Comparison Sorts (cont.)

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	

Announcements

- Quiz 2 due tomorrow *morning*

Lecture Outline

- Comparison-based Sorting
  - **Review**
  - Simple algorithms
    - InsertionSort
    - SelectionSort
  - Fancier Algorithms: HeapSort
  - Fancier algorithms using Divide-and-Conquer
    - Intro
    - MergeSort
    - QuickSort

Comparison Sorting Definitions

- Problem: We have *n* comparable items in an array, and we want to rearrange them in such that for any index i and j,

  $$\boxed{\text{if } i < j, \quad \text{then } A[i] \le A[j]}$$

- Notable Variations:
  - **Stable sort**: if there are ties, preserve the original ordering
  - **In-place sorts**: don't use more than O(1) "auxiliary space"

- Why "comparison sorting"?
  - If our elements can do more than just a pairwise comparison, we can use different techniques

Sorting: The Big Picture

- Comparison-based sorting algorithms
  - 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: Ω(n log n)

- Techniques for handling huge data sets:
  - External sorting

- Specialized algorithms: O(n)
  - BucketSort, RadixSort

Lecture Outline

- Comparison-based Sorting
  - Review
  - **Simple algorithms**
    - InsertionSort
    - SelectionSort
  - Fancier Algorithms: HeapSort
  - Fancier algorithms using Divide-and-Conquer
    - Intro
    - 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: $O(n^2)$ Randomized case: $O(n^2)$
- Characteristics:

  Stable: Y In-place: Y

Demo:
https://docs.google.com/presentation/d/10b9aRqpGJu8pUk8OpfqUIEEm8ou-zmmC7b_BE5wgNg0/present

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: $O(n^2)$ Randomized case: $O(n^2)$

- Characteristics:

  Stable: N In-place: Y

Demo: https://docs.google.com/presentation/d/1p6g3r9BpwTARjUylA0V0yspP2temzHNJEJjCG41I4r0/edit

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 < ~15)
  - Java's built-in sort prefers InsertionSort for arrays <47 items
- But …
  - There are other algorithms which are more efficient *for non-small arrays that are not already "mostly sorted"*

Aside: We won't cover Bubble Sort

- It doesn't have good asymptotic complexity: $O(n^2)$
- It's not particularly efficient with respect to common factors
- Basically, almost everything it is good at, some other algorithm is at least as good at
- Some people seem to teach it just because someone taught it to them

- For fun see: "Bubble Sort: An Archaeological Algorithmic Analysis", Owen Astrachan, SIGCSE 2003 http://www.cs.duke.edu/~ola/bubble/bubble.pdf

Lecture Outline

- Comparison-based Sorting
  - Review
  - Simple algorithms
    - InsertionSort
    - SelectionSort
  - **Fancier Algorithms: HeapSort**
  - Fancier algorithms using Divide-and-Conquer
    - Intro
    - MergeSort
    - QuickSort

Naïve HeapSort

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

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

- Time:

  Best-case: O(n log n) Worst-case: O(n log n) Randomized case: O(n log n)

- Characteristics:

  Stable: N In-place: ______

Demo: https://goo.gl/EZWwSJ
    ```

In-place HeapSort

- Idea: Put everything in a **MAX** heap ; successively `deleteMax`
  - 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:

  Best-case: O(n log n) Worst-case: O(n log n) “Average” case: O(n log n)

- Characteristics:

  Stable: N In-place: Y!

Demo:
https://docs.google.com/presentation/d/1SzcQC48OB9agStD0dFRgccU-tyjD6m3esrSC-GLxmNc/present

Aside: "AVLSort" and "DataStructureSort"

- We can also use a balanced tree to:
  - `add` each element: total time $O(n \texttt{ log } n)$
  - Do an in-order traversal $O(n)$

- But a balanced tree cannot be made in-place, and constants worse than HeapSort
  - Both are $O(n \texttt{ log } n)$ in worst, best, and average case
  - Neither sorts parallelizes well

- Don't even think about trying to sort with a hash table …

Lecture Outline

- Comparison-based Sorting
  - Review
  - Simple algorithms
    - InsertionSort
    - SelectionSort
  - Fancier Algorithms: HeapSort
  - Fancier algorithms using Divide-and-Conquer
    - **Intro**
    - 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*]

Lecture Outline

- Comparison-based Sorting
  - Review
  - Simple algorithms
    - InsertionSort
    - SelectionSort
  - Fancier Algorithms: HeapSort
  - Fancier algorithms using Divide-and-Conquer
    - Intro
    - **MergeSort**
    - 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:
    - "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…

8	2	9	4	5	3	1	6

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)

