ (not magic 😊)

arr	2	4	8	9	1	3	5	6
-----	---	---	---	---	---	---	---	---

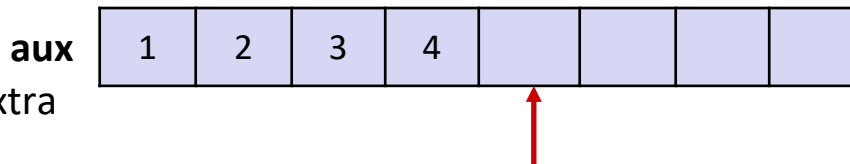


❖ Merge

▪ Use 3 cursors and an extra auxiliary array

▪ When done, copy the extra array back to the original

aux	1	2	3	4				
-----	---	---	---	---	--	--	--	--



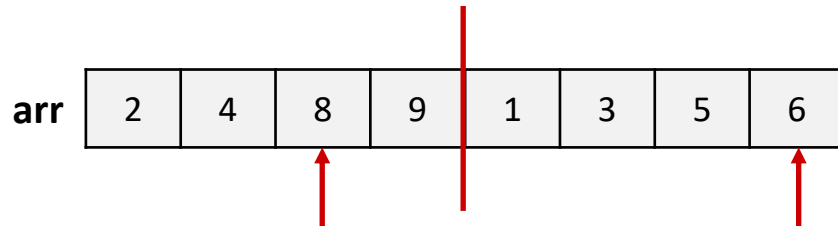
MergeSort: Merging Example (6 of 10)

❖ Start with:



❖ Return from left and right recursion

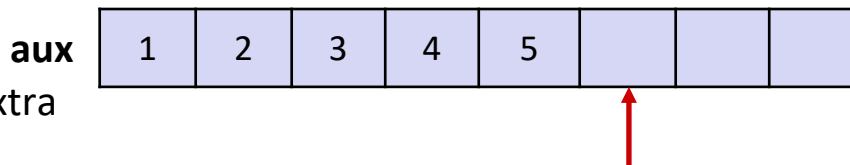
▪ (not magic 😊)



❖ Merge

▪ Use 3 cursors and an extra auxiliary array

▪ When done, copy the extra array back to the original



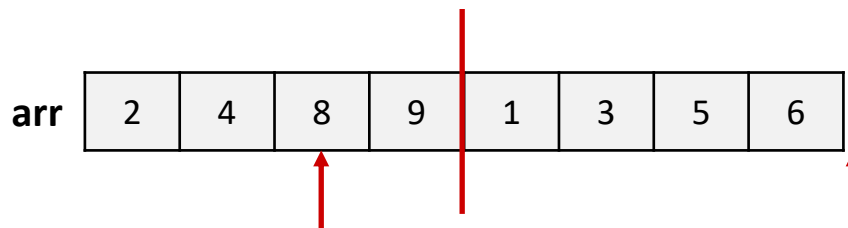
MergeSort: Merging Example (7 of 10)

❖ Start with:



❖ Return from left and right recursion

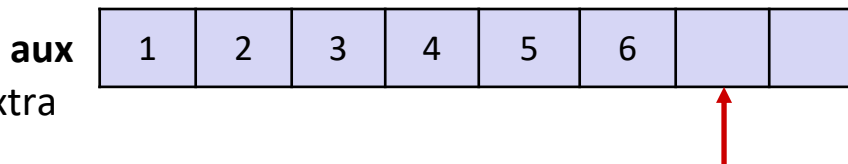
▪ (not magic 😊)



❖ Merge

▪ Use 3 cursors and an extra auxiliary array

▪ When done, copy the extra array back to the original



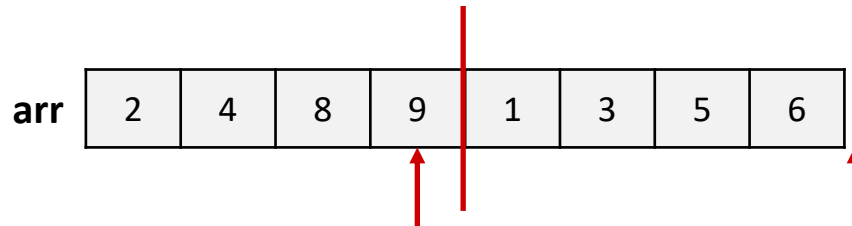
MergeSort: Merging Example (8 of 10)

