Outline

- merge sort
- heap sort
- quick sort

merge sort

- given two arrays a1 and a2, both of size n
- assume one of the two arrays, a1, contains the input data
- if n == 1, the array is sorted, copy a1 to a2
- otherwise, split the two arrays into equal-sized parts (within one)
- recursively merge-sort the two sub-arrays
- merge the two sorted sub-arrays into the final array
- depth of recursion is log n, so time for sorting is O(n log n)
- space requirement is 2n
- see this example: https://upload.wikimedia.org/wikipedia/commons/c/cc/Merge-sort-example-300px.gif
- in-class exercise (in groups of 2 or 3): write code to implement merge sort, assuming you have the method to merge two arrays into a third one

heap sort

- insert all the elements into a heap
- remove all the elements from a heap
- since the heap grows from the left of the array, insert the elements from left to right, making room for each new element by removing it from the array just as the heap needs the additional space
- in-class exercise: given a method to insert items into a heap, write a method to insert all the elements of an array into a heap
- since the heap shrinks from the right of the array, the element removed from the heap can be put back into the array just to the right of the heap
- in-class exercise (in small groups or individually): given a method to insert items into a heap (`void insertHeap(E value, E[] heap, int heapSize)`) and a method to remove items from a heap (`E removeHeap(E[] heap, int heapSize)`), write a heapsort method
- each insertion and removal takes at most time O(log n), so heapsort takes at most O(n log n), and space n
- see this example: https://en.wikipedia.org/wiki/File:Sorting_heapsort_anim.gif

quick sort

- given an array to sort, pick an arbitrary element, called a pivot
- arrange all the elements so everything smaller than the pivot is to the left of the pivot, and everything larger than the pivot is to the right of the pivot
- consider the sub-arrays to the left of the pivot and to the right of the pivot
- if these sub-arrays were sorted, the entire array would be sorted
- the sub-arrays can be sorted by calling quick-sort recursively

quick sort:
partitioning the sub-arrays

- partitioning can be done with a single pass through the array:

1. select the first array element as the pivot, and set index p = 0
2. set index first = 1, last = array.length - 1
3. while (array[first] < pivot) first++;
4. while (array[last] > pivot) last--;
5. now array[first] > pivot and array[last] < pivot, so swap array[first] and array[last]
6. go back to step 3 until all elements < pivot are to the left of all the elements > pivot (and first == last)
7. swap array[p] and array[first] to put the pivot in between

   see the example at https://en.wikipedia.org/wiki/File:Sorting_quicksort_anim.gif

quick sort: analysis

- if the pivot is approximately in the middle, the depth of recursion will be O(log n)
- at each depth of recursion, the total amount of work done is O(n) to rearrange the array to the left and right of the pivot
- so with good pivot selection, quicksort is O(n log n)
- with random pivot selection, quicksort is O(n log n)
- with the worst possible pivot (smallest or largest), e.g. if the array is already sorted, the depth of recursion is O(n), and quicksort is O(n^2)