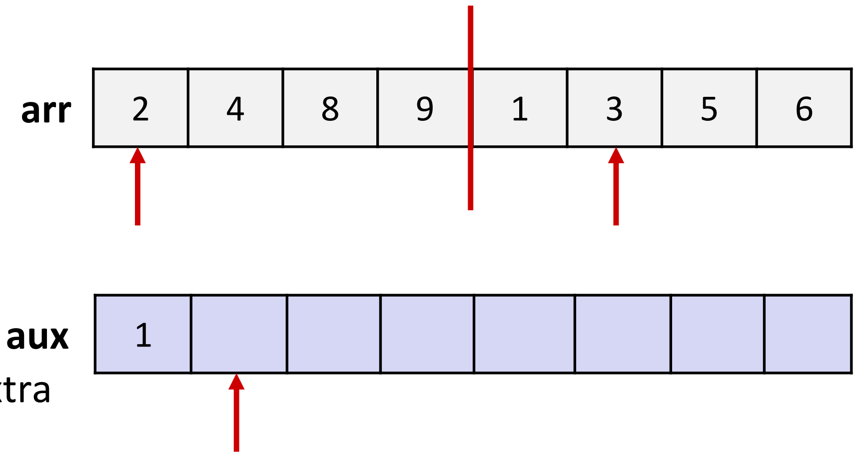
arr	2	4	8	9	1	3	5	6

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

aux								

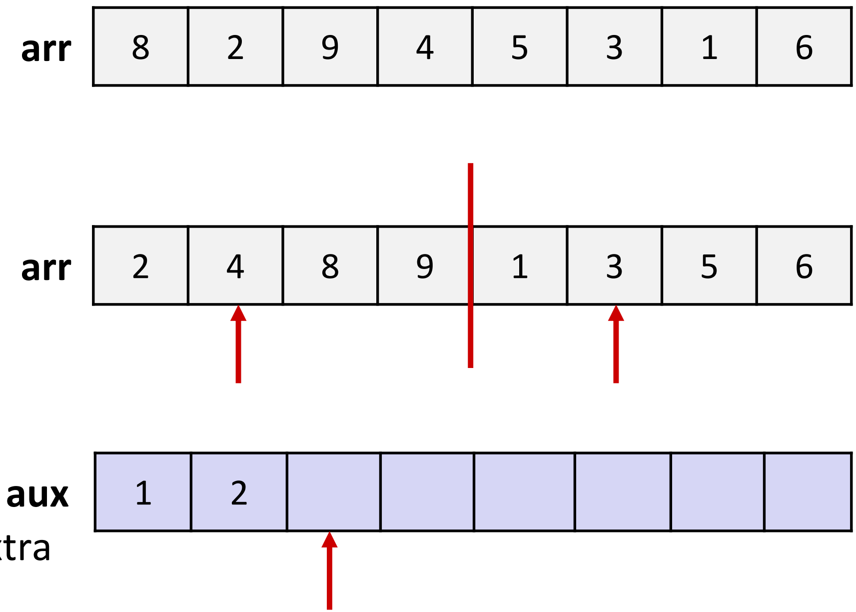
MergeSort: Merging Example (2 of 10)

- 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:

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

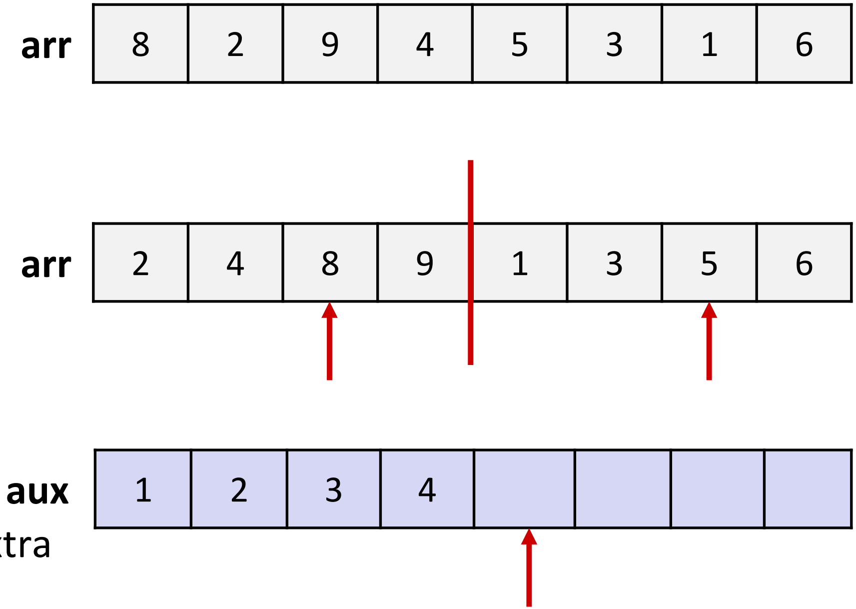
- Merge
  - Use 3 cursors and an extra auxiliary array
  - When done, copy the extra array back to the original

aux

1	2	3					

MergeSort: Merging Example (5 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 (6 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			

