We focus on implementing two recursive sorts.

1. Questions?

2. 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 – we’ll do this in lab.

3. Merge sort.

   (a) Very simple in theory: split a list into two sublists; sort the sublists; merge them.

   (b) The merge operation: from two sorted lists construct a new sorted list containing all the values.

   (c) The division does not depend on the data; the two lists are always within size 1 of each other.

   (d) 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.

   (e) If the sort used on smaller lists is merge sort, itself, this sort is $O(n \log n)$.


   (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.

5. Some reminders about logarithms.

   (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 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.