❖ Start with:



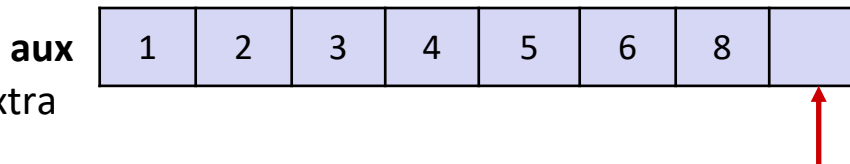
❖ Return from left and right recursion

▪ (not magic 😊)



❖ Merge

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



MergeSort: Merging Example (9 of 10)

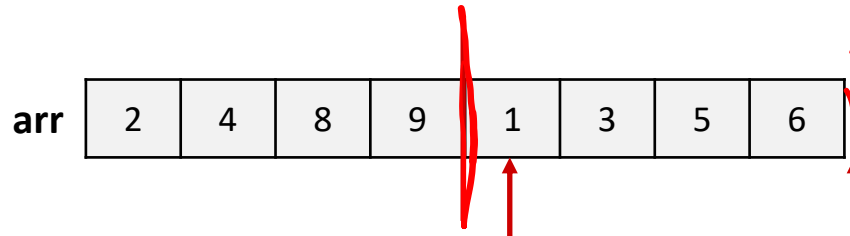
❖ Start with:

arr	8	2	9	4	5	3	1	6
-----	---	---	---	---	---	---	---	---

❖ Return from left and right recursion

▪ (not magic 😊)

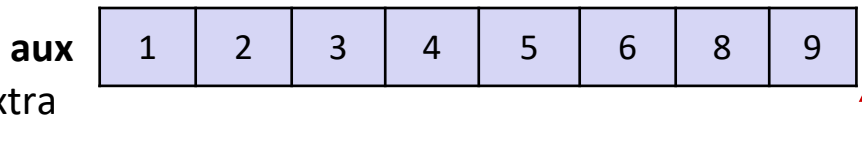
arr	2	4	8	9	1	3	5	6
-----	---	---	---	---	---	---	---	---



❖ Merge

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

aux	1	2	3	4	5	6	8	9
-----	---	---	---	---	---	---	---	---



MergeSort: Merging Example (10 of 10)

❖ Start with:

arr	8	2	9	4	5	3	1	6
-----	---	---	---	---	---	---	---	---

❖ Return from left and right recursion

▪ (not magic 😊)

arr	2	4	8	9	1	3	5	6
-----	---	---	---	---	---	---	---	---

❖ Merge

▪ Use 3 cursors and an extra auxiliary array

▪ When done, copy the extra array back to the original

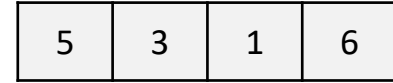
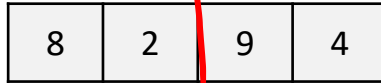
aux	1	2	3	4	5	6	8	9
-----	---	---	---	---	---	---	---	---

arr	1	2	3	4	5	6	8	9
-----	---	---	---	---	---	---	---	---

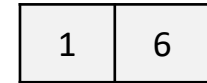
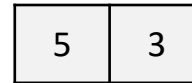
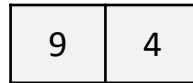
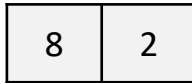
MergeSort: Recursion Example (1 of 3)



Divide

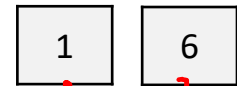
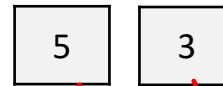
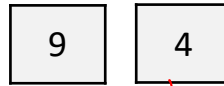
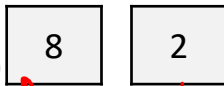


Divide

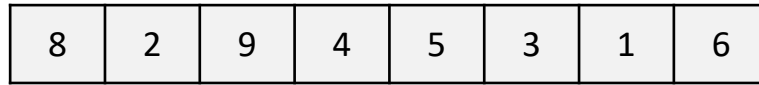


One Element

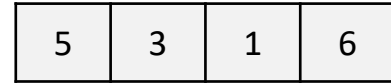
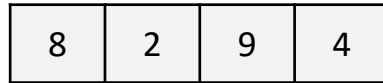
(done recurring!)



