CSE 373: Data Structures and Algorithms

Lecture 19: Comparison Sorting Algorithms

Instructor: Lilian de Greef
Quarter: Summer 2017

Today

- Intro to sorting
- Comparison sorting
  - Insertion Sort
  - Selection Sort
  - Heap Sort
  - Merge Sort

Sorting

Now looking at algorithms instead of data structures!

Introduction to Sorting

- Stacks, queues, priority queues, and dictionaries all focused on providing one element at a time
- But often we know we want “all the things” in some order
  - Humans can sort, but computers can sort fast
  - Very common to need data sorted somehow
    - Alphabetical list of people
    - List of countries ordered by population
    - Search engine results by relevance
    - List store catalogue by price
    - …
- Algorithms have different asymptotic and constant-factor trade-offs
  - No single “best” sort for all scenarios
  - Knowing one way to sort just isn’t enough

More Reasons to Sort

General technique in computing:

Preprocess data to make subsequent operations faster

Example: Sort the data so that you can

- Find the k^{th} largest in constant time for any k
- Perform binary search to find elements in logarithmic time

Whether the performance of the preprocessing matters depends on

- How often the data will change (and how much it will change)
- How much data there is

The main problem, stated carefully

For now, assume we have *n* comparable elements in an array and we want to rearrange them to be in increasing order

Input:

- An array `A` of data records
- A key value in each data record
- A comparison function

Effect:

- Reorganize the elements of `A` such that for any `i` and `j`, if `i < j` then
- (Also, `A` must have exactly the same data it started with)
- Could also sort in reverse order, of course

An algorithm doing this is a **comparison sort**

Variations on the Basic Problem

1. Maybe elements are in a linked list (could convert to array and back in linear time, but some algorithms needn't do so)
2. Maybe ties need to be resolved by "original array position"
    - Sorts that do this naturally are called
3. Maybe we must not use more than $O(1)$ "auxiliary space"
    - Sorts meeting this requirement are called
4. Maybe we can do more with elements than just compare
    - Sometimes leads to faster algorithms
5. Maybe we have too much data to fit in memory
    - Use an "   " algorithm

Sorting: The Big Picture

Surprising amount of neat stuff to say about sorting:

Simple algorithms: $O(n^2)$	Fancier algorithms: $O(n \log n)$	Comparison lower bound: $\Omega(n \log n)$	Specialized algorithms: $O(n)$	Handling huge data sets
Insertion sort	Heap sort		Bucket sort	External sorting
Selection sort	Merge sort		Radix sort	
Shell sort	Quick sort			
…	…			

(space for notes from demo)

Insertion Sort

- Idea: At step `k`, put the `k`th element in the correct position among the first `k` elements

- Alternate way of saying this:
  - Sort first two elements
  - Now insert 3rd element in order
  - Now insert 4th element in order
  - …

- "Loop invariant": when loop index is `i`, first `i` elements are sorted

- Time?

  Best-case ______ Worst-case ______ "Average" case ______

Selection sort

- Idea: At step `k`, find the smallest element among the not-yet-sorted elements and put it at position k

- Alternate way of saying this:
  - Find smallest element, put it 1st
  - Find next smallest element, put it 2nd
  - Find next smallest element, put it 3rd …

- "Loop invariant": when loop index is `i`, first `i` elements are the `i` smallest elements in sorted order

- Time?

  Best-case _________ Worst-case _________ "Average" case ________

Insertion Sort vs. Selection Sort

- Different algorithms
- Solve the same problem
- Have the same worst-case and average-case asymptotic complexity
  - Insertion-sort has better best-case complexity; preferable when input is "mostly sorted"
- Other algorithms are more efficient *for large arrays that are not already almost sorted*
  - Insertion sort may do well on small arrays

The Big Picture

Surprising amount of juicy computer science: 2-3 lectures…

Simple algorithms: $O(n^2)$	Fancier algorithms: $O(n \log n)$	Comparison lower bound: $\Omega(n \log n)$	Specialized algorithms: $O(n)$	Handling huge data sets
Insertion sort Selection sort Shell sort …	Heap sort Merge sort Quick sort (avg) …		Bucket sort Radix sort	External sorting

