We work on developing a selection of sorts.

1. Questions?

2. Implementing a *key* function to render values from an object to compare, for our sorting purposes.

3. A review of functions as objects, from Lecture 21.

4. Sorting techniques we consider:
   
   (a) Bubble sort. An attempt to reverse each out-of-order pair, in several passes by comparing pairs.
   
   (b) Selection sort. Like bubble sort, scan across the array, but keep track of the *location* of the smallest value. After the scan, swap this value with the leftmost value.
   
   (c) Insertion sort. A collection of sorted values is built up by inserting a new, random value in each pass. Compare with selection sort by observing the movement of maximum values.
   
   (d) Quicksort. A low and high values are segregated and then considered recursively.
      
      i. Notice that we can identify where any value is correctly located within the sorted list, without actually sorting the list. For example, the smallest value should end up in location 0. We’ll use a technique, called *partitioning* that finds the appropriate place for a *pivot* by putting smaller values to its left and larger values to its right.
      
      ii. Once partitioned, we can sort (somehow!) the smaller and larger values independently.
      
      iii. This potentially is a fast sort, unless your data is in-order, or in reverse-order. Why?
      
      iv. A fix: pick a pivot randomly. Other fixes?

5. Some reminders about logarithms (Lecture 22 slides are also helpful).

   (a) Logarithms are defined by the relationship \( n = b^{\log_b n} \).
   
   (b) The value \( \log_b n \) is, essentially, the number of times \( n \) must be divided by \( b \) to reduce it to 1.
   
   (c) In practice, we often use \( b = 2 \) because we often develop problem solutions that involve dividing the size of inputs in half.
   
   (d) Since \( \log_a n = \log_a b \cdot \log_b n = c \log_b n \) we can see that logarithms of various bases differ by a constant multiplier.
   
   (e) Computer scientists write \( O(\log n) \) to describe logarithmic function growth, no matter the base.

6. See Lecture 22 for notes on big-O notation.

   (a) It might be helpful to consider the number of times a function is called, multiplied by the number of operations that are performed in that function, to get an upper bound on the run-time of a particular algorithm.