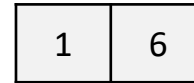
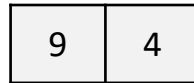
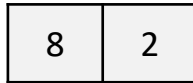
MergeSort: Recursion Example (2 of 3)



Divide

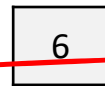
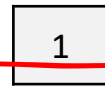
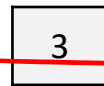
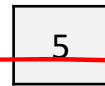
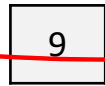
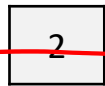
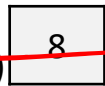


Divide

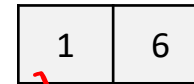
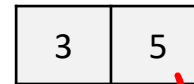
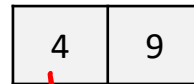
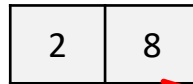


One Element

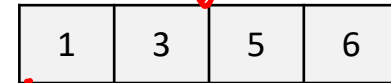
(done recurring!)



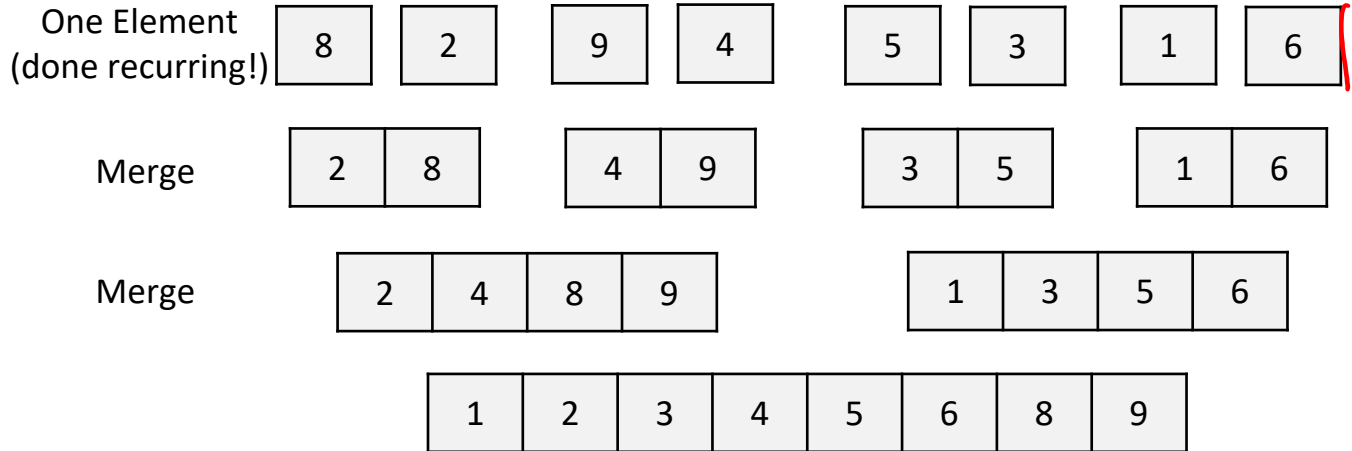
Merge



Merge