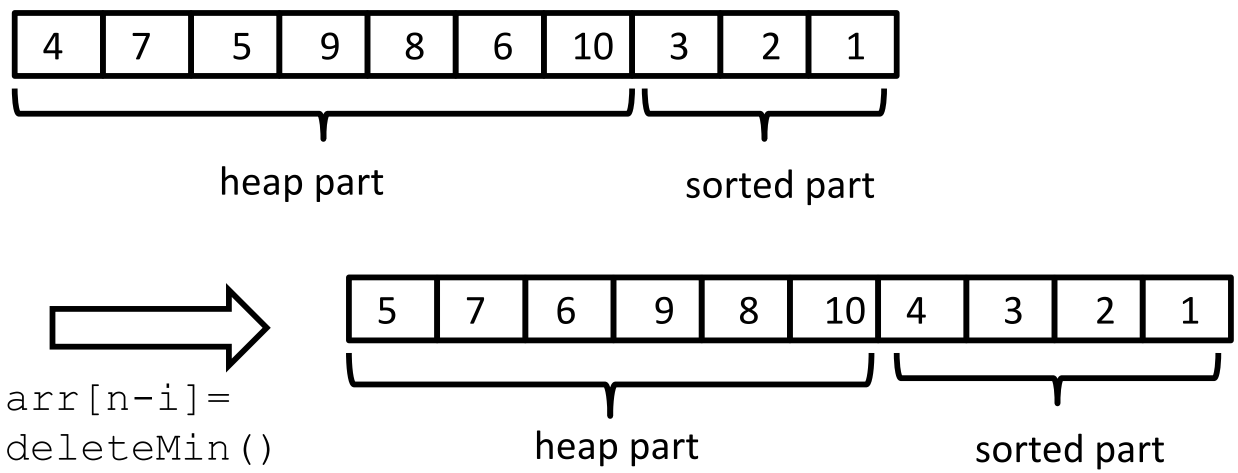
Heap sort

- Sorting with a heap:
  - `insert` each `arr[i]`, or better yet use `buildHeap`
  - ```
    for(i=0; i < arr.length; i++)
        arr[i] =
    ```
- Worst-case running time:
- We have the array-to-sort and the heap
  - So this is not an in-place sort
  - There’s a trick to make it in-place…
    ```

In-place heap sort

But this reverse sorts – how would you fix that?

- Treat the initial array as a heap (via `buildHeap`)
- When you delete the `i`th element, put it at `arr[n-i]`
  - That array location isn't needed for the heap anymore!



“AVL sort”

- We can also use a balanced tree to:
  - `insert` each element: total time $O(n \log n)$
  - Repeatedly `deleteMin`: total time $O(n \log n)$
    - Better: in-order traversal $O(n)$, but still $O(n \log n)$ overall

- Compared to heap sort
  - both are $O(n \log n)$ in worst, best, and average case
  - neither parallelizes well
  - heap sort is can be done in-place, has better constant factors

Design decision: which would you choose between Heap Sort and AVL Sort? Why?

“Hash sort”???

Finding min item in a hashtable is O(n), so this would be a slower, more complicated selection sort

Divide and conquer

Very important technique in algorithm design

1. Divide problem into smaller parts