MergeSort: Merging Example (7 of 10)

- Start with:

arr								
arr	8	2	9	4	5	3	1	6

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

dregs

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		

MergeSort: Merging Example (8 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	

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

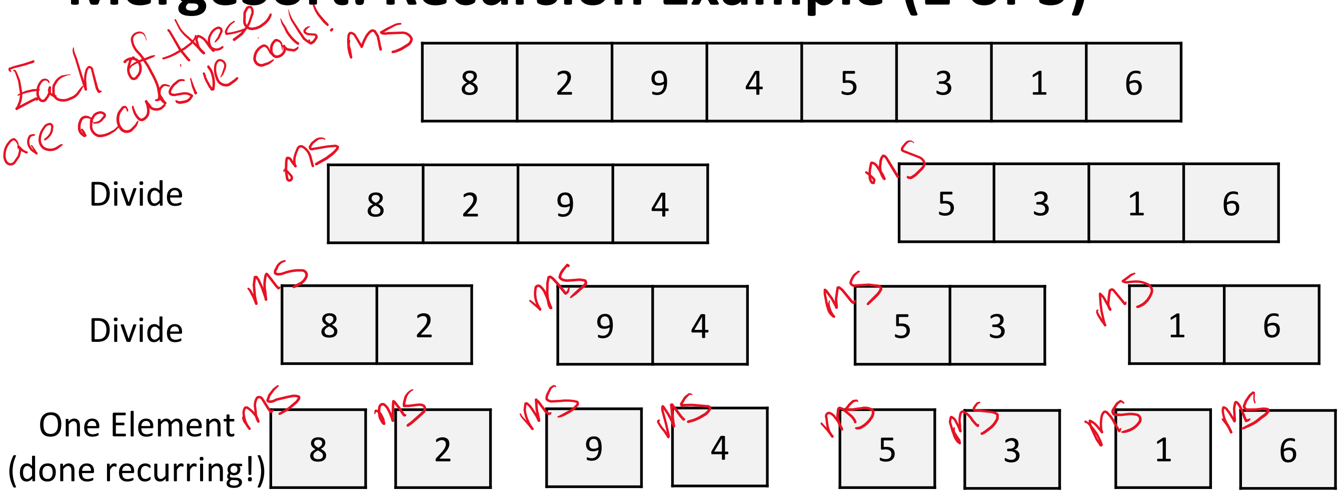
- Return from left and right recursion
  - (not magic ☺)

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)



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]

work done as the recursive call returns

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)

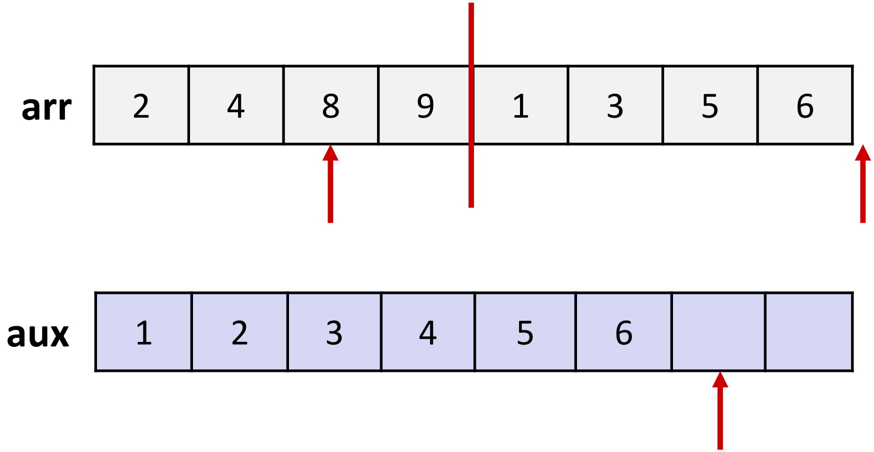
One Element (done recurring!)	8 2	9 4	5 3	1 6
Merge	2 8	4 9	3 5	1 6
Merge	2 4 8 9		1 3 5 6	
	1 2 3 4 5 6 8 9			

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

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

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

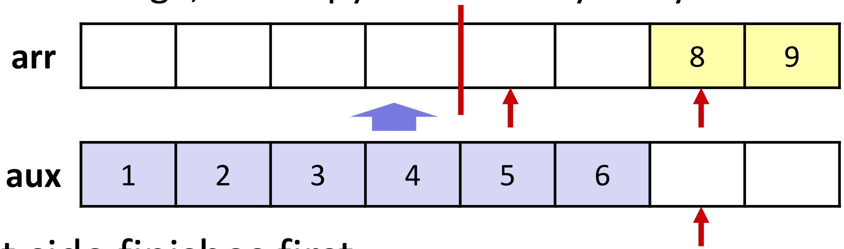
- Remember the final steps of our merge example?