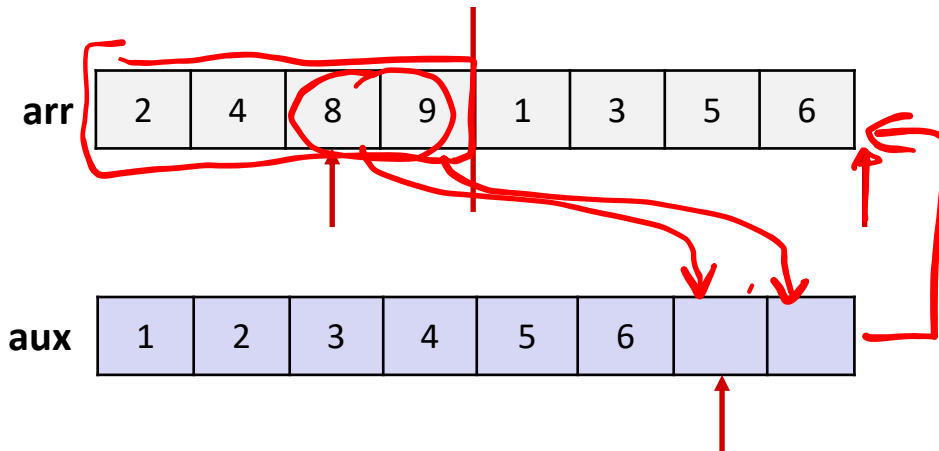
MergeSort: Recursion Example (3 of 3)



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

Optimizations: Reducing “Dregs Copies” (1 of 2)

- ❖ Remember the final steps of our merge example?

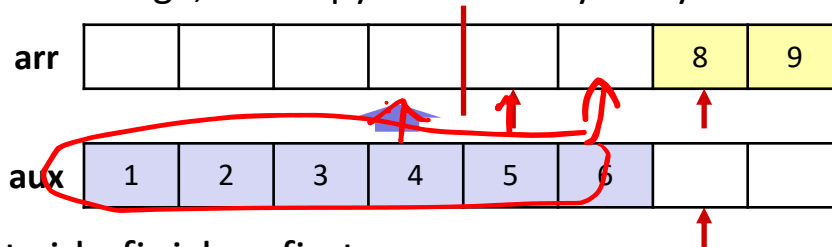


- ❖ It's wasteful to copy 8 & 9 to the auxiliary array, and then immediately copy them back into the original array!

Optimizations: Reducing “Dregs Copies” (2 of 2)

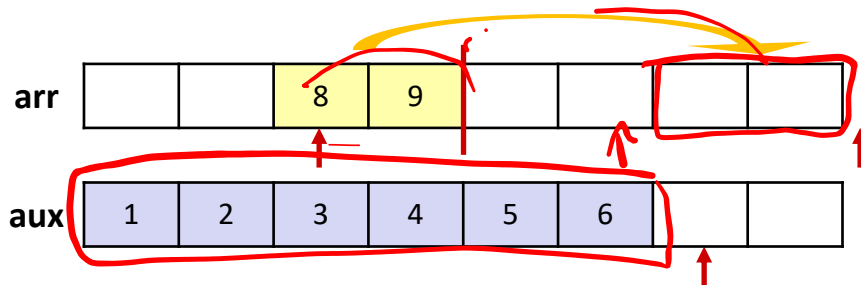
❖ If left side finishes first:

- 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:

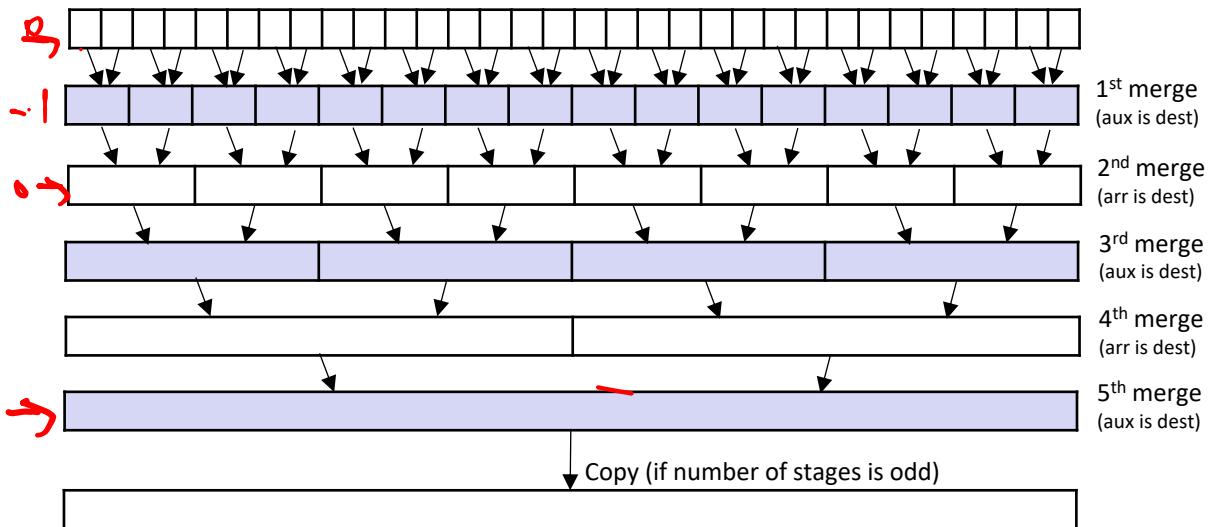
- Allocate a single auxiliary array of size n at beginning to use throughout
- Reuse “slices” of size $(hi - lo) / 2$ within that array at every merge