2. Independently solve the simpler parts
    - Think recursion
    - Or parallelism

3. Combine solution of parts to produce overall solution

Two great sorting methods are fundamentally divide-and-conquer
(Merge Sort & Quicksort)

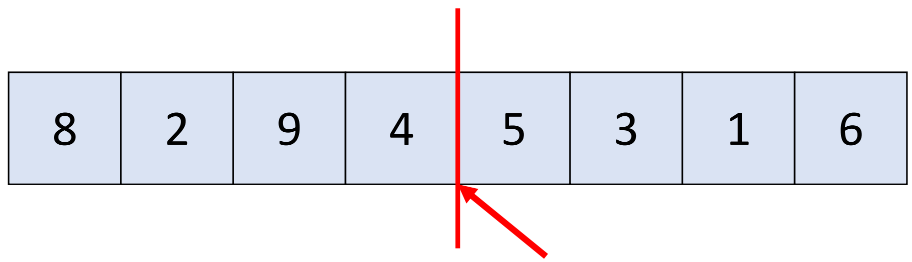
Merge Sort

Merge Sort: recursively…

- Sort the left half of the elements
- Sort the right half of the elements
- Merge the two sorted halves into a sorted whole

(space for notes from demo)

Merge sort

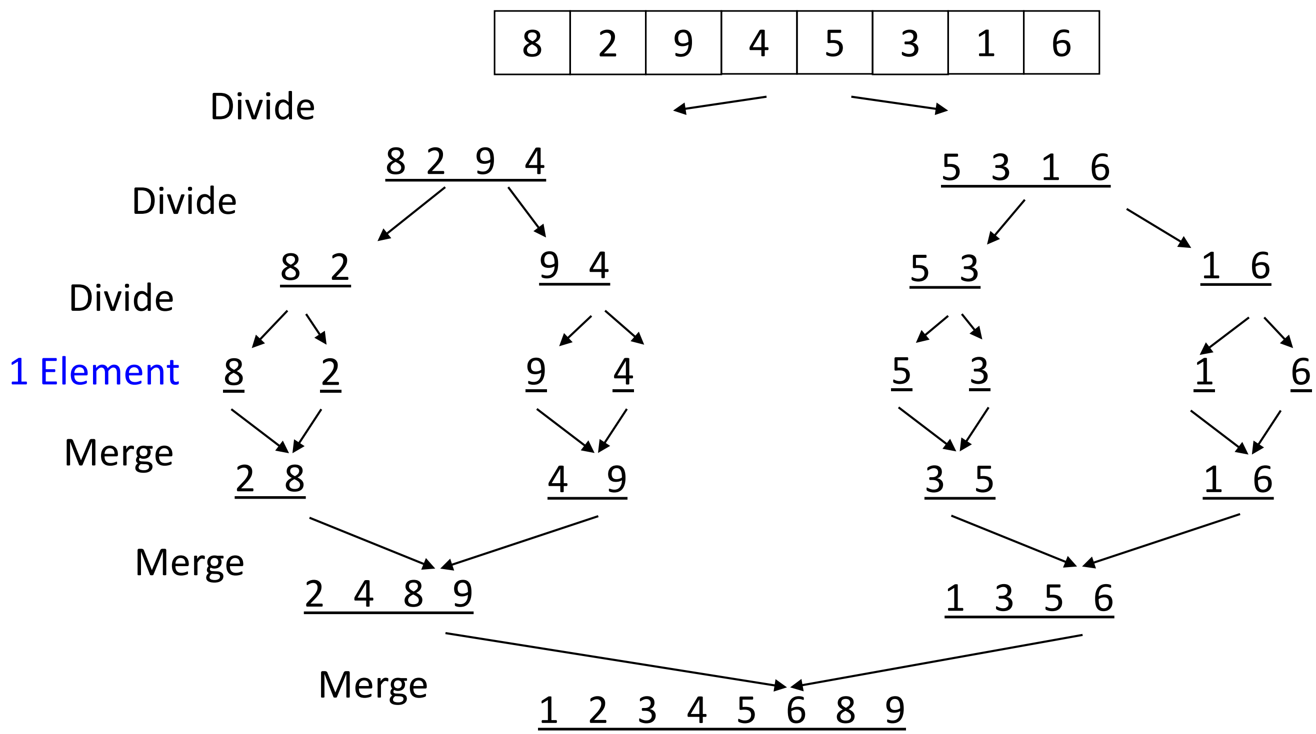


- To sort array from position `lo` to position `hi`:
  - If range is 1 element long, it is already sorted!
  - Else:
    - Sort from `lo` to `(hi+lo)/2`
    - Sort from `(hi+lo)/2` to `hi`
    - Merge the two halves together
- Merging takes two sorted parts and sorts everything
  - $O(n)$ but requires auxiliary space…

Merge Sort: Example focused on merging

Start with:	8	2	9	4	5	3	1	6	Main array
After recursion: (not magic ☺)	2	4	8	9	1	3	5	6	Main array
Merge: Use 3 “fingers” and 1 more array									Auxiliary array
(After merge, copy back to original array)									Main array

Merge Sort: Example showing recursion

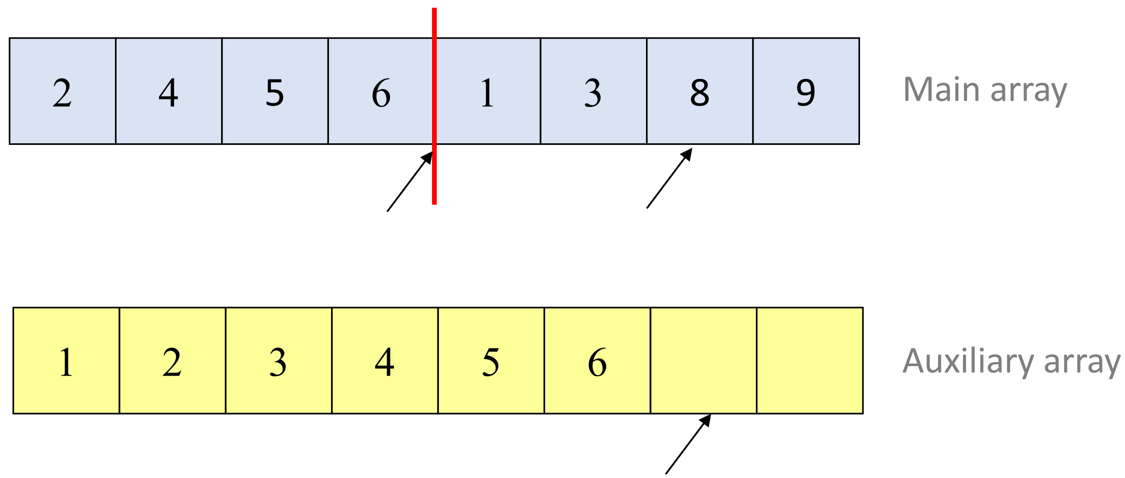


One way to practice on your own time:

- Make yourself an unsorted array
