Outline

- reminder: equality and comparisons in Java
- reminder: sorting
- reminder: selection sort
- reminder: bubble sort
- reminder: insertion sort
- Shell sort
- Merge sort

Java equality reminder

- four kinds of equality:
  - `==` is true if and only if two references are to the *exact same object* (or for basic types)
  - `object1.equals(object2)` is `true` if object1's `equals` method (*in its wisdom*) decides they are the same
  - `object1.compareTo(object2)` is `0` if the `compareTo` method decides they are the same
  - `object1.compare(object2, object3)` is similar
- the behavior of the last three depends on the implementation:
  - ideally, they compare the *contents* of the object
  - ideally they are all consistent with each other

  but:
  - they may not do what you think
  - they may have bugs

properties of the equals method

- the equals method should be:
  - reflexive: a.equals(a) should always be true
  - symmetric: a.equals(b) == b.equal(a)
  - transitive: if a.equals(b) and b.equals(c) iff a.equals(c)
  - consistent: successive identical calls should return the same result
  - if a is not null, a.equals(null) == false
- the equals method can be overridden for any class
- if equals is not overridden, it defaults to ==

ordering objects in Java

- the mathematical function `Integer.signum` returns -1 if its argument is negative, 1 if it is positive, and 0 if it is zero
- in Java, there are two kinds of comparison:
- `Comparable<T>` interface, specifies the `int compareTo(T x)` method
- `Comparator<T>` interface, specifies the `int compare(T x1, T x2)` method (which could be static, but isn't)

compare and compareTo

- both comparison methods should have the following properties
  - symmetric: Integer.signum(compare(a, b)) == -Integer.signum(compare(b, a))
  - transitive: if (((compare(a, b) > 0) && ((compare(b, c) > 0)), then ((compare(a, c) > 0)
  - consistent: if (compare(a, b) == 0), then sgn(compare(a, c)) should be the same as sgn(compare(b, c))
- when building ordered linked lists, heaps, or when sorting, can use either interface, e.g. class Collections has two different methods for sorting,
  - static <T extends Comparable<? super T>> void sort(List<T> list)
  - static <T> void sort(List<T> list, Comparator<? super T> c)

applications of sorting

- finding duplicate elements in a list
  - eliminating duplicates: making a list of elements, where each element must occur at most once
- preparing for future binary search
  - dictionary, phone book
- presenting data in an appropriate format, e.g. for printing (bank statement sorted by date)
- comparing two lists to find out which elements are in one, the other, or both
- merging multiple sorted collections into a new sorted collection, with or without duplicates

in-class exercise

- sort by first name
- what algorithm did you use?

Java sorting

- Java has

```
public static void sort(x[] items);
public static void sort(x[] items, int fromIndex, int toIndex);
```

- for type x being any one of:
  - `int`
  - `Object` -- the objects must be `Comparable`
  - `Comparator`, in which case the last argument is the comparator object
- as mentioned above, there is also a sort method declared on lists of objects that are either `Comparable`, or can be compared by a `Comparator.compare()` method
  - too much of a good thing might be confusing!

reminder: selection sort

- start with an unsorted array a
- find the smallest element in the array (at index i)
- swap the element at index i with the element at index 0
- now, find the smallest element in 1..a.length-1
- swap the elements at index i and at index 1
- now, the sub-array with elements 0..1 is sorted
- now, find the smallest element in 2..a.length – 1
- ...
- in-class exercise (in groups of 2 or 3): write code to implement selection sort
- in-class exercise: how long does selection sort take?

reminder: bubble sort

- start with an unsorted array a
- loop through all the elements of array a, swapping any that are not sorted
- repeat until the array is sorted
- in-class exercise: how long might this take? how long does it take in the best case? how do these compare to selection sort?
- historical trivia: if the data is stored on a magnetic tape, how many passes of the tape are needed to sort the entire tape? (this is assuming the tape can store much more data than the memory of the computer)
- current trivia: bubble sort can be done in parallel on n processors...

reminder: 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?

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))

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