❖ 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: Runtime Analysis (1 of 3)

❖ MergeSort sorts n elements by:

- Returning immediately if $n=1$

➔ Doing 2 subproblems of size $n/2$ + then an $O(n)$ merge otherwise

❖ Runtime expression?

- $T(1) = c_1$

- $T(n) = 2T(n/2) + c_2n$

MergeSort: Runtime Analysis (2 of 3)

$$T(1) = c_1$$

$$T(n) = 2T(n/2) + c_2n \quad \text{First expansion}$$

$$= 2(2T(n/4) + c_2n/2) + c_2n \quad \text{Second expansion}$$

$$= 4T(n/4) + 2c_2n$$

$$= 4(2T(n/8) + c_2n/4) + 2c_2n \quad \text{Third expansion}$$

$$= 8T(n/8) + 3c_2n$$

$$= \underline{2^k T(n/2^k) + kc_2n} \quad \text{kth expansion}$$

If I want $n/2^k = 1$, let $k = \log n$

Then $T(n) = 2^k T(n/2^k) + kc_2n$

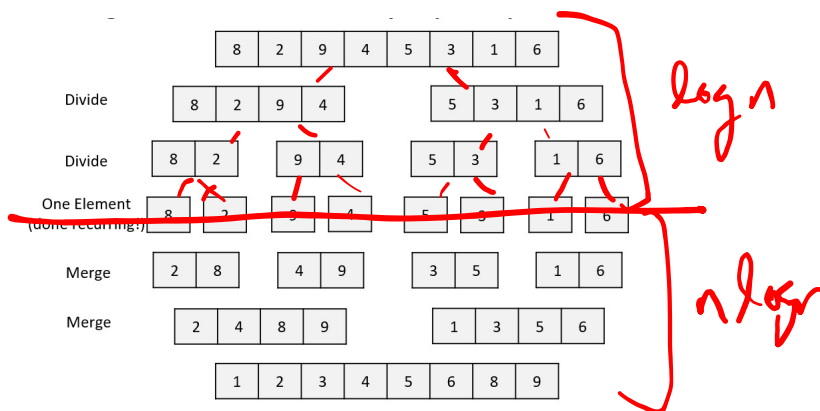
$$= 2^{\log n} T(1) + \log n c_2n$$

$$= c_1n + c_2n \log n$$

$$= O(n \log n)$$

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’s runtime 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



MergeSort: Characteristics

❖ Execution:

- Merge sorted subarrays as it “recurs upward” (ie, returns from recursive calls)

❖ Characteristics:

- Stable: yes
- In-place: no

❖ Time: always $O(n \log n)$

```
mergeSort(arr, startIdx, endIdx) {
    if (startIdx == endIdx
        || startIdx + 1 == endIdx) {
        return;
    }

    midIdx = (endIdx - startIdx) / 2
              + startIdx;
    mergeSort(arr, startIdx, midIdx);
    mergeSort(arr, midIdx, endIdx);
    merge(arr, startIdx, midIdx,
          endIdx);
}
```

MergeSort: Final Thoughts

- ❖ We've discussed arrays, but you may need to sort linked lists
 - One approach for other algorithms:
 - 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 new disk accesses*