- Try using one of the sorting algorithms on it
- You know you got the right end result if it comes out sorted
- Can use the same example for merge sort as the previous slide to double check in-between steps

Some details: saving a little time

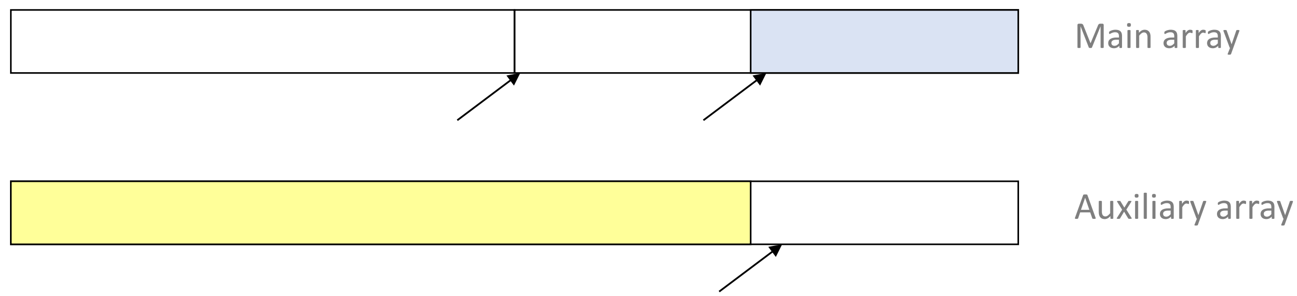
- What if the final steps of our merge looked like this:



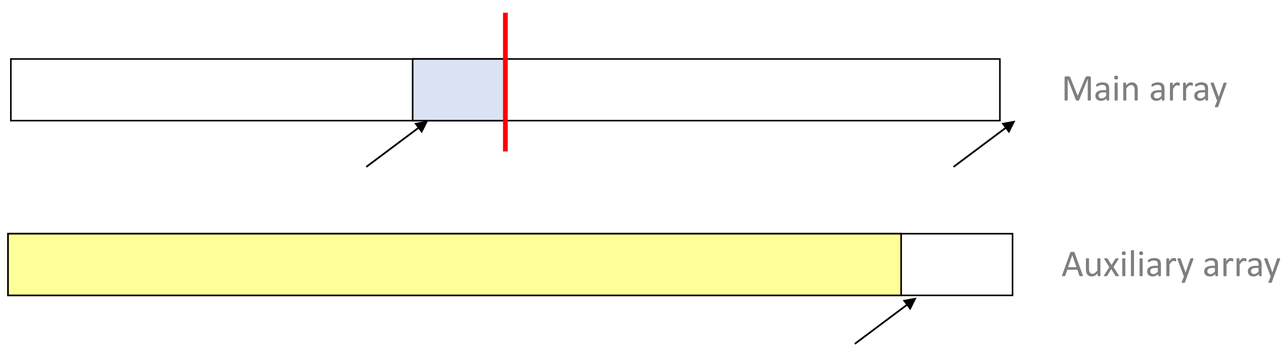
- Wasteful to copy to the auxiliary array just to copy back…

Some details: saving a little time

- If left-side finishes first, just stop the merge and copy back:



- If right-side finishes first, copy dregs into right then copy back



Some details: saving space and copying

Simplest / Worst:

Use a new auxiliary array of size `(hi-lo)` for every merge

Better:

Use a new auxiliary array of size `n` for every merging stage

Better:

Reuse same auxiliary array of size `n` for every merging stage

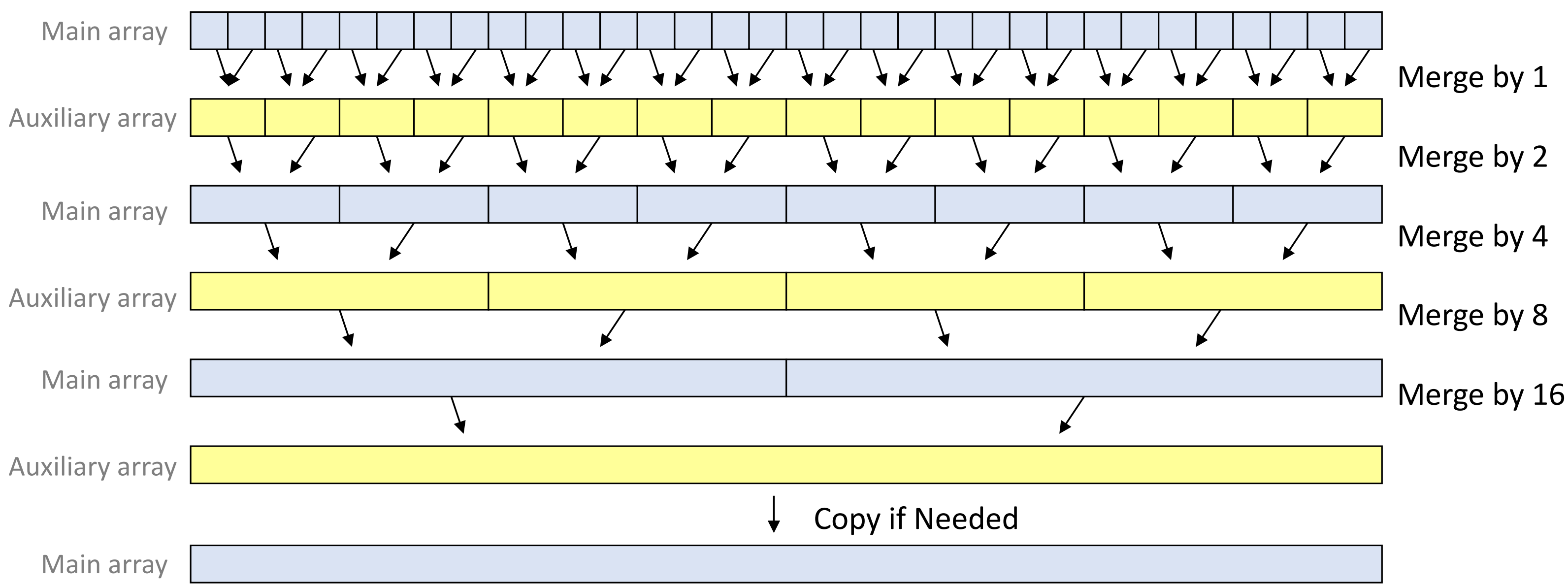
Best (but a little tricky):

Don't copy back – at 2nd, 4th, 6th, … merging stages, use the original array as the auxiliary array and vice-versa

- Need one copy at end if number of stages is odd

Swapping Original / Auxiliary Array (“best”)

- First recurse down to lists of size 1
- As we return from the recursion, swap between arrays



(Arguably easier to code up without recursion at all)

