Sorting

Quicksort

- Divide elements to be sorted into 2 groups, sort the 2 groups by recursive calls, and then combine the 2 groups into a single array of sorted values (i.e., **divide-and-conquer**)

```
int* quicksort(int A[ ]) {
  if (length of A < 2)
    return A
  else {
        pick some x in A;   // x will be a "pivot point"[1]

        A1 = all elements y in A  such that y < x;   // partitions around the pivot pt.
        A2 = all elements y in A  such that y > x;
        A3 = all elements y in A such that y == x;

        A1 = quicksort(A1);
        A2 = quicksort(A2);

        return concatenation of A1, A3, and A2;
     }
   }
```

Performance depends on selection of pivots (hope it's always near the median value)

But, in general, # times you can continue making 2 partitions out of n items is O(log n)
Doing each partitioning requires O(n) to arrange the items

So this algorithm is **O(n log n)**   …But can be **O(n^2)**

…Again, what if array is already in sorted order???
…What if array is already in descending sorted order???

Characteristics of These O(n log n) Sorting Algorithms

- **Not as simple to implement as O(n^2) sorting algorithms**
- **Worst case time is as good as it gets for sorting**
- **Quicksort is somewhat better suited to not having all elements of array in memory at one time**

[1] Commonly used strategies are to pick the 1st element, pick the element in the middle position, or pick the element in a randomly chosen position.