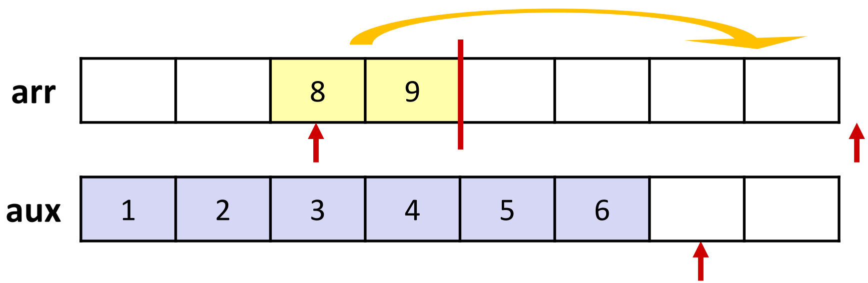
- It’s wasteful to copy 8 & 9 to the auxiliary array, and then immediately copy them back into the original array!

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

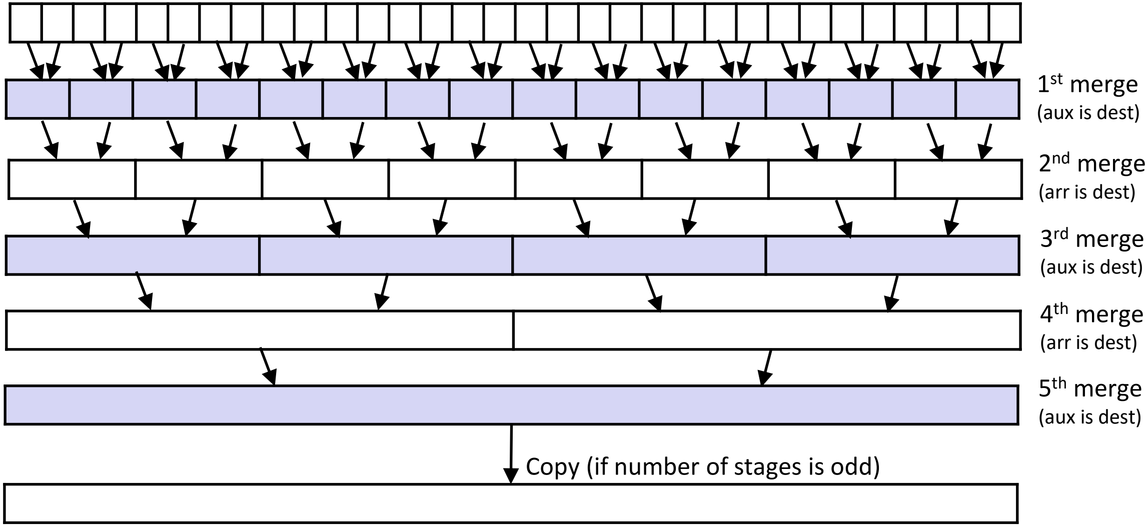


Optimizations: Reducing Temp Arrays (1 of 2)

- 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)

$T(1) = c_1$

First expansion

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

Second expansion

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

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

Third expansion

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

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

k^{th} expansion

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

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

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

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

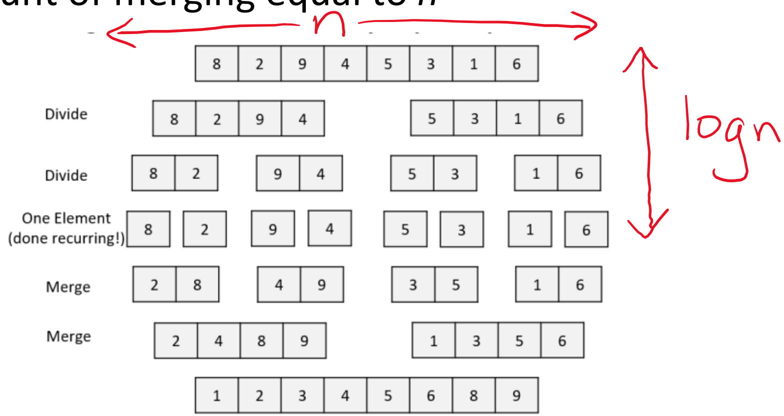
$= c_1 n + c_2 n \log n$

$= O(n \log n)$

MergeSort: Runtime Analysis (3 of 3)

- More intuitively, this recurrence comes up often enough you should "just know" it's $O(n \texttt{ log } n)$

- 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)

```
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);
}
```

- Characteristics:
  - Stable: yes
  - In-place: no

- Time: always O(n log n)

MergeSort: Final Thoughts

- 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 new disk accesses)*