Linked lists and big data

We defined sorting over an array, but sometimes you want to sort linked lists

One approach:

- Convert to array:
- Sort:
- Convert back to list:

Merge sort works very nicely on linked lists directly

- Heapsort and quicksort do not
- Insertion sort and selection sort do but they're slower

Merge sort is also the sort of choice for external sorting

- Linear merges minimize disk accesses
- And can leverage multiple disks to get streaming accesses

Analysis

Having defined an algorithm and argued it is correct, we should analyze its running time and space:

To sort *n* elements, we:

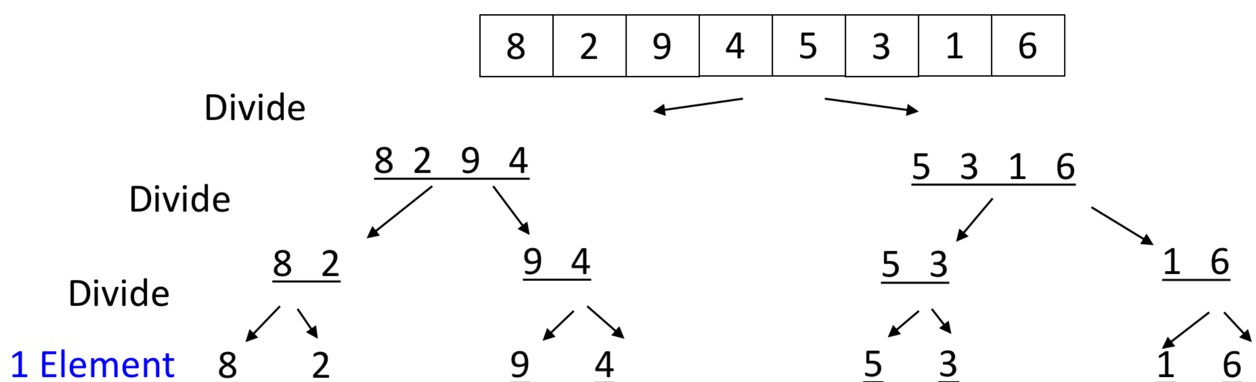
- Return immediately if *n*=1
- Else do 2 subproblems of size and then an merge

Recurrence relation:

Analysis intuitively

This recurrence is common, you just "know" it's $O(n \texttt{ log } n)$

Merge sort is relatively easy to intuit (best, worst, and average):
- The recursion "tree" will have height
- At each level we do a *total* amount of merging equal to



Analysis more formally

(One of the recurrence classics)

For simplicity, ignore constants (let constants be)

$T(1) = 1$

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

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

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

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

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

....

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

We will continue to recurse until we reach the base case, i.e. T(1) for T(1), $n/2^k = 1$, i.e., $\log n = k$

So the total amount of work is $2^k T(n/2^k) + kn = 2^{\log n} T(1) + n \log n = n + n \log n = O(n \log n)$

Divide-and-Conquer Sorting

Two great sorting methods are fundamentally divide-and-conquer

1. Merge Sort:
    - 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

2. Quicksort:
    - Pick a “pivot” element
    - Divide elements into “less-than pivot” and “greater-than pivot”
    - Sort the two divisions (recursively on each)
    - Answer is “sorted-less-than”, followed by “pivot”, followed by ”sorted-greater-than”

Quicksort Overview (sneak preview)

1. Pick a pivot element

2. Partition all the data into:
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than the pivot

3. Recursively sort A and C

4. The final answer is “as simple as A, B, C” *(also is an American saying)*

Cool Resources

- http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
- http://www.sorting-algorithms.com/
- https://www.youtube.com/watch?v=t8g-iYGHpEA

Seriously, check them out!