Outline

- insertion sort
- merge sort
- heap sort
- quick sort
- Shell sort

insertion sort

- in selection sort, find the smallest element, and put it in the next position
- in insertion sort, take the next element, and put it in the right place
- trivia: card players typically use insertion sort to arrange their decks
- the sub-array at the beginning of the array is already sorted, just as in selection sort
- the next element e is always taken from the next index in the array
- elements greater than e (in the sorted part of the array) are *shifted up* one position, to free the position where element e will be stored
- in-class exercise (in groups of 2 or 3): write code to implement insertion sort
- in-class exercise: how long does insertion sort take?

merging sorted data

- given two sorted arrays a1, a2 of sizes n1, n2
- fill an array a of size n=n1+n2 with the sorted data from both arrays
- keep an index for each of the input arrays: i1, i2
- compare a1[i1] to a2[i2]
- put the smallest into a[i], and increment both i and the index for the array from which the data was taken
- if either array is "finished" (all its data is already in the final array), just copy the remaining data from the other array into the final array
- continue until all the data has been transferred to the bigger array

merging sorted data exercises

- in-class exercise: how long does it take to merge two arrays into one?
- in-class exercise: how much space does it take to merge two arrays into one array?
- in-class exercise (in groups of 2 or 3): write code to implement merging two arrays into a 3rd. All three arrays are given, and you should verify that `a.length = a1.length + a2.length`

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, int heapSize)`) and a method to remove items from a heap (`E removeHeap(E value, 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 $O(n^2)$

Shell sort

- named after Donald Shell
- tries to do insertion sort on nearly-sorted arrays, when insertion sort is most efficient
- divide array into sub-arrays of 2 or 3 elements
- sort the sub-arrays
- combine the sub-arrays in sorted order
- subtlety: sub-arrays are not adjacent, they are the collection of elements separated by gap
- intuitively, the array can be thought of as being re-arranged into gap rows of n/gap columns (but without moving the data in the array)
- each column is sorted (in place in the array) using insertion sort
- this can move data quickly towards its final position in the array
- selecting the gap affects the performance of the algorithm
- for good performance, the initial gap can be floor(n/2), subsequent gaps floor(gap/2.2) (but not less than 1)

Shell sort specifics

- set the gap as described above
- the first sub-array consists of a[0], a[0+gap], and possibly a[0+gap+gap]
- the second sub-array consists of a[1], a[0+gap], and possibly a[1+gap+gap], and so on
- perform insertion sort on each sub-array
- update the value of the gap as described above
- the first sub-array consists of a[0 + n * gap] for all values of n which give valid indices in the array
- the second sub-array consists of a[1 + n * gap], and so on
- when gap==1, the array is sorted

Shell sort analysis

- insertion sort works well on nearly-sorted arrays
- shell sort is better than insertion sort, because it applies insertion sort to nearly-sorted arrays
- if the gap is a sequence of numbers of the form 2^{k-1} (for decreasing k, e.g. 31, 15, 7, 3, 1), the performance is $O(n^{3/2})$, that is, O(n sqrt(n))