**Computer Science 134C**
*Introduction to Computer Science, in Python*
Lecture #25 (Merge Sort, Binary Search)
*November 12, 2018*

**Keywords**
binary search, decoration,
instrumentation, merge sort

We (finally!) finish up sorting.

1. Questions?
2. Instrumenting quicksort: how is it inefficient?
   (a) The reason this sort runs fast is because we can typically find the correct location for the pivot near the center of the list. When we then use this pivot to split the remaining sort "in two halves" we depend on these "halves" being approximately the same size.
   (b) If the list is inorder then the pivots appear on either end of the list. One of the two sub-lists is trivially empty, while the other is *size* $n - 1$. This is problematic.
   (c) How can we solve this? Randomization.
   (d) We can *instrument* our code to keep track of the actual accounting.
3. Merge sort.
   (a) This sort is very simple, in theory: divide the list in half, sort both lists, and then merge the lists back together.
   (b) The division does *not* depend on the data; the two lists are always within size 1 of each other.
   (c) The merging process is simple: take the smallest value from either list, or if only one list remains, all of the remaining values from that list. This is difficult to do *in place*, but it's easy to merge into a new list in $O(n)$ time.
   (d) If the sort used on smaller lists is merge sort, itself, this sort is $O(n \log n)$.
4. Binary search: just like using a dictionary.
   (a) When searching for the correct location for a value in an ordered list, binary search runs in $O(\log n)$ time.
   (b) We investigate an iterative solution, meant to work even when the value is not found in the